Tutorial: Xilinx ISE 14.4 and Digilent Nexys 3

This tutorial will show you how to:
- Part I: Set up a new project in ISE 14.4
- Part II: Implement a function using Schematics
- Part III: Implement a function using Verilog HDL
- Part IV: Simulate the schematic/Verilog circuit using the ISim + Verilog test fixture
- Part V: Constraint, Synthesize, Implement, Generate, and Program for Nexys 3 FPGA board

I assume that you're using a DSL lab machine, or that you've installed Xilinx ISE 14.4 on your own machine as some of the following procedures may be different depending on the version of ISE.

Part I: Set up a new project in ISE 14.4

Attention: Make sure to use the appropriate version of the ISE, 64 bit navigator for a 64 bit OS, and 32 bit for 32 bit. If you don't pay attention to this, there will be unexpected behavior in the ISE software and thing may not work properly!

1. Open the **Xilinx ISE Design Suit 14.4**. You can click on the ISE icon on the desktop, or search **Start → All Programs → Xilinx ISE Design Suite 14.4 → ISE Design Tools → Project Navigator**

    The screen should look something like the following, the ISE always defaults to the last open project unless none where open before just like the following:
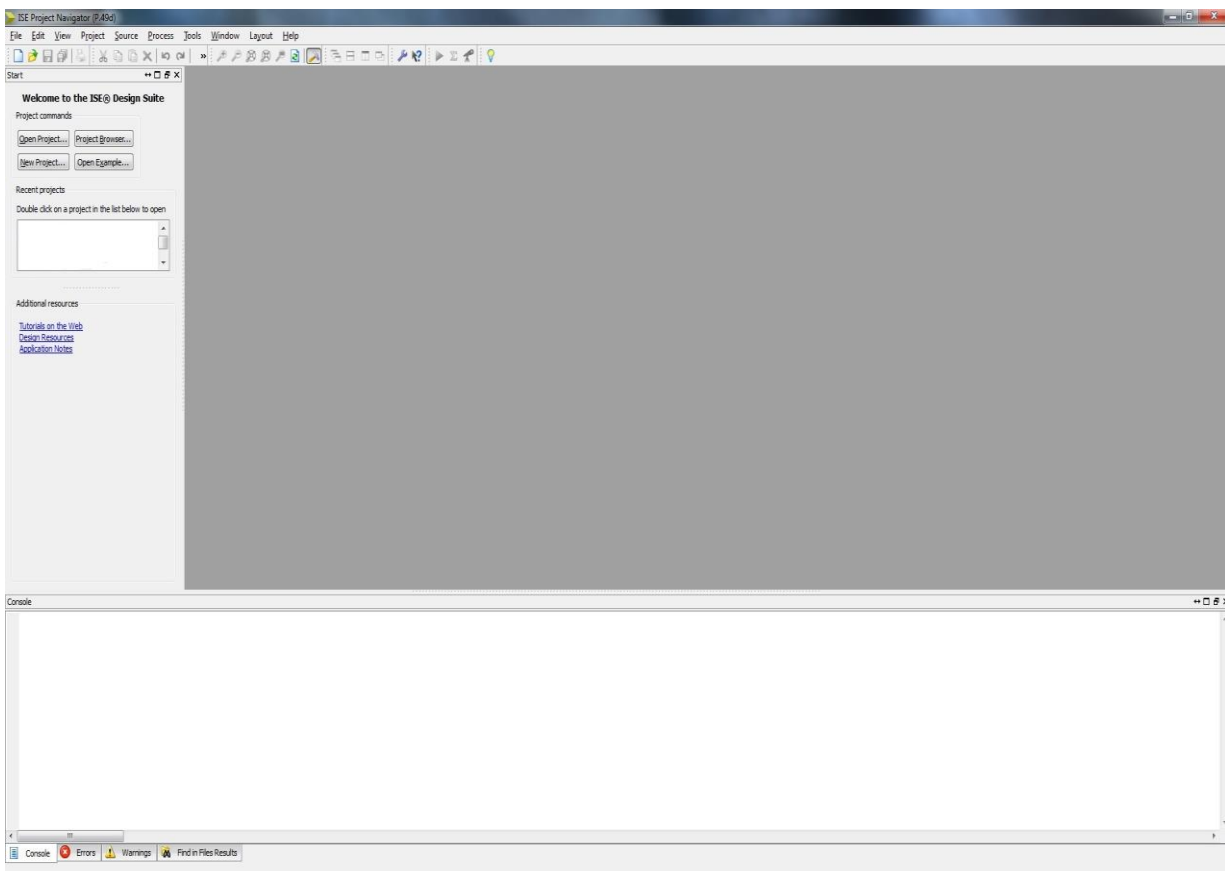
2. Now either press the New Project tab or select **File → New Project…** and change the **Name** and **Location** to whatever you like.

The selected Top Level Source Type is **Schematic** because that's what we're planning on using first. This is not critical, as you can always add a new source file of any type later. The dialog box for the project wizard looks like:
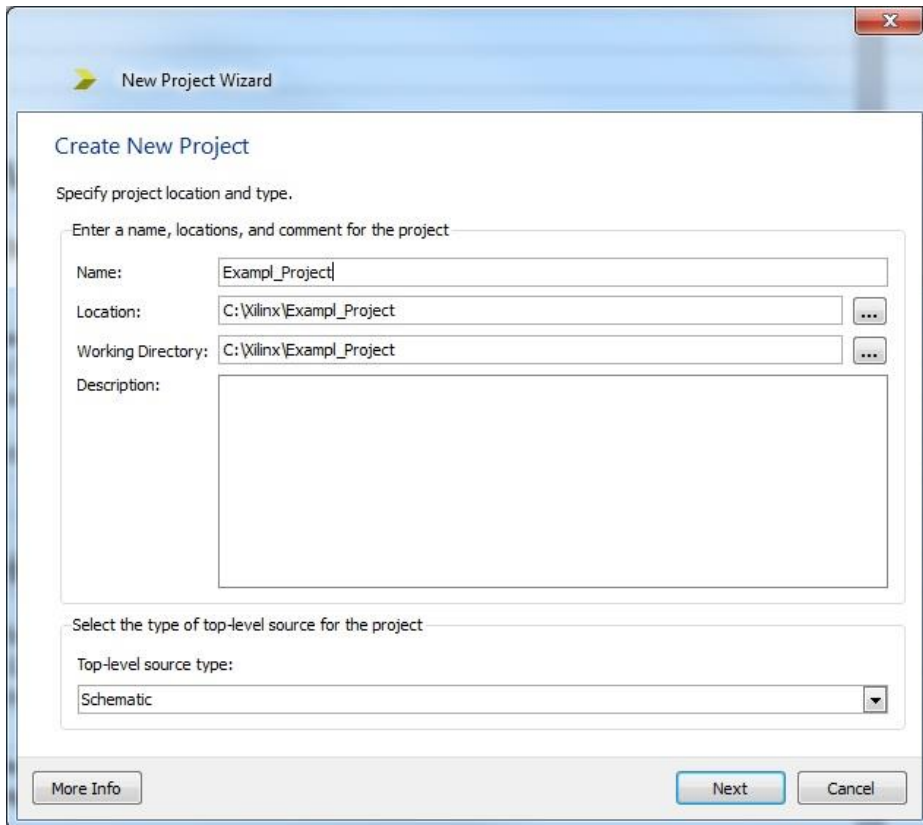

3. Click NEXT and in the next dialog box you should fill in the fields as shown here. You can do this in two ways, one is to select an **"Evaluation Development Board"** from the drop down list, and in our case you should select **"Nexys 3"**. This will automatically fill out the board information in the next five sections. If the board does not exist in the list then you can set correct choices according to the following image.

We are using a **General Purpose** product in the **Xilinx Spartan6** family. The specific chip on the Nexys 3 board is an **XC6SLX16** in a **CSG324** package and the **–3** speed grade.

Please make sure that the Synthesis Tool is **XST**, the Simulator is the **ISim**, and the Preferred Language is **Verilog**. This is very important for proper operation.
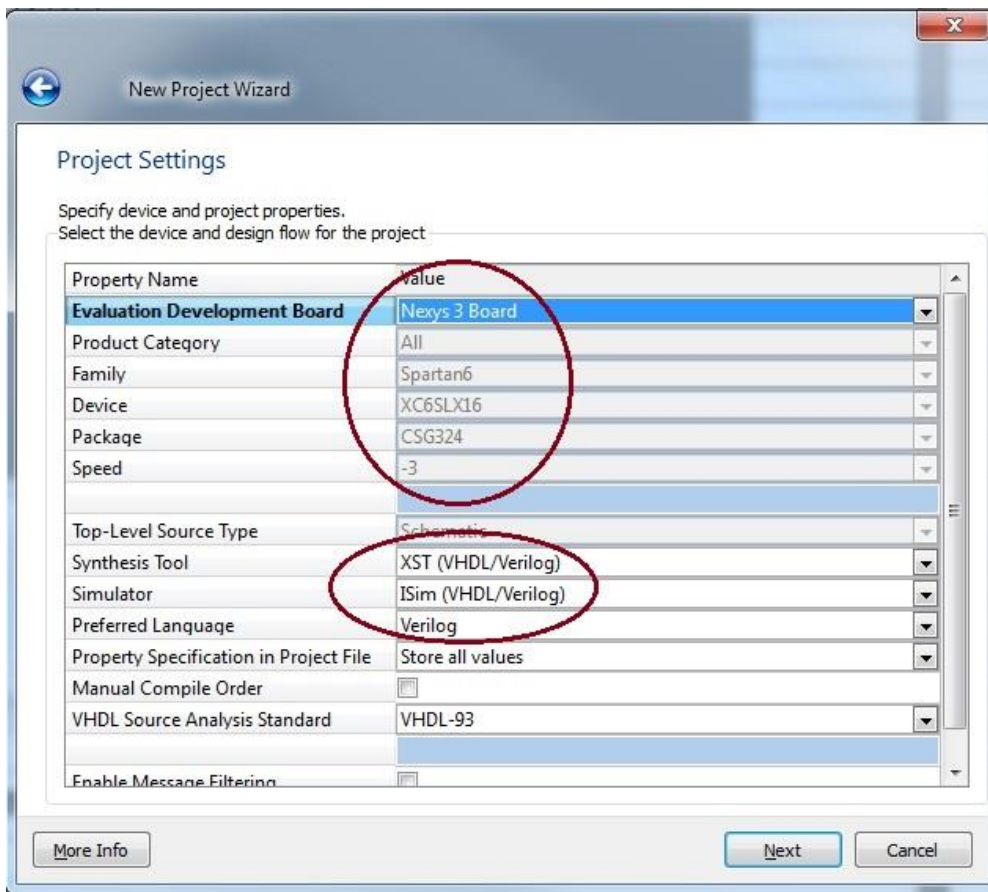
4. Click NEXT and review the project summary page and then click FINISH, it is always good to double-check the summary to prevent headaches due to the problems you can face while implementing your design if the information is incorrect.
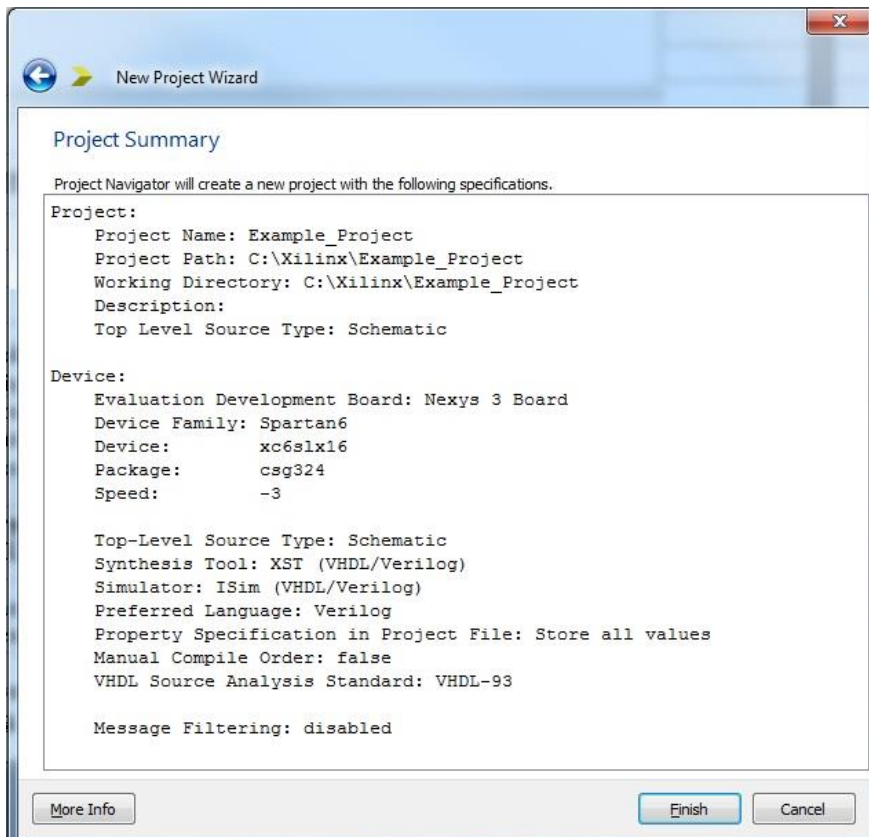
# Part II: Implement a function using Schematics

