Lab1

Introduction to the Xilinx Design Flow

Hardware Hacking

Goals:

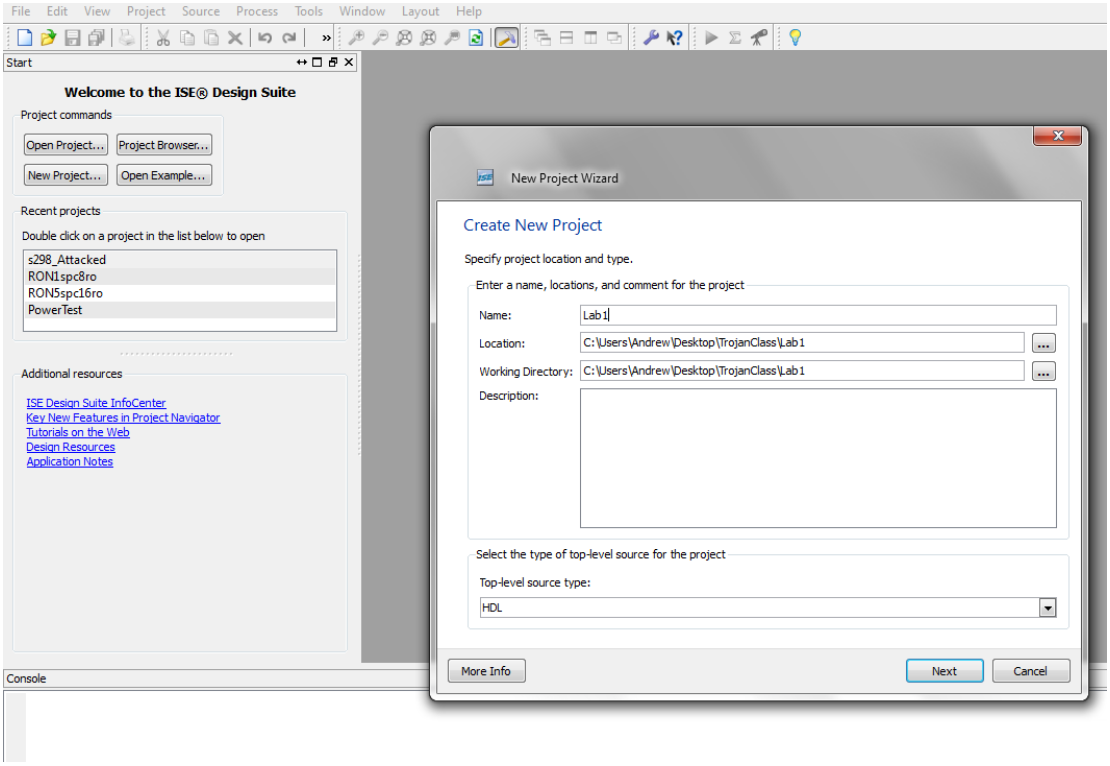
In this lab we will walk you through the basic procedures to work with VHDL files in the Xilinx ISE environment to produce simulations and FPGA programming files using a ripple adder example.

Items to be submitted:

