

# Tutorial: Working with Verilog and the Xilinx FPGA in ISE 9.2i

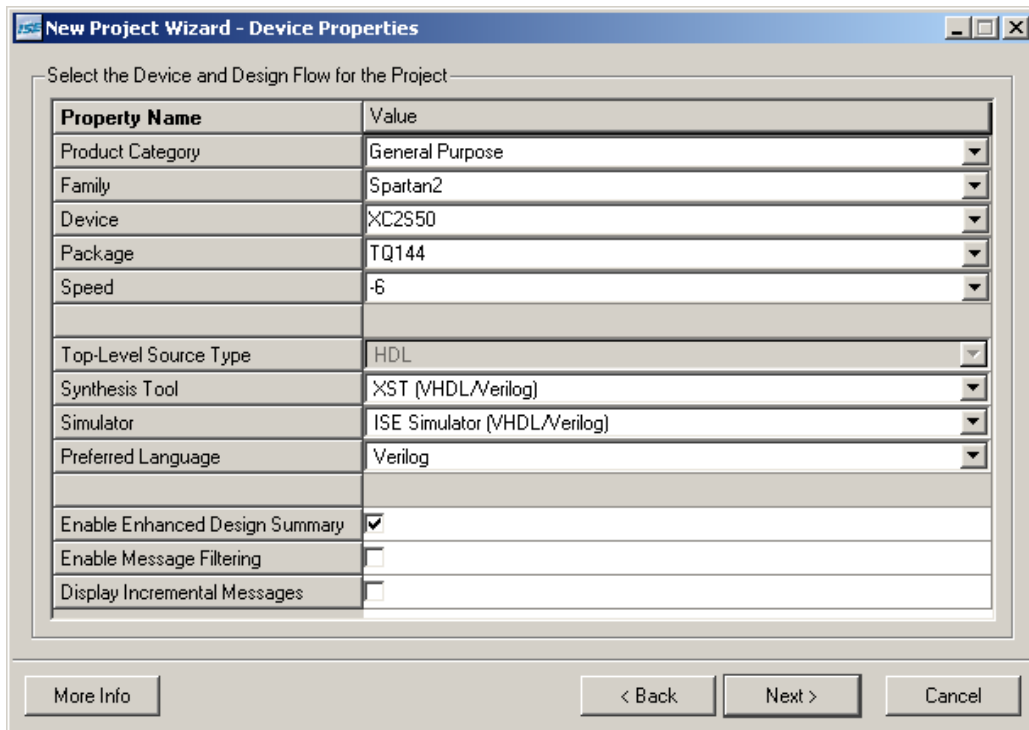
**This tutorial will show you how to:**

- Use Verilog to specify a design
- Simulate that Verilog design
- Define pin constraints for the FPGA (.ucf file)
- synthesize the design for the FPGA
- Generate a bit file
- Load that bit file onto the FPGA in your lab kit

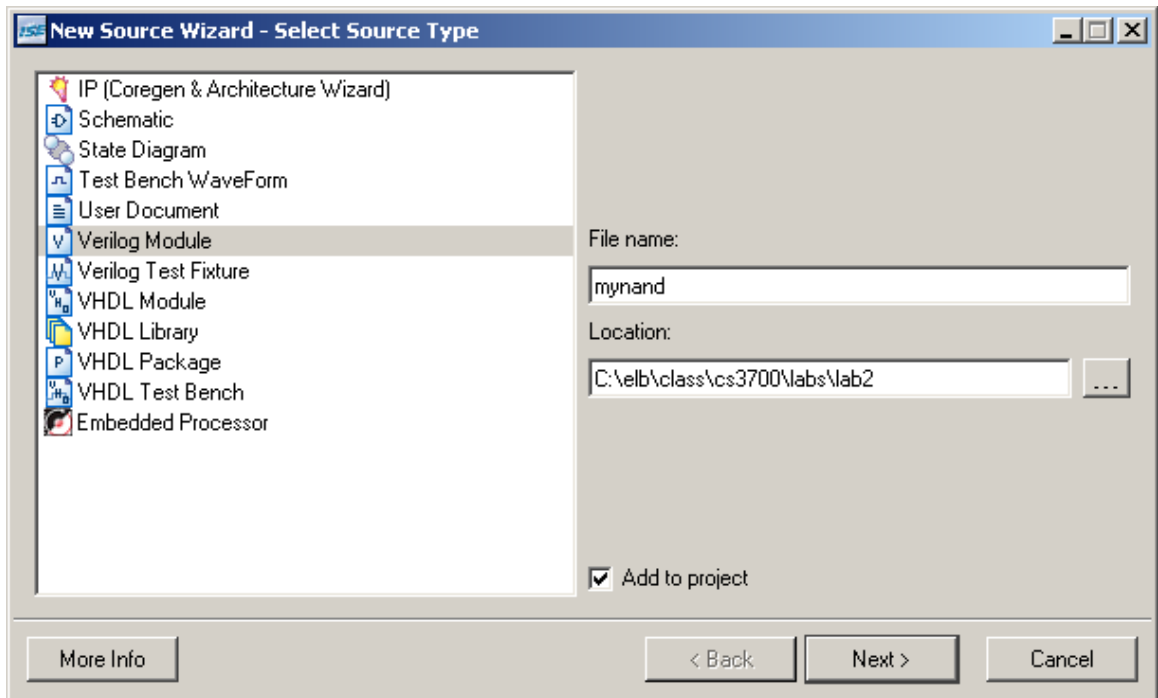
I assume that you're using a DSL lab machine, or that you've installed Xilinx ISE 9.2i on your own machine. This tutorial will use **ONLY** the XSA-50 board which is the small board with the FPGA, **NOT** the larger XST board that has the prototyping area that was used in Lab1. You will also need to load the XSTOOLS from the Xess company. More details about that later.

