
ECE 281 IN CLASS EXERCISE 2

HALF-ADDER IN VHDL (DUE LESSON 8)

OVERVIEW

So far in this course you have been learning how to design basic combinational logic circuits. However, most logic hardware prototyping is done using a hardware description language such as VHDL. VHDL allows you to boil the essence of a hardware schematic down to a few lines of code. In this in-class exercise you will be implementing a half-adder logic design in VHDL.

OBJECTIVES

The objectives of this in-class exercise are for you to:

- Implement and test a simple logic design (half-adder) using VHDL
- Gain more experience using tools (Git, Xilinx Vivado)

DESIRED END STATE

You will provide a live demo to your instructor or provide proof of functionality to receive credit (Due Lesson 8). You must also turn in the files you used through Git. Your instructor will be able to view your ICE2 folder through Git. The folder will contain at least the following:

1. VHDL files used (half-adder implementation and testbench) in a **code** folder
2. Constraints (.xdc) file in a **code** folder
3. .bit file used to program board

INTRODUCTION

VHDL

VHDL, which stands for Very High Speed Integrated Circuits (VHSIC) Hardware Description Language, is a language that allows you to **describe hardware**. It was developed in 1981 by the DoD and remains one of the two most common HDLs (the other being Verilog).

It is important to note that **VHDL is not a programming language**. Again, you use it to describe hardware. Thus, it is important to become familiar with common constructs, or idioms, that allow you to build the desired hardware.

Finally, since you are describing hardware, you should be aware that **most statements can be considered concurrent rather than happening sequentially**. The exception to this is some statements within a process. More details on VHDL will be provided later in this in-class exercise and throughout the course.

HALF-ADDER AND DESIGN FLOW

A half-adder takes two single-bit inputs and outputs their sum. You can read more about half-adders in Section 10.1 of your textbook. A truth table, schematic symbol, and logic equations for a half-adder are shown in Figure 1. We will follow the basic digital design steps shown in Figure 2.

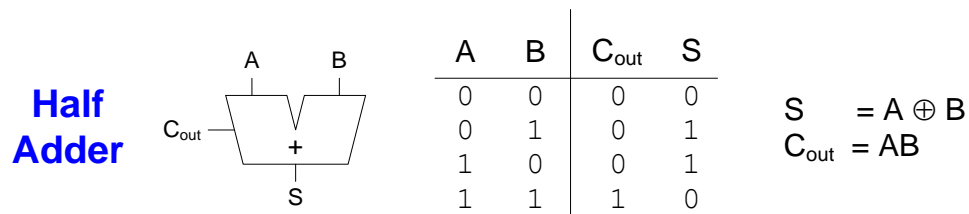


Figure 1 – Half-adder schematic symbol, truth table, and logic equations

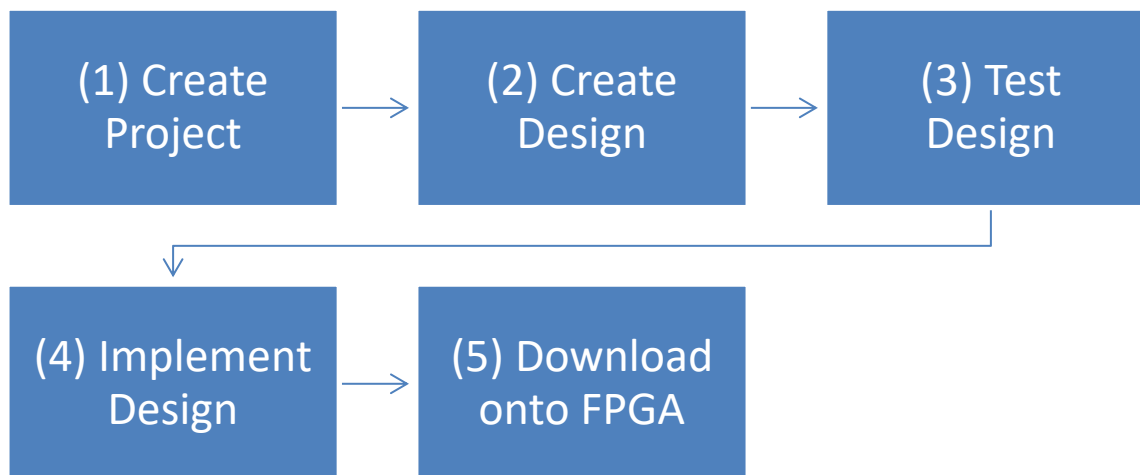
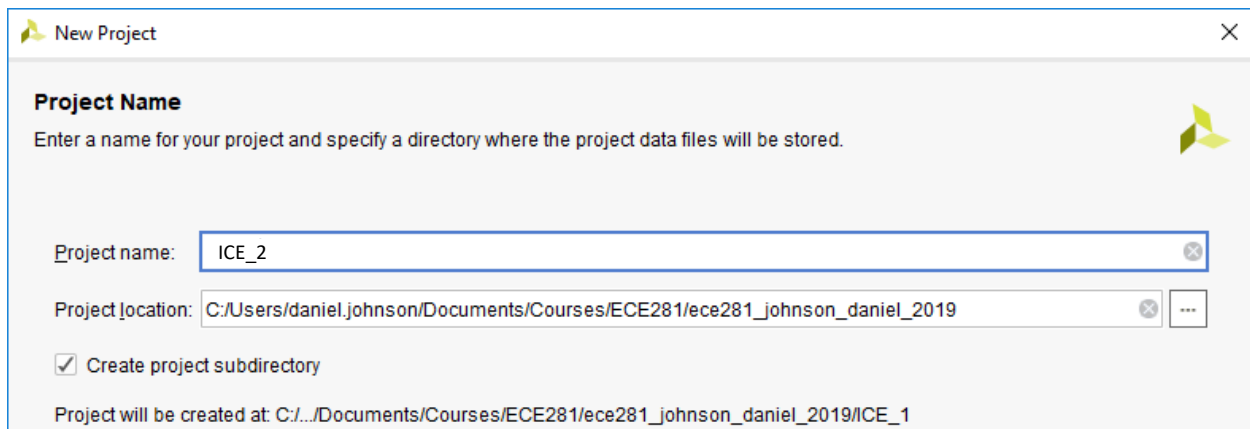


