Introduction

This lab introduces the concept of modular design by guiding you through the process of creating a two-bit full adder from multiple source files, as well as how to use a testbench file to simulate and verify your design.

Objectives

After completing this lab, you will be able to:

- Create modular designs
- Design a testbench file to verify Input/Output behavior
- Use the testbench file with Vivado’s Simulator Tool

Design Description

In this lab we will design a two-bit full adder, as shown in the diagram below.

To do this we will build it from the ground up, starting with a Half Adder. A half adder consists of 2 inputs that pass through an XOR gate for the “Sum” output, and an AND gate for the “Carry” output.
As seen in the Full Adder diagram below, we can link 2 Half Adders together, plus an additional OR gate, to build a Full Adder. This will be done in the VHDL design file, where we will instantiate multiple Half Adder components and link them together using internal signals. These internal signals are shown in the diagram as gray lines instead of black.

With the Full Adder designed, we can then link multiple Full Adders together, cascading them to create a repeatable structure for adding larger numbers. For the purposes of this lab we will stop with 2 Full Adders, leaving us with a 2-bit Full Adder design.
Part 1. Adding/Editing Multiple Design Sources

**Step 1-1:** Create a new project as you did in Lab 0. Once you are at the *Main Project Window*, click on *Add Sources*, then select *Add or create design sources*.

![Add Sources](image1)

**Step 1-2:** In the *Add or Create Design Sources* window, click on the `[+] button` and create 3 new design source files. Name the files as shown in the image below, then click *Finish*.

![Add or Create Design Sources](image2)
Step 1-3: When the *Define Modules* window comes up, click on the *half_adder.vhd* file from the left window first. Change the *Architecture name* to *dataflow* then use the *button* to add the same ports as you did in Lab 0 for the half-adder design. Once you are done, select the *full_adder.vhd* file from the left window.

![Image of Define Modules window with half_adder.vhd selected]

Step 1-4: For the *full_adder.vhd* file, change the *Architecture name* to *mixed*. For the Ports, follow the Full Adder diagram in the design description at the beginning of this Lab. You should have 3 inputs and 2 outputs as shown in the image below.

![Image of Define Modules window with full_adder.vhd selected]
Step 1-5: Finally, select the `full_adder_2bit.vhd` file. Change the Architecture name of this file to `structural`, then add the same ports here as you did for the full adder source file. However, for ports `a`, `b`, and `sum`, check the Bus box. Then in the MSB (Most Significant Bit) column, change the value to 1 for each of those ports. Then click on OK.

Step 1-6: When you look at the Sources window in the Project Manager, you should see the 3 design source files listed under the Design Sources folder. If the `full_adder_2bit.vhd` file is not shown as the Top Level file, right click on it and select Set as Top.
**Step 1-7:** Double click on the `half_adder.vhd` file to open it for editing.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity half_adder is
  Port ( a : in STD_LOGIC;
         b : in STD_LOGIC;
         sum : out STD_LOGIC;
         carry : out STD_LOGIC);
end half_adder;

architecture dataflow of half_adder is
begin
  sum <= a XOR b;
  carry <= a AND b;
end dataflow;
```

**Step 1-8:** In the Architecture block of the `half_adder.vhd`, add the same expressions for the sum and carry ports as you did for the Half Adder in Lab 0, then save the file.

**Step 1-9:** Open the `full_adder.vhd` file for editing.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity full_adder is
  Port ( a : in STD_LOGIC;
         b : in STD_LOGIC;
         carry_in : in STD_LOGIC;
         sum : out STD_LOGIC;
         carry_out : out STD_LOGIC);
end full_adder;

architecture mixed of full_adder is
begin
end mixed;
```
**Step 1-10:** In the `Architecture` block of the `full_adder.vhd` file, before the `begin` statement, add a component declaration of your `half_adder` as shown below. Adding components is simple, just copy the entity declaration of the component you want to use in your current architecture, paste it immediately after the architecture declaration, and replace the name `entity` with `component`, and at the end replace the name of the entity with the word `component`, as shown below.

```vhdl
architecture mixed of full_adder is
component half_adder is
    port(
        a : in STD_LOGIC;
        b : in STD_LOGIC;
        sum : out STD_LOGIC;
        carry : out STD_LOGIC
    );
end component;
```

**Step 1-11:** After the component declaration, add the intermediate signals which will be used for interconnecting the instantiated components.

```vhdl
signal s1, c1, c2 : STD_LOGIC;
begin
```

**Step 1-12:** Finally, after the `begin` statement, we are going to map the ports between `half_adder` and `full_adder` which is referred to as structural modeling, and make an OR gate out of dataflow statements. This is why we named the Architecture for the `full_adder ‘mixed’` to indicate a mixed modeling style. Make sure to save the file.