## Setting up a New Project and specifying a circuit in Verilog

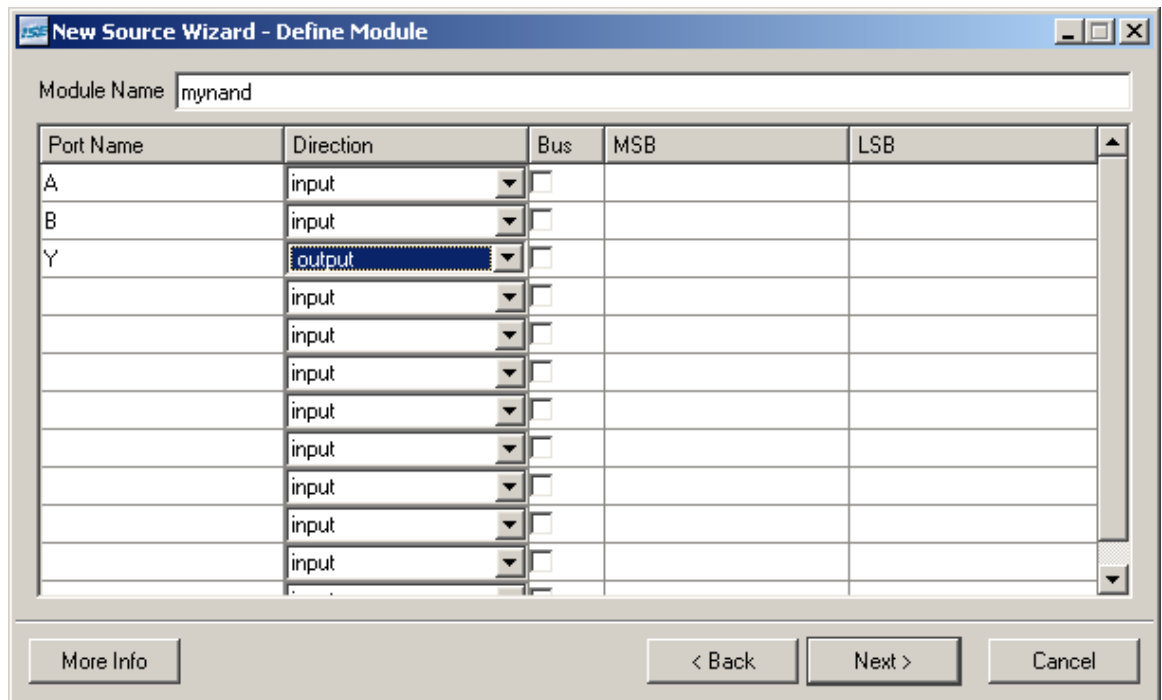
1. Create a new project like you did for Lab1. Make sure that you use the **Spartan2** Device Family, **xc2s50** Device, **tq144** Package, **-6** Speed Grade. This time you can specify **HDL** as the Top-Level Source Type with **XST** as the Synthesis Tool, **ISE** as the Simulator, and **Verilog** as the language.