Figure 2 – Generalized design process for digital design projects

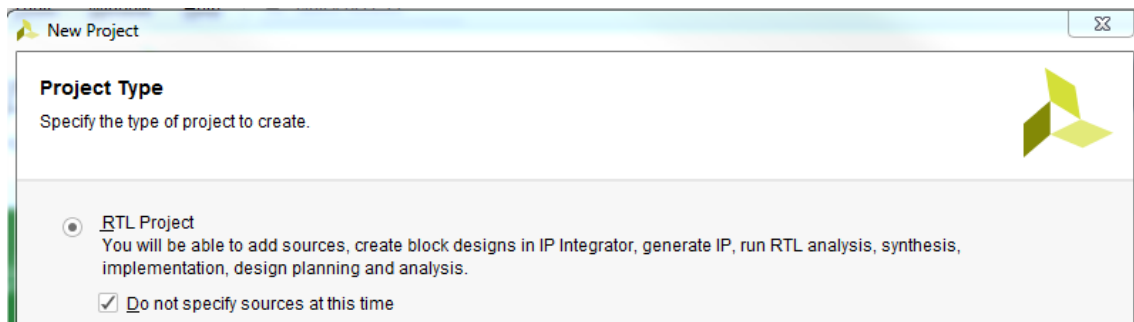
1. CREATE A VIVADO PROJECT

At this point you should NOT have a ICE2 directory in your git repository. Instead, Vivado will create that directory for you when you create a project.

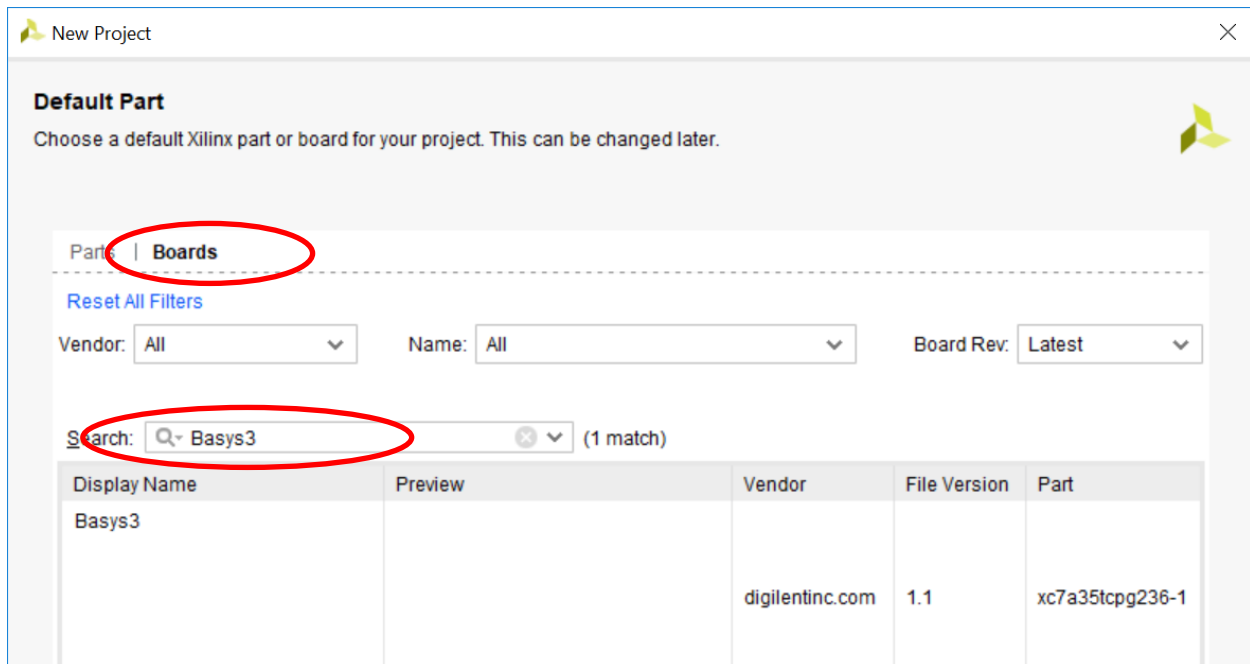
- If you have not already installed the board files, **make sure Vivado is closed**, and then copy the **basys3** directory from:
 - [Teams](#)
 - [Provided Install Directory](#)
- to your Vivado installation at:
 - C:\Xilinx\Vivado\2018.2\data\boards\board_files
- Launch Vivado (yes, sometimes it takes a minute)
- Click File→New Project
- First Window
 - Click “Next”
- Second Window
 - **Project Name:** ICE_2
 - **Location:** Click the “...” and navigate to your git repository
 - Leave “Create project subdirectory” checked
 - Click “Next”



- Third Window (Project Type)
 - Select “RTL Project”
 - Leave “Do not specify sources at this time” checked
 - Click “Next”



