UNIVERSITY OF WATERLOO

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

# ECE-124 LAB MANUAL

## V1.9A
## LAB 2

Course: ECE-124 Digital Circuits and Systems

# 1 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.

## 1.1 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 (Adder and Multiplexer) designs
4) **CREATE** a digital design for a simple Arithmetic Logic Unit (ALU) and confirm with simulation

## 1.2 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 from Lab 1
2 Design Re-use in VHDL – with Structural coding style

Then the Lab will be getting into background details for the Lab2 project

3 Project Setup for Lab2
4 Lab2 Part A: New VHDL Component – What is a Seven Segment Decoder function?
5 Lab2 Part B: New VHDL Component - What is a Multiplexer or MUX function?
6 Lab2 Part C: Creating a Simple Logic Processor from a Multiplexer Design
7 Lab2 Part D: New VHDL Component – What is an Adder function?
8 Lab2 Part E: A Little bit on Signal Concatenation
9 Lab2-Project Brief

## 1.3 Lab2 Activities

### 1.3.1 Recall from Lab1:

In the Lab1, some basic gate functionality was created in the FPGA. Some of the tools and utilities available within the Quartus FPGA development environment were briefly explored. The top-level design was schematic based. A subordinate block in schematic form was developed and added into the design hierarchy. Later the design went through a process to "synthesize" the design into a logical gates representation so that functional simulations could be completed. Later a functional STIMULUS file was created to stimulate the synthesized gate design in simulation. After completion, the simulation results were compared against the truth tables for the gates implemented.

A VHDL design that was functionally equivalent to the schematic version was also created. Then it was added to the top level of the FPGA design.

As a final design step, some output polarity control was added (one in schematic form, one in VHDL form).

Simulations were used to confirm that the design, whether in schematic or VHDL form, were equivalent.

Then the two methods for design entry were used to create another pair of blocks. This second pair of blocks offered the functionality of a Signal Polarity state control. The concepts of Active-HI and Active-LO signals were covered. The new enhancements were added to the top level design and further simulations were accomplished.

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

### 1.3.2 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.

### 1.3.3 Lab2 VHDL – Architecture Styles

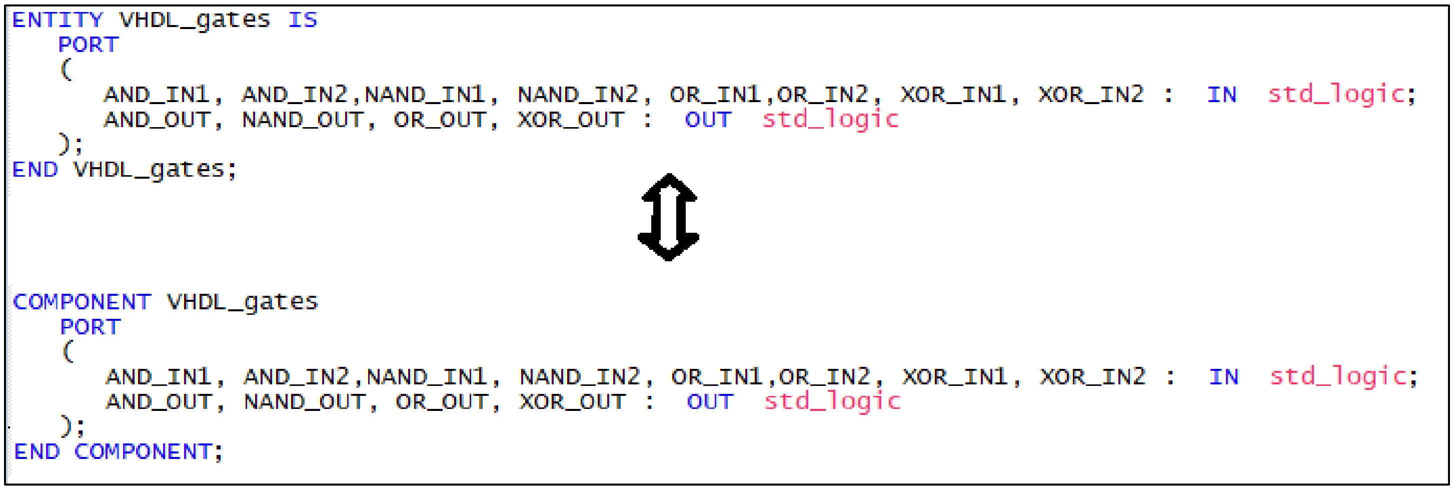
For Lab2 there are just two VHDL coding styles being used in the Architecture section:

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

Lab1 was VERY BASIC in scope, and it used the only one style of VHDL coding (DATAFLOW) that just basic Boolean equations and two-input gates. Lab2 will use that style as well and will also use a second VHDL style called STRUCTURAL.

Quite often a VHDL design is constituted from smaller VHDL blocks connected to form a more complex VHDL function. Using a hierarchy of VHDL blocks in this manner is the STRUCTURAL VHDL style. This will be the style required at the top-level of the design for Lab2 and remaining labs.

Before we begin to use Structural VHDL, we first must understand the Component declaration. It looks very much like an Entity declaration (see below). See an example Entity syntax below for a VHDL file called VHDL_gates and a companion Component declaration in another file that could use the VHDL_gates file. They are very close in syntax content. One thing to remember in the Component Declarations is that the port names must match those defined in the Entity declarations of the VHDL file being used. The nets to the Component Instances may require unique names.



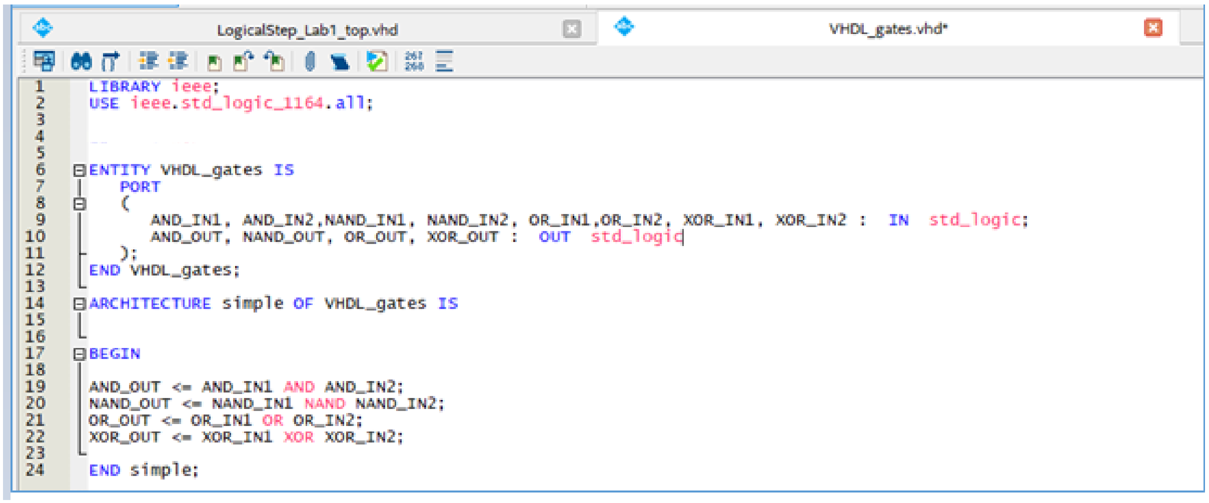
```
ENTITY VHDL_gates IS
   PORT
   (
      AND_IN1, AND_IN2,NAND_IN1, NAND_IN2, OR_IN1,OR_IN2, XOR_IN1, XOR_IN2 :  IN  std_logic;
      AND_OUT, NAND_OUT, OR_OUT, XOR_OUT :  OUT  std_logic
   );
END VHDL_gates;
```

```
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;
```

After a Component is declared inside a VHDL architecture section there is still the signal hook-up to its interfaces to be done. This is done by Port Mapping inputs, outputs or internal signals to an instance of a component. The connections to the instance must be done in the same order as the input, output ports order as outlined in the Component port list.

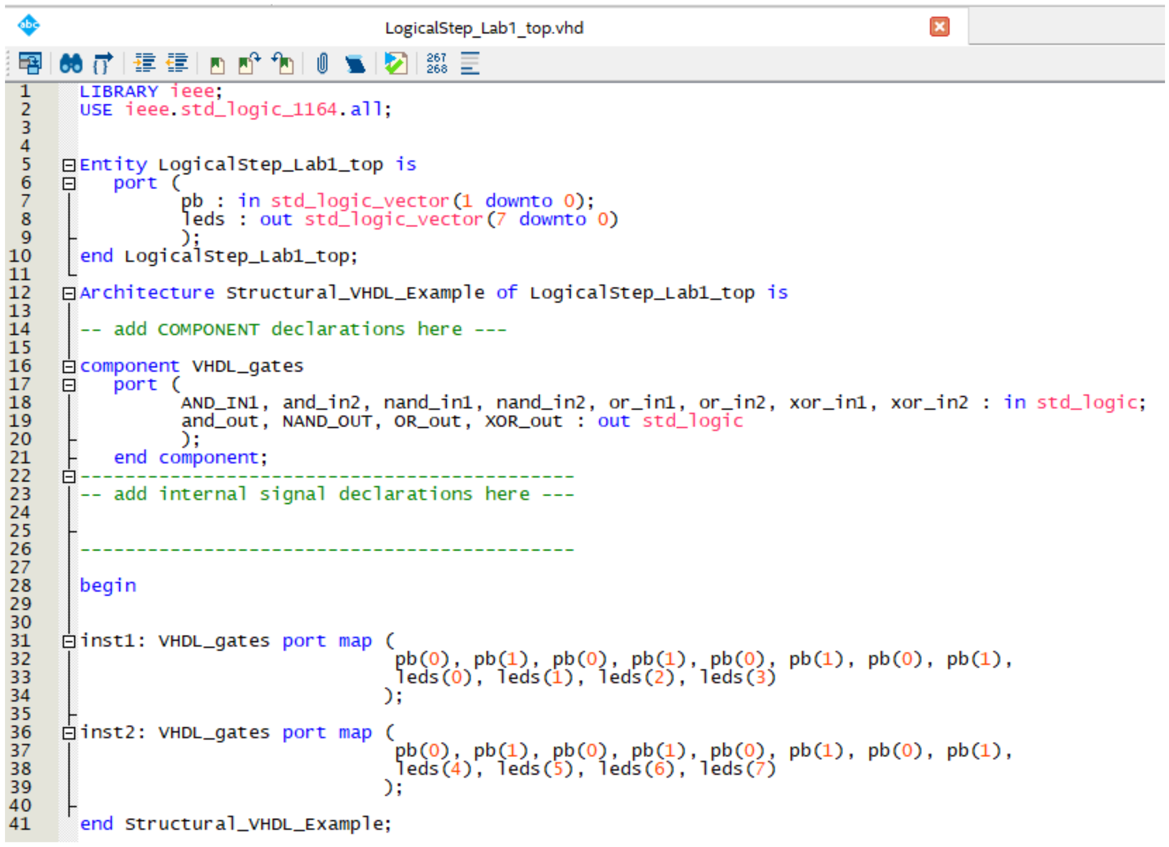
Basic Signal Types such as **std_logic** and **std_logic_vector(msb downto lsb)** are used in VHDL designs to carry logic values. The std_logic type is used for a single logic bit connection. The std_logic_vector type is used to bundle multiple logic signals of type std_logic. The (msb downto lsb) denote the starting and ending index values of the individual std_logic signals in the group.