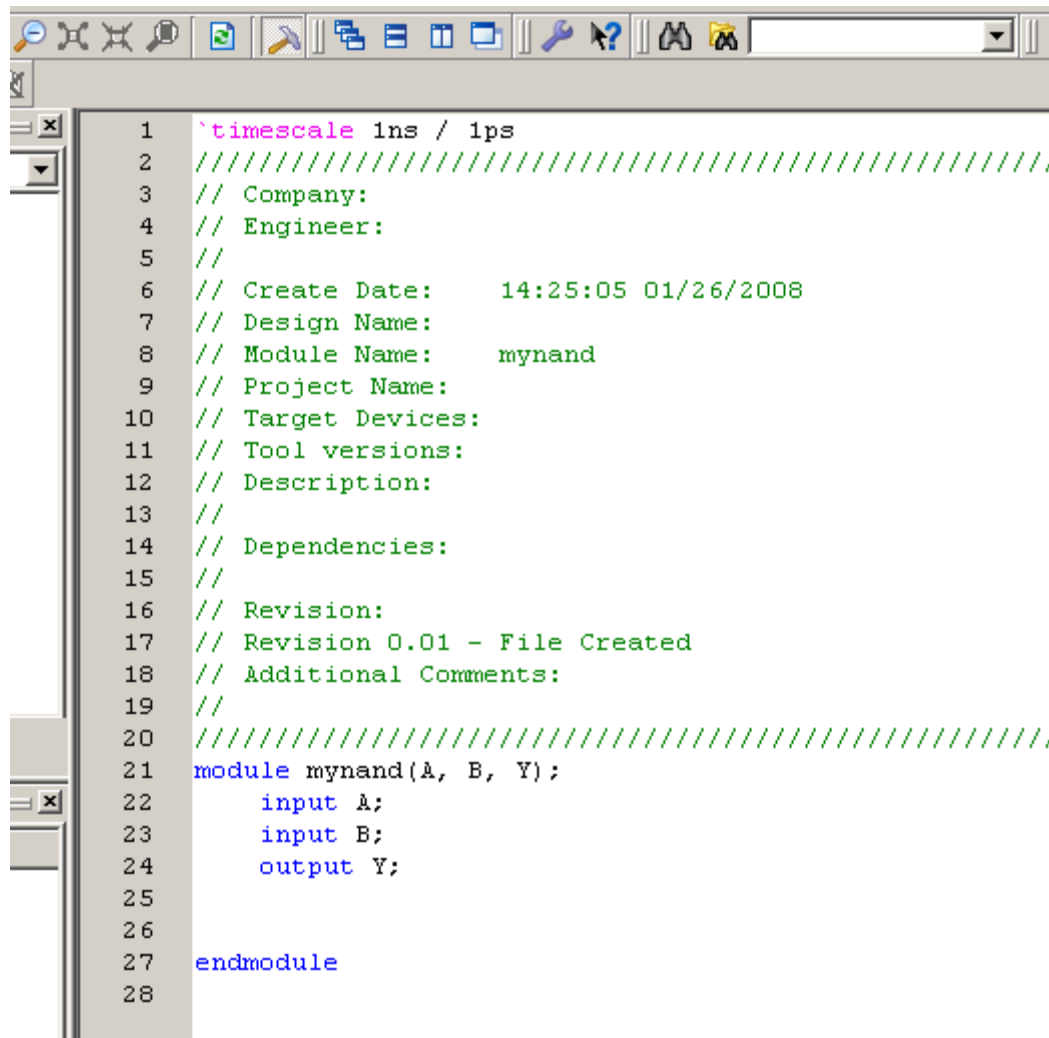
- Now you want to open a new source file, but this time you want a **Verilog Module** as your source. I'm calling my example **mynand**.



- When you press **Next** you'll get a dialog box that lets you define the inputs and outputs of your new module. I'm adding two inputs (**A** and **B**), and one output named **Y**. Remember that Verilog is case sensitive!

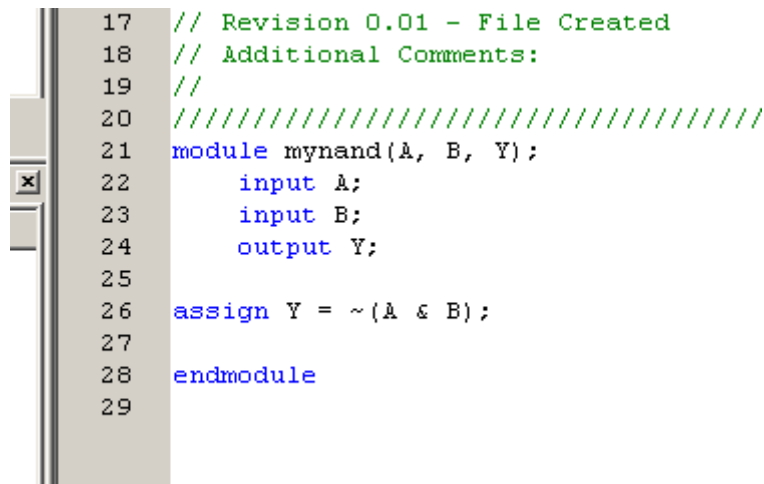


4. When you **Finish**, you'll have a template for a Verilog module that you can fill in with your Verilog code. It looks like this (note that you can also fill in the spots in the comment header with more information):



```
1  `timescale 1ns / 1ps
2  ////////////////////////////////////////////////////////////////////
3  // Company:
4  // Engineer:
5  //
6  // Create Date:    14:25:05 01/26/2008
7  // Design Name:
8  // Module Name:    mynand
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 mynand(A, B, Y);
22     input A;
23     input B;
24     output Y;
25
26
27 endmodule
28
```