- Fourth Window (Default Part)
 - Click on **Boards**
 - Select **Basys3** from the “Display Name” drop down menu
 - Click “Next”



- Fifth Window (New Project Summary)
 - Click “Finish”

2. CREATE CODE AND IMAGES DIRECTORIES AND ADD FILES TO THEM

Navigate to your ICE2 folder in Windows Explorer (file manager) and ensure there is:

- A **code** directory with the following given files
 - halfAdder.vhd and halfAdder_tb.vhd
 - In future labs, you might copy in the ECE_template.vhd and ECE_template_tb.vhd files instead (and then rename them)
 - Constraints file (Basys3_Master.xdc)

Add/stage and commit your changes

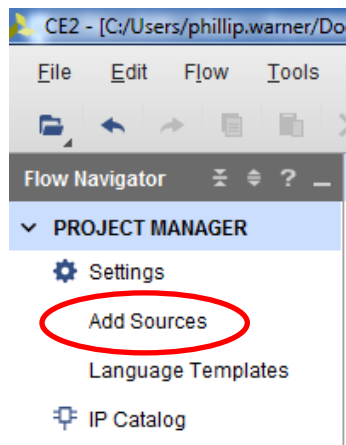
- Prepend all commit statements for ICE2 with “ICE2 – “. For instance, your first commit might be:
 - “ICE2 – created initial project files and README”

If you did not copy the provided .gitignore file to your git repo folder (not the ICE2 folder), please do so now. This will force Git to ignore all files and directories other than the ones we specify. Alternatively, when you add files, only add the files you wish to commit. This is important as Vivado will create a LOT of files that we do not want in your repo. Alternatively, only commit files you specifically want. If you ever find that git won't let you add a file, check to see if the .gitignore will allow it to pass.

3. CREATE DESIGN

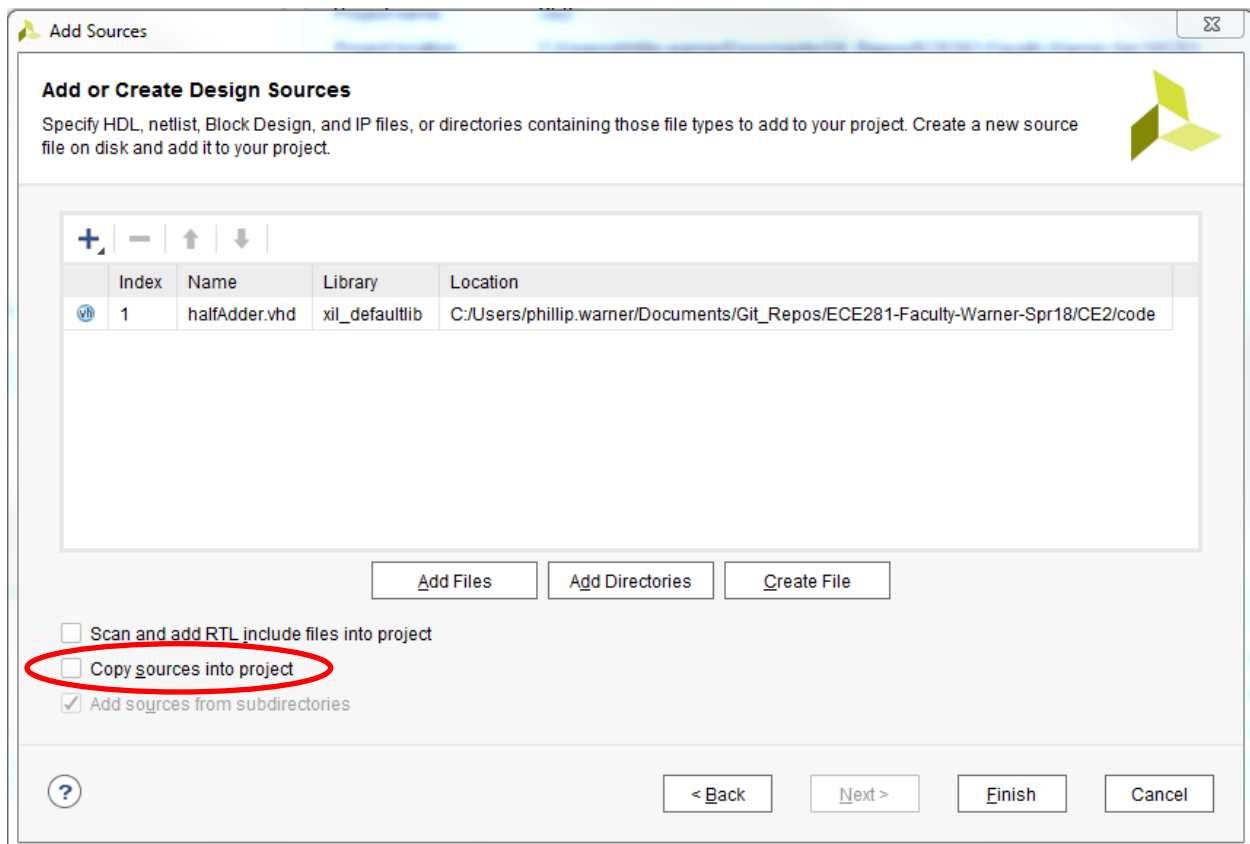
ADD SOURCES (DESIGN, SIMULATION, AND CONSTRAINTS)