The Lab1 VHDL file called VHDL_gates.vhd is shown below:



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


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 OF VHDL_gates IS


BEGIN

AND_OUT <= AND_IN1 AND AND_IN2;
NAND_OUT <= NAND_IN1 NAND NAND_IN2;
OR_OUT <= OR_IN1 OR OR_IN2;
XOR_OUT <= XOR_IN1 XOR XOR_IN2;

END simple;
```

If we were to write the LogicalStep_Lab1_top as a VHDL file so that it could use the above VHDL_gates file as a component, then the LogicalStep_Lab1_top could look something like that shown in the figure below with the Component INSTANCES (inst1, inst2) added in the bottom section.



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


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

Architecture Structural_VHDL_Example of LogicalStep_Lab1_top is

-- add COMPONENT declarations here ---

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 here ---


--------------------------------------------

begin


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

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

end Structural_VHDL_Example;
```

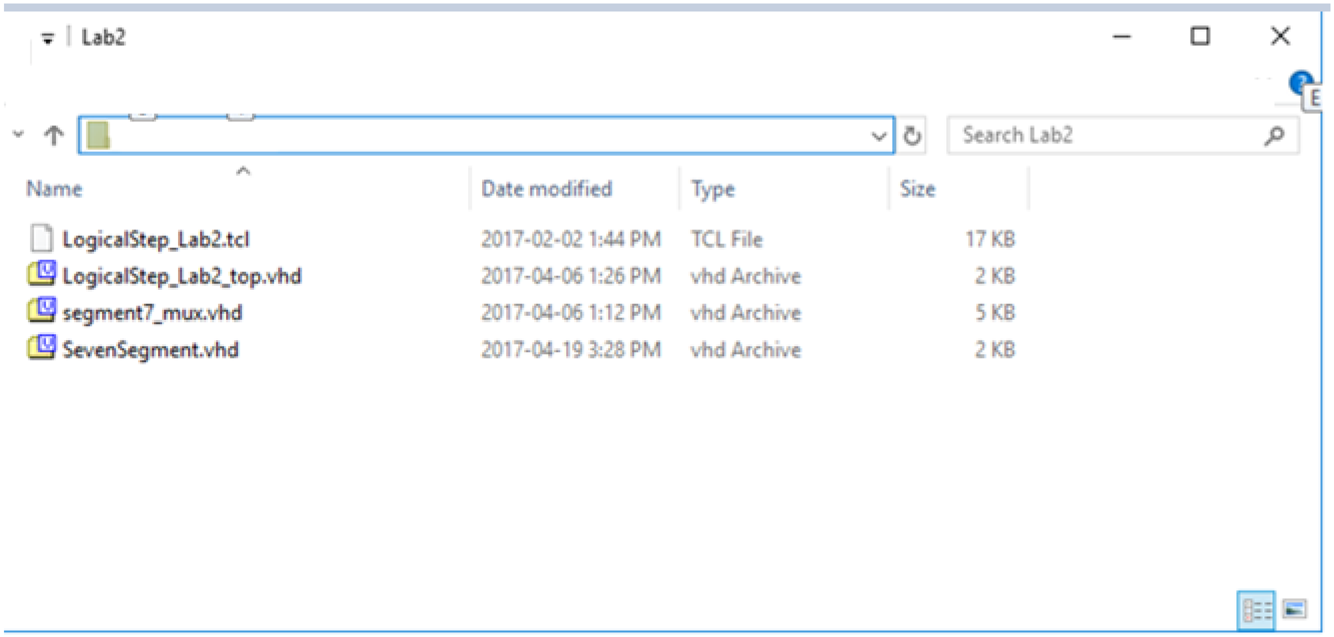
This example above can be used as a reference for the component instantiation exercises in the lab session to connect signals at the top level of a design to ports of components..

Some syntax generalities to notice from the above example:

1) The last element in the list of ports etc. must not have a terminating semi-colon (see lines 8,17,31,33).
2) VHDL signal names or port names are NOT case sensitive in VHDL (see lines 16, 17).
3) **Individual std_logic elements can be separated from a std_logic_vector by using the index (e.g.: pb(0)).**
4) The ports in the example components and instances in the Lab1 VHDL example are shown as 1-bit wide ports. VHDL also offers syntax for multi-bit ports to be used in the Entity and Component declarations This can be seen in the Lab2 entities and components for the sevensegment.vhd file shown later.

### 1.3.4 Project Setup for Lab2

Start the LAB2 like what was done in Lab1 by creating a new project folder on the C: Drive/users/<name>/…/ECE-124 folder. Using the Windows File Explorer go to your ECE-124 folder directory and create a Lab2 subfolder. Go to LEARN and download the Lab2 Zipped folder "Lab2" into the ECE-124/Lab2 folder. Extract the contents into the Lab2 project folder.



| Name | Date modified | Type | Size |
|---|---|---|---|
| LogicalStep_Lab2.tcl | 2017-02-02 1:44 PM | TCL File | 17 KB |
| LogicalStep_Lab2_top.vhd | 2017-04-06 1:26 PM | vhd Archive | 2 KB |
| segment7_mux.vhd | 2017-04-06 1:12 PM | vhd Archive | 5 KB |
| SevenSegment.vhd | 2017-04-19 3:28 PM | vhd Archive | 2 KB |

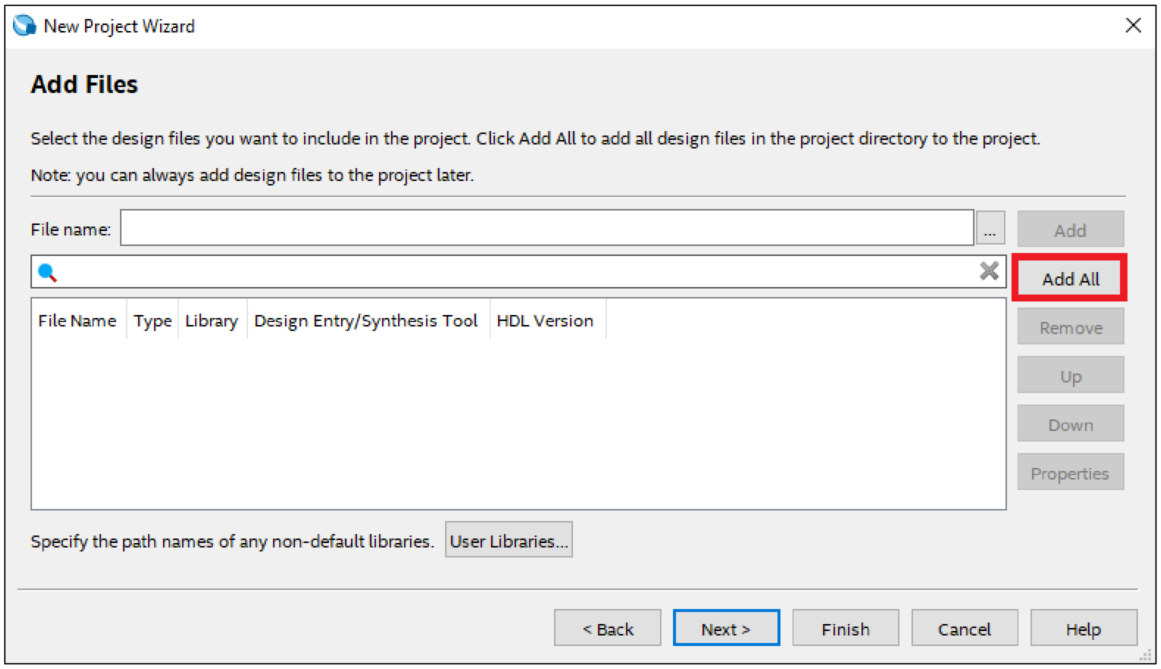
Start up the Quartus Prime platform and begin a new project (Using FILE>New Project Wizard).

Click NEXT to go to the next slide.
The project parameters will now be entered (**MAKE SURE THAT THERE ARE NO SPACES USED**).

Project Folder: **C:/users/<name>/ECE-124/Lab2**
Project Name: **LogicalStep_Lab2**
Project Top Level: **LogicalStep_Lab2_top**

Click NEXT to go to the next slide.

Then click NEXT on the New Project Wizard Dialog Window. Then click NEXT again (with Empty Project). Then the dialog box below shows up.



New Project Wizard

**Add Files**

Select the design files you want to include in the project. Click Add All to add all design files in the project directory to the project.

Note: you can always add design files to the project later.

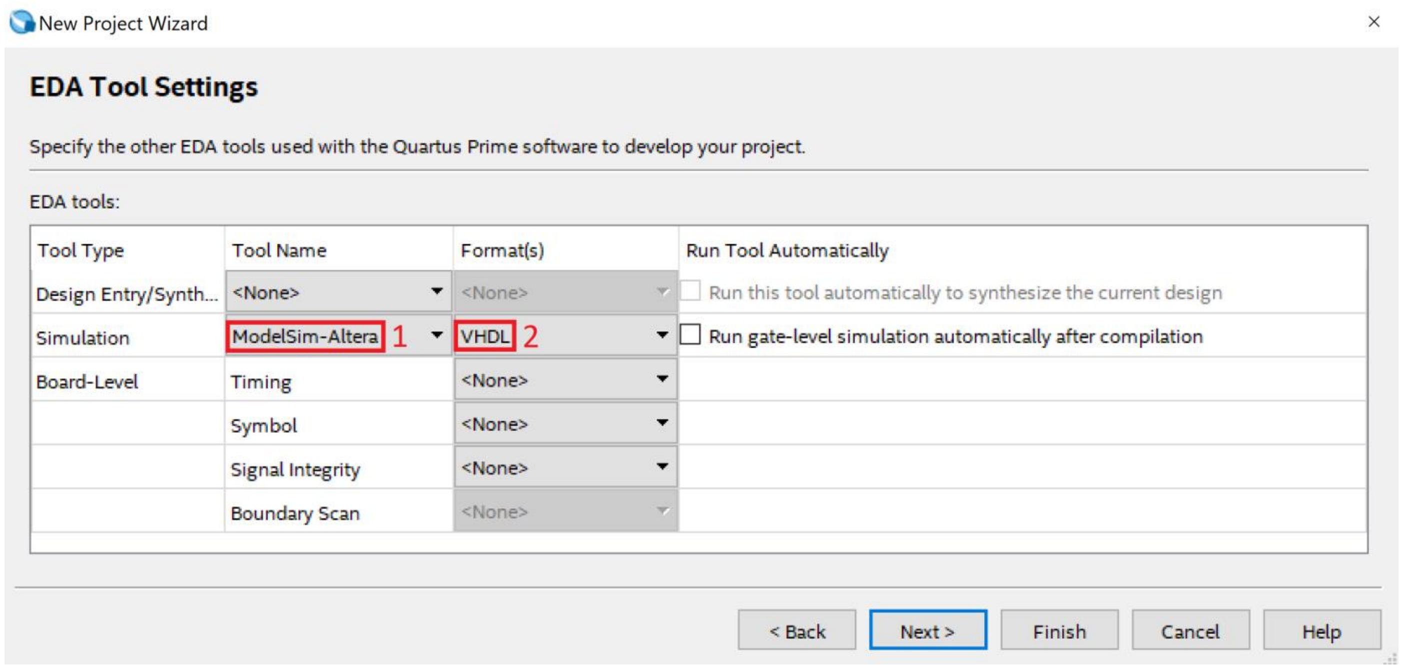
File name:

| File Name | Type | Library | Design Entry/Synthesis Tool | HDL Version |
|---|---|---|---|---|

Specify the path names of any non-default libraries.

Click on the ADD All button. This will bring the starter design files (downloaded from Learn) into the Quartus LogicalStep_Lab2 project.

Click NEXT on dialog windows for Family, Device & Board Settings until you reach the following window (EDA Tool Settings). Then choose (1) **Modelsim-Altera** and (2) **VHDL** from the drop-down menu according to the following figure:


New Project Wizard

EDA Tool Settings

Specify the other EDA tools used with the Quartus Prime software to develop your project.

EDA tools:

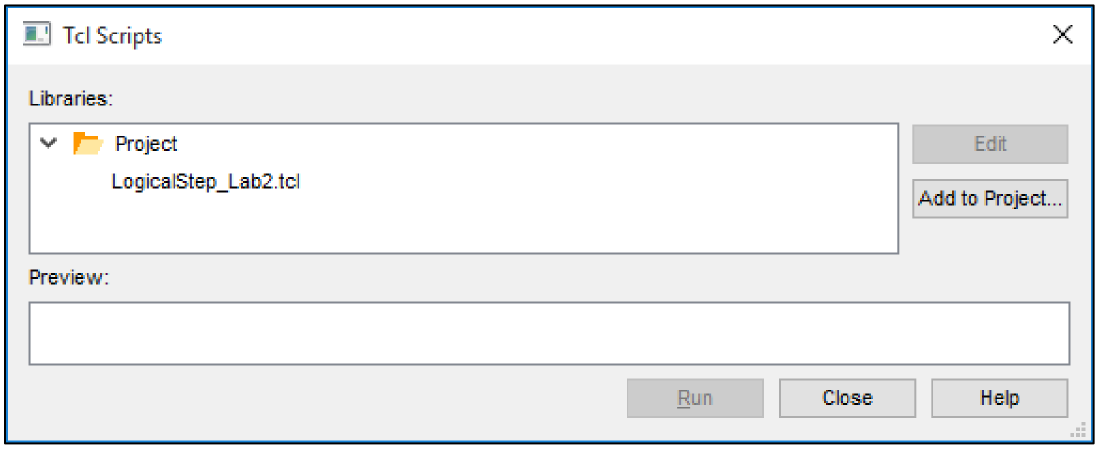
| Tool Type | Tool Name | Format(s) | Run Tool Automatically |
|---|---|---|---|
| Design Entry/Synth... | <None> | <None> | ☐ Run this tool automatically to synthesize the current design |
| Simulation | ModelSim-Altera 1 | VHDL 2 | ☐ Run gate-level simulation automatically after compilation |
| Board-Level | Timing | <None> | |
| | Symbol | <None> | |
| | Signal Integrity | <None> | |
| | Boundary Scan | <None> | |

Now the Project setup is entered. Click FINISH.