1. Now you should have a new project that targets the correct Xilinx part and other features of the ISE system. Notice the window to the left, and also notice the four tabs Start, Design, Files, …, and in the case of an open schematic a Symbols tab will appear. Ensure that the "**Implementation"** choice is selected on the design pane.
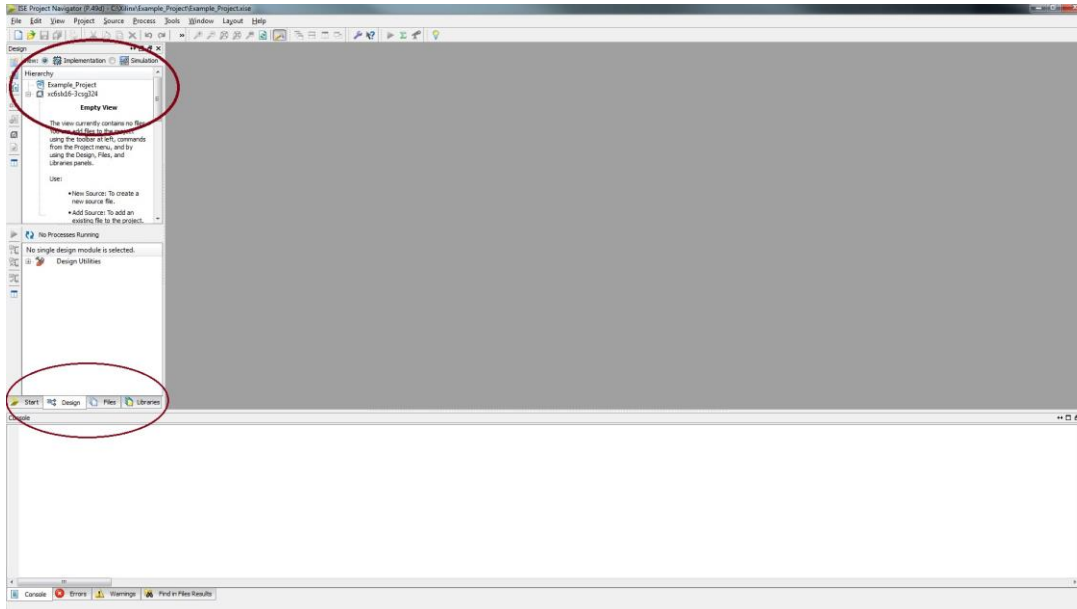
2. Now you can create a new schematic in your project. Choose **Project → New Source** or right click on the Hierarchy section of the design windows to get the dialog box that adds a new source file to your project (or use the **New Source** widget on the left vertical tool bar). Source files can be of many types. Add a Schematic and name it **simple_logic** for example. Make sure to both fill in the **File Name** and select the **Schematic** type from the list on the left. I've filled in my dialog box as shown below. Now click NEXT and after observing the summary page click FINISH.
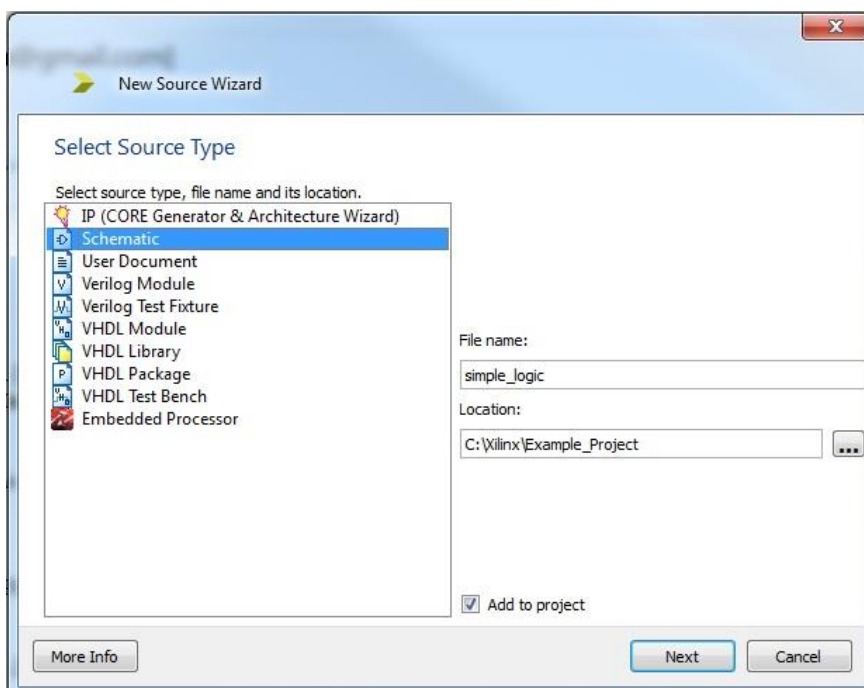
3. Now you have a blank schematic view, and also a Design Summary view in the main pane of the ISE window. You can switch between windows in the main pane with the tabs along the bottom. The (empty) schematic looks like below. Also note the Hierarchy created and the number of processes such as "Synthesis", "Implement", "Generate", etc. you can run on it.
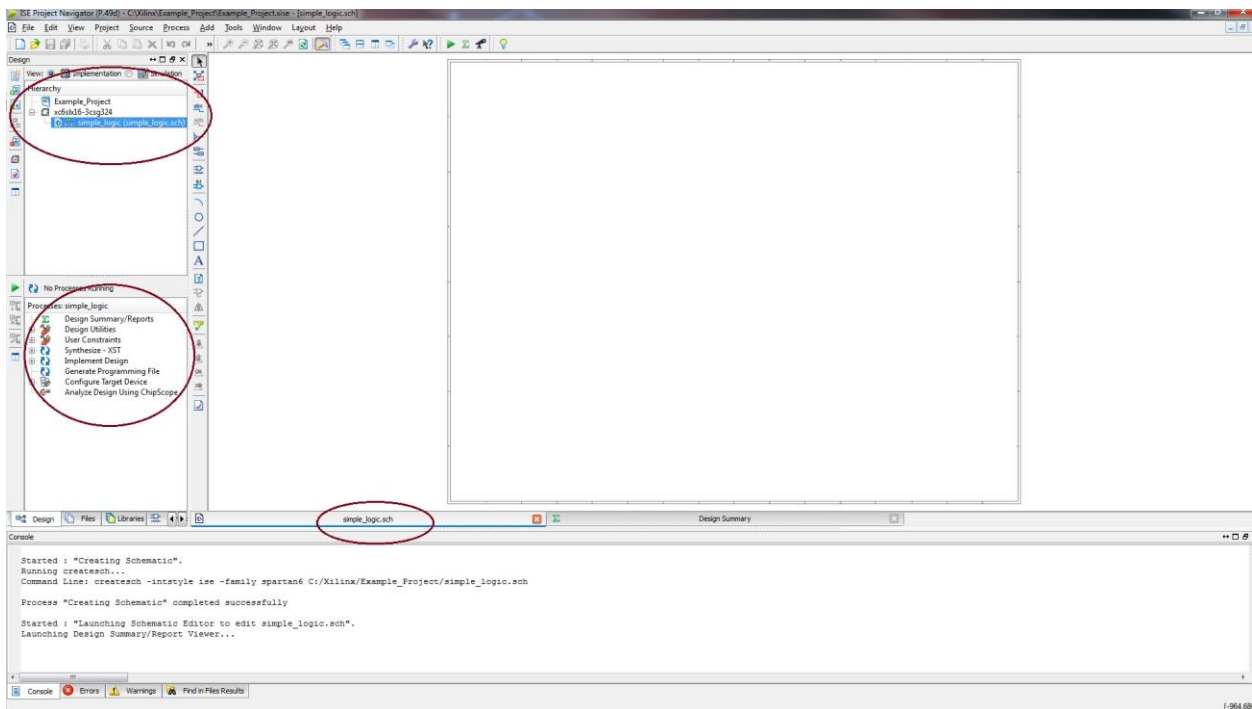
4. If you don't see this exact view, you may be looking at a different tab in a window. Each pane has tabs at the bottom that let you switch to look at different things. For example, for the left pane in the image above there are multiple tabs to allow different things, "Files", "Snapshot", "Libraries", and "Symbols" tabs that you can click on and get different information about the project. The lower left pane is the Processes tab and Options tab that show different tools and steps. The main window in the figure above has a "simple_logic.sch" tab for the new schematic, and a "Design Summary" tab. If you open more schematics, or other types of files (like Verilog files) the main pane will have additional tabs.

Now you can select components from the **Symbols** tab of the upper left pane and drag them to your schematic. You can narrow down your choices using the **Categories**, or by typing the first few characters of the symbol you're looking for in the **Symbol Name Filter**, or just scroll through the lists and see what's there. The important category for now is **Logic:** General logic gates. See the Lab handouts for restrictions on which logic gates you should use!

I'll grab some components from those Categories to make a very simple schematic to implement **(A & (!B)) | (B & C)** and drop them into the schematic page. Now we need to add wires and I/O markers by using the tools on the vertical bar between the schematic page and the side pane. Also pay attention to the **"Options"** tab at the bottom of the left pane. This section gives you some options on the schematic such as "selecting the entire wire branch" or "selecting line segments". This is good to know if you wanted to only remove a piece of wire and not everything that it is attached to.
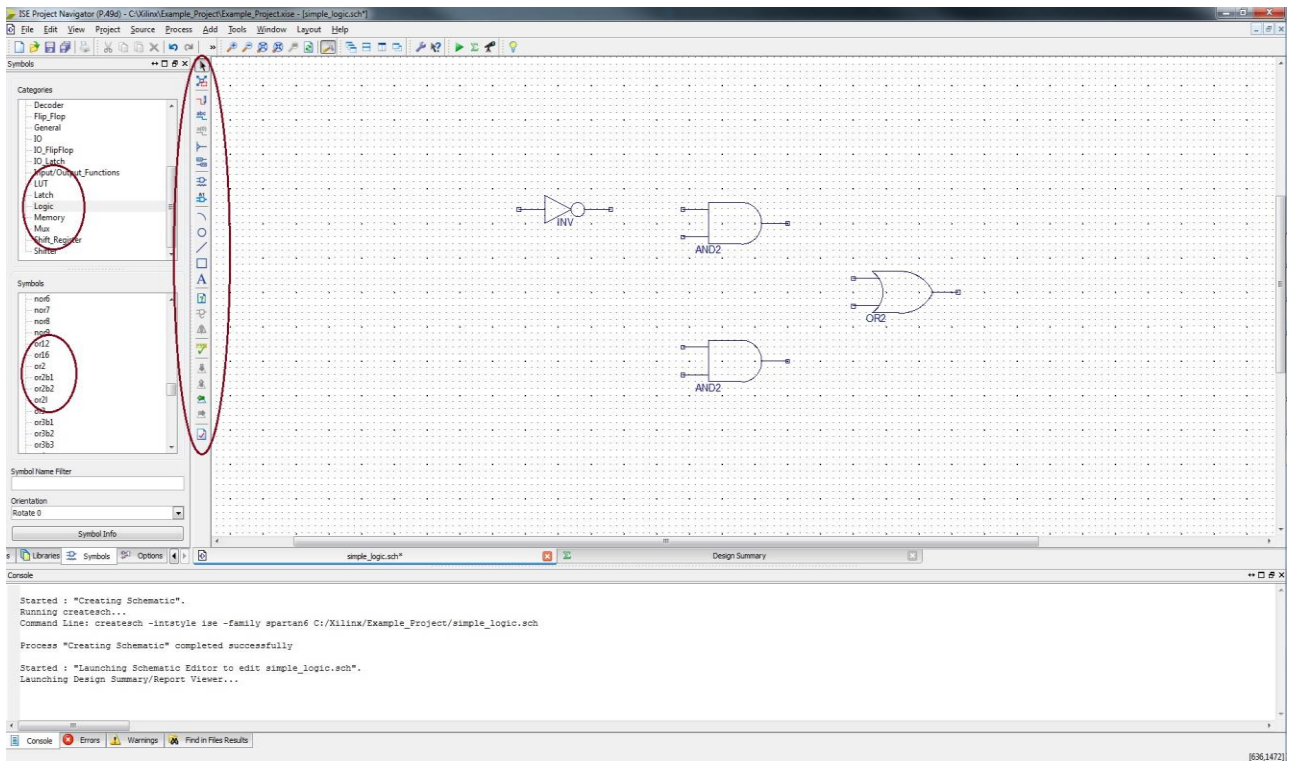
5. Use the wiring tool to wire up the components. It is in the tool bar and looks like a red line and a pencil. You could also use **Add →Wire** from the menu. I'm using the following components (from the **Logic** category) in this example:

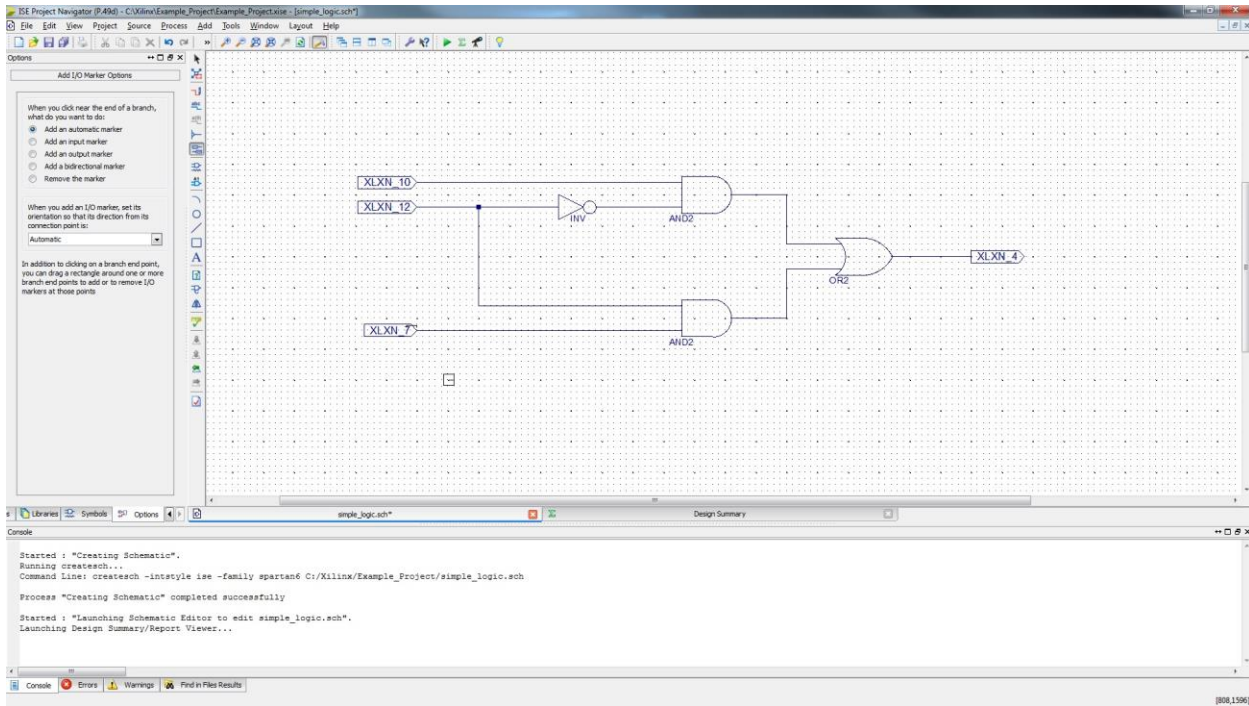   a. **and2:** a two-input AND gate
    b. **or2:** a two-input OR gate
    c. **inv:** an inverter
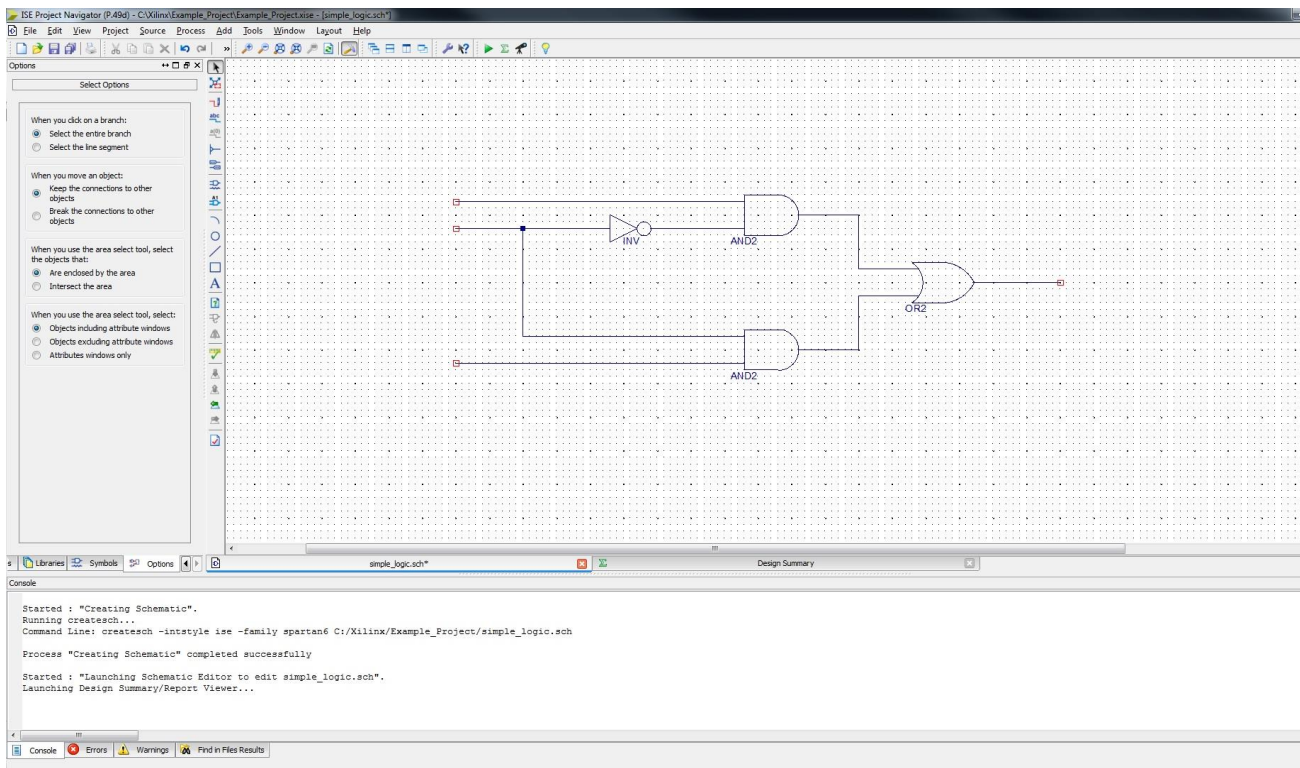
6. Now place **I/O Markers** to the inputs and outputs. I've used the **I/O Marker** widget that looks like two little labels. You can also use the **Add → I/O Marker** command from the menu. Click on the endpoints of the wire to add the marker.

7. You should always change the name of the marker to whatever you want but choose a good identifier. You should double click the marker, or select the marker and right click to get a menu and choose **Edit → Properties**. Then click on "Nets" and then edit the "Name", also observe the Port Polarity, then click OK. I'm calling the inputs **A**, **B** and **C** and the output **F**. An I/O Marker dialog box looks like:
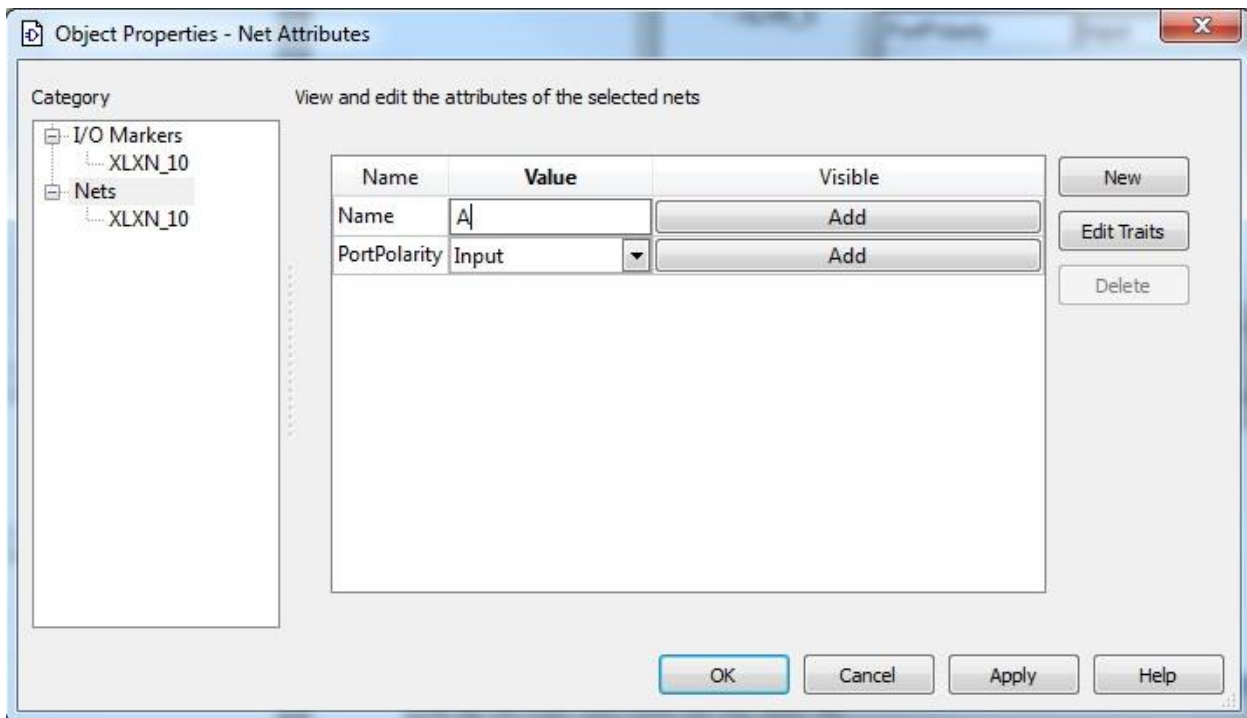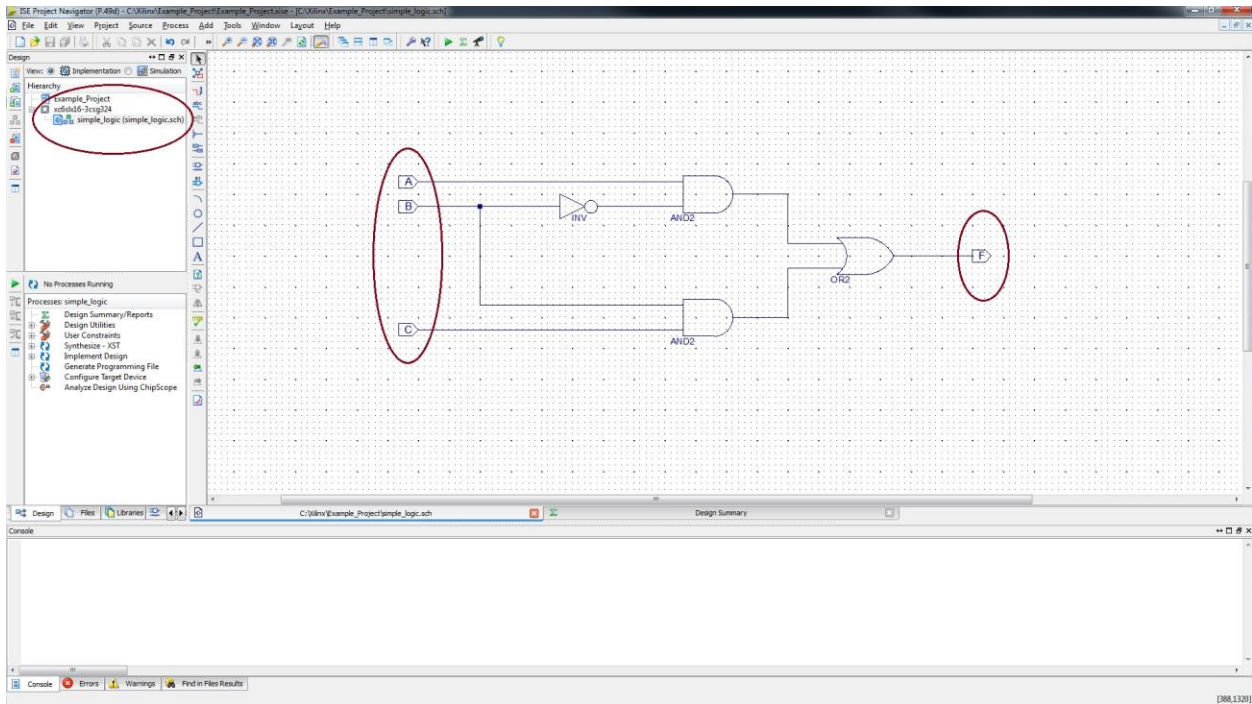
8. Now the resulting schematic looks like the following and it is ready for simulation or synthesis. Once your schematic is saved it should show up in the **Sources** pane in the upper left of the ISE screen.

When you are happy with your schematic (and your schematic will likely include other logic gates from the Logic category that I haven't used in this example!), save it. Hopefully you don't have any errors or warnings. If you do, you need to fix them.

Part III: Implement a function using Verilog

This schematic representation of the function **F = (A & (!B)) | (B & C)** is now ready for further processing either for simulation or implementation on the Nexys 3 board. Now we will explore the implementation of the same circuit using a "**Verilog**" module instead of a schematic, and both with have the same functionality.

1. Just like step 2 of the schematic capture, we need to first add a new source. So again choose **Project → New Source** or right click on the Hierarchy section of the design windows to get the dialog box that adds a new source file to your project (or use the **New Source** widget on the left vertical tool bar). This time choose **"Verilog Module"** and give it a file name.

Attention: it is important that you create the correct form of Verilog file for the specific use. Remember to ONLY use **"Verilog Module"** for *Implementation* and **"Verilog Test Fixture"** for *simulation* purposes. Do not do this the other way or things will not wok right
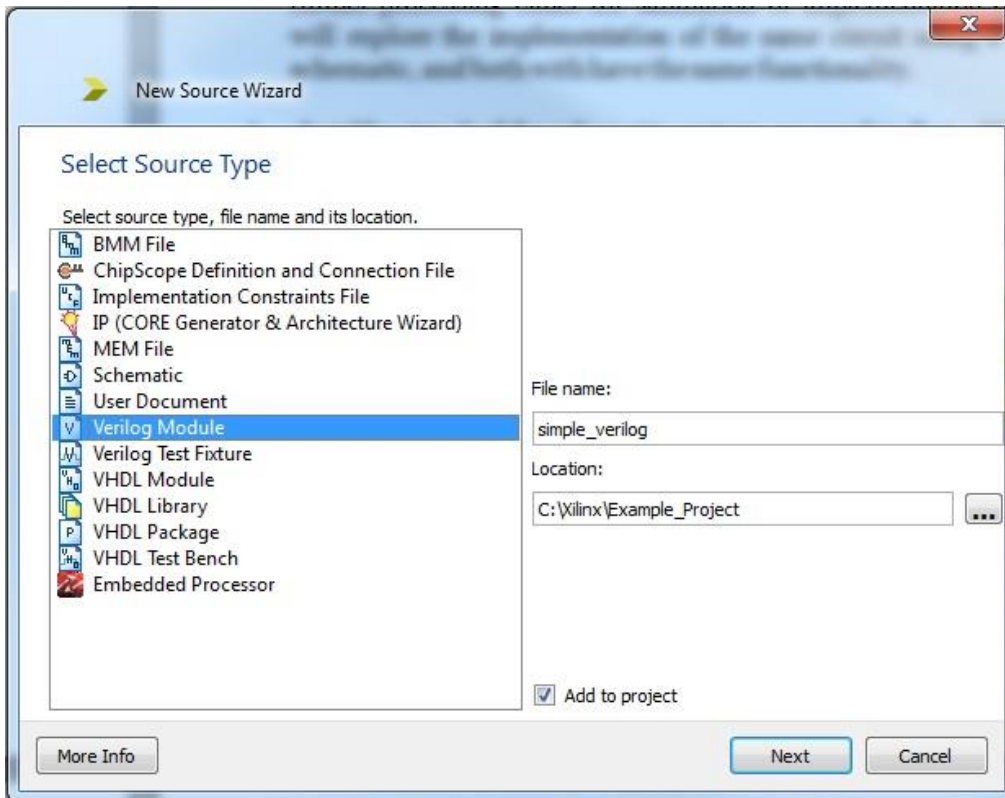
2. Click NEXT and you should see the module definition box. Here you can setup I/O names with correct polarity and a choice for buses and the width which we will be using in the future labs. Note that you do not have to add anything here right away and you can always add the I/O definitions to the module's header when it is created. After you're done click NEXT and then observe the summary page for a quick review of your I/O list.
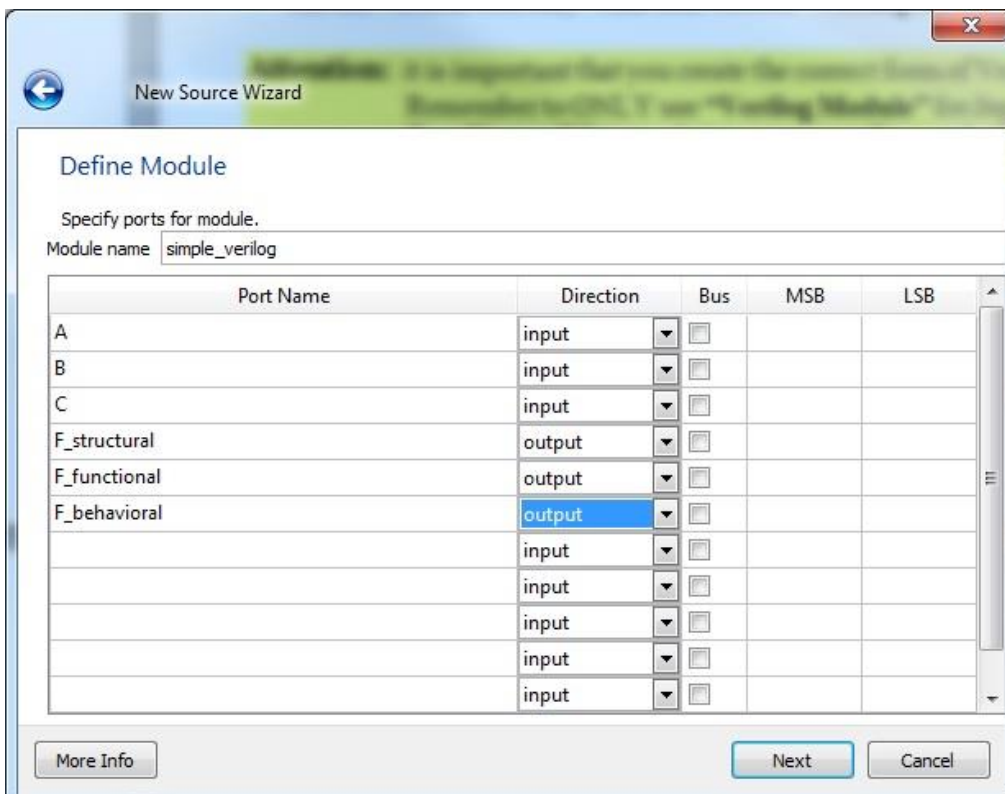
3. Now you should have a tab for the Verilog file opened in the ISE main pane. There is a line reading "**`timescale 1ns / 1ps**". Please leave line alone and never delete it. It just lets the ISE know that each unit of time should be 1ns and have the resolution up to 1ps for any timing purposes (more on this later).