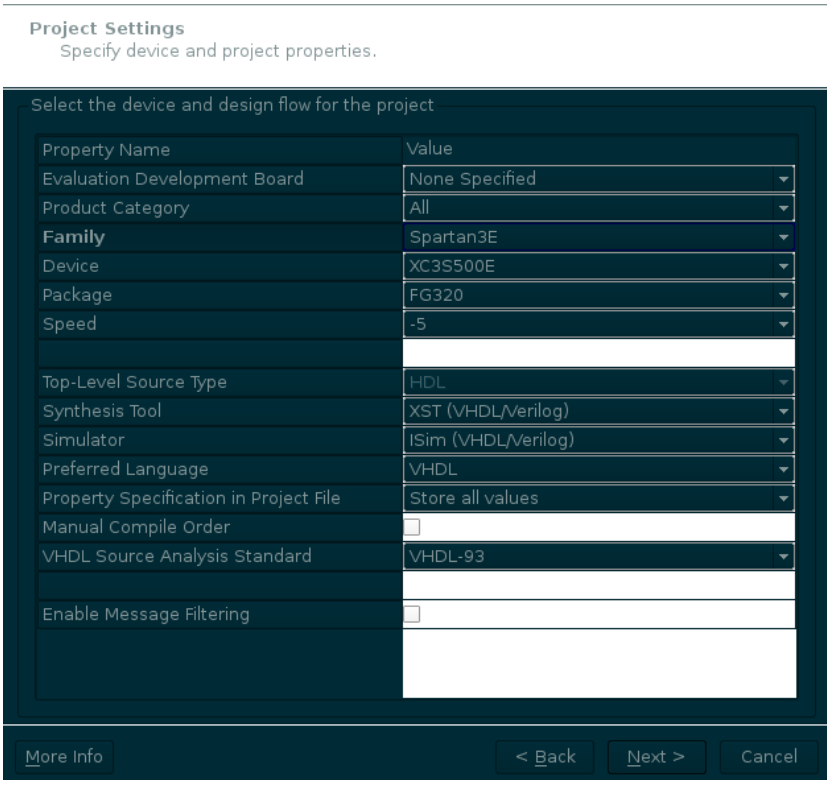
- Full Adder test bench
- Full Adder simulation waveform
- Completed Ripple adder
- Ripple adder test bench
- Ripple adder simulation waveform
- Demonstrate programmed FPGA to the TA

Part 1: Simulating VHDL with ISIM in ISE.

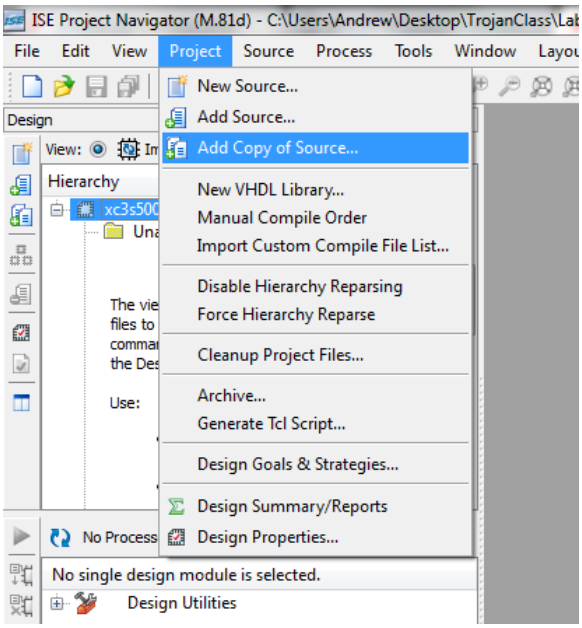
When you first open the ISE project navigator you will be greeted with a start screen that has options to open a project, start a new project, etc. Click “New Project...” then navigate to an appropriate directory to store the project and call it Lab 1 as shown below.



Proceed by clicking next, and then fill in the following settings. Pay particular attention to the FPGA to the family, device, and package settings.