On the leftmost pane (*Flow Navigator*), click on **Add Sources**

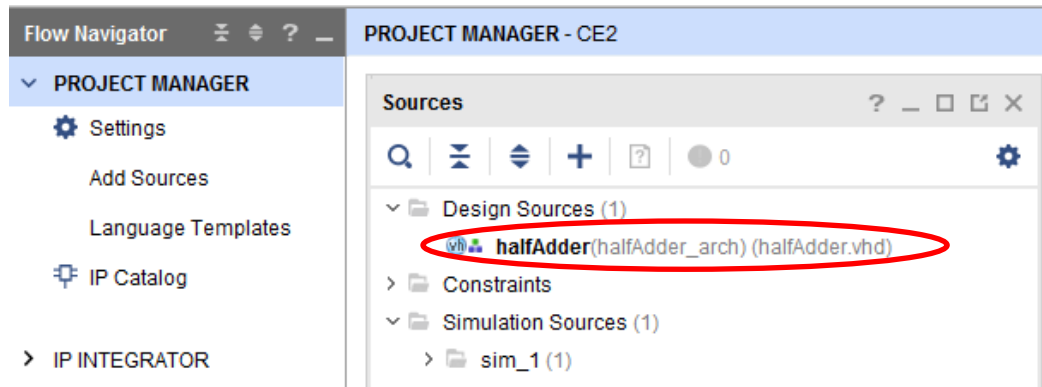


Add Sources window

- Select the “Add or create **design sources**” radio button and click Next.
- Click on **Add Files** button in middle of window
- Navigate to your ICE2/code folder, select halfAdder.vhd, and click OK
- **Uncheck Copy sources into project** (we want to edit the file in the **code** folder) and click **Finish**



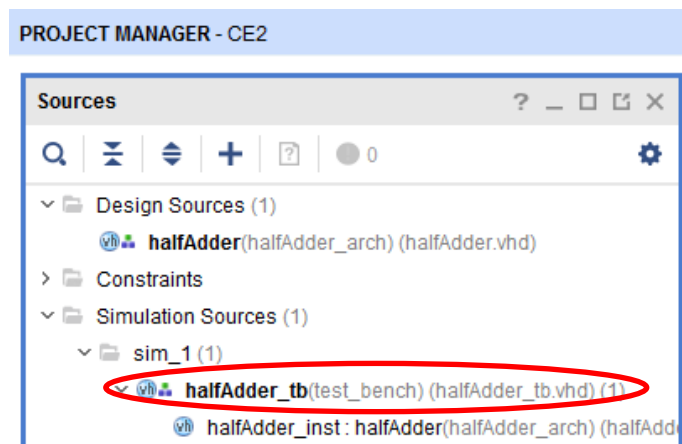
The VHDL module should now be listed in your Sources window. The **bolded** name is based off of the entity name, and the file name is in the right-most parentheses.



Add another source, but this time select “Add or create **simulation** sources” in the initial window

- Add **halfAdder_tb.vhd** (“tb” stands for test bench)
- Again, make sure “Copy sources into project” is **unchecked**

The test bench file should now be added into your Simulation Sources:

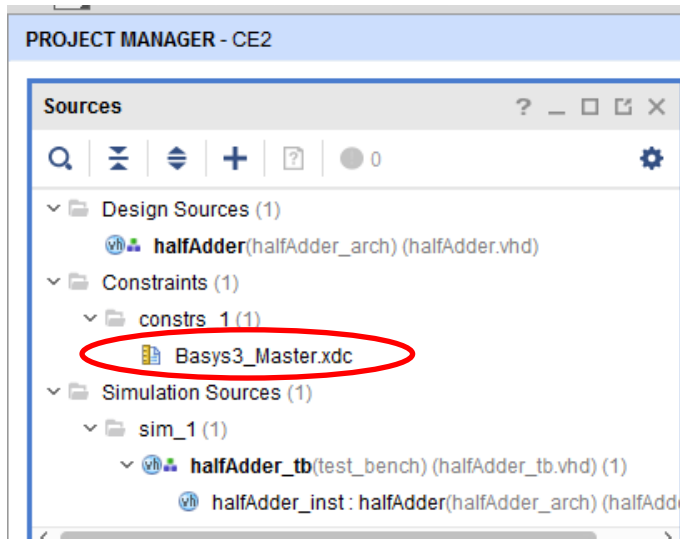


If for some reason, it is not the top Simulation Source as shown above, right-click on it and select “Set as Top”.

Add the final source, but this time select “Add or create **constraints**” in the initial window

- Add **Basys3_Master.xdc**
- Make sure “Copy constraints files into project” is **unchecked**

The constraints file should now be added into your Constraints:



EDIT YOUR HALF-ADDER IMPLEMENTATION

Open up your halfAdder.vhd file in a text editor. The Vivado editor is fine, but the following screenshots were taken from Notepad++

FILE HEADER

The first thing you will notice is that the file contains a header. Comments in VHDL are usually colored green or gray and are preceded by a '--' for single-line comments.

Filename, author(s), etc need to be filled out correctly. Edit as applicable.

```
11  --| -----
12  --|
13  --| FILENAME      : halfAdder.vhd
14  --| AUTHOR(S)    : Capt Warner
15  --| CREATED      : 01/17/2017
16  --| DESCRIPTION   : This file implements a one bit half adder.
17  --|
18  --| DOCUMENTATION : None
19  --|
20  --| -----
```

The next section of the header shows the REQUIRED FILES. This file has no specific "Files" dependencies, so NONE is appropriate there.

```
20  --| -----
21  --|
22  --| REQUIRED FILES :
23  --|
24  --| Libraries : ieee
25  --| Packages  : std_logic_1164, numeric_std, unisim
26  --| Files     : LIST ANY DEPENDENCIES
27  --|
28  --| -----
```