When connecting (mapping or associating) ports/signals to instantiated components, VHDL allows for two separate association styles: implicit/in-order (positional), and explicit/out-of-order (named). The signal or port that we can map to an instantiated component’s port can be an internal signal of the current architecture, or port of the current entity. The example below shows the explicit named port mapping style, where we first specify the name of the instantiated component’s port, followed by the connect/map symbol “=>”, followed by the name of the signal/port we wish to make the connection to. It is similar to creating a wire-list, and the instantiated component’s port mappings do not have to be in the same order as defined in the component declaration.

Alternatively, we could use the positional port mapping style where we only specify the names of the signals and/or ports the component’s ports are being connected to, but we have to pay
particular attention that the position of the associations have to follow the same order as in the component declaration. For example:

```
ha1_positional: half_adder port map (a, b, s1, c1);
ha2_positional: half_adder port map (carry_in, s1, sum, c2);
```

For now let’s use named association port mapping, as shown below:

```
begin
  ha1: half_adder port map (
    a => a,
    b => b,
    sum => s1,
    carry => c1);
  ha2: half_adder port map (
    a => carry_in,
    b => s1,
    sum => sum,
    carry => c2);
  or_gate: carry_out <= c1 or c2;
end mixed;
```

**Step 1-13:** You will notice now in the Sources window that the hierarchy looks different. This shows how the files are linked together in the file hierarchy and will change again once we edit the `full_adder_2bit.vhd` file.
Step 14: Finally, open the `full_adder_2bit.vhd` file for editing.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity full_adder_2bit is
  Port ( a : in STD_LOGIC_VECTOR (1 downto 0);
       b : in STD_LOGIC_VECTOR (1 downto 0);
       carry_in : in STD_LOGIC;
       sum : out STD_LOGIC_VECTOR (1 downto 0);
       carry_out : out STD_LOGIC);
end full_adder_2bit;

architecture structural of full_adder_2bit is
```

Step 15: As you did in the `full_adder.vhd` file, add a component in the `Architecture` block of the `full_adder_2bit.vhd`. This time you will be adding a `full_adder` component.

```
architecture structural of full_adder_2bit is
  component full_adder is
    port(
      a : in STD_LOGIC;
      b : in STD_LOGIC;
      carry_in : in STD_LOGIC;
      sum : out STD_LOGIC;
      carry_out : out STD_LOGIC
    );
  end component;
```

Step 16: Add the internal signal `cin_fa1` as shown below. We are going to need this internal architecture signal to interconnect the two `full_adder` components.

```
signal cin_fa1 : std_logic;
begin
```
**Step 1-17:** After the *begin* statement, map the ports between the *full_adder_2bit* entity and the *full_adder* components, as shown below. Then save the file.

```vhdl
26 begin
27
28 fa0: full_adder port map (  
29   a => a(0),
30   b => b(0),
31   carry_in => carry_in,
32   sum => sum(0),
33   carry_out => cin_fai);
34
35 fa1: full_adder port map (  
36   a => a(1),
37   b => b(1),
38   carry_in => cin_fai,
39   sum => sum(1),
40   carry_out => carry_out);
41
42 end structural;
```

**Step 1-18:** This is what your *Sources* window should look like after editing and saving the three source files.

**Step 1-19:** If you elaborate the design, the schematic of *full_adder_2bit* should look like this.
**Step 1-20:** You will notice that inside each of the *full_adder* components in the schematic, there is a small (+) button.

![Diagram of full_adder components](image)

**Step 1-21:** If you click on the (+) button, the *full_adder* component will expand to show its own internal components. You will notice this schematic closely resembles the Full Adder diagram from the beginning of the lab.

![Diagram of half_adder components](image)

**Step 1-22:** You can further expand it by clicking on the (+) buttons inside the *half_adder* components, revealing the entire inner workings of the Full Adder.

![Diagram of half_adder components](image)
Part 2. Simulating with Testbench Files

**Step 2-1:** Before we move onto the simulation, click on *Add Sources* and select *Add or create simulation sources*.

**Step 2-2:** In the *Add or create simulation sources* window, click on the **button and select *Add files*. Then navigate to the *Testbench* file you downloaded at the beginning of the Lab.
Step 2-3: Make sure that *Copy sources into project* is selected then click *Finish*.

Step 2-4: Go back to the *Project Manager* and look at the *Sources* window under the *Simulation Sources* folder. If the added *Testbench* file is not *Top Level*, right click the file and select *Set as Top*. 

![Simulation Sources Folder](image-url)
**Step 2-5:** Double click on the `ha_tb.vhd Testbench` file to bring it up in the editing window. As you will be writing your own `Testbench` files for the `full_adder` and `full_adder_2bit`, we will go over some important details about how a `Testbench` file is composed.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity half_adder_tb is
  --A testbench is always an empty entity with no ports
end half_adder_tb;

architecture half_adder_tb_stimulus of half_adder_tb is

  component half_adder is
    port(
      a : in STD_LOGIC;
      b : in STD_LOGIC;
      sum : out STD_LOGIC;
      carry : out STD_LOGIC
    );
  end component;

  --stimulus signals initialized to '0'
  signal a_s, b_s, sum_s, carry_out_s : std_logic := '0';

begin
  --Instantiate a Unit Under Test (UUT) and connect it to
  UUT: half_adder port map (a => a_s, b => b_s, sum => sum_s);
end;
```