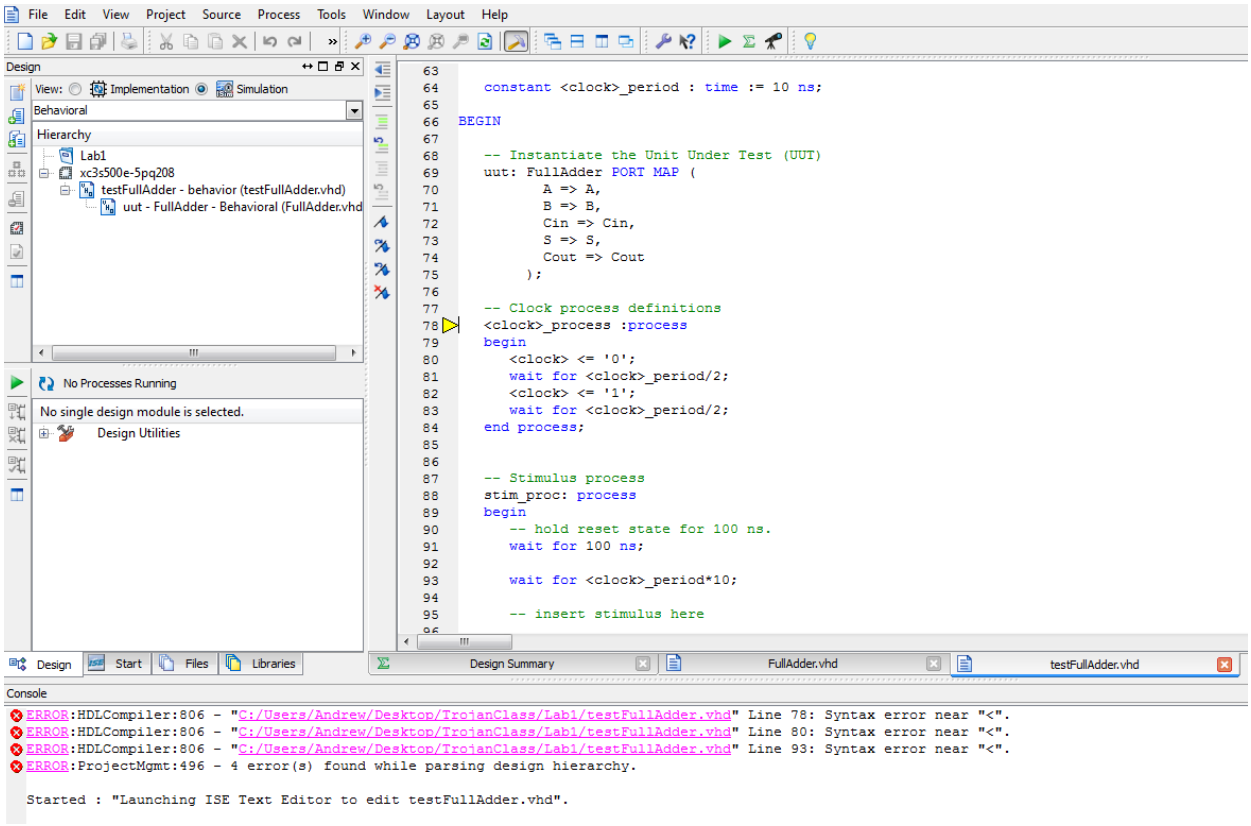


Proceed until the project has been created. Then, add a copy of the included source for a full adder from Project > Add Copy of Source... as shown below.



If successful, you will see the code for the full adder module in the main pane of the environment. Before we use this module to build a ripple adder, it is advisable that we verify this lower level module using a simulation. To simulate with ISIM, we must first create a test bench. Add a new test bench by navigating to Project > New Source. Select “VHDL Test Bench,” call the file “testFullAdder,” and use the default file path as shown below:

If you've done this correctly, the result will be a .vhd test bench file with a few syntax errors. These errors happen, because ISE attempts to find a clock signal (by looking for ports called "clk," "clock," etc.) to automatically setup a simulated clock process. When it fails to find one, it creates syntactically incorrect template code.



This module, however, is not sequential, and thus, we remove all lines in the process, "<clock>_process", along with all lines that otherwise contain "<clock>."