The final section of the header shows naming conventions to use throughout this course. The purpose of the naming conventions is primarily to make debugging easier and to make it obvious what your signals are used for. For simple

designs this is not as important, but it becomes critical when you develop more complex designs. Thus, we develop good habits now.

```
28  --+-----+
29  --|
30  --| NAMING CONVENTIONS :
31  --|
32  --|   xb_<port name>           = off-chip bidirectional port ( _pads file )
33  --|   xi_<port name>           = off-chip input port           ( _pads file )
34  --|   xo_<port name>           = off-chip output port          ( _pads file )
35  --|   b_<port name>            = on-chip bidirectional port
36  --|   i_<port name>            = on-chip input port
37  --|   o_<port name>            = on-chip output port
38  --|   c_<signal name>          = combinatorial signal
39  --|   f_<signal name>          = synchronous signal
40  --|   ff_<signal name>         = pipeline stage (ff_, fff_, etc.)
41  --|   <signal name>_n          = active low signal
42  --|   w_<signal name>          = top level wiring signal
43  --|   g_<generic name>         = generic
44  --|   k_<constant name>        = constant
45  --|   v_<variable name>        = variable
46  --|   sm_<state machine type>  = state machine type definition
47  --|   s_<signal name>         = state name
48  --|
49  --+-----+
```

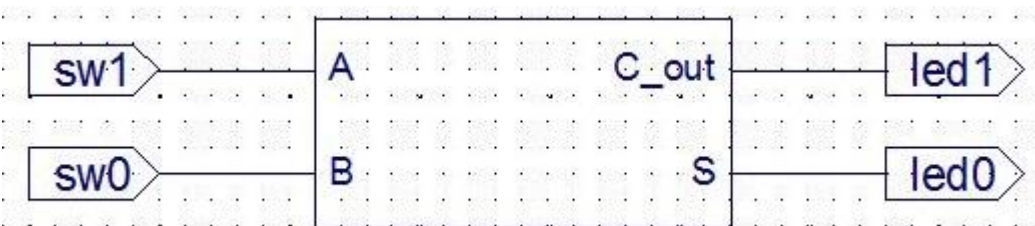
INCLUDED LIBRARIES

The next section shows a few libraries that we are adding for use. These statements are very similar to importing modules or includes in other languages. Note, the unisim library includes can be deleted as we are not using them currently.

```
50  library ieee;
51      use ieee.std_logic_1164.all;
52      use ieee.numeric_std.all;
53
54  library unisim;
55      use UNISIM.vcomponents.ALL;
```

MODIFY THE HALF-ADDER ENTITY

An entity in VHDL is simply a declaration of a hardware module's input and outputs. In other words, it describes the interface of a black box. For instance, the block diagram in the figure below shows that the half-adder has two inputs (A and B) and two outputs (C_out and S).



This interface can then be described in VHDL as an entity. The figure below shows a partially completed entity for the halfAdder.

```
57 -- entity name should match filename
58 entity halfAdder is
59   port(
60     i_A    : in  std_logic; -- 1-bit input port
61     i_B    : in  std_logic;
62     o_S    : out std_logic -- 1-bit output port
63     -- (NOTE: NO semicolon on LAST port only!)
64     -- TODO: Carry port
65   ); -- the semicolon is here instead
66 end halfAdder;
```

Note, the port names are preceded by an 'i_' for input and an 'o_' for output per the naming convention previously shown.

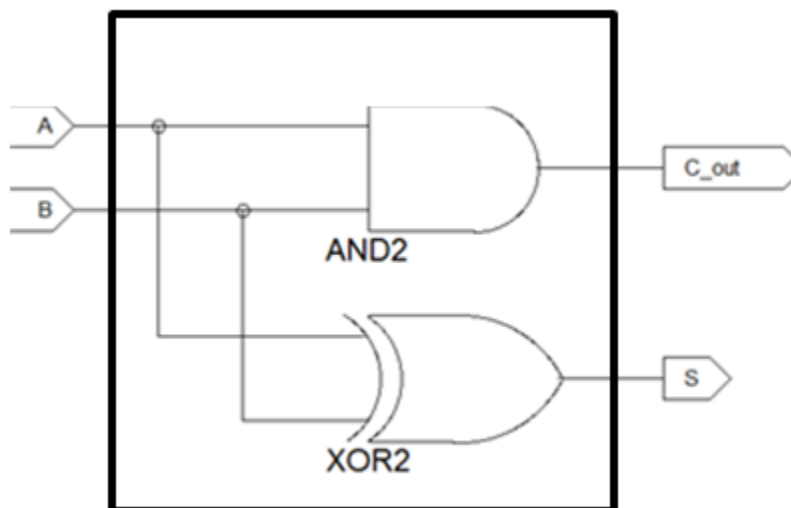
The inputs and output ports are signals described by: portName : mode : signalType.

Std_logic and std_logic_vector are the two most common types you will use. They produce logical signals of '1' or '0', where the vector is more like a bus (multiple digital signals in parallel).

Add an output for the Carry named o_Cout.

MODIFY THE HALF-ADDER ARCHITECTURE

The architecture is the guts of an entity. It is the internal logic that makes it *do* something. For instance, the architecture of the half-adder may look like the AND or XOR gates connected inside the half-adder entity below.



This is partially realized in the given VHDL code below:

```
68 architecture halfAdder_arch of halfAdder is
69     -- this is where you would include components declarations and signals if you needed them
70
71 begin
72     -- this is where you would map ports for any component instantiations if you needed to
73
74     -- *concurrent* signal assignments
75     o_S    <= i_A xor i_B;
76     -- TODO: Carry signal assignment
77
78 end halfAdder_arch;
```

Note, the architecture is “of halfAdder” since that is the name of the component the architecture is for. Since this is a simply design, the architecture’s *behavior* can be described with only two lines of code. The first assignment statement is provided. **Add the second statement required to implement o_Cout.** Participation activity 7.2.5 in your Zybook may be helpful if you are stuck.