   Next is the module header that contains the module name and the list of I/O as its parameters. If you notice I have declared multiple outputs to implement the same circuit using different HDL forms when writing in Verilog. These will all implement the same function **F** represented by the schematic in Part I.

   Also you should notice that the Verilog file is added to the Hierarchy next to the schematic as a part of this project. Now we're ready to implement the Verilog description of the function **F** in the three intended forms.
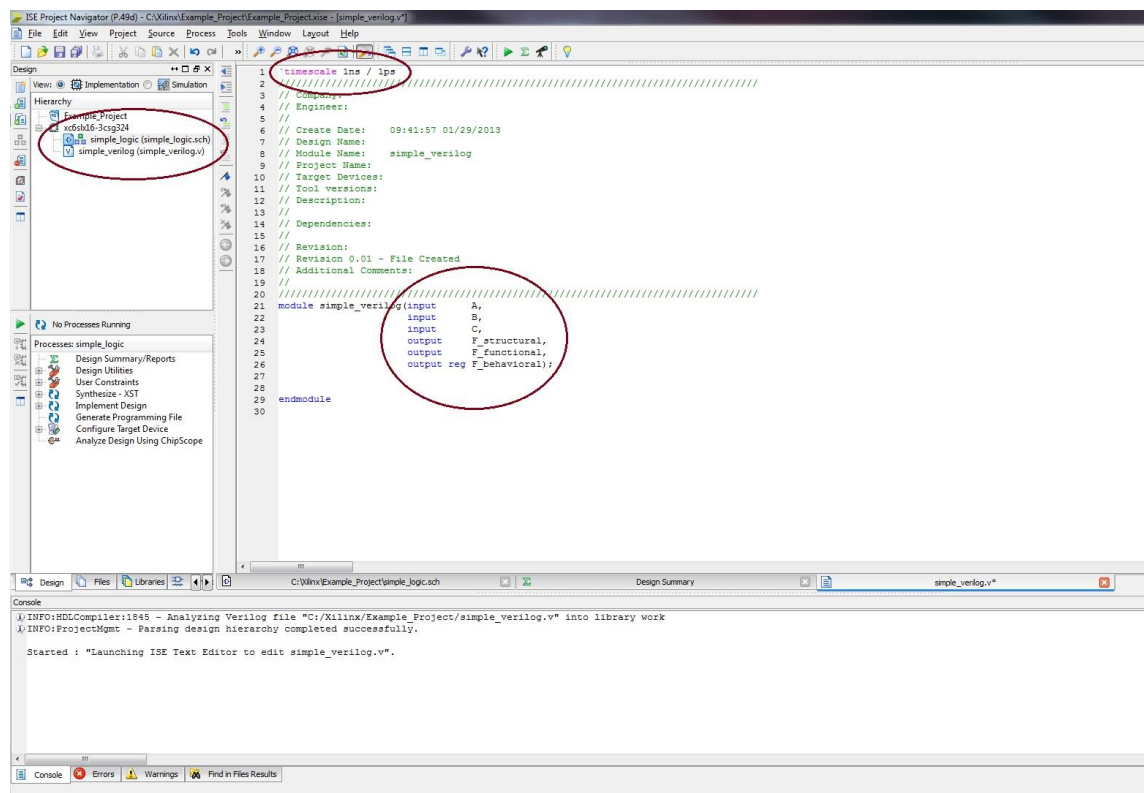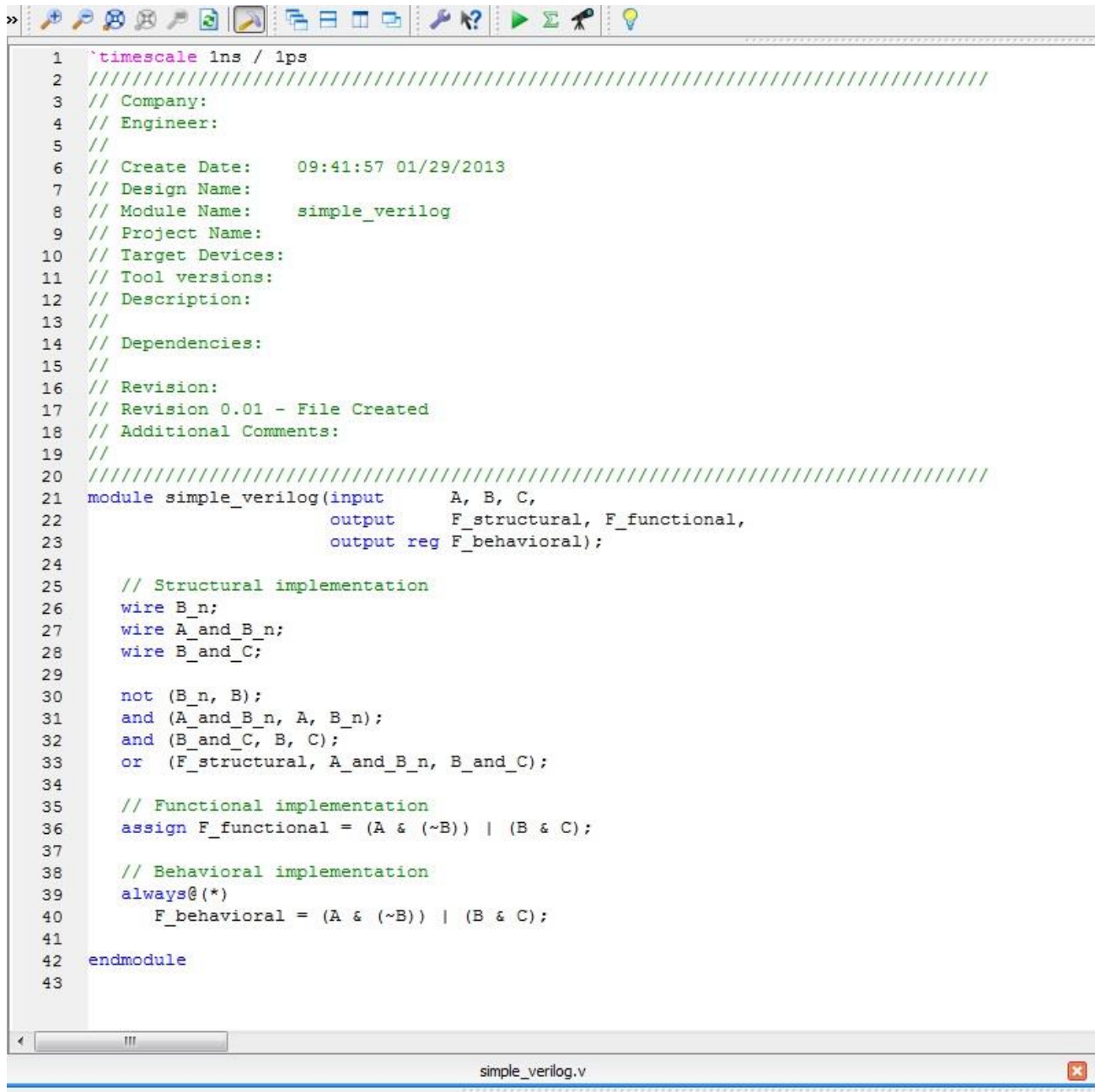
4. For the implementation the three forms we have different formats. Sometimes we want to declare internal wires to connect things easily and in a more organized fashion, but if you didn't declare these wires, ISE will assume a single bit wire for the use.

   The Structural version: This is done using gate primitives that are automatically taken from the Xilinx libraries (just as you would with a schematic) by calling their name and passing parameters, so for a two input gate we have the format "gate (output, input1, input2)" and this is just like doing a schematic in words.

   The Functional version: Using the **"assign"** keyword to assign the results of the function expression to the output. The expression of the function looks a lot like how you would write it down on paper.

   The Behavioral version: Using a synthesis directive called an **"Always Block"** we can implement the same function. The difference is that it only wakes up and assign the output

when any of the inputs are changes, hence the (*) which means "any change". We will be using this format extensively in the future labs. One thing to notice is that any output being assigned inside of an always block needs to be declared as a **"reg"** for synthesis purposes, and that you cannot use the "assign" keyword inside of such block.

```verilog
1    `timescale 1ns / 1ps
2    //////////////////////////////////////////////////////////////////////////////////
3    // Company:
4    // Engineer:
5    //
6    // Create Date:    09:41:57 01/29/2013
7    // Design Name:
8    // Module Name:    simple_verilog
9    // Project Name:
10   // Target Devices:
11   // Tool versions:
12   // Description:
13   //
14   // Dependencies:
15   //
16   // Revision:
17   // Revision 0.01 - File Created
18   // Additional Comments:
19   //
20   //////////////////////////////////////////////////////////////////////////////////
21   module simple_verilog(input       A, B, C,
22                         output      F_structural, F_functional,
23                         output reg F_behavioral);
24
25      // Structural implementation
26      wire B_n;
27      wire A_and_B_n;
28      wire B_and_C;
29
30      not (B_n, B);
31      and (A_and_B_n, A, B_n);
32      and (B_and_C, B, C);
33      or  (F_structural, A_and_B_n, B_and_C);
34
35      // Functional implementation
36      assign F_functional = (A & (~B)) | (B & C);
37
38      // Behavioral implementation
39      always@(*)
40         F_behavioral = (A & (~B)) | (B & C);
41
42   endmodule
43
```

simple_verilog.v

5. You should always save your work multiple times as there always a slight chance that the tool may crash and you will lose your work. It is important to understand that hardware CAD tools are massive and complex so there is always a chance for unexpected or faulty behavior.

Now the circuit is ready for simulation or implementation on the board. It is important to notice that most processes are the same for both the schematic and the Verilog version of your design.

Part IV: Simulate the schematic/Verilog circuit using the ISIM and a Verilog test fixture

Now that you have a saved schematic, you need to simulate its behavior. The simulator we'll use is the **ISE** built in simulator, which is essentially a Verilog simulator. The schematic that you just drew will be saved as a piece of Verilog behind your back if you choose to simulate it. In order to simulate the circuit you need:

Testbench: is a file that becomes a top module to your design and applies inputs to your circuit, and potentially checks that the outputs are correct. This will be another Verilog file written slightly differently than circuit implementation. The testbench will instantiate one copy of your circuit, and call it **UUT** for "Unit Under Test". You will then write the Verilog statements that set the inputs to your circuit (the UUT), and looks at the outputs produced by your circuit. You need to know only very basic Verilog syntax to do this.