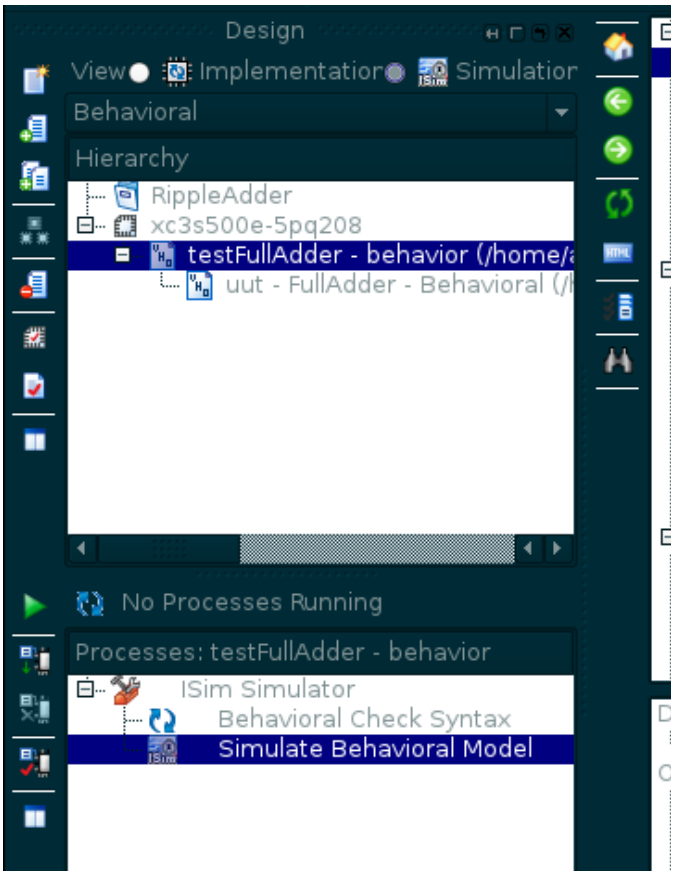
Upon successful completion, saving the project again will remove the syntax errors.

```
54    signal A : std_logic := '0';
55    signal B : std_logic := '0';
56    signal Cin : std_logic := '0';
57
58    --Outputs
59    signal S : std_logic;
60    signal Cout : std_logic;
61    -- No clocks detected in port list. Replace <clock> below with
62    -- appropriate port name
63
64
65  BEGIN
66
67     -- Instantiate the Unit Under Test (UUT)
68     uut: FullAdder PORT MAP (
69            A => A,
70            B => B,
71            Cin => Cin,
72            S => S,
73            Cout => Cout
74          );
75
76     -- Stimulus process
77     stim_proc: process
78     begin
79
80        --Fill this in.
81
82        wait;
83     end process;
84
85  END;
86
```

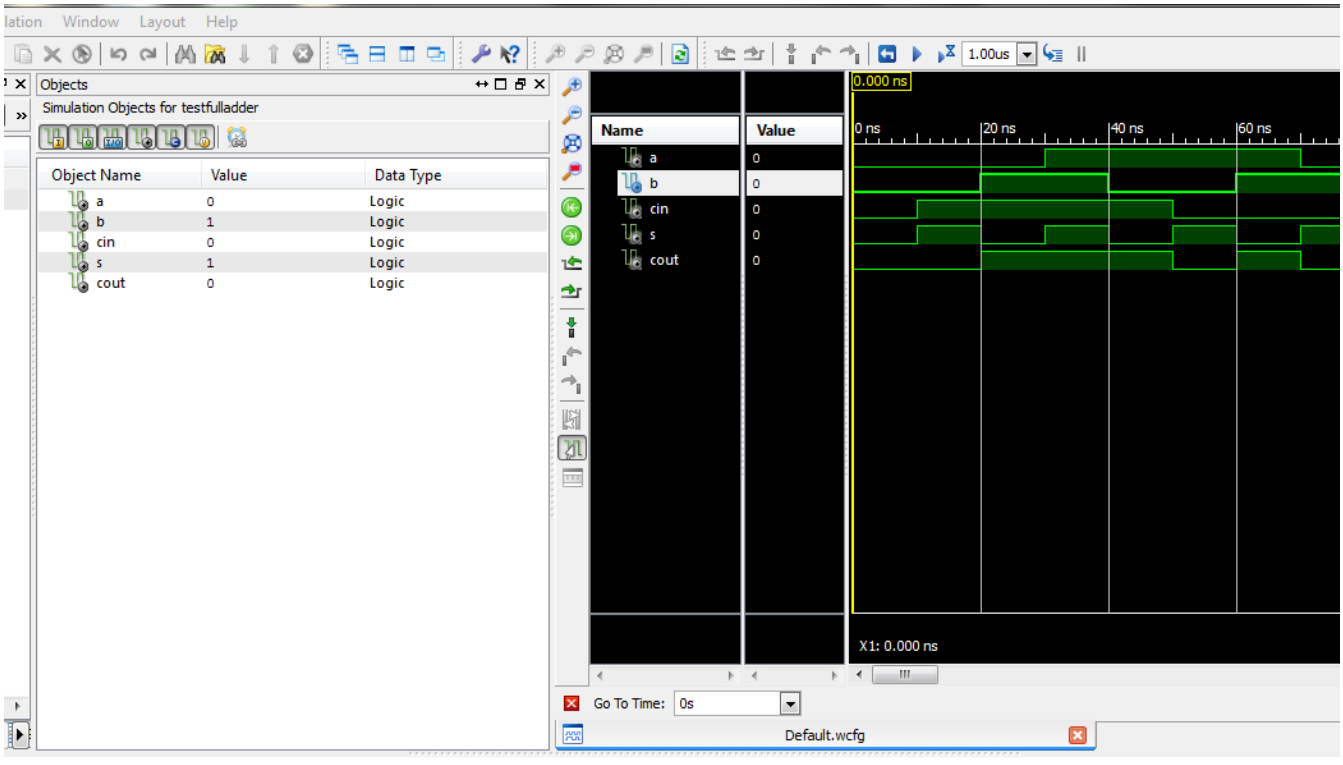
In order to complete the test bench, stimuli must be added to the process, “stim_proc.” These stimuli are often programmed by supplying an input, holding it for a time period, and then supplying new inputs. The piece of code below should serve as a reminder of the syntax and a hint for the remaining test cases. Please complete the test bench by providing an exhaustive set of test cases (there are 8).

```
-- Stimulus process
stim_proc: process
begin

   A <= '0';
   B <= '0';
   Cin <= '0';
   wait for 10 ns;

   Cin <= '1';
   wait for 10 ns;

   -- more steps

   wait;
end process;
```