**Step 2-6:** The purpose of a `Testbench` is to generate signals that can be applied to an entity so that the resulting output can be compared with expected values. Below is a diagram demonstrating this concept.

![Testbench Diagram]

---

As you will be writing your own `Testbench` files for the `full_adder` and `full_adder_2bit`, we will go over some important details about how a `Testbench` file is composed.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity half_adder_tb is
  --A testbench is always an empty entity with no ports
end half_adder_tb;

architecture half_adder_tb_stimulus of half_adder_tb is

  component half_adder is
    port(
      a : in STD_LOGIC;
      b : in STD_LOGIC;
      sum : out STD_LOGIC;
      carry : out STD_LOGIC
    );
  end component;

  --stimulus signals initialized to '0'
  signal a_s, b_s, sum_s, carry_out_s : std_logic := '0';

begin
  --Instantiate a Unit Under Test (UUT) and connect it to
  UUT: half_adder port map (a => a_s, b => b_s, sum => sum_s);
end;
```
**Step 2-7:** When creating a Testbench file, the first thing to note is that a Testbench file always has an empty entity with no ports.

```
4 entity half_adder_tb is
5    -- A testbench is always an empty entity with no ports
6 end half_adder_tb;
```

**Step 2-8:** As this is a Testbench file for the half_adder, the half_adder is included as a component.

```
8 architecture half_adder_tb_stimulus of half_adder_tb is
9
10 component half_adder is
11    port(
12        a : in STD_LOGIC;
13        b : in STD_LOGIC;
14        sum : out STD_LOGIC;
15        carry : out STD_LOGIC
16    );
17 end component;
```

**Step 2-9:** These are the stimulus signals that will be used to simulate the ports, initialized at ‘0’ for the start of the simulation.

```
19    -- Stimulus signals initialized to '0'
20 signal a_s, b_s, sum_s, carry_out_s : std_logic := '0';
```

**Step 2-10:** Next we use a Unit Under Test, to map the ports of the half_adder component to the internal Testbench stimulus signals.

```
22 begin
23    -- Instantiate a Unit Under Test (UUT) and connect it to internal stimuli
24    UUT: half_adder port map (a => a_s, b => b_s, sum => sum_s, carry => carry_out_s);
```

**Step 2-11:** We set the simulated input signals to generate every possible combination of inputs by staggering the change in the signals from high to low. We can then compare the generated output to the expected results.

```
38    a_s <= not a_s after 10ns;
39    b_s <= not b_s after 20ns;
40
41 end half_adder_tb_stimulus;
```
**Step 2-12:** Run a *Behavioral Simulation* as you did in Lab 0, then restart and run the simulation for the default 40 ns.

**Step 2-13:** Click *Zoom to Fit* and you should have something like this. This helps demonstrate where a *Testbench* file can be helpful. In Lab 0, you used the *force clock* command on each input port. However, there are many designs where it will be impractical to individually set patterns of each input signal. With *Testbench* files, we can quickly set up all the signals in just a few lines of code.
Step 2-14: Here is a useful tip for when you want to include images of these simulations in your reports. If you click on the double arrow (>>) button in the top left corner of the simulation toolbar, you will see the Waveform Options window. You can change the colors of just about everything in the simulation window, including the background color.

![Waveform Options window](image1.png)

Step 2-15: You can also remove the shaded areas under the signal waveforms by unchecking the Draw waveform shadow box. To collapse this window, click on (<<).

![Waveform Options window](image2.png)
Part 3. Creating Constraints Using I/O Planning

**Step 3-1:** Go back to *RTL Analysis* in the *Flow Navigator*. Then in the upper toolbar change *Default Layout* to *I/O Planning*.

**Step 3-2:** This is another way to set the port constraints for a design. What is shown in the screen capture below is the pin layout of the FPGA chip.
Step 3-3: In the bottom window are all of the ports you need to map to pins on the FPGA. These are the Top Level Design ports.

Step 3-4: To begin mapping the ports, first change the I/O Std to LVCMOS33 which is the 3.3-volt standard. Then change the Site of the pins as shown below. This can be done by either dragging and dropping each of the ports on to the specific pin in the grid array above, or by using the drop-down menu under Site.
Step 3-5: Go up and click on *File* and select *Save Constraints*. Name the constraint file whatever makes sense to the project.

![Save Constraints](image1)

Step 3-6: Go back to *Project Manager* and look in the *Sources* window under the *Constraints* folder.

![Project Manager](image2)
**Step 3-7:** If you double click on the constraints file to open it in the right window, you will see a more condensed version of the *Master XDC* file from Lab 0.