1. First you need to ensure that the ISE more is changed to **"Simulation"** from implementation. Go to the top left pane and change the **"View"** field to simulation. The design window will then change slightly with different options. Referring back to the same step in creating a **"New Source"** create a **"Verilog Test Fixture"** to create a Verilog file that will contain the test code.
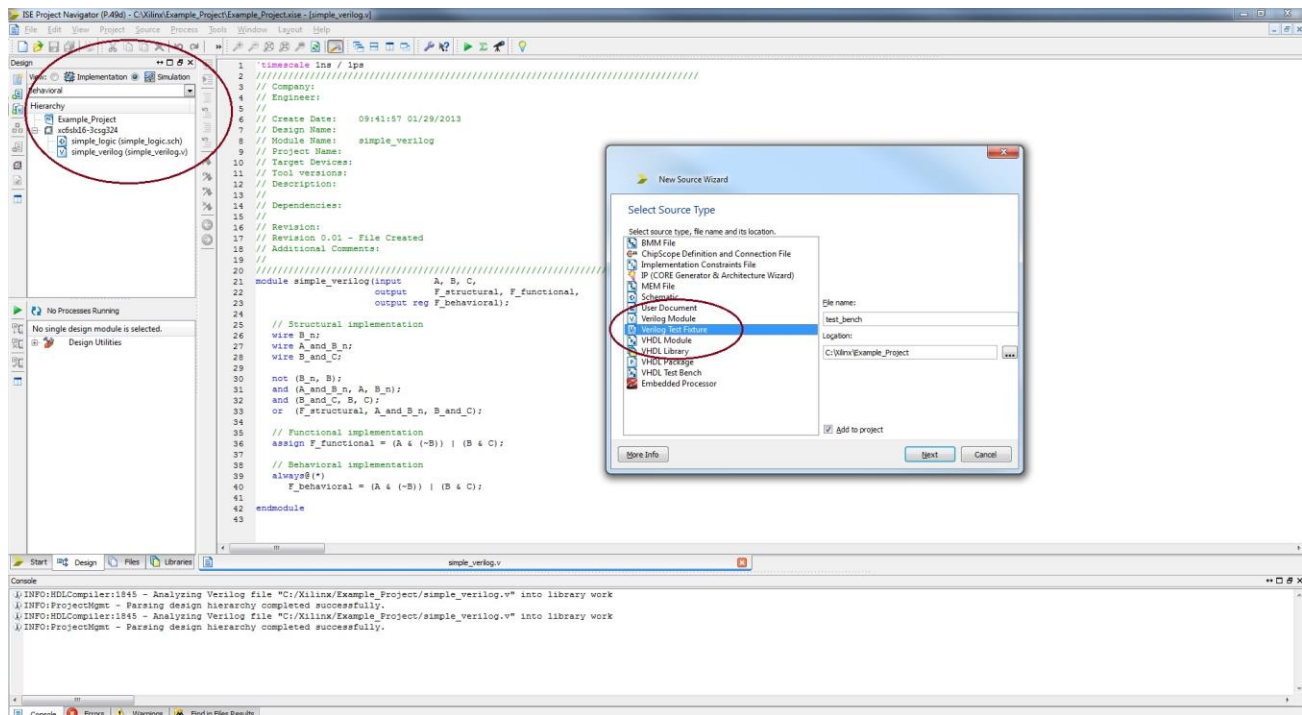
2. Click NEXT and choose which design you want to associate the test bench with. This is very important as you will have multiple modules or schematics in the future and you need to be sure which design will be going under test using the test bench. In this case I will just choose the "simple_verilog" module to be tested. The procedure for testing the schematic version is exactly the same, you just have to choose the appropriate source to be associated with the test bench.
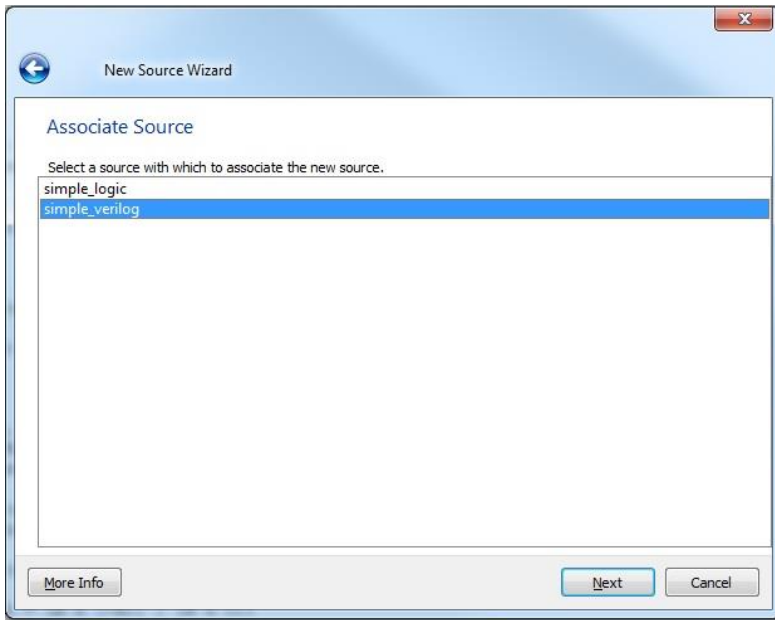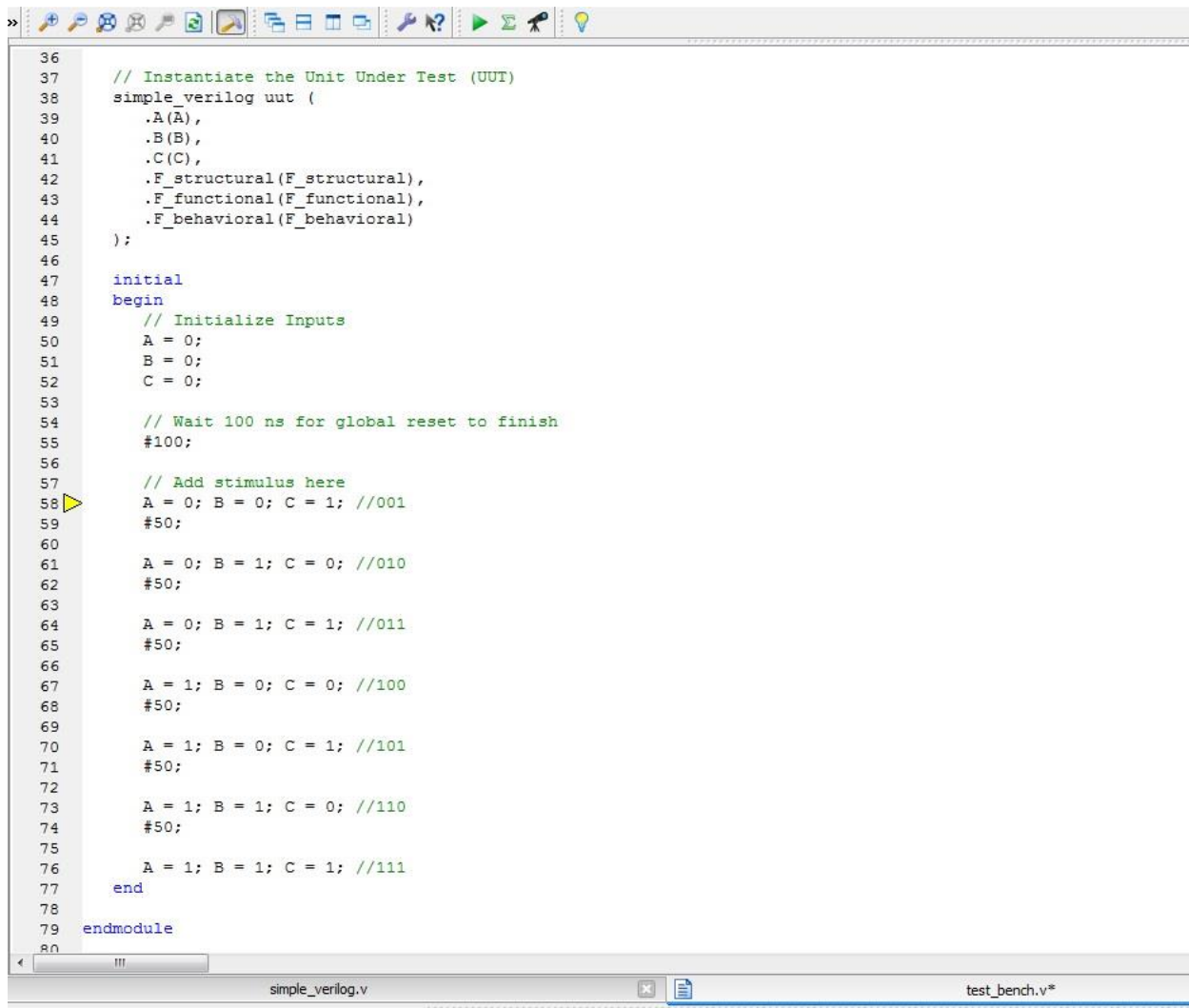
3. Click NEXT and after observing the summary click FINISH. Now you'll get a new piece of Verilog code generated for you. This Verilog code instantiates the **"simple_verilog"** module as the **UUT**, and includes some other stuff related to how the **UUT** is connected to the testbench. It looks like this:

```verilog
19  // Revision:
20  // Revision 0.01 - File Created
21  // Additional Comments:
22  //
23  //////////////////////////////////////////////////////////////////////////////
24
25  module test_bench;
26
27      // Inputs
28      reg A;
29      reg B;
30      reg C;
31
32      // Outputs
33      wire F_structural;
34      wire F_functional;
35      wire F_behavioral;
36
37      // Instantiate the Unit Under Test (UUT)
38      simple_verilog uut (
39          .A(A),
40          .B(B),
41          .C(C), |
42          .F_structural(F_structural),
43          .F_functional(F_functional),
44          .F_behavioral(F_behavioral)
45      );
46
47      initial begin
48          // Initialize Inputs
49          A = 0;
50          B = 0;
51          C = 0;
52
53          // Wait 100 ns for global reset to finish
54          #100;
55
56          // Add stimulus here
57
58      end
59
60  endmodule
61
62
```

simple_verilog.v   test_bench.v

4. You can now write your test bench code as an **initial** block right before the **endmodule**. Basically you set the values of your inputs, and tell the simulator how long to wait between each change on the inputs. The results will eventually be plotted on a waveform for you. Verilog syntax for setting a variable is very simple, and the **#50** notation just means for the simulation to wait for 50 ticks of the simulation clock before moving on to the next statement. A very simple test bench for this circuit looks like the following. I added the lines between **initial** and **end** to drive the inputs with different values so that we can see what the circuit result is. Note that each statement in Verilog ends with a semicolon, and you can put multiple statements on a line if you like.

    Later in upcoming labs we will explore different options that we can write the sequencing of the inputs to prevent having to hardcode lots of input changes, you can see that if we had two more inputs then we had to write out 32 different possibilities for 5 inputs. We will see how using a "For" loop in the test bench will make things a lot and shorter.

```
36
37   // Instantiate the Unit Under Test (UUT)
38   simple_verilog uut (
39   .A(A),
40   .B(B),
41   .C(C),
42   .F_structural(F_structural),
43   .F_functional(F_functional),
44   .F_behavioral(F_behavioral)
45   );
46
47   initial
48   begin
49   // Initialize Inputs
50   A = 0;
51   B = 0;
52   C = 0;
53
54   // Wait 100 ns for global reset to finish
55   #100;
56
57   // Add stimulus here
58   A = 0; B = 0; C = 1; //001
59   #50;
60
61   A = 0; B = 1; C = 0; //010
62   #50;
63
64   A = 0; B = 1; C = 1; //011
65   #50;
66
67   A = 1; B = 0; C = 0; //100
68   #50;
69
70   A = 1; B = 0; C = 1; //101
71   #50;
72
73   A = 1; B = 1; C = 0; //110
74   #50;
75
76   A = 1; B = 1; C = 1; //111
77   end
78
79   endmodule
80
```
simple_verilog.v                                        test_bench.v*

5.  We usually want to test all possible inputs to be able to draw a better conclusion on whether the circuit is functioning correctly. After you're satisfied with the input setting of your test bench make sure to save. Always observe the console window to look out for errors after saving.

    Now you are ready to simulate your Verilog circuit. Observe that the test bench Verilog file is now the top module to your "simple_verilog" module in the simulation design

view. It is very important to have the test bench file selected for simulation or things will go wrong. After selecting and highlighting the test bench file in the design windows, you can check your test bench for syntactic errors by clicking the **"Behavioral Check Syntax"** in the processes windows right below. If your code is correct syntactically then you'll get a green check mark. This is not crucial as if you run the simulation without it then the syntax will be automatically checked by the ISim and will not run if there are errors in your test bench. Now you can double-click the **"Simulate Behavioral Model"** to see the waveform generated by the ISim.
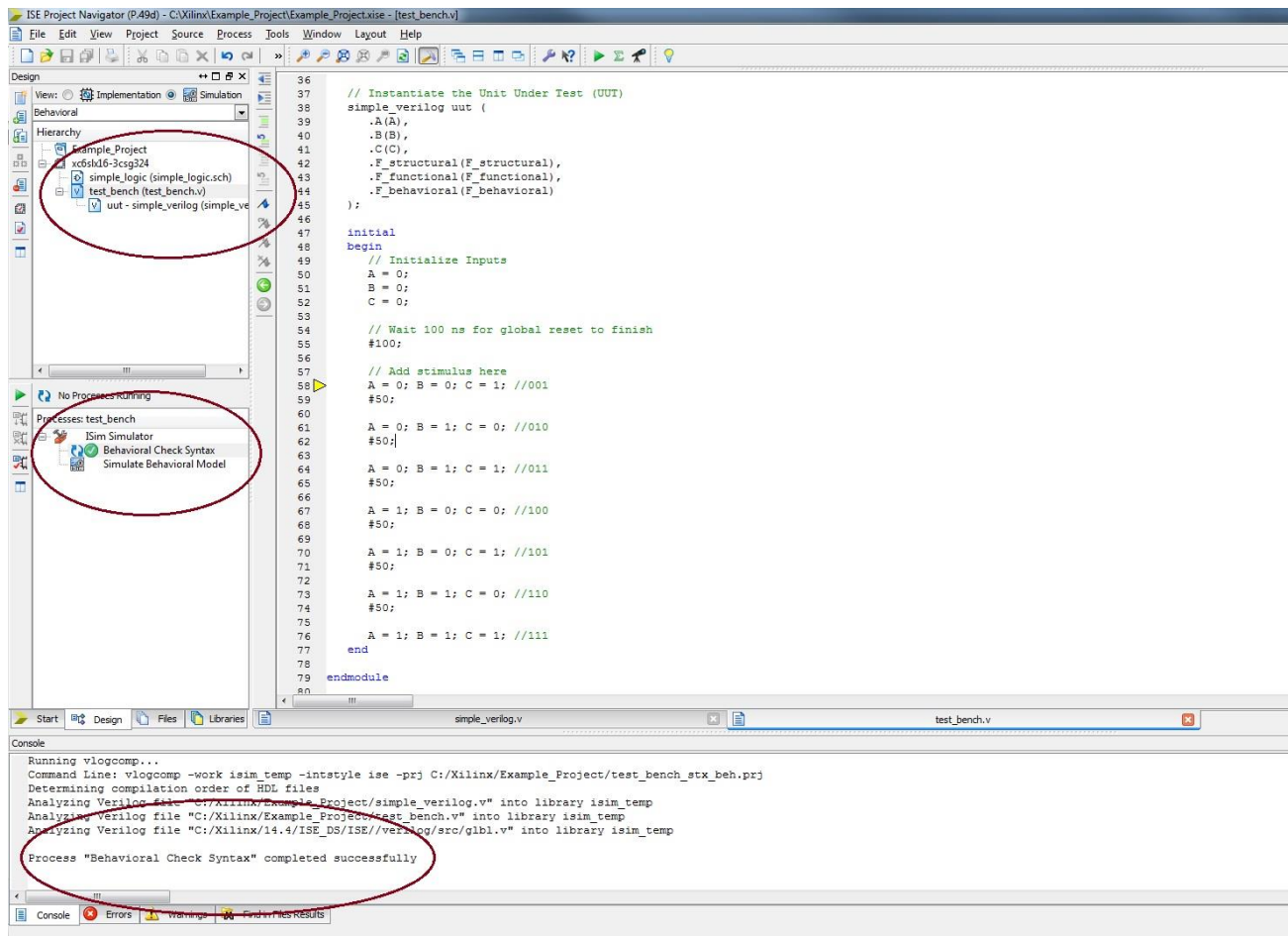


7. Double-clicking the **Simulate Behavioral Model** will fire up the simulator on your testbench file. Because your testbench includes an instance of your schematic (the **UUT**) and some commands to drive signals into your schematic, this will result in simulating your schematic. The output will be displayed as waveforms as shown. Note that the simulator is by default set up to simulate for 1000ns, so all the stuff I did is bunched up at the beginning of the simulation (the first 0ns). I had to zoom out a little to see this view. The values reported for A, B, S, and F are the values seen at the blue bar. You can pick up (with the mouse) and move the blue bar to see the values at different points in the simulation.