When you have a complete test bench, simulate it by switching to simulation view, highlighting the “testFullAdder” module, and double clicking “Simulate Behavioral Model.”

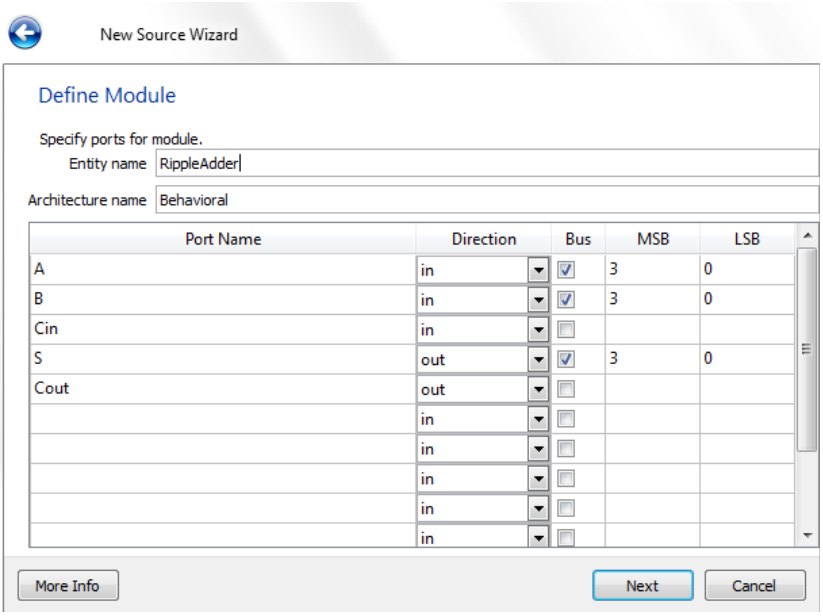


ISIM will launch to simulate the full adder according to your test bench. To view the waveform at time zero, press “ctrl+G” and type 0ns. Then, zoom in or out until you can interpret the waveform. Use the waveform to verify the operation of the full adder before proceeding.