```
1 | set_property IOSTANDARD LVCMOS33 [get_ports {a[1]}]
2 | set_property IOSTANDARD LVCMOS33 [get_ports {a[0]}]
3 | set_property IOSTANDARD LVCMOS33 [get_ports {b[1]}]
4 | set_property IOSTANDARD LVCMOS33 [get_ports {b[0]}]
5 | set_property IOSTANDARD LVCMOS33 [get_ports {sum[1]}]
6 | set_property IOSTANDARD LVCMOS33 [get_ports {sum[0]}]
7 | set_property IOSTANDARD LVCMOS33 [get_ports carry_in]
8 | set_property IOSTANDARD LVCMOS33 [get_ports carry_out]
9 | set_property PACKAGE_PIN T16 [get_ports {a[1]}]
10 | set_property PACKAGE_PIN M13 [get_ports {a[0]}]
11 | set_property PACKAGE_PIN P15 [get_ports {b[1]}]
12 | set_property PACKAGE_PIN G15 [get_ports {b[0]}]
13 | set_property PACKAGE_PIN M15 [get_ports {sum[1]}]
14 | set_property PACKAGE_PIN M14 [get_ports {sum[0]}]
15 | set_property PACKAGE_PIN D18 [get_ports carry_out]
16 | set_property PACKAGE_PIN R18 [get_ports carry_in]
```

**Step 3-8:** With these steps completed you can move on to Synthesizing and Implementing your design.
Part 4. Viewing the Implemented Design

Step 4-1: After your design finishes implementing, select *Open Implemented Design*.

Step 4-2: Under *Implemented Design*, we have the *Netlist* window. There are two folders here. The *Nets* folder and the *Leaf Cells* folder. You can think of Nets as the “wires” that connect the pins in the circuit together. This means that the *Nets* folder is basically a wire list. A Leaf is used in Hierarchical design to describe a cell in its most simple state, containing no instances of other cells. The *Leaf Cells* folder here contains all of the leaf cells inside the top-level module, *full_adder_2bit*. You’ll notice that each has an abbreviated description next to it, such as *IBUF* for Input Buffer. Among these are *LUT3* and *LUT5*. These are the Look Up Tables used in the Hardware Implementation of the design. Click on the one described as *LUT3* as shown below.
**Step 4-3:** In the *Cell Properties* window below, click on *Truth Table* to see the truth table for this Look Up Table.
Step 4-4: In the window on the right we see a diagram detailing the FPGA device resources used in the implemented design. You’ll notice that the design is divided into four regions. These are individual clock regions.

Step 4-6: If you click on a spot on the diagram and hold it as you drag the mouse, you can zoom in to the selected region to get a more detailed look at the smaller components.
Step 4-7: In the image below, the green blocks are DSPs, the red blocks are RAMs, and the blue blocks are SLICEs. Zoom in further to the highlighted region shown below.

Step 4-8: The individual Slices are now visible with their internal makeup shown in greater detail. Each slice contains four 6-input LUTs, with three MUXs to generate 8-input functions from the four LUTs. There is also a series MUXs and XOR gates that act as a Carry Chain, which can be cascaded for wider addition and subtraction functions. Finally there are eight Flip-Flops, four of which are DFF only, while the other four can be used as DFF or Latches.
**Step 4-9:** You’ll notice in this slice, the two bottom LUTs are highlighted. That means these are used in our implemented design.
Step 4-11: Go back to Project Summary and look under the Utilization window. Click on the Post-Implementation tab then select Table. You can see that two LUTs are utilized. These are the two we saw highlighted the previous image. Keep in mind that this is 2 LUTs out of an available 17600. This means we are barely using a fraction of the available LUTs on the board.

Step 4-11: With these steps completed, you can move on to writing the Testbench files for the full_adder and full_adder_2bit. Once you have done this you can then program the ZYBO board as you did in Part 4 of Lab 0.