By looking at the waveform we can see that all three different forms of expressing the function in Verilog (structural, functional, and behavioral) are all holding the same behavior throughout the simulation. You can click on the waveform in different places (the yellow line is where in the range of time in the waveform it was clicked) and you can see values quickly for all I/O in the **"Name"** and **"Value"** sections to the left of the waveform.

The simulation is now done. You can always go back to the test bench and make a more complicated set of tests and re-simulate the circuit.
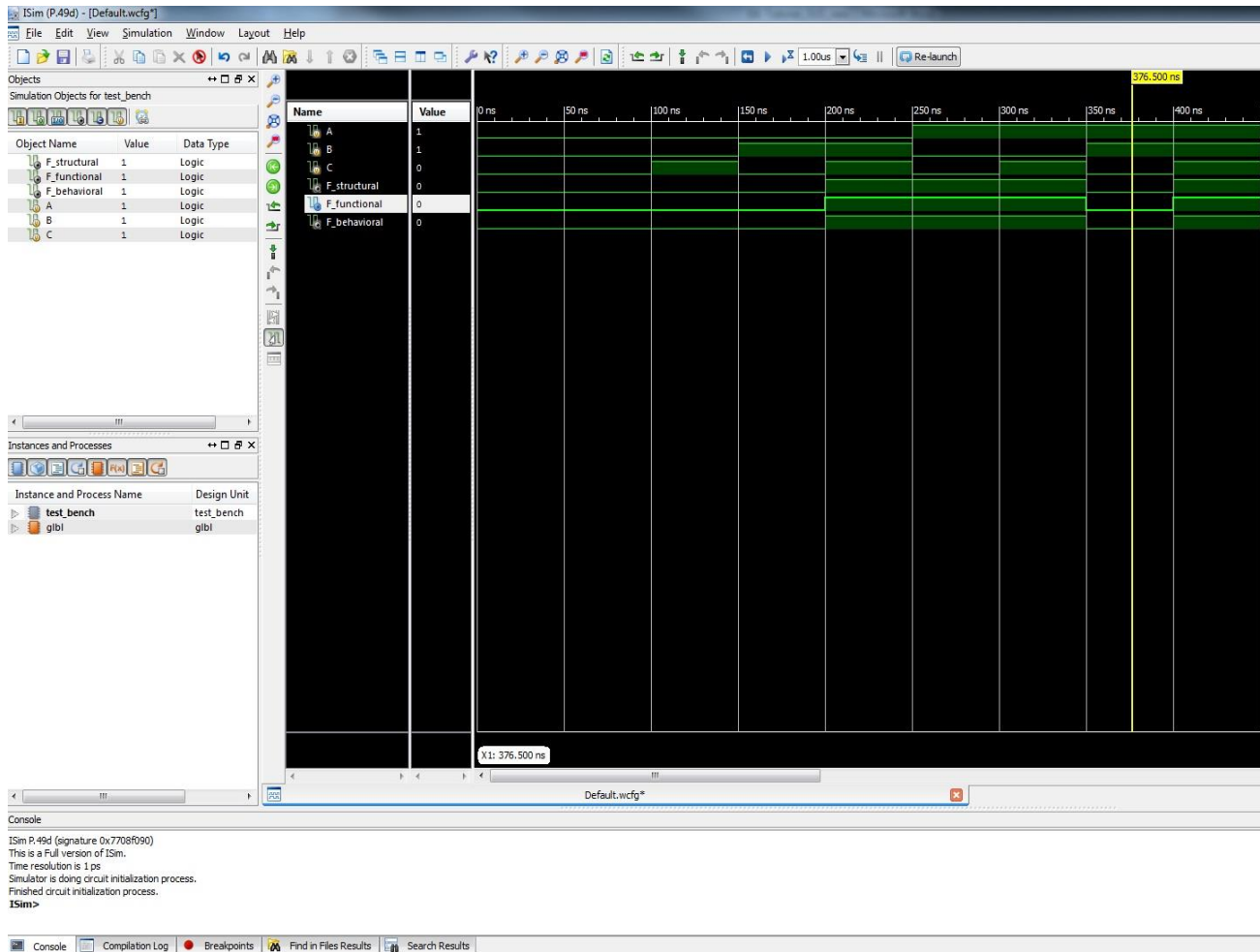


It is good to observe the other capabilities of your simulation windows. The console is where all of the simulator messages will be printed. Look for errors or warning about your design.

The left panes will allow you to dig into your design hierarchy to grab signals that are deep in the design and may not be set in the top module, recall that the test bench is only created for one module, so if you have a multiple level hierarchy then you need to use these windows to navigate to the desired signal and drag-and-drop into the simulation windows.

Sometimes we want the values of the test bench results for the Verilog module's output to be printed into the console of the simulator windows. We can then add the Verilog print statement called **$display** in between our test bench code. This will allow us to monitor signals in large designs and long waveforms with many signals being looked at. We can also format the printed statement to what we want it to show on the console for better readability of the results. This is called a **Self-Checking** testbench.

Below are some $display statements added to the current test bench to show the effects it will have. Never the less, you should always put a $display statement at the beginning and the end of your test code to indicate the starting and actual finishing of the simulations.

```
47      initial
48      begin
49          $display("The simple_verilog module simulation is starting ...");
50
51          // Initialize Inputs
52          A = 0;
53          B = 0;
54          C = 0;
55
56          // Wait 100 ns for global reset to finish
57          #100;
58          $display("ABC = %b%b%b, F_structural = %b, F_functional = %b, F_behavioral = %b", A, B, C, F_structural, F_functional, F_behavioral);
59
60          // Add stimulus here
61          A = 0; B = 0; C = 1; //001
62          #50;
63          $display("ABC = %b%b%b, F_structural = %b, F_functional = %b, F_behavioral = %b", A, B, C, F_structural, F_functional, F_behavioral);
64
65          A = 0; B = 1; C = 0; //010
66          #50;
67          $display("ABC = %b%b%b, F_structural = %b, F_functional = %b, F_behavioral = %b", A, B, C, F_structural, F_functional, F_behavioral);
68
69          A = 0; B = 1; C = 1; //011
70          #50;
71          $display("ABC = %b%b%b, F_structural = %b, F_functional = %b, F_behavioral = %b", A, B, C, F_structural, F_functional, F_behavioral);
72
73          A = 1; B = 0; C = 0; //100
74          #50;
75          $display("ABC = %b%b%b, F_structural = %b, F_functional = %b, F_behavioral = %b", A, B, C, F_structural, F_functional, F_behavioral);
76
77          A = 1; B = 0; C = 1; //101
78          #50;
79          $display("ABC = %b%b%b, F_structural = %b, F_functional = %b, F_behavioral = %b", A, B, C, F_structural, F_functional, F_behavioral);
80
81          A = 1; B = 1; C = 0; //110
82          #50;
83          $display("ABC = %b%b%b, F_structural = %b, F_functional = %b, F_behavioral = %b", A, B, C, F_structural, F_functional, F_behavioral);
84
85          A = 1; B = 1; C = 1; //111
86          #50;
87          $display("ABC = %b%b%b, F_structural = %b, F_functional = %b, F_behavioral = %b", A, B, C, F_structural, F_functional, F_behavioral);
88
89          $display("The simple_verilog module simulation has ended ...");
90      end
91
```

simple_verilog.v   test_bench.v*

And the resulting console output for the above test bench code is the following:

```
Console
This is a Full version of ISim.
Time resolution is 1 ps
Simulator is doing circuit initialization process.
The simple_verilog module simulation is starting ...
Finished circuit initialization process.
ABC = 000, F_structural = 0, F_functional = 0, F_behavioral = 0
ABC = 001, F_structural = 0, F_functional = 0, F_behavioral = 0
ABC = 010, F_structural = 0, F_functional = 0, F_behavioral = 0
ABC = 011, F_structural = 1, F_functional = 1, F_behavioral = 1
ABC = 100, F_structural = 1, F_functional = 1, F_behavioral = 1
ABC = 101, F_structural = 1, F_functional = 1, F_behavioral = 1
ABC = 110, F_structural = 0, F_functional = 0, F_behavioral = 0
ABC = 111, F_structural = 1, F_functional = 1, F_behavioral = 1
The simple_verilog module simulation has ended ...
ISim>
```

Console | Compilation Log | Breakpoints | Find in Files Results | Search Results

Part V: Constraint, Synthesize, Implement, Generate Bitstream, and Program the Nexys 3 FPGA board

Overview: Now that you have a correctly simulating Verilog module, you will have the ISE tools synthesize your Verilog or schematic to something that can be mapped to the Xilinx FPGA. That is, the Verilog code will be converted by ISE to some gates that are on the FPGA. To be even more specific, ISE will convert the Verilog description into a set of configuration bits that are used to program the Xilinx part to behave just like the Verilog code. Those configuration bits are in a **.bit** file and are downloaded to the Xilinx part in this next section of the tutorial.

For the purposes of this tutorial I will choose to put the Verilog version on the Nexys board, but the process is exactly the same for any other design form i.e. schematics. I will use the first three toggle switches on the board for **A, B,** and **C,** and the first three LEDs for **F**.
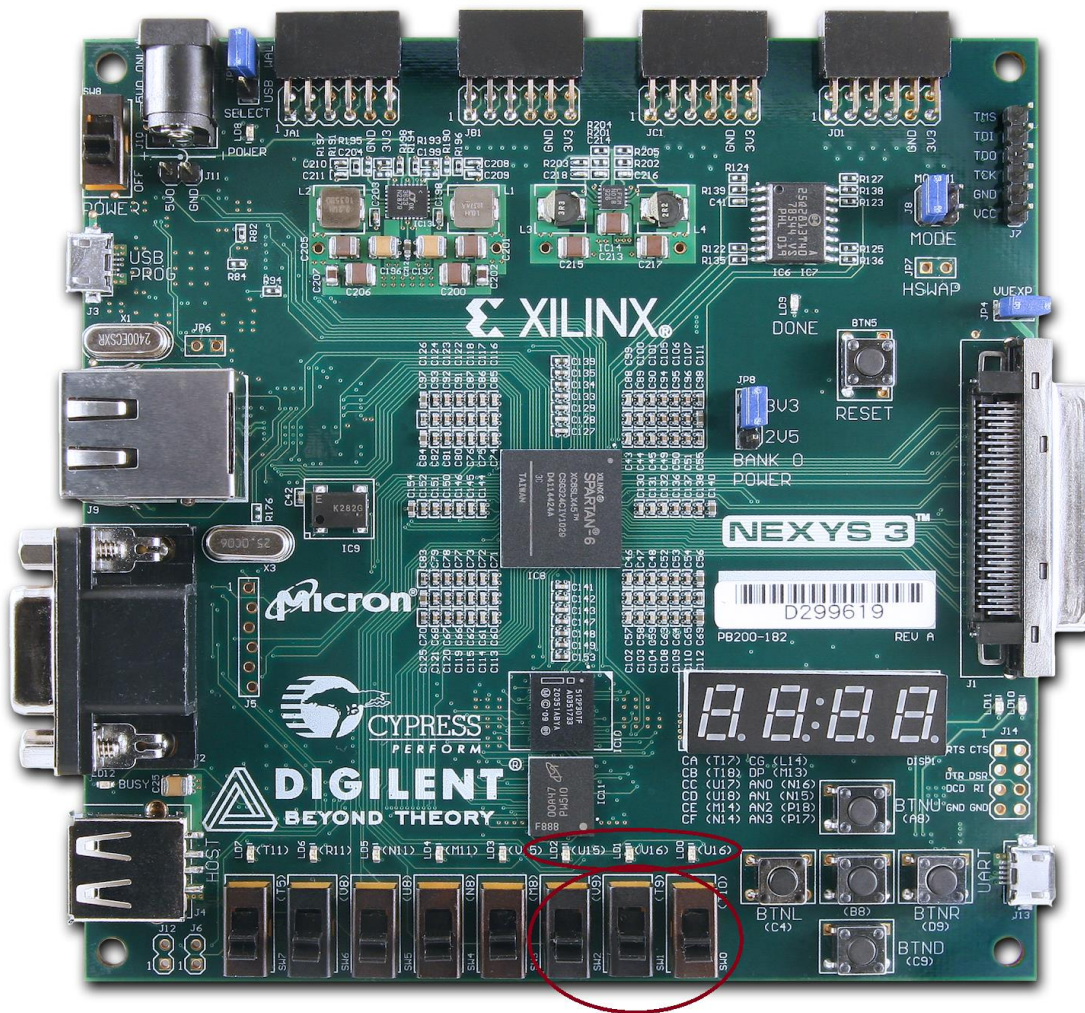
UCF (User Constraints File): Because we're headed towards putting this on the Xilinx FPGA on the Nexys 3 board, we need to set some constraints. In particular, we need to tell ISE which pins on the Xilinx chip we want **A**, **B**, **C**, and **F** assigned to so that we can access those from switches and LEDs on the Nexys 3 board. For that we need a "User Constraints File".