**<u>IMPORTANT: Do this step next</u>.**

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 below should appear:


Tcl Scripts

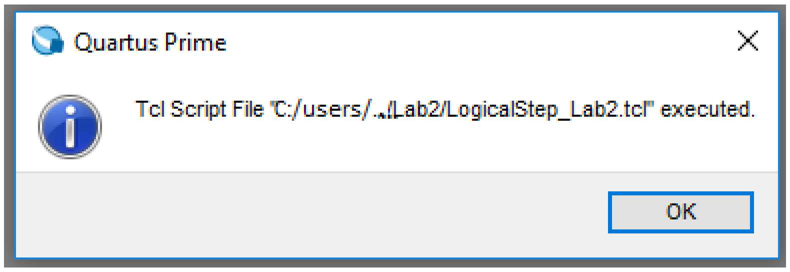
Libraries:

Project
LogicalStep_Lab2.tcl

Preview:

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

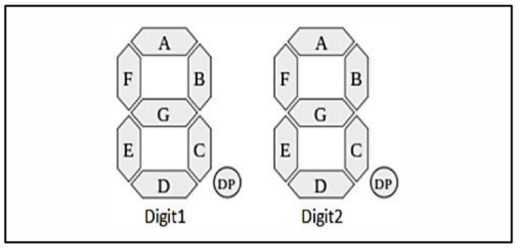
The following should appear when it is finished.



Tcl Script File "C:/users/...(Lab2/LogicalStep_Lab2.tcl" executed.

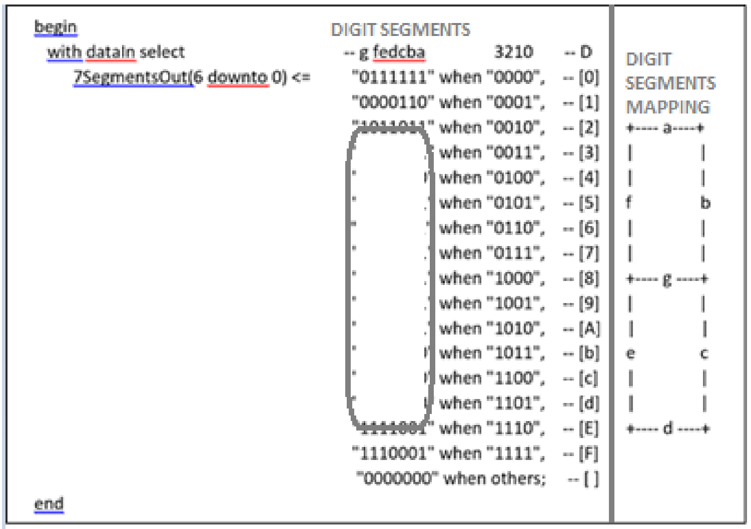
Click the OK Button and close the TCL Scripts window.

## 1.4 Lab 2 Part A: NEW VHDL Component - What is a Seven Segment Decoder?



The LogicalStep board has two seven segment displays available.

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 snip of a VHDL example of this function is shown below with some seven-segment codes hidden.

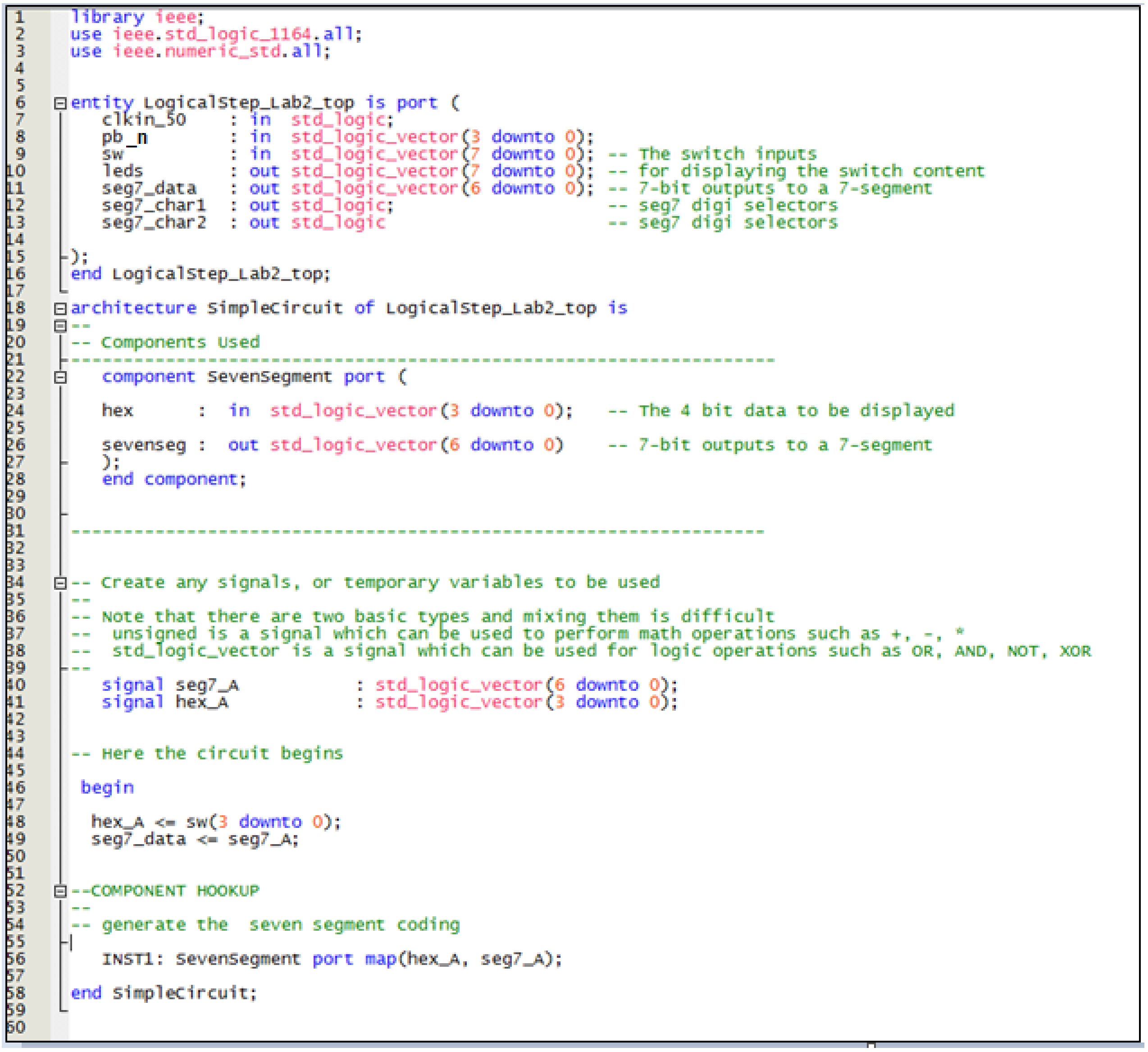


```
begin                                          DIGIT SEGMENTS
  with dataIn select                             -- g fedcba       3210      -- D     DIGIT
     7SegmentsOut(6 downto 0) <=                  "0111111" when "0000",  -- [0]   SEGMENTS
                                                  "0000110"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

Question: For this example, how big of a task do you feel it would be to enter the function above with just schematic gates??

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

Earlier, the top level design file (of 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 screenshot below, that the pins are declared in the Entity section rather than in a schematic.



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


entity LogicalStep_Lab2_top is port (
   clkin_50     : in  std_logic;
   pb_n         : 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_Lab2_top;

architecture SimpleCircuit of LogicalStep_Lab2_top is
--
-- Components Used
-------------------------------------------------------------------
   component SevenSegment port (

   hex      :  in  std_logic_vector(3 downto 0);   -- The 4 bit data to be displayed

   sevenseg :  out std_logic_vector(6 downto 0)    -- 7-bit outputs to a 7-segment
   );
   end component;


------------------------------------------------------------------


-- Create any signals, or temporary variables to be used
--
-- Note that there are two basic types and mixing them is difficult
--  unsigned is a signal which can be used to perform math operations such as +, -, *
--  std_logic_vector is a signal which can be used for logic operations such as OR, AND, NOT, XOR
--
   signal seg7_A         : std_logic_vector(6 downto 0);
   signal hex_A          : std_logic_vector(3 downto 0);


-- Here the circuit begins

 begin

  hex_A <= sw(3 downto 0);
  seg7_data <= seg7_A;


--COMPONENT HOOKUP
--
-- generate the  seven segment coding

   INST1: SevenSegment port map(hex_A, seg7_A);

end SimpleCircuit;
```

In the above example, the component for the SevenSegment decoder is already declared as well as two signal busses (hex_A and seg7_A). Note how (between the "begin" and "end" statements) the busses (signal groupings) are connected:

Signal name being assigned is on the left hand side (hex_A), followed by "<=" and then the signal connection or value is on the right hand side (sw(3 downto 0)).

Note also, how the instantiation of the component "Sevensegment" is done. YOU must add the above VHDL code to your LogicalStep_Lab2_top.vhd file.

### 1.4.1 Hunting for “BUGS”

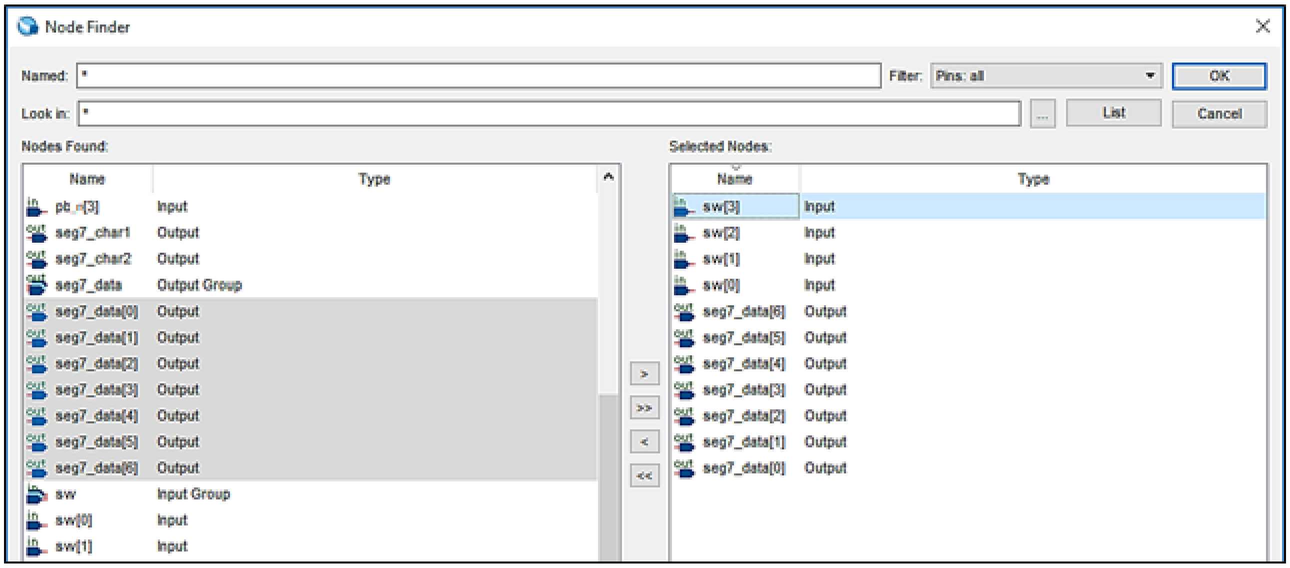
Next, run an ANALYSIS and SYNTHESIZE compilation process to allow a functional simulation model to be created. This can be done by going to the Processing TAB and then selecting “Processing>Start>Analysis and Synthesis” option.