5. Now you can fill in the rest of the Verilog module to implement some Boolean function. I'll implement a NAND for this example. You can use any of the Verilog techniques shown in Section 2.10 of your text. I'll use a continuous assignment statement: **assign Y = ~(A & B);** as shown below, then I'll save the file.



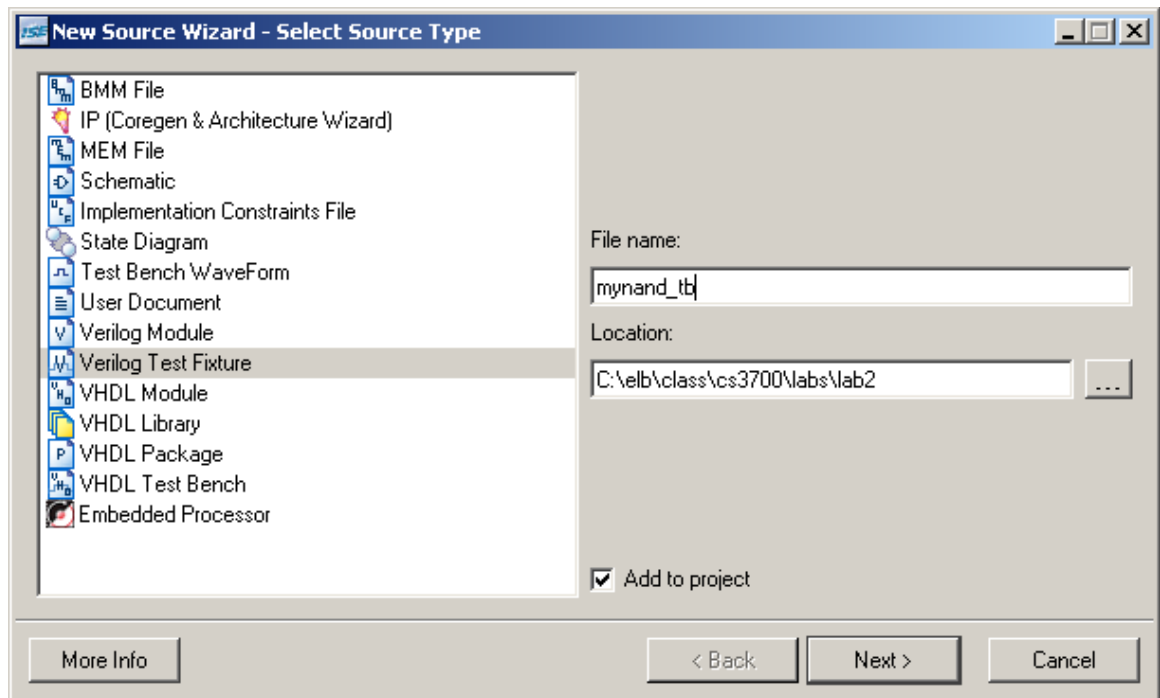
```
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 ////////////////////////////////////////////////////////////////////
21 module mynand(A, B, Y);
22     input A;
23     input B;
24     output Y;
25
26     assign Y = ~(A & B);
27
28 endmodule
29
```

## Simulating your Circuit:

- Now that you have a saved Verilog source file, you can simulate its behavior. We'll use the **ISE simulator** with a testbench just like in Lab1.

To simulate the (Verilog) circuit:

1. Go to the top left pane and change the **Sources For:** field to be **Behavioral Simulation**.
2. This changes the options in the bottom left pane to the simulator options. In that pane select **Create New Source**. This will bring up the **New Source Wizard**. In that dialog type in the name of your testbench file, and make sure to select **Verilog Test Fixture** in the list on the left. I will name my testbench **mynand\_tb** (where the tb stands for testbench). The box looks like:



3. The **Next** dialog asks you which source you want the testbench constructed from. I'll choose **mynand**, of course. The code that gets generated looks a lot like the code from the previous tutorial, but it is actually a little nicer. Note that it already has an initial block with the inputs (**A** and **B** in this case) already initialized to 0