1. First, you need to ensure that you're in the Implementation view and that the module you're trying to implement on the board is set as the top module in the top left pane in order to get the options to synthesize, implement, and generate the design. Now if you look at the bottom left pane you can see a number of processes you can run on this Verilog top module.
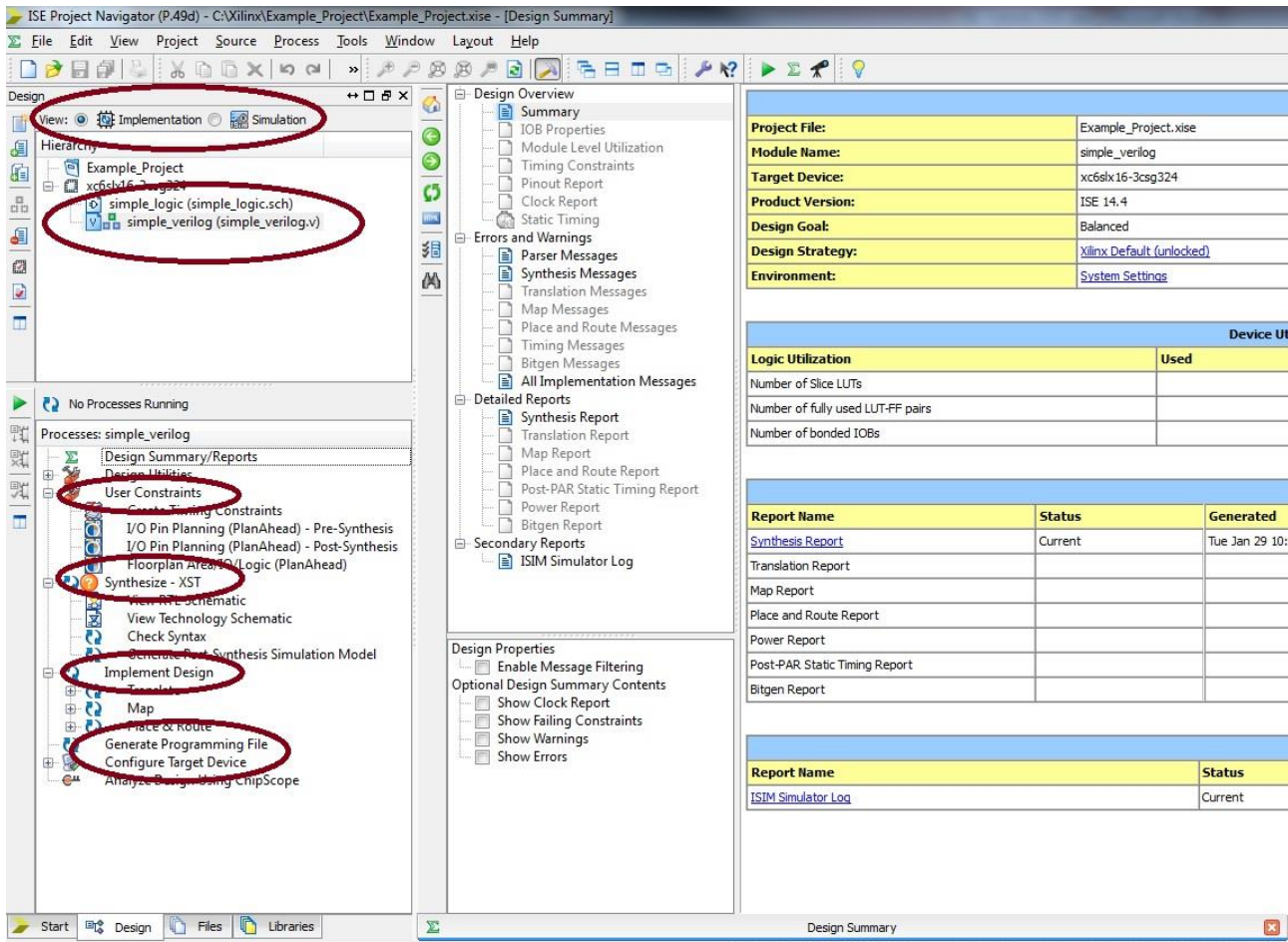
2. **Constraint:** Now it starts with creating a floor plan by setting the UCF file. To do this take a look at the **User Constraints** drop down option in the bottom left pane. We can set the pins in two different ways. Double click on the **I/O Pin Planning (Planahead) – Pre-Synthesis** since we want to set our pins before the synthesis process so they are included in it. This should bring up a message box for adding a new UCF file to your design, so click yes and this will kick start another Xilinx tool called **Plan Ahead.** This program allows you to set all constraints on all I/O pins in the design. Please follow these steps carefully.
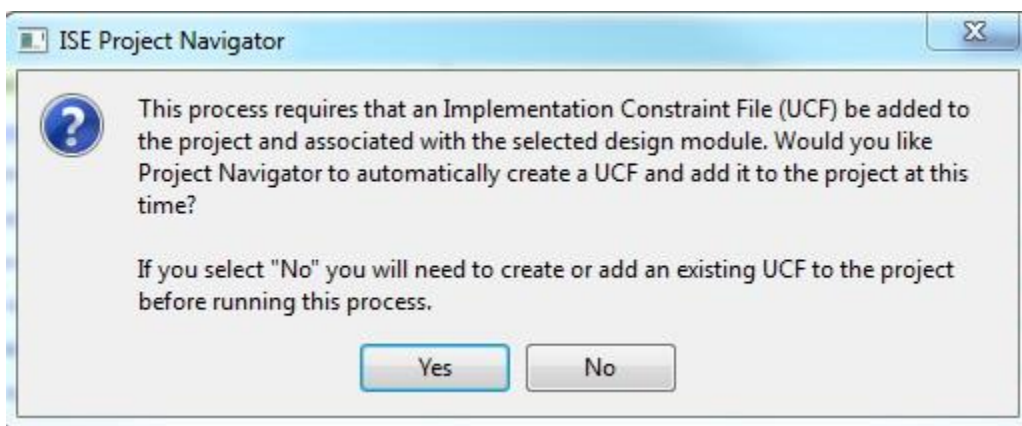
3. **Pin Assignment:** We need to edit the initial UCF for the details of connections between the ports and pins in the design, so as **Plan Ahead** opens (it looks a lot like ISE), take a look at the horizontal bottom pane. After expanding the **Scalar Ports** drop-down you should see all of you I/O pins. The first thing to do is to set all of the pins to **LVCMOS33** right away, if you don't do this, things might not work correctly on the board. Click on the name of the pin (A, B, C, …) and then look at the properties table line listed in front of it. The only two properties that we want to change are the **I/O Std** and **Site**. Remember, the I/O standard is always and should be set to LVCMOS33 (3.3 V Low-Voltage-Complimentary-Metal-Oxide-Semiconductor) for the Nexys 3 board. The Site is the Pin# (FPGA board Pin Identifier). You can find all sorts of the information including the module schematics and the Pin #s for all of the Nexys 3 Peripherals by studying the Nexys 3 manual found at http://www.digilentinc.com/Data/Products/NEXYS3/Nexys3_rm.pdf.

In order to change a property for any port, click on the rectangular space in the correct column and the correct row for the respective port. This click causes a drop-down menu to appear and then you can select the right choice. You can either type or select the correct choice. If there is a choice already selected then clicking on the text will allow you to change it, the hit enter. I looked up the three switches and the three LEDs, you can also look closely at the actual switch or LED on the board and you will see an identifier in parentheses (i.e. T10 for SW0 referring to Pin #T10 connecting to Switch 0, the first toggle switch from the right). Set the correct identifier to the correct port in your design in Plan Ahead as follows:

And your resulting UCF file should look something like the following:
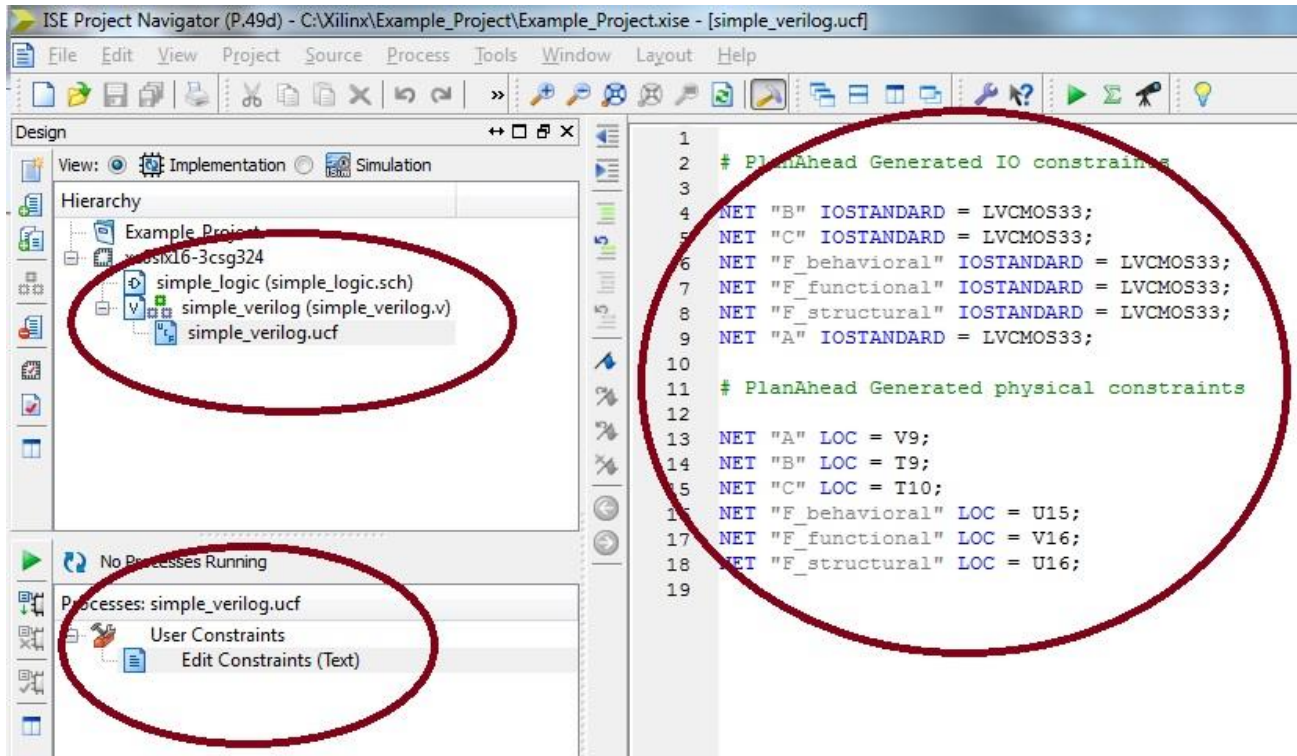
After you're done changing every port to LVCMOS33 and putting the right Pin # for all the ports in the design, click the **SAVE** button from the top menu in the page and close down Plan Ahead.

4. **Synthesize:** Now the design is ready for more processing and the next in line is to synthesize. This process will create a structural representation of the design (similar to compiling C code into assembly code). Do this by first ensuring that the top module is selected and highlighted in the top left pane of ISE and then simply double clicking **Synthesize – XST** in the lower left pane. After the process is done, you will either get a green check mark (everything is peachy), a yellow attention mark (there are warnings!), or a red x mark (there are errors!), and orange question mark just means "out of date". In the case of errors you need to investigate them by looking at the Errors tab on the bottom pane and fix, and then rerun synthesis. In the case or warnings make sure to review them and validate they are safe, or in the case of green you're ready for the next step.

The synthesis process also creates a couple of more useful things that you should explore and study. One is the synthesis report full of information about timing, resource usage details, and etc. The other is generated RTL schematic; sometimes it is very useful to see what the XST made out of your Verilog description, or even your version of schematic. You can view these (highly recommended) by expanding **Synthesize – XST** for the schematic and the Design Summary page in the main pane for the report.
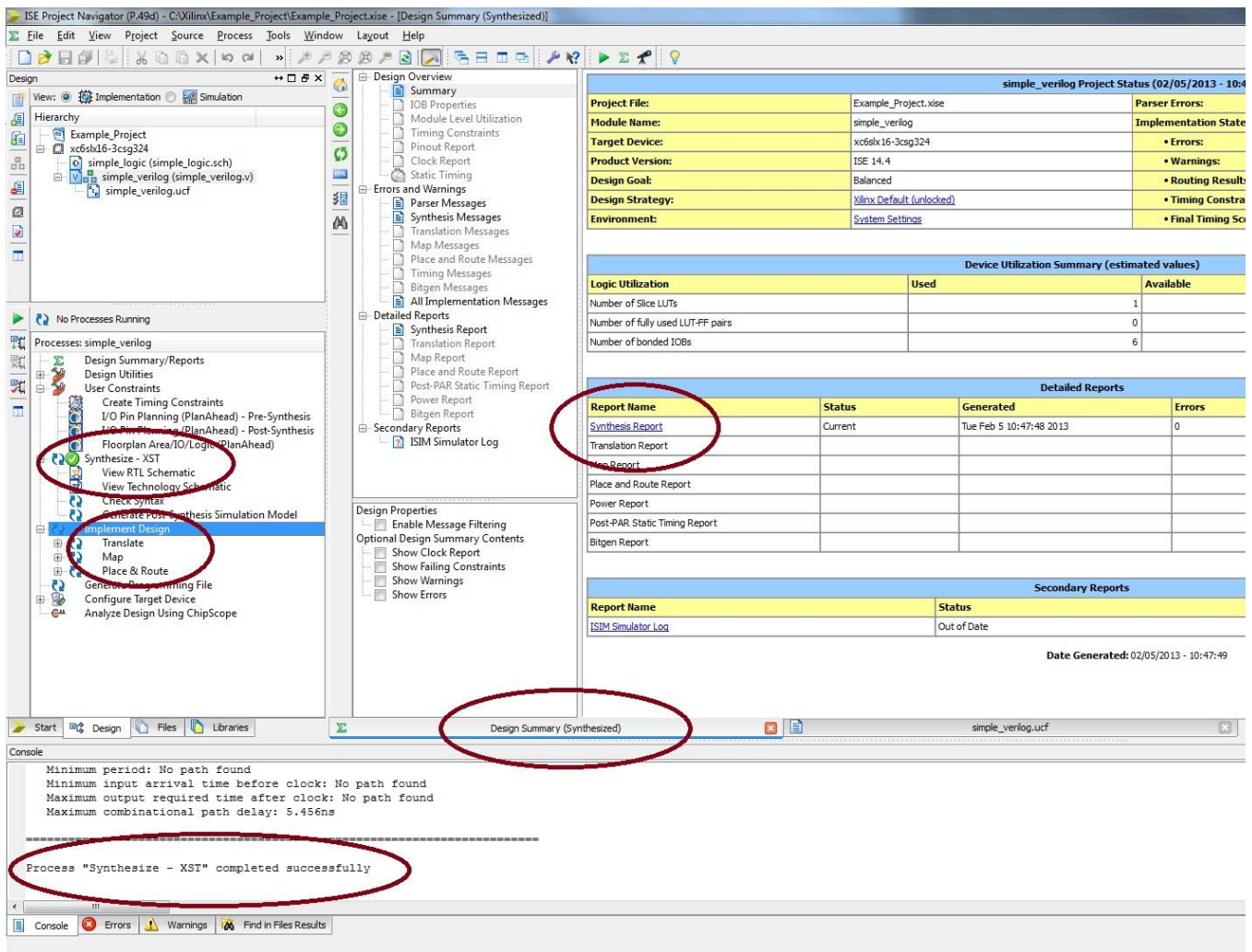
5. **Implement**: Next step is to define the hardware configuration. With your top module source file selected (simple_verilog.v in this case), double click the **Implement Design** process in the **Processes** tab. This will translate the design to something that can physically be mapped to the particular FPGA that's on our board (the xc6slx16-3csg324). You should see a green check mark if this step finishes without issues. If there are issues, you need to read them for clues about what went wrong and what you should look at to fix things. If you expand this **Implement Design** tab (which is not necessary) you will see that the **Implement Design** process actually consists of three parts:

a. **Translate**: Translate is the first step in the implementation process. The Translate process merges all of the input netlists and design constraint information and outputs a Xilinx NGD (Native Generic Database) file. The output NGD file can then be mapped to the targeted FPGA device.