3. TEST DESIGN

At this point your testbench VHDL file (halfAdder_tb.vhd) should already be added. If not, go back and add it per the directions in Step 3 – Add Sources.

- **TIP:** Generally, your testbench files will have the same file name as the component you are testing with them, but with a “_tb” at the end.

EDIT YOUR HALF-ADDER TESTBENCH

Open your testbench file in a text editor. Note the header has halfAdder.vhd as a REQUIRED FILE.

The entity in this file is the testbench itself, which is empty as shown below.

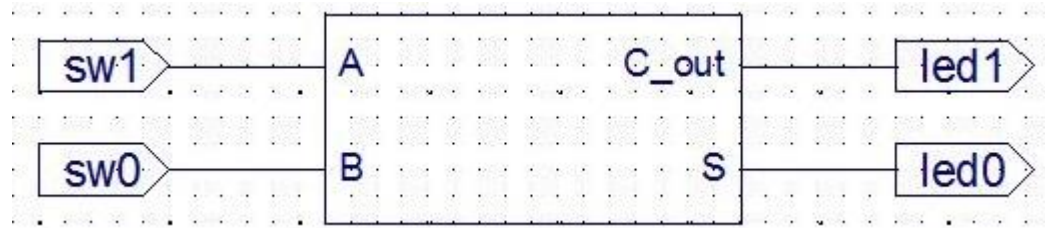
```
57 entity halfAdder_tb is
58 end halfAdder_tb;
```

It is empty because it does not need any external interfaces. Instead, we want to observe the signals generated by the hardware components *inside* the testbench. These hardware unit under test (UUT) inputs will then be stimulated by the testbench internal architecture, which realizes our test signals. To do this, first we have to declare what hardware components will be inside of our test bench:

```
60 architecture test_bench of halfAdder_tb is
61
62     -- declare the component of your top-level design unit under test (UUT)
63     component halfAdder is
64     port(
65         i_A    : in  std_logic; -- 1-bit input port
66         i_B    : in  std_logic;
67         o_S    : out std_logic -- 1-bit output port
68         -- (NOTE: NO semicolon on LAST port only!)
69         -- TODO: Carry port
70     ); -- the semicolon is here instead
71 end component;
```

This syntax is almost *identical* to what you saw in the implementation file. The only difference is now the halfAdder is called a *component* instead of an *entity*. Thus, you can **copy in the port information from your implementation file to the corresponding component in your testbench.**

Now that your halfAdder hardware is inside your testbench entity, we are going to declare some additional signals that correspond to the switches and leds per our previous block diagram:



You can think of these signals as *wires* that we place inside our testbench box. This is partially realized in the given testbench snippet shown below. **Note, since these are top level wires, we would normally prepend the names with “w_” instead of “i_” or “o_”,** but we will stick with the current names for this exercise. Note, the code also sets an initial value of ‘0’. **Declare the two missing signals.**

```

74      -- declare signals needed to stimulate the UUT inputs
75      signal i_sw1 : std_logic := '0';
76      -- TODO: sw0 signal
77
78      -- also need signals for the outputs of the UUT
79      signal o_led1 : std_logic := '0';
80      -- TODO: led0 signal

```

So, now we have a testbench box with a half-adder component and four wires in it. *Nothing is connected yet.* To actually “wire up the hardware”, we need to map the ports. This is partially realized in the code snippet below.

```

83      begin
84          -- PORT MAPS -----
85
86          -- map ports for any component instances (port mapping is like wiring hardware)
87          halfAdder_inst : halfAdder port map (
88              i_A      => i_sw1, -- notice comma (not a semicolon)
89              i_B      => i_sw0,
90              o_S      => o_led0 -- no comma on LAST one
91              -- TODO: map Cout
92          );

```

Wire up the o_Cout signal to o_led1.

Now all that is left is to assign values to the test input wires (i_sw0 and i_sw1) inside the **test plan process**:

```

99  -- Test Plan Process -----
100 -- Implement the test plan here. Body of process is continuously from time = 0
101 test_process : process
102 begin
103
104     i_sw1 <= '0'; i_sw0 <= '0'; wait for 10 ns;
105     -- TODO: rest of test plan
106
107     wait; -- wait forever
108 end process;
109 -----

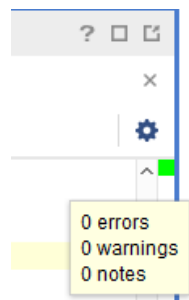
```

How many test cases will you need?