**NOTE: WE WILL NOT BE DOWNLOADING THIS DESIGN DUE TO PIN PROPERTY CONSTRAINTS (pin drive settings) AT THIS STAGE OF THE LAB.THE LOGICALSTEP_LAB2_TOP design is for synthesis and simulation ONLY.**

For this Lab, there are a few small “logic errors planted” into the provided SevenSegment.vhd file that must be discovered during functional simulations and must be corrected to meet the Lab2 project requirements later. There are three bugs in the SevenSegment.vhd table file.

Create a NEW **University Program VWF** simulation file (FILE>NEW>University Program VWF). When the simulation window appears, go to the Node Finder window and 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.



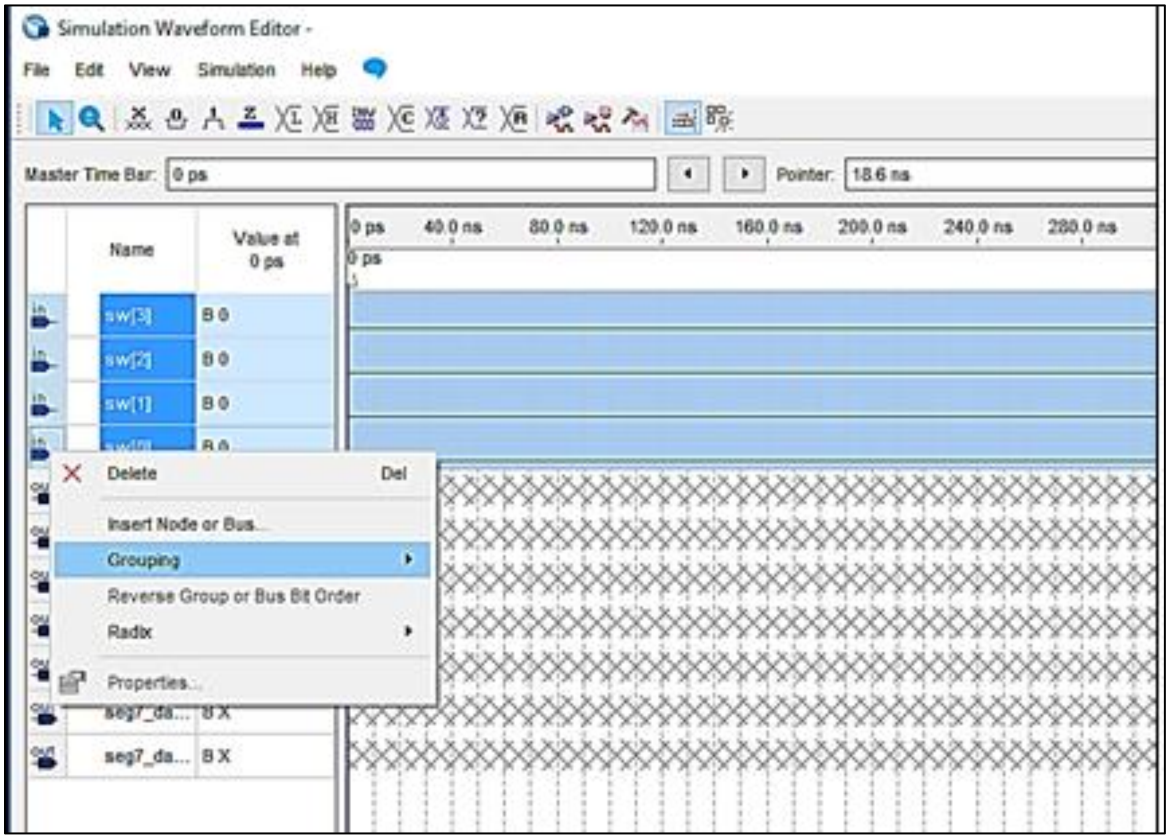
Nodes Found:

| Name | Type |
|---|---|
| pb_n[3] | Input |
| seg7_char1 | Output |
| seg7_char2 | Output |
| seg7_data | Output Group |
| seg7_data[0] | Output |
| seg7_data[1] | Output |
| seg7_data[2] | Output |
| seg7_data[3] | Output |
| seg7_data[4] | Output |
| seg7_data[5] | Output |
| seg7_data[6] | Output |
| sw | Input Group |
| sw[0] | Input |
| sw[1] | Input |

Selected Nodes:

| Name | Type |
|---|---|
| sw[3] | Input |
| sw[2] | Input |
| sw[1] | Input |
| sw[0] | Input |
| seg7_data[6] | Output |
| seg7_data[5] | Output |
| seg7_data[4] | Output |
| seg7_data[3] | Output |
| seg7_data[2] | Output |
| seg7_data[1] | Output |
| seg7_data[0] | Output |

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



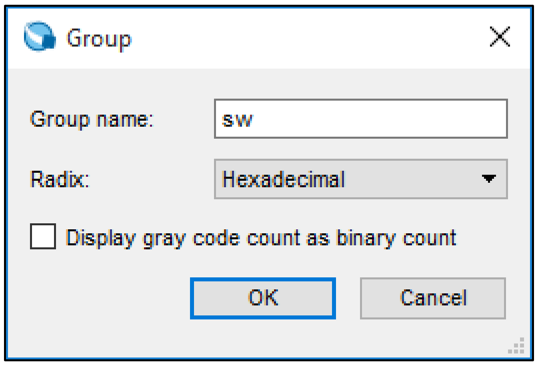
Within the simulation Window, one can group the individual nodes into “groups” or “buses”. This can often save interpretation time of the simulation results. Start with the sw[3..0] nodes. SELECT the nodes in the following order with the Control Key continually pressed:

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 the above figure.

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

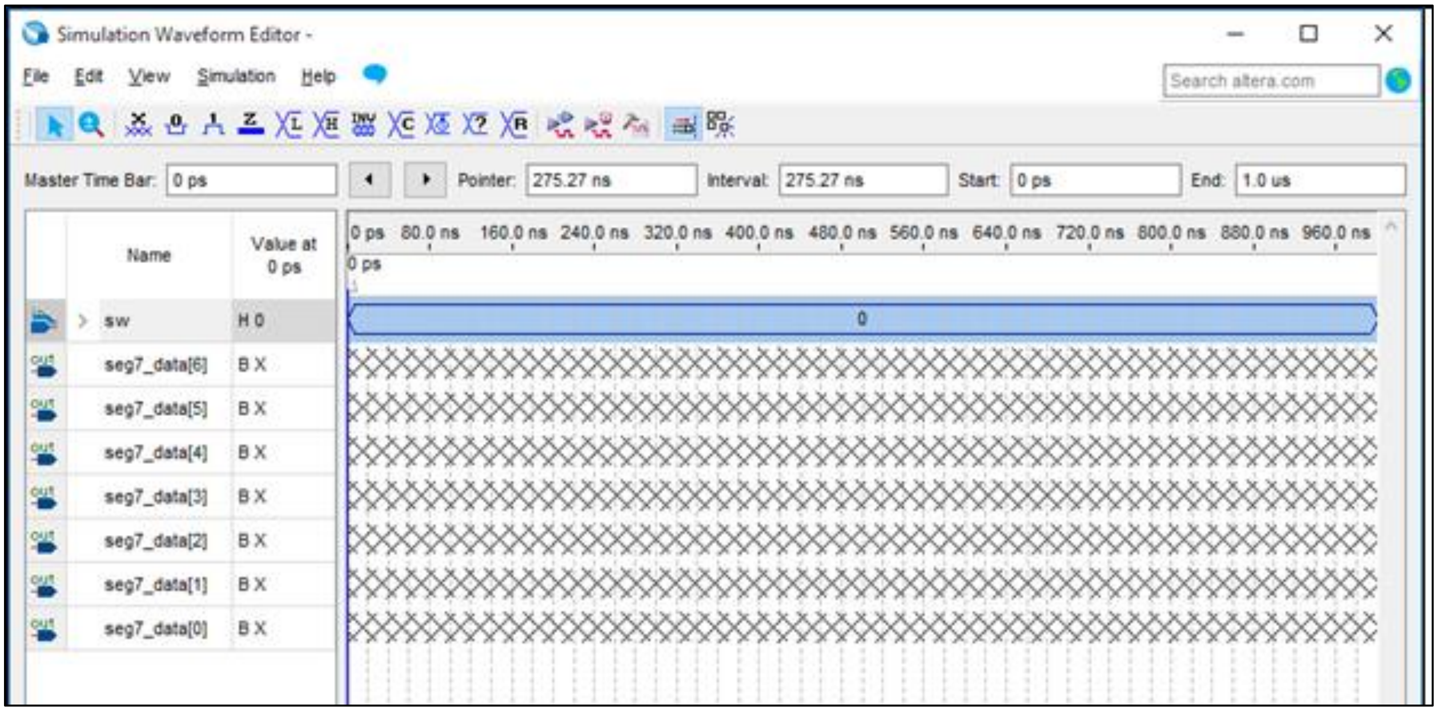

Group

Group name: sw

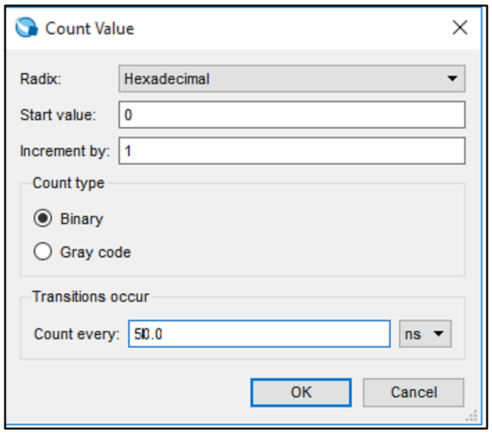
Radix: Hexadecimal

Display gray code count as binary count

Below, 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.



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 below will appear. Set the counting to increment by one every 50 nsec.



Count Value

Radix: Hexadecimal

Start value: 0

Increment by: 1

Count type

[✓ Binary]