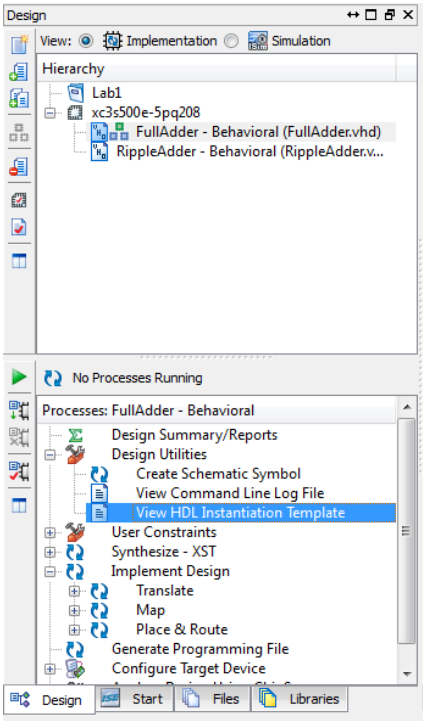


Part 2: Working with components

You are now ready to implement the ripple adder. To do this, navigate to Project > New Source. Complete the specifications for the adder in the window that appears as shown below:



This ripple adder will consist of four full adders. You will need to declare the FullAdder in the architecture preamble, and then make four instances with the appropriate connections. ISE provides a helpful tool to display the declaration and instantiation templates of a module. Access this by highlighting the module of interest, then navigating to Design Utilities > View HDL Instantiation Template under the processes pane as shown below.



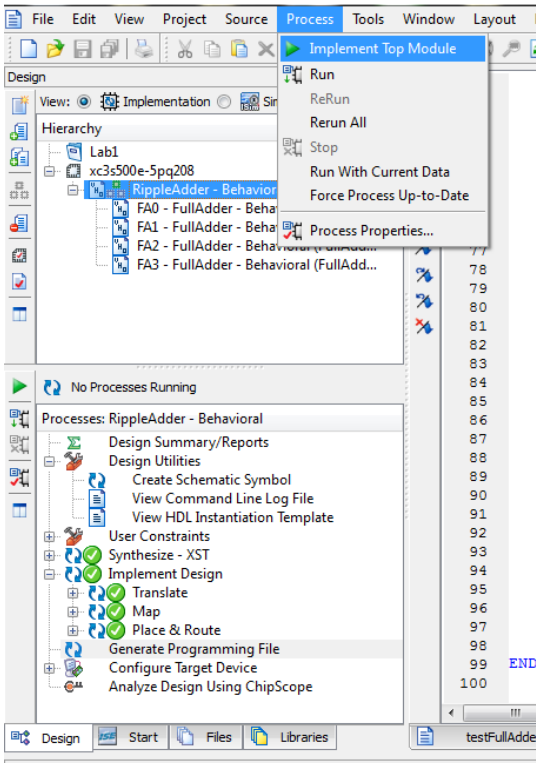
Paste the declaration of the full adder in the architecture preamble of the ripple adder. Then, declare a 3-bit std_logic_vector, C, below the full adder declaration. Paste four instances of the ripple adder, each with a different name, and complete the connections. The required declarations along with the first instance is shown below.

```
32  entity RippleAdder is
33      Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
34             B : in  STD_LOGIC_VECTOR (3 downto 0);
35             Cin : in  STD_LOGIC;
36             S : out  STD_LOGIC_VECTOR (3 downto 0);
37             Cout : out  STD_LOGIC);
38  end RippleAdder;
39
40  architecture Behavioral of RippleAdder is
41
42     COMPONENT FullAdder
43     PORT(
44        A : IN std_logic;
45        B : IN std_logic;
46        Cin : IN std_logic;
47        S : OUT std_logic;
48        Cout : OUT std_logic
49        );
50     END COMPONENT;
51
52     signal C : std_logic_vector(1 to 3);
53
54  begin
55
56     FA0: FullAdder PORT MAP(
57        A => A(0),
58        B => B(0),
59        Cin => Cin,
60        S => S(0),
61        Cout => C(1)
62     );
63
64  end Behavioral;
```