CHECK SYNTAX AND SIMULATE YOUR PROJECT

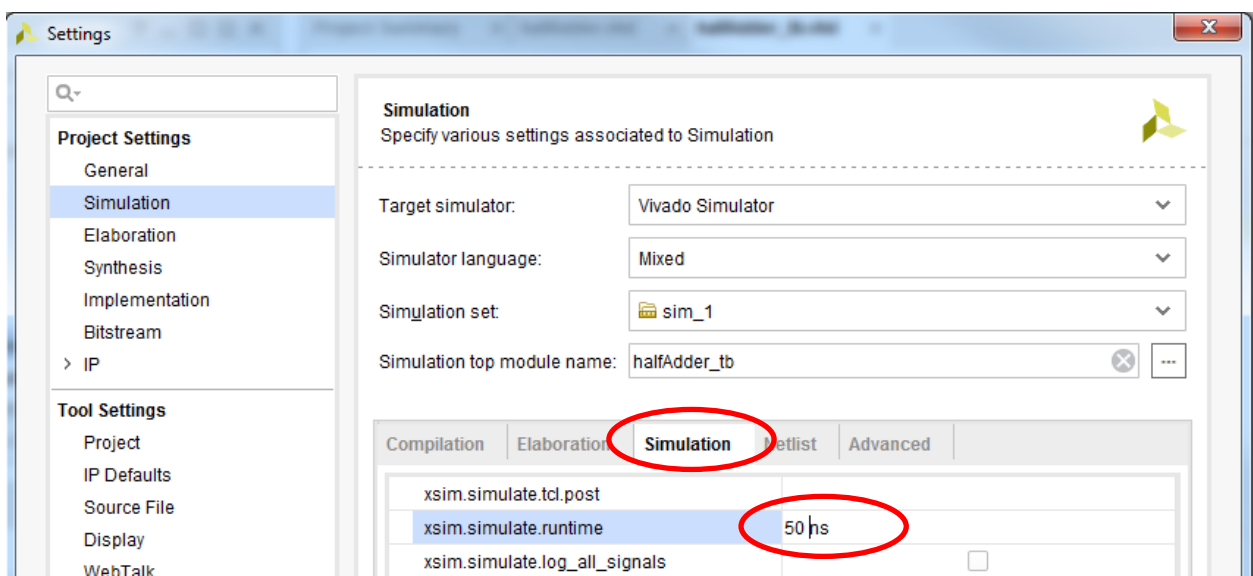
CHECK SYNTAX


While you edit your code in Vivado, it checks the syntax. If everything is good you should see a green box in the upper right-hand corner of your editor window. If you hover your mouse over it you will see that there are 0 errors. Otherwise, red marks along the right-hand margin indicate locations of syntax errors.

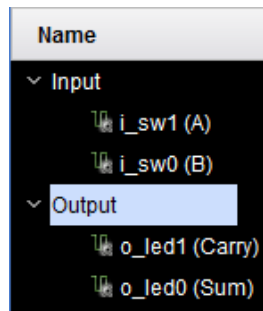


SIMULATE PROJECT

1. In the Flow Navigator, right click on “Run Simulation” and select “Simulation Settings”
2. Click on the **Simulation** tab and change the runtime to **50 ns** or an otherwise appropriate length based on your test plan process. Click OK.



3. Left click on **Run Simulation** and then click on **Run Behavioral Simulation**
 - a. You can ignore/cancel any firewall access popups
4. If for some reason a waveform window did not pop up, check the Messages and Log tabs near the bottom of your window. Look for error messages.
5. Once the waveform window is in view, click on the Zoom Fit icon  or adjust the zoom window as needed to see the time periods of interest
 - a. Verify that the waveforms show your design is working correctly (inputs and outputs should match truth table)
6. Try changing the outputs to different colors by right clicking on the signals and selecting **Signal Color**.
7. Rename signals to show what they correspond to (e.g., A) and define virtual busses for your inputs and outputs as shown in the figure below. To define a virtual bus, simply highlight multiple signals and right click. Then, select **“Virtual Bus”** and name it as desired.



Note, the wave editor also allows you to view data in different formats. Right click on a number in the “Value” column and select **Radix**.

REMEMBERING THE WAVE CONFIGURATION

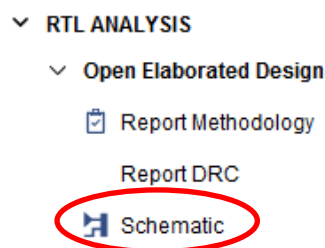
Since you put in all of that work to have a nice wave form, you can optionally save your wave configuration file (File→Save Wave Configuration or simply click on the Save icon in the editor). Saving it to the main project folder is fine. Click OK to add it to your project. Now future runs of your simulation will use the same settings!

VIEW THE RTL AND TECHNOLOGY SCHEMATICS

RTL SCHEMATIC

A Register Transfer Level (RTL) schematic is a gate-level schematic that shows you how your VHDL code is initially being interpreted. It is important to verify that the circuit matches what you expect. Note, this schematic is not optimized for your hardware chip.

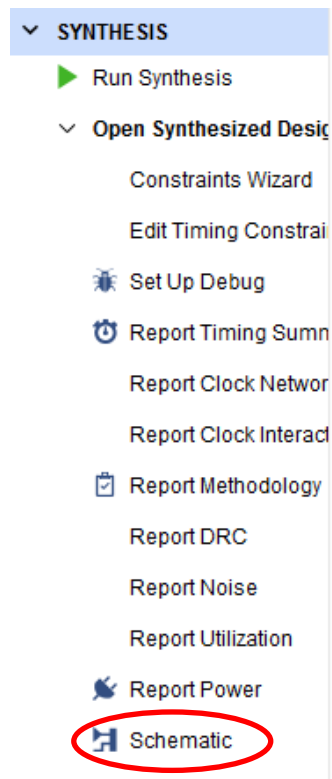
In the Flow Navigator, click on the RTL Analysis **Schematic**. Click OK to continue.