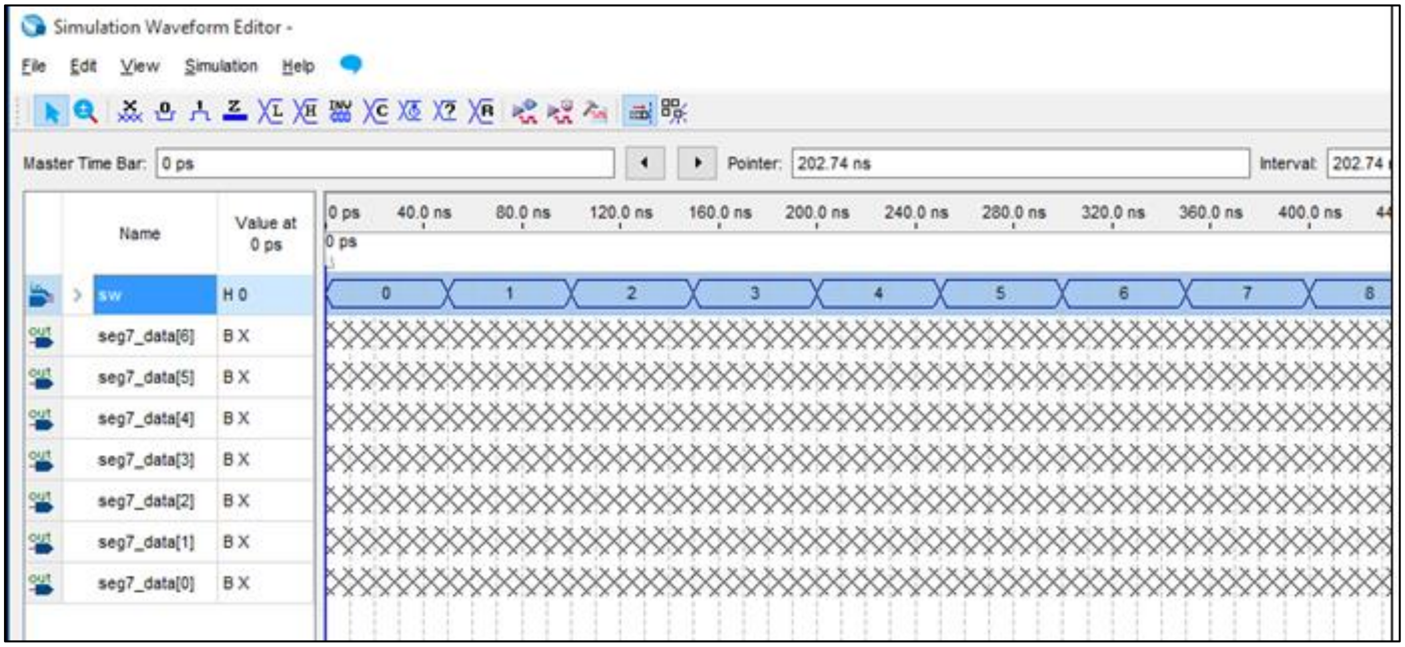
Gray code

Transitions occur

Count every: 50.0 ns

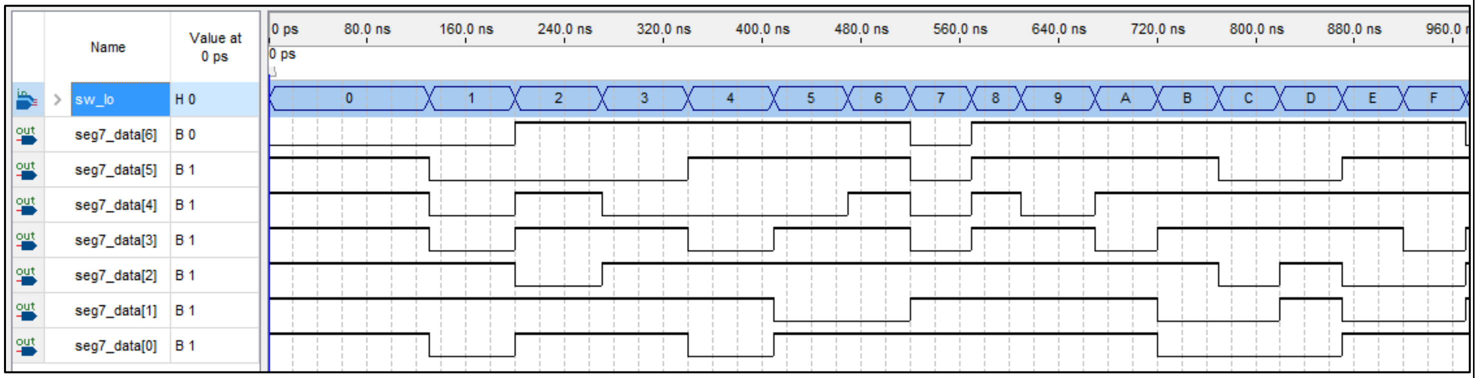
Click OK.

Now you should see the stimulus like the screenshot below:



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 below. Notice that for each HEX value in the simulation, there is a set of column segment bit values. Compare your simulation results with those in the simulation. 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 below.



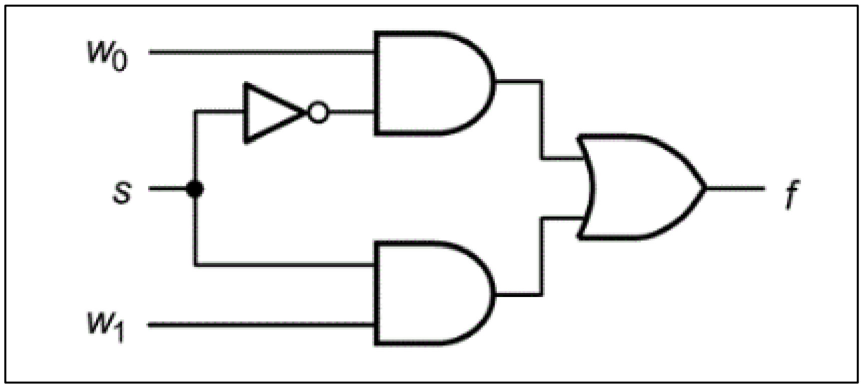
After noting the simulation differences for the HEX values, open the SevenSegment.vhd file (inside Lab2 project directory) to correct the appropriate set of **row** segment values. The seg7_data[0] bits are in the segment "**A**", seg7_data[1] bits are in the segment "**B**", etc. Make the changes and 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 the simulation above.

Save the repaired seven segment design. It is required for your Lab2 Demo.

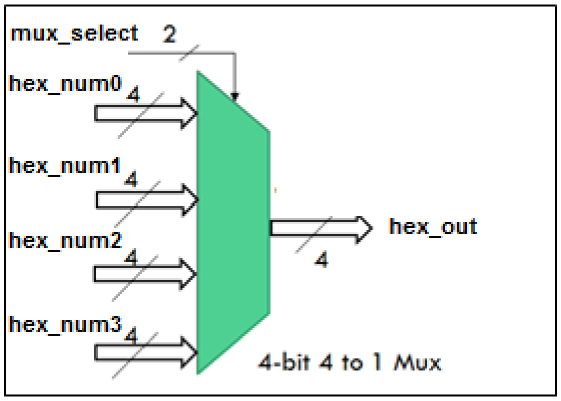
## 1.5 Lab 2 Part B: NEW VHDL Component - What is a Multiplexer or MUX function?

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 …)

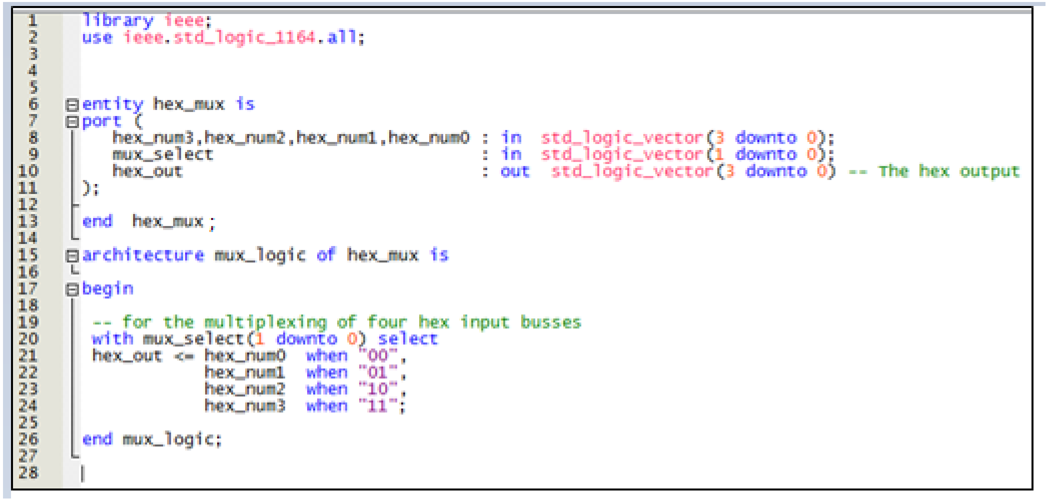


The figure on the left is a simple 2 to 1 multiplexer or MUX function. Its output function “f” will pass thru the w0 input value when the “select” control input “s” is in a LOW state (or a “0”). When “s” is HIGH (or “1”) the output function “f” will equal the value from the w1 input.



A graphical representation example of a QUAD-bit 4 to 1 multiplexer is shown below for your reference on the left. A VHDL companion is shown below. All busses are 4 bits wide. The 2-bit selector can direct any of 4 input ports to the output port.

Create a copy of this hex_mux.vhd file in your Lab2 project folder as shown below.

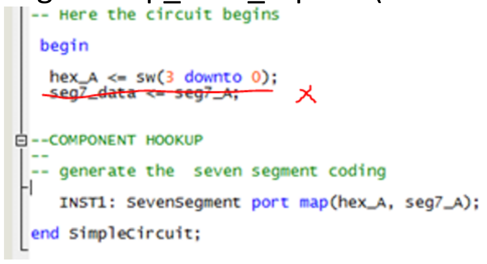


```
library ieee;
use ieee.std_logic_1164.all;


entity hex_mux is
port (
    hex_num3,hex_num2,hex_num1,hex_num0 : in  std_logic_vector(3 downto 0);
    mux_select                          : in  std_logic_vector(1 downto 0);
    hex_out                             : out  std_logic_vector(3 downto 0) -- The hex output
);

end  hex_mux ;

architecture mux_logic of hex_mux is

begin

 -- for the multiplexing of four hex input busses
 with mux_select(1 downto 0) select
 hex_out <= hex_num0  when "00",
            hex_num1  when "01",
            hex_num2  when "10",
            hex_num3  when "11";

end mux_logic;

```

### 1.5.1 Connecting to the External Seven Segment Displays with a Special Multiplexer (segment7_mux)

With the SevenSegment design debugged from Part A, we will now use it in Part B of this Lab. There are two seven segment displays on the LogicalStep board so two SevenSegment decoders will be required. Therefore, there must be a second **instance** of the SevenSegment decoder added to the next version of the LogicalStep_Lab2_top design.

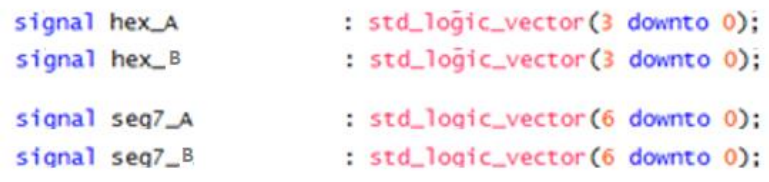
**Disconnect the SevenSegment decoder outputs (INST1) from the seg7_data pins** that were used in Part A. **This can be done by removing the "seg7_data <= seg7_A"** line from the LogicalStep_Lab2_top file (added in Part A).



```
-- Here the circuit begins

begin
 hex_A <= sw(3 downto 0);
 ~~seg7_data <= seg7_A;~~

--COMPONENT HOOKUP
--
-- generate the  seven segment coding
  INST1: SevenSegment port map(hex_A, seg7_A);
end SimpleCircuit;
```

Declare a second 4-bit wide **SIGNAL** group called hex_B. This signal group will have the same width (4 bits) as the hex_A signal bus when declared.

Declare a second 7-bit wide **SIGNAL** group called seg7_B. (seg7_B will look identical in declaration as seg7_A).



```
signal hex_A          : std_logic_vector(3 downto 0);
signal hex_B          : std_logic_vector(3 downto 0);

signal seg7_A         : std_logic_vector(6 downto 0);
signal seg7_B         : std_logic_vector(6 downto 0);
```

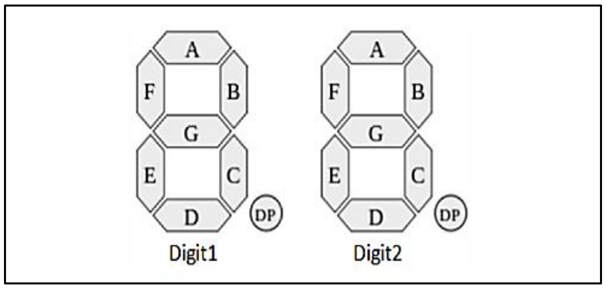
Now connect the sw[7..4] switch inputs to this new signal group called hex_B.

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

INST2: SevenSegment port map (… …);

Connect the other end of the hex_B bus to the INST2 SevenSegment decoder inputs. Connect the output port of INST2 will connect to seg7_B.

Recall:



**Note the orientation** of Digit1 and Digit2 on the board. Your FPGA design will need to use the provided seven segment multiplexer function.

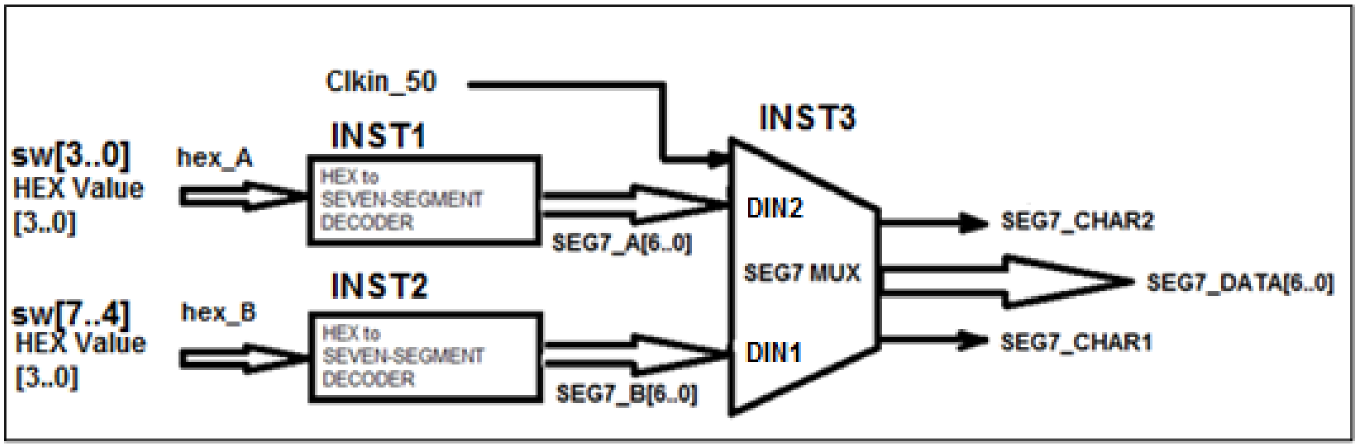
A **unique** (14 input to 7 output) MUX (segment7_mux) will be added to your design for this purpose on the LogicalStep board. It is like having seven 2-to-1 muxes in parallel with an internally controlled select signal inside it (some counter logic is inside for this). This mux **is not suitable** for other purposes.

```
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;
```

Make a declaration like the one above for this component by inserting it in the declaration area (just below the sevensegment component declaration in the LogicalStep_Lab2_top.vhd file). Also, instantiate the **segment7_mux** function instance as INST3 below the INST2 instance just added. For example:

INST3: segment7_mux port map (……);

Refer to the diagram below for the signal connections among instances INST1, INST2 and INST3.