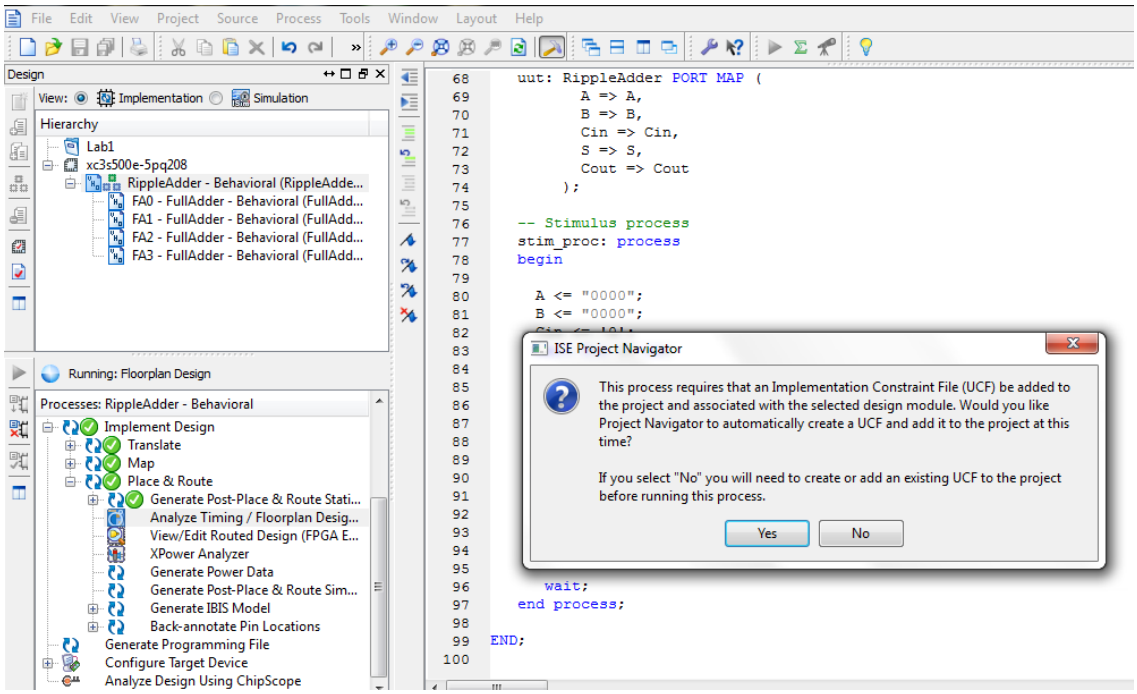
Complete the implementation using all four instances. With this simple four-bit adder there are already 2^13=8192 possible input combinations - far too many to check exhaustively. Design a test bench that uses a few test cases. Submit your completed ripple adder, test bench, and a waveform screenshot.