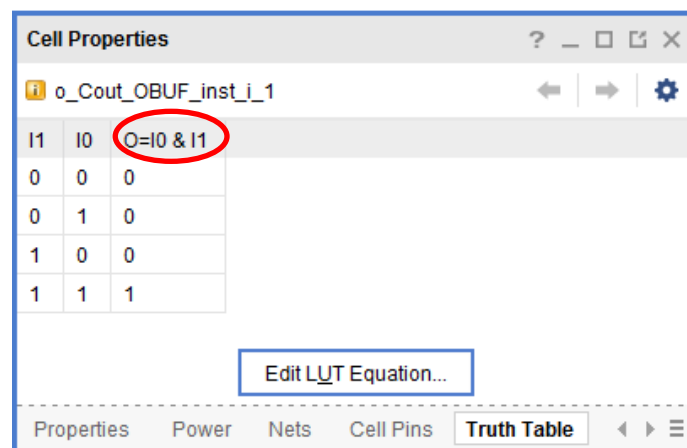
TECHNOLOGY SCHEMATIC

Now that you have verified that your design is functionally correct, it is time to verify that the synthesized hardware is correct by viewing the technology schematic. This schematic, unlike the RTL one, is optimized for your specific hardware platform (i.e., it shows your design in terms of FPGA logic elements).

Click on “**Run Synthesis**” in the Flow Navigator. After the process completes, click on **Schematic** to view the optimized schematic for your FPGA chip as shown in the picture provided on the next page.



Notice that each port is connected to a buffer and that some of your logic (i.e., the XOR) is realized using a lookup table (LUT). If you click on any of the two LUTs you should see, you also can view their associated truth table by clicking on the “Truth Table” tab at the bottom of the “Cell Properties” sub-window. Notice an equation is also given for the output as circled below.



Do the truth tables for the LUTs make sense for your design? Now would be a good time for another Git commit.

4. IMPLEMENT DESIGN

EDIT THE CONSTRAINTS FILE

1. Click on the Project Manager in the Flow Navigator
2. Double click on the Basys3_Master.xdc file in the Sources sub-window to open it.
3. Get your BASYS 3 board out and look at the text surrounding the switches and LEDs. Note, the labels underneath the switches are the physical pin locations on your BASYS 3 board. For example, SW0 is connected to pin V17. Find (use CTRL+f) **V17** in the constraints file. You should see it first on line 12 as shown below.

```
11 | ## Switches
12 | #set_property PACKAGE_PIN V17 [get_ports {sw[0]}]
13 | #set_property IOSTANDARD LVCMOS33 [get_ports {sw[0]}]
```

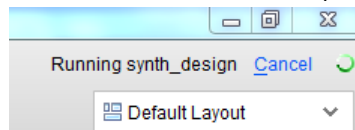
4. Replace the highlighted portions (**sw[0]**) shown above with the name of the node (signal) that should connect to the switch. In this case, we want to connect **i_B** from our halfAdder entity to this switch. In the future we will utilize the sw[0] naming convention so changing the xdc file may not be required.
 - a. Note, the node names correspond to the signal names you defined for your half-adder interface in halfAdder.vhd and NOT the signal names in your testbench.
5. Uncomment the two lines by removing the '#' signs. The final result should look similar to below:

```
11 | ## Switches
12 | set_property PACKAGE_PIN V17 [get_ports {i_B}]
13 | set_property IOSTANDARD LVCMOS33 [get_ports {i_B}]
```

6. Repeat steps 3-5 to connect **sw[1]** to **i_A**
7. Repeat steps 3-5 to connect **led[0]** to **o_S**
8. Repeat steps 3-5 to connect **led[1]** to **o_Cout**
9. Save the constraints file (CTRL+s)

IMPLEMENT AND GENERATE BITSTREAM

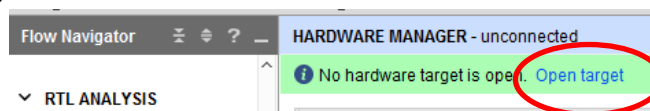
10. In the Flow Navigator, click on **Generate Bitstream**
 - a. Click **YES** to run Synthesis and Implementation as required and then **OK** to start the job
 - b. The upper right-hand of the Vivado window will show a quick view of the status



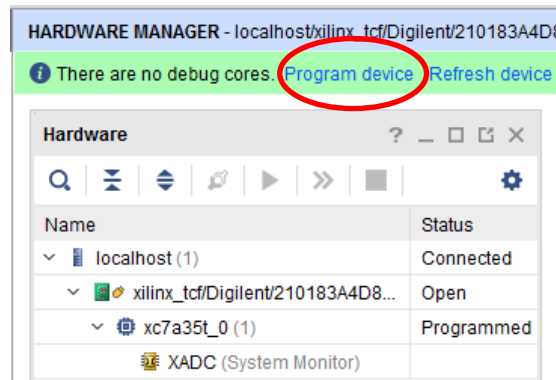
11. You should get a window reporting successful Bitstream Generation.

5. DOWNLOAD ONTO FPGA AND TEST FUNCTIONALITY

1. Plug in your BASYS 3 board and turn it on.
2. Open the Hardware Manager (bottom of Flow Navigator)
3. Click on the **Open target** link and then click on **Auto Connect**



- Click on **Program device**



- By default the Bitstream (.bit) file that you last created will be selected for programming. Click **Program**
- After a successful program, test your design by flipping switches and verifying that the LED outputs are correct.
- If your hardware works correctly, demo it to an instructor. Alternatively, you may document your functionality by taking a short video or a series of pictures. Include the documentation in your README (a link is fine for a video).
 - Note, your FPGA will not maintain its programming after it loses power
 - Instead of opening your project for programming in the future, you can directly open the Hardware Manager from the main Vivado menu. Simply select the correct .bit file.
- Commit your README changes along with your .bit file.

7. WRAPPING UP...

- Double check that you have everything **committed** to your git repo.
- Then **push** them to your Bitbucket repo.
- Make sure all your work appears in Bitbucket as you expect.
- Congratulations, you have completed In Class Exercise 1!**