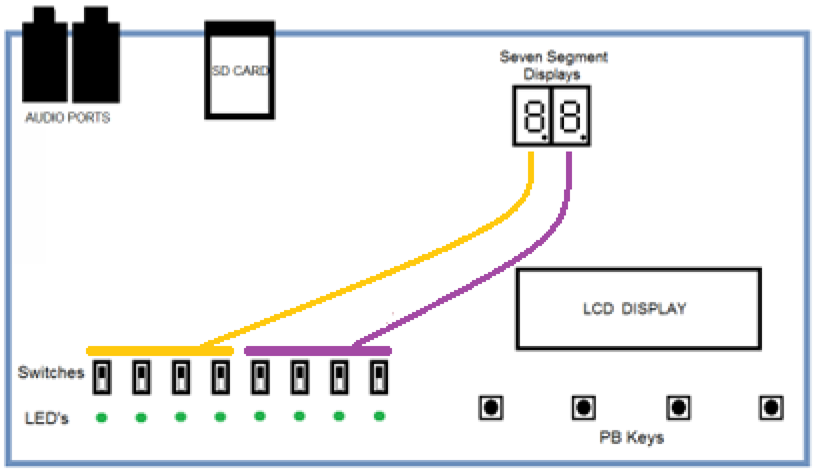
The port mapping to all instances must be done in the same order as declared in the respective Component declaration.

For INST1, hex_A is already connected at the input. Seg7_A is already connected to the INST1 output. Similarly, for INST2 the input connection is to hex_B and its output connection is to seg7_B.

For INST3, the port mapping should match the order of the ports in the SEG7_MUX component declaration port list. So, for the "clk" port, connect the "clkin_50" signal (coming for the input pin "clkin_50" listed in the Entity Section of the LogicalStep_Lab2_top file). Next, the INST1 and INST2 **outputs (seg7_A, seg7_B)** must be connected to the INST3 **inputs** (for **DIN2, DIN1** resp.). Then connect the 7-bit output data port **DOUT** to the **seg7_data(6 downto 0)** port. Then, connect the DIG1 output of INST3 to the output port **seg7_char1** output pin and finally, connect the DIG2 output to the **seg7_char2** output pin.

Do a FULL compile on the design and download it to the FPGA.

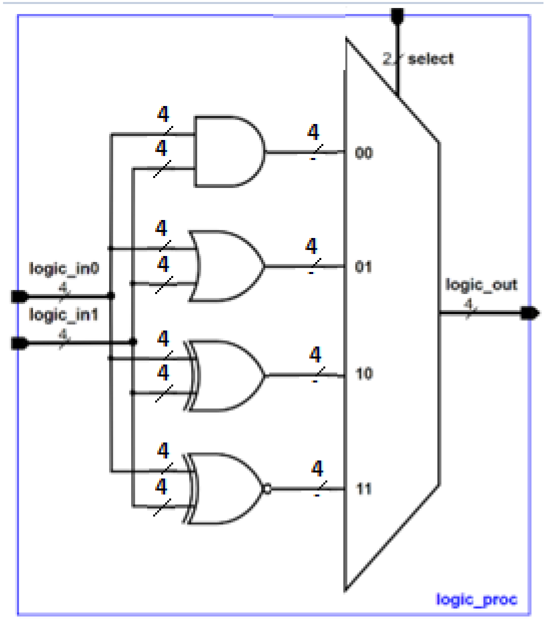
The Dual seven-Segment display should follow the two sets of HEX inputs from the switches sw(3:0) (hex value should be displayed on Digit2) and sw(7:4) (hex value should be displayed on Digit 1).



The digits should be correctly displayed if you were able to debug the sevensegment.vhd file earlier.

This functionality will be incorporated later into your Lab2 Project for demonstration.

## 1.6 Lab2 Part C: Creating a Simple Logic Processor from a Multiplexer Design



Using your knowledge of designing a VHDL 4 to 1 MUX from Part B, to create a simple Logic Processor that has just **two** 4-bit inputs (logic_in0(3 downto 0), logic_in1(3 downto 0)) and a two-bit port for a selecting any one of four logic operations. Then inside the Logic Processor, there will be four different selections available (AND, OR, XOR, XNOR).

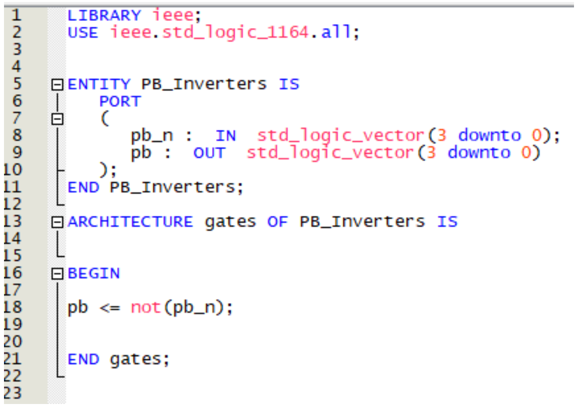
The hex_mux VHDL file described earlier, shows how the with/select construct can be used. For logic processing, the AND function could be inserted inside the with/select construct portion of the hex_mux code replacing the hex_num0 input line in the construct with something like:

(logic_in0 **AND** logic_in1) when “00”.

Make sure that you save this new Logic Processor block with the same name as what you use in the entity section.

### 1.6.1 Experimenting with the Logic Processor Design and Adding Push Buttons as Selectors

Before adding the Logic_Processor function to your LogicalStep_Lab2_top file there will be another function to be created first. This VHDL component will be a function that inverts the pb_n inputs. The outputs from this function will be the pb’s used to select the logic function choice in your Logic_Processor function.



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


ENTITY PB_Inverters IS
    PORT
    (
        pb_n :  IN  std_logic_vector(3 downto 0);
        pb  :  OUT  std_logic_vector(3 downto 0)
    );
END PB_Inverters;

ARCHITECTURE gates OF PB_Inverters IS


BEGIN

pb <= not(pb_n);