Part 3: Programming the FPGA

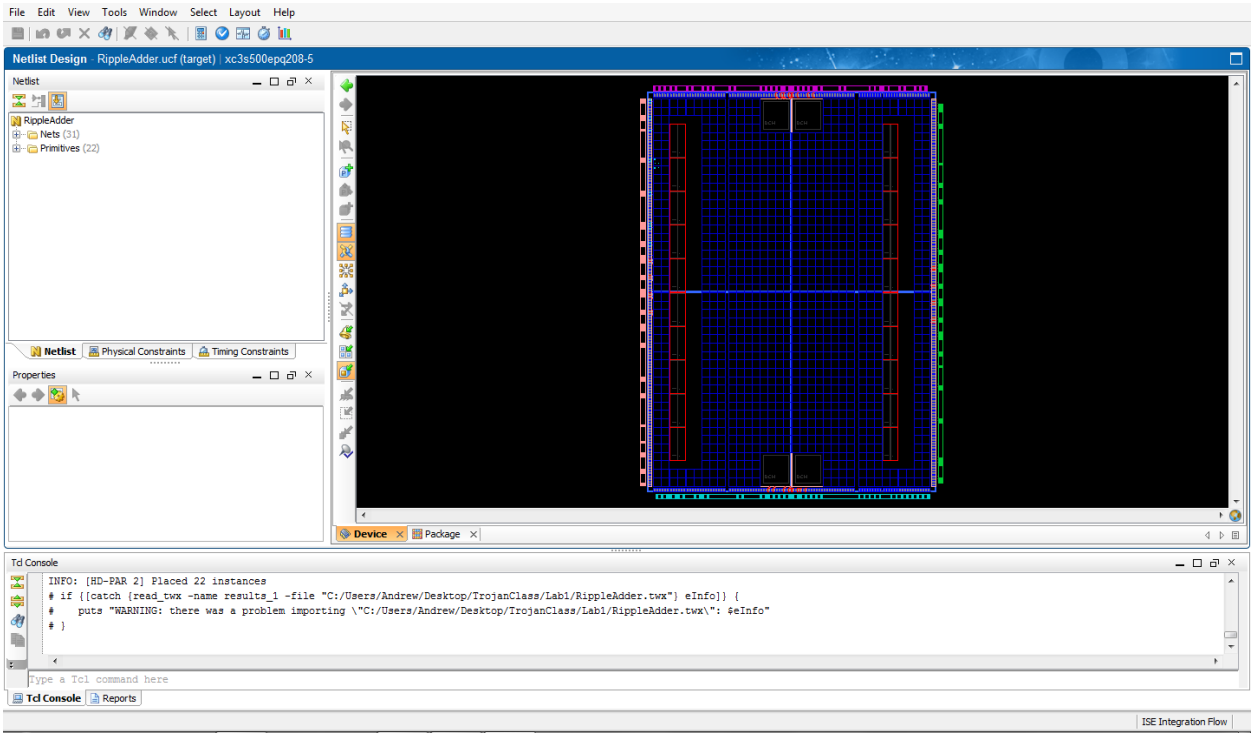
In order to simulate a design, we must first implement the top module by navigating to Process > Implement Top Module. If there are no problems with the design, the processes in the processes pane will complete successfully and display a green check mark as shown.



Next, lets check the floorplan of the design by launching PlanAhead from: Implement Design > Place & Route > Analyze Timing / Floorplan Design. A prompt will be displayed since we haven't supplied a user constraints file. Click yes to generate a blank UCF, then re-run the processes.



The next attempt to launch PlanAhead should prove successful. You will see a representation of the placement of modules in the FPGA. The LUTs used to implement the module will be shown in light blue (there are very few in this design).



If we were to program an FPGA with this design, none of the inputs and outputs would be connected properly. We must specify these connections by editing the UCF file directly. Exit PlanAhead and return to the ISE window. Double-click the UCF file to edit it. The syntax is as shown below, but the pins may differ. Connect the outputs to the LEDs and the inputs to switches and/or buttons. Consult the master UCF file located at http://www.digilentinc.com/Products/Detail.cfm?Prod=NEXYS2 for LOC values. The actual values will differ from those shown here.

```
1  NET "A[0]" LOC = P205;
2  NET "A[1]" LOC = P203;
3  NET "A[2]" LOC = P202;
4  NET "A[3]" LOC = P199;
5
6  NET "B[3]" LOC = P11;
7  NET "B[2]" LOC = P14;
8  NET "B[1]" LOC = P15;
9  NET "B[0]" LOC = P18;
10
11 NET "S[0]" LOC = P139;
12 NET "S[1]" LOC = P137;
13 NET "S[2]" LOC = P135;
14 NET "S[3]" LOC = P133;
```

Once this is complete, saving the file will reset the state of the processes again. If you were to run PlanAhead again, the placement should change to optimally accommodate the output routing, though doing this is not required for the lab. Re-run the processes and then double-click the “Generate Programming File” step in the processes pane to produce a .bit.

Connect the FPGA to the computer via the port labeled “USB PROG.”

Last, open Digilent Adept (if it is not installed, it is available at http://www.digilentinc.com/Products/Detail.cfm?Prod=ADEPT2). If the connections are correct, the FPGA should appear beside “Connect:” Browse for the generated programming file and program the FPGA as shown below. Verify that the design works on the FPGA and demonstrate this to the TA.