b. **Map**: Mapping is the process of assigning a design's logic elements to the specific physical elements that actually implement logic functions in a device. The Map process creates an NCD (Native Circuit Description) file. The NCD file will be used by the PAR process.

c. **Place and Route** (PAR): PAR uses the NCD file created by the Map process to place and route your design. PAR outputs an NCD file that is used by the bitstream generator (BitGen) to create a (.bit) file. The Bit file (see the next step) is what's used to actually program the FPGA. In this part the actual transistor configuration and wire routing is decided.

6. **Generate:** In this step all information resulting from the previous steps are gathered and put into a Bit format that the USB programmer on the FPGA board (in our case Nexys3) understands. Simply double-click the **Generate Programming File** to generate a .bit file which will be used in the final step to program the board. Now the design is ready to be put on the board so we can physically see its functionality.
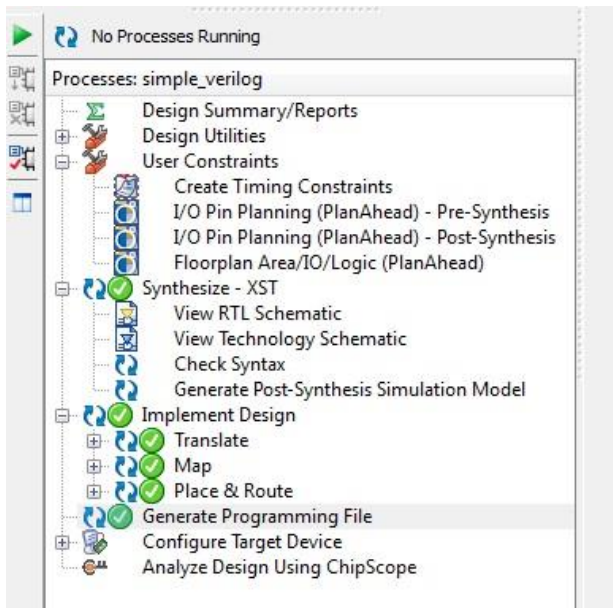

Attention: Before proceeding to the next step connect the board via the USB cable to the PC you're using, and turn on the power, the next step ensure proper connection to the board. Also note that you do not have to do steps 4, 5, and 6 individually. These processes are dependent on each other, if one needs a preceding process to be updated then it will automatically run that process before it runs itself. So you can just proceed to step 7 and watch ISE do everything.

7. **Program:** You can start the programming process by double clicking **Configure Target Device** and ISE will launch yet another Xilinx tool called **iMpact.** A warning box appears complaining about "No iMpact project file exists…", so just click OK to launch iMpact as it will automatically read your existing project.
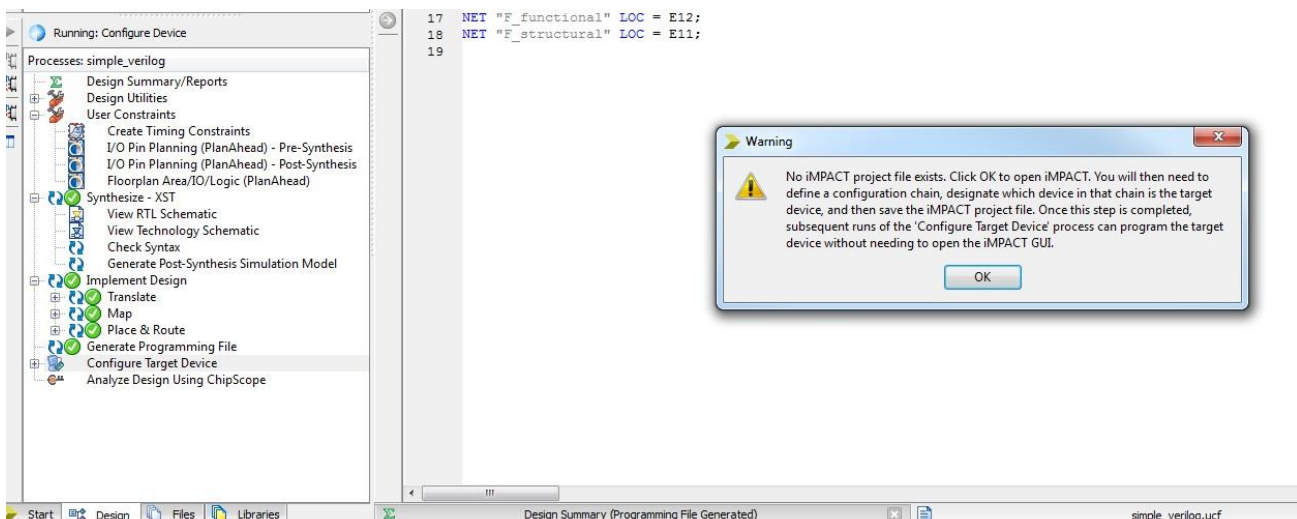
a. In the ISE iMpact window which again looks a lot like ISE, double click **Boundary Scan** in the top left pane. In the boundary scan windows in the main pane, where it says "Right click to Add Device or Initialize chain" right-click in the middle of the page and select **Initialize Chain** or just press Ctrl + I. This will ensure there is a good connection to your board and it can communicate with it.
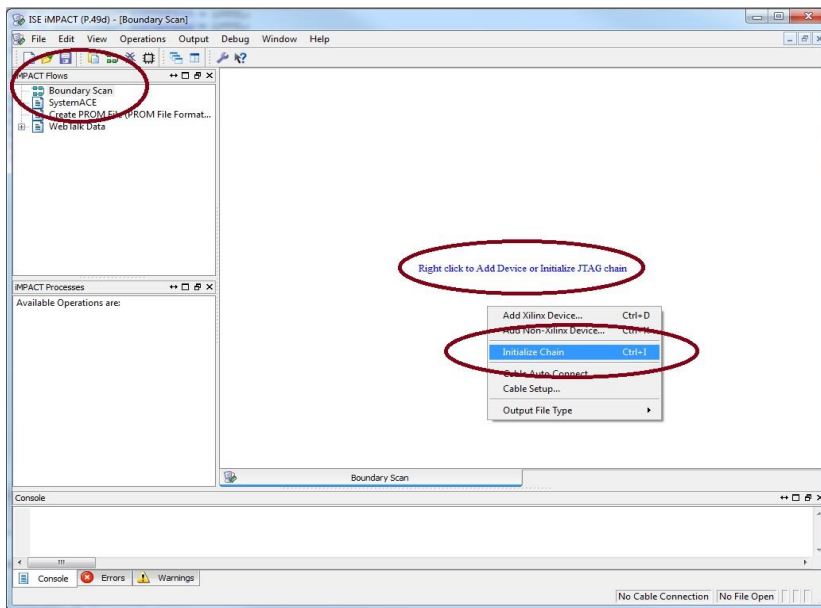

b. After iMpact verifies that the cable is connected it will prompt you to load your **.bit** file that you generated in step 6. Note that this file selection window doesn't always default to your existing project so you may need to navigate to your project folder and locate the .bit file. This file is always named to your top module so in our case it is **simple_verilog.bit.** Double-click or select the bit file and click open, again make sure it is the right file.
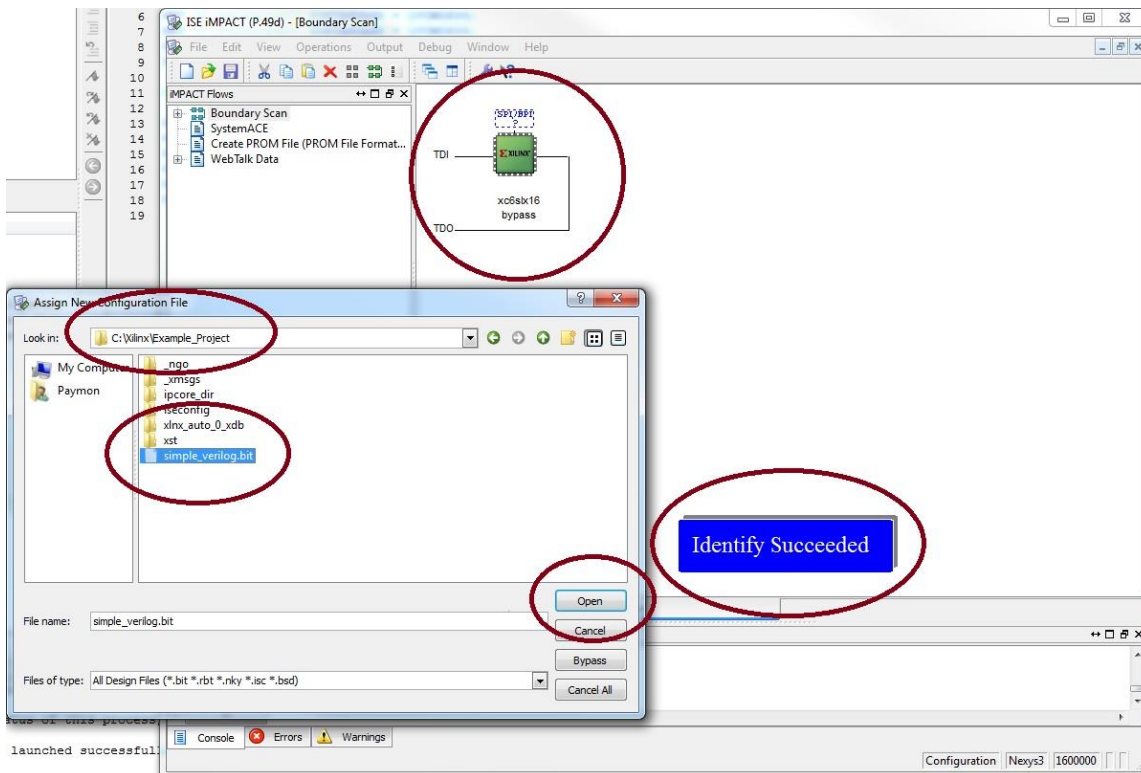
c. After the bit file is read in, iMpact prompts you to attach a PROM controller, just click NO to skip this step since we're not putting anything in the Flash memory.
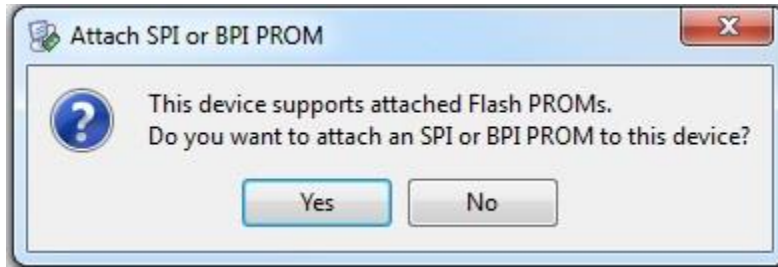
d. In the next dialog box you would be verifying which device on the board you're targeting but in our case we only have the FPGA chip to program, so click Ok and the preparation for programming the board is complete
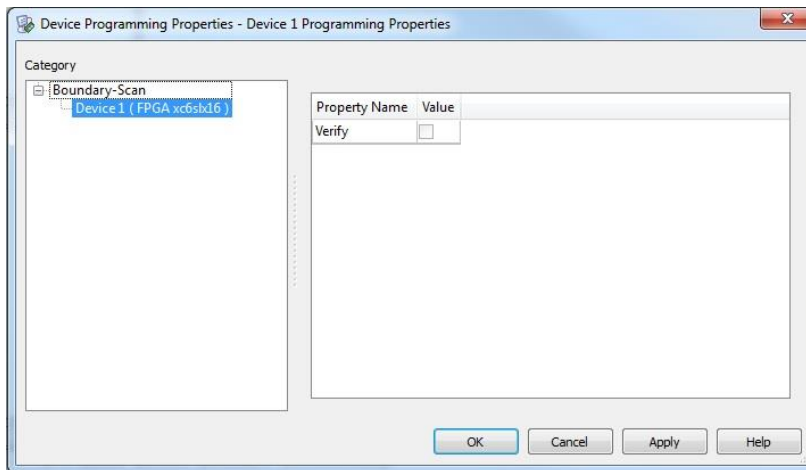
e. All that is left is to right click on the green chip icon with the Xilinx logo in the main pane and click **Program**. After the communication bar finishes, your design is programmed to the Nexys 3 FPGA board.

Original author: Prof. Brunvand
Revision and reconstruction: Paymon Saebi