END gates;
```

Create a new VHDL block called PB_Inverters with the example code at the left. Declare a Component of this block at the top level and make an Instance of it as well.

Connect the four pb_n input pins at the top-level to the inputs of that instance of the PB_Inverters. Declare a new signal group called pb(3 downto 0) and connect it to the PB_Inverters output port.

To your LogicalStep_Lab2_top.vhd file add a Component declaration in the declaration section to include the Logic Processor you created earlier.

For your Logic Processor, add an instance for it in the top level file (LogicalStep_Lab2_top). Connect the hex_B and hex_A signal groups, previously defined to the 2 input ports (logic_in0, logic_in1 resp.) of the Logic Processor instance.

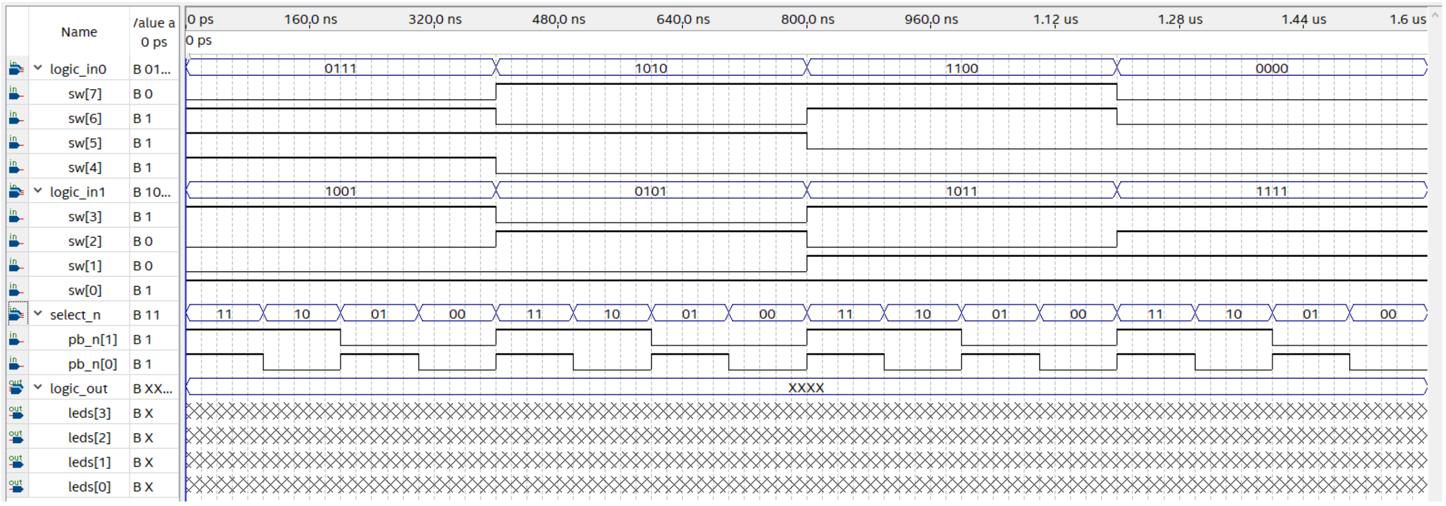
Continuing with the Logic Processor connections from above, connect the Logic_Processor select port (2 bits) to pb (1 downto 0).

Connect the Logic Processor outputs to leds (3 downto 0).

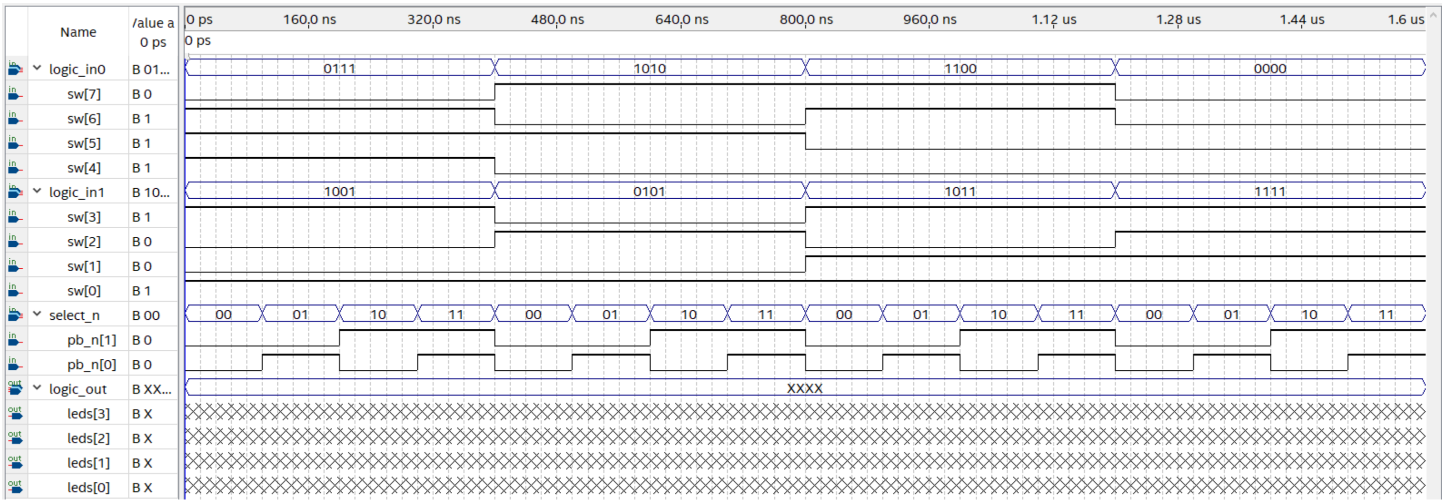
Do an **<u>Analysis and Synthesis</u>** compile. Correct any compile errors. to create a gate-level model for simulation.

Then start up a new simulation window to simulate this function with the following stimulus. In the simulation window add the nodes for all pb_n, sw inputs and leds outputs. Under the EDIT TAB, set the End Time to 1.6 usec. Then group, sw(7 downto 4), sw(3 downto 0), pb_n(1 downto 0), and leds (3 downto 0) and group name them as **logic_in0**, **logic_in1**, **select_n**, and **logic_out**, respectively.

Note that the pb_n input pins <u>MUST BE represented as ACTIVE LOW</u>. Set the radix to **Binary** for all signals. The logic_in0 and logic_in1 values are 400 ns long each. The select_n grouping values are 100 ns. The simulation End time is 1.6 us. You should see the leds output pins change to match the logic operation being done on the inputs. The simulation stimulus for the **<u>FIRST student</u>** on each Team is to be like the following:



The stimulus to be used by the **SECOND student** on each team is to be like the following:



Each student of a team is to Save the respective Logic Processor simulation for his/her Lab2 Report. On Learn, see the rubric on this.

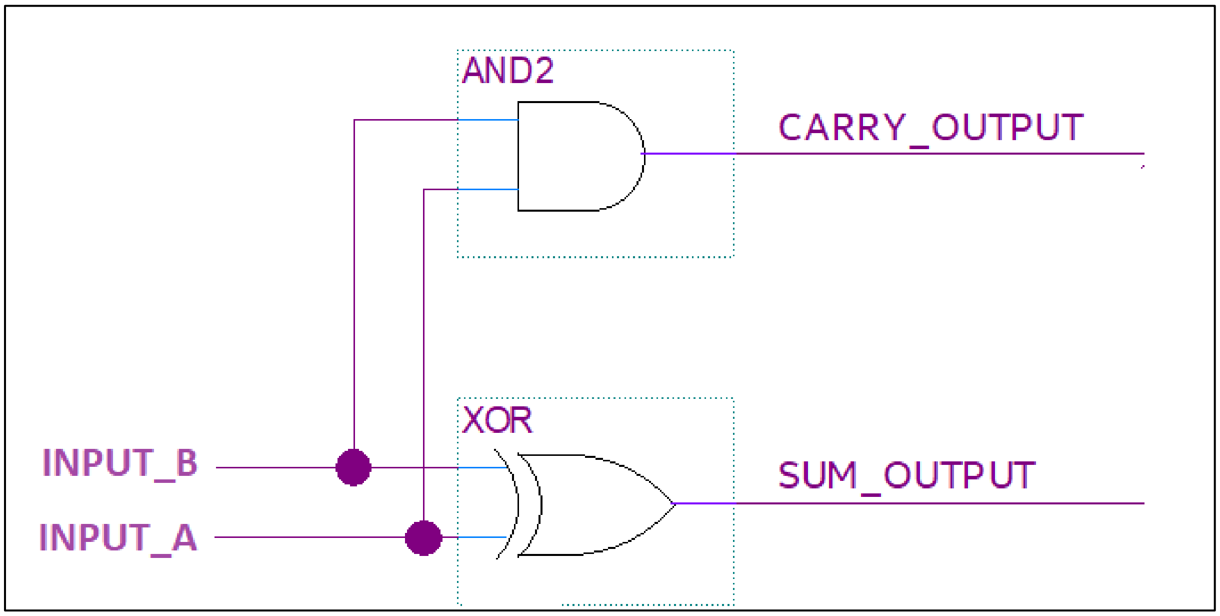
## 1.7 Lab2 Part D: NEW VHDL Component - What is an Adder function?

An Adder can be of very useful in digital logic designs. It can vary in size and in performance.

To begin, we will discuss a typical adder design just having two single-bit logic inputs. The Adder will generate two outputs based on input logic levels. See the following truth table:

| INPUT_B | INPUT_A | SUM OUTPUT | CARRY OUTPUT |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

This functionality can be implemented with logic gates in the following arrangement:



Now this is all well and good for adding just two single-bit operands. But what if we want to build an Adder that must add inputs having more than just one bit? Inputs that represent values greater than just a 1 or a 0 (ie: for a single-bit operand) will have to use more than one bit for each input quantity.

In such situations, there must be multiple bits per input. How do we systematically process multiple bits for an input? And if there is a "Carry_Output" from a lower order bit adder, how do we include it into the higher order single-bit adders? We need to add a modest amount of complexity to the above single-bit logic adder design.

Let's expand the truth table that we just used.

First, we will change the name of the SUM_OUPUT to <u>HALF ADDER SUM OUTPUT</u> and the Carry_Output to <u>HALF ADDER CARRY OUTPUT</u>, respectively. The yellow highlighted part of the table below is identical to the previous one except that it is repeated to cover the span of the new input of CARRY_IN values.

Now the CARRY_IN is inserted into non-highlighted part of the table.

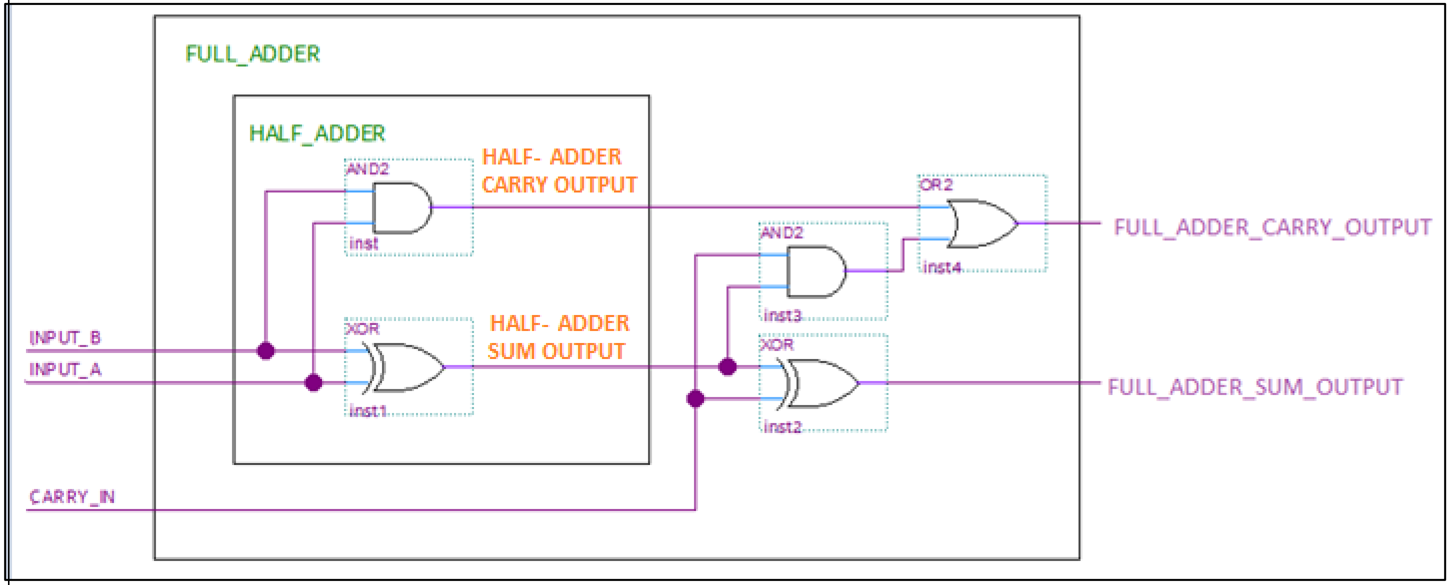
| INPUTS TO HALF ADDER | | INPUTS TO FULL ADDER | | | OUTPUTS | |
|---|---|---|---|---|---|---|
| INPUT _B | INPUT _A | HALF ADDER SUM OUTPUT | HALF ADDER CARRY OUTPUT | CARRY_IN (from lower adder carry-out) | FULL ADDER SUM OUTPUT | FULL ADDER CARRY OUTPUT |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | | 1 | 0 |
| 1 | 0 | 1 | 0 | | 1 | 0 |
| 1 | 1 | 0 | 1 | | 0 | 1 |
| 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | | 0 | 1 |
| 1 | 0 | 1 | 0 | | 0 | 1 |
| 1 | 1 | 0 | 1 | | 1 | 1 |

The CARRY_IN input (along with the two outputs of the HAFL-ADDER) is now included to generate a FULL ADDER function.

Specifically, when CARRY_IN is “0”, the FULL_ADDER_SUM_OUTPUT is the same as the HALF_ADDER_SUM_OUTPUT and the FULL_ADDER_CARRY_OUTPUT is the same as the HALF_ADDER_CARRY_OUTPUT.

But when the CARRY_IN is a “1”, the FULL_ADDER outputs must change to generate the proper FULL_ADDER_SUM_OUTPUT and FULL_ADDER_CARRY_OUTPUT signals.

The above truth table can be represented by the circuit below. The original gates in the highlighted part of the above table, are what is described as a “HALF-ADDER”. When the new gates are added to handle the CARRY_IN input, the whole circuit is described as a FULL_ADDER.



The FULL ADDER circuit is described above in Schematic form. For part of the Lab2 project you have to implement this FULL ADDER function in VHDL form. The above figure is just for a single-bit FULL adder. Let's call it a full_adder_1bit file.

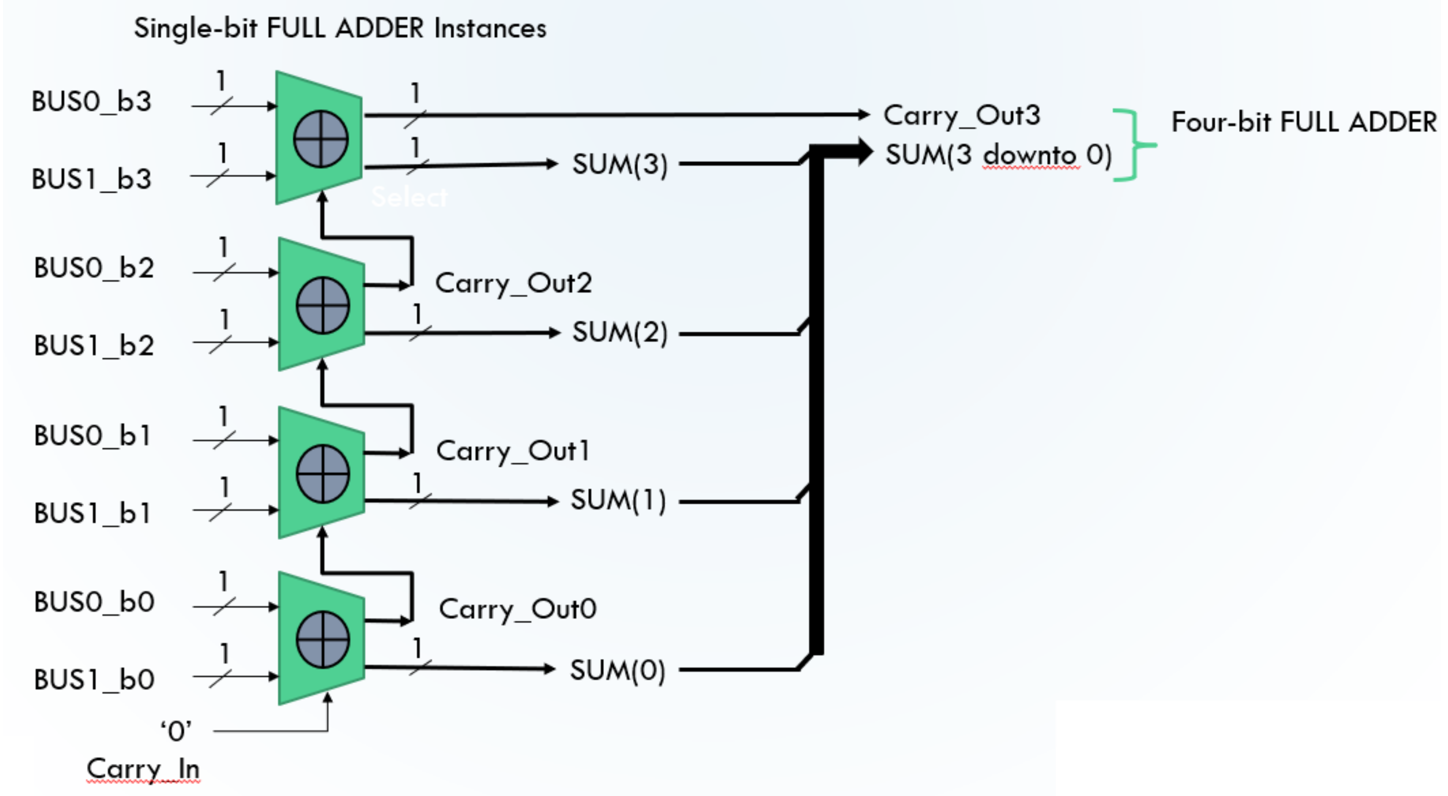
To create larger, multi-bit adders you will have to "daisy-chain" the carry-out of each single-bit Full_Adder to the carry-in of the next single-bit Full_Adder. For example, a four bit FULL_ADDER will require FOUR of the above designs, daisy-chained together.

Because the full_adder_1bit can be used and re-used (with instances), your full_adder_4bit design could have the full_adder_1bit files declared as a Component in the declaration section of the full_adder_4bit file. Then use four instances of the full_adder_1bit, daisy-chained together with the Carry signals.

**Create VHDL** files for the full_adder_1bit.vhd and full_adder_4bit .vhd designs in the Lab2 project folder. Complete the design of the full_adder_1bit.vhd **first** and then the full_adder_4bit.vhd file.

Then in the full_adder_4bit.vhd file, declare the full_adder_1bit as a component and add four instances of it. Then connect signals to the instances as shown below.

## LAB 2 PART D: NEW VHDL: ADDER GATE LOGIC (CONTINUED)



As mentioned above, use four daisy-chained instances of your full_adder_1bit to implement the full_adder_4bit (VHDL Structural design style).

#### 1.7.1.1 Experimenting with the Full_Adder_4bit Design

To test your full_adder_4bit.vhd design at the LogicalStep_Lab2_top.vhd level we want to use the Display digits on the LogicalStep board.

Disconnect the hex_A and hex_B signals from the two sevensegment instances (INST1, INST2).

Connect hex_A and hex_B to the 4-bit inputs of an instance of the full_adder_4bit. Also, assign a '0' (use single quotes) to the cin port of the full_adder_4bit.

Connect the full_adder_4bit hex_sum output to the INST1 sevensegment instance.

The connection for the Carry_Out signal from the full_adder_4bit is explained in the next section (Part E).

### 1.7.1.2 Lab2 Part E: A Little Bit of Signal Concatenation VHDL

Next, a concatenation of “000” and the full_adder_4bit carry_out signal to the INST2 input. Follow the example below to accomplish this.

In order to group signals to other signal groups, you can use the **concatenation operator** in VHDL. The VHDL concatenate operator is ampersand (&).

For example, assuming that signal_A is a 3-bit signal and signal_B is a 1-bit signal you can group these two and create a new 4-bit signal named signal_C.

```
--Declare a new 4-bit signal
signal signal_C          : std_logic_vector (3 downto 0);
--Concatenation
signal_C  <=  signal_A  &  signal_B;
```

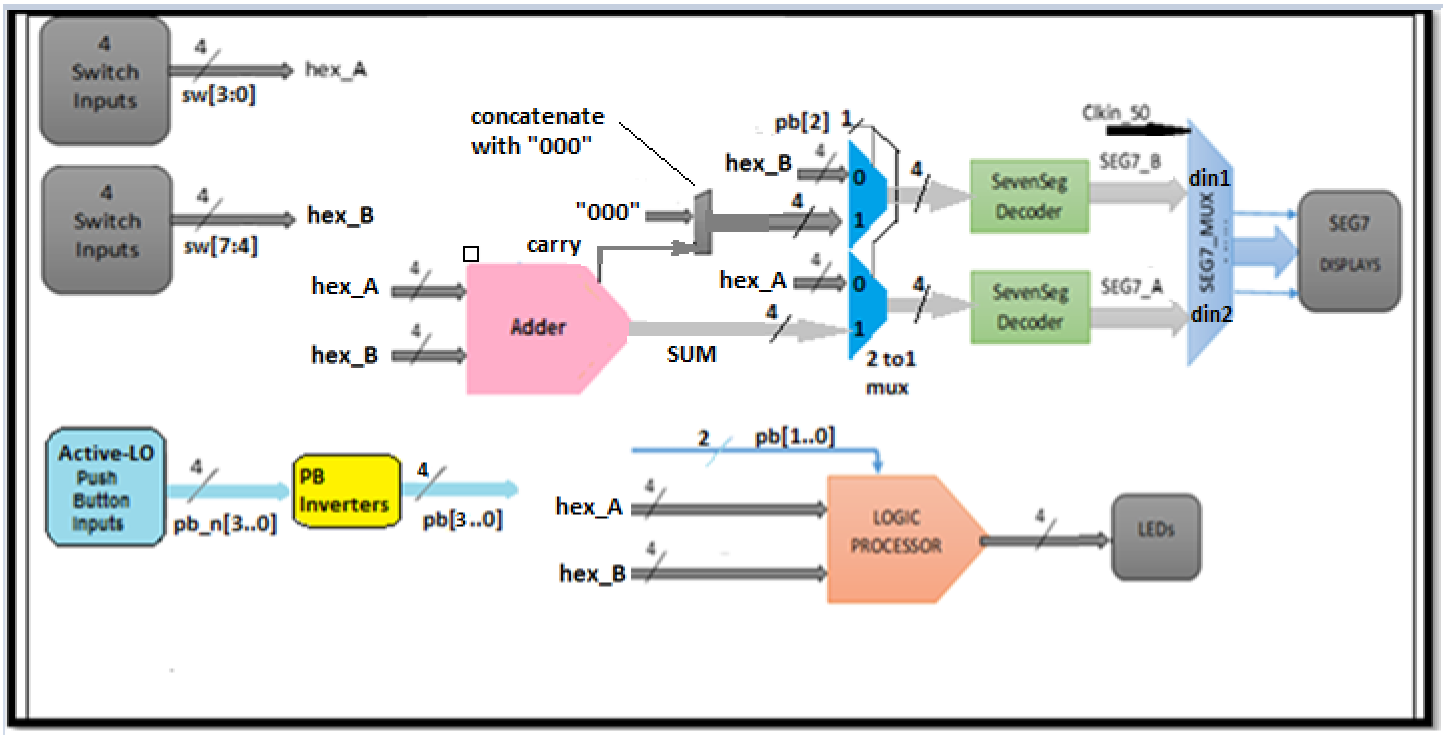
In the concatenation operator example shown above, signal_A will be the MSB of the signal_C group after combining signals. You can use this operator to group onto a 4-bit signal group.
Whatever you name the new 4-bit group signal, connect this new signal grouping as a hex quantity to the INST2 input.

**NOTE:** The most significant Carry_Out signal for the full_adder_4bit can be considered as the MOST SIGNIFICANT BIT of the SUM (ie: a Bit 5 in this example).

Do a FULL Compilation of the design and download it to the LogicalStep board. Test the 4 bit adder design by setting the hex values for hex_A and hex_B with the switches. The hex formatted sum should appear on Digits 1 and 2.

### 1.7.2 Lab2 Project Brief for Demonstration and Lab2 Report

The previous parts of Lab2 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 the computer. The schematic diagram of the top level of the lab2 project is shown below.



**NOTES:**

VHDL STRUCTURAL STYLE MUST BE USED AT THE TOP LEVEL (LogicalStep_Lab2_top.vhd). NO DATAFLOW CONSTRUCTS WILL BE ALLOWED (logic keywords AND, OR, XOR, NAND, INV etc. or with/select constructs etc.) AT THE TOP LEVEL.

The PB_Inverters, various multiplexers, logic_processor, and full_adder_4bit VHDL files have been developed in the previous part of this lab.

Based on your knowledge of multiplexer design, create a new VHDL mux design file for a 2 to 1 multiplexer with two 4-bit input ports and one 4-bit output port. A single-bit Selector port will also be required. Then instantiate all modules at the top level and connect them with appropriate internal signals that you are declaring.

The ALU project for Lab2 will consist of two input hex values. Operand_A will be connected to the ALU over a signal group "hex_A". SW(3 downto 0) will be used for Operand A. Operand_B will be connected to the ALU over a signal group "hex_B". SW(7 downto 4) will be used for Operand B.

There will be an inverter function that can invert all four of the active_low pb_n(3 downto 0). But this design will only need two of the inverted outputs (pb(1 downto 0) from that block.

The Logic Processor will process the two operands in a bit-wise fashion. The Logic Processor will offer 4 different logic operations on the operands and the logical results will appear on leds(3 downto 0).

The pb_n(1 downto 0) will be used to select the logic function type according to the following table:

| Pb_n(1) | Pb_n(0) | Logic Function |
|---|---|---|
| 1 | 1 | AND |
| 1 | 0 | OR |
| 0 | 1 | XOR |
| 0 | 0 | XNOR |

The 4 bit Full Adder and operands are connected to a small multiplexer network.

When pb(2) is '0' (so pb_n(2) is therefore to be a '1'), the multiplexers direct the two operand values to the digit displays (Operand_A on DIGIT2 and Operand_B on DIGIT1).

When pb(2) is '1' (so pb_n(2) is therefore to be a '0'), the multiplexers direct the Adder values to the digit displays (SUM on DIGIT2 and ("000" & carry) on DIGIT1).

Do a FULL Compile on the design, resolve any errors, download to the LogicalStep board to test it out. Have the Lab2 Project ready for demonstration during the next Lab Session.

EACH STUDENT MUST SUBMIT A REPORT. You can have the common Team design for your VHDL files. BUT The simulations and annotations MUST BE YOUR OWN WORK. <u>**Only TEXT fonts will be accepted**</u>.

For your Lab2 Report, you can take screenshots of your VHDL code files but please make sure they are READABLE. Please add your Lab_Session number, Team number, Lab2_REPORT, and both student names at the TOP of each requested VHDL file.

<u>ALL VHDL FILES ARE REQUIRED FOR THIS REPORT (except for the segment7_mux.vhd code).</u>

You can use screenshots to capture your <u>Logic Processor simulation</u> for your Lab2 report. Please use a graphics editor to insert <u>textual</u> annotations to point out areas of interest. (see ECE-124 Simulation Tips and the Example Report files on Learn). No cursive fonts or handwritten characters will be accepted.

Please refer to the Lab2 Report rubric on LEARN.

There are many reports being submitted and the management of all of these files for downloading and marking among the different TA's etc. can be rather INTENSE. So please help out everyone by following the naming formats as described below. <u>**THIS IS REQUIRED**</u>.

See the Required Lab2 report format description below.

## LAB2 Report Submission Requirements

ONE pdf report file (in PDF format with Portrait or Landscape orientation) **for each Group.**

Report Name: **LS20x_Tyy_ LAB2_REPORT_Group_#(.pdf)**

The entire **report must be a single PDF file**. The **filename** must be the same as the title above with the /session/team number/Lab2_REPORT/group number etc.

The report is due in the Learn Dropbox folder on the DUE DATE specified on Learn. A late report submission will have a mark reduction of 5% for the first 24 hours and 10% per day afterward.

The Report Content order:

| |
|---|
| 1)TITLE PAGE: showing the name of the file (e.g.: LS201_Txx_Lab2_REPORT_Group_XX) |
| 2) LogicalStep_Labx_**top.** vhd file (can be a snip(s)):<br><br>- All Team Student Name(s) at the top of the VHDL File<br>- Show all VHDL code for this top-level file. It must use the **STRUCTURAL VHDL** style only.<br>- Try to organize the code into a logical flow for good readability.<br>- Add comments preceded by '- -' to describe the VHDL code |
| 3) Subordinate VHDL files (such as any "Component".vhd files --- can be a snip(s)):<br><br>- All Team Student Name(s) at the top of the VHDL File<br>- Show all the other project VHDL code files next (segment7_mux.vhd is not required).<br>- Try to organize the file code into a logical flow for good readability (**STRUCTURAL/Dataflow VHDL**)<br>- Add comments preceded by '- -' to describe the VHDL coding |
| 4) Supporting documentation requested for in the Lab Report Rubric on Learn:<br>In general for reports:<br>- Simulations with comments (test inserted with a graphics editor) to highlight interesting points.<br>- **Your annotations should illustrate to the MARKER that you understand what is being displayed.**<br>- **Insert the annotated simulation into your report.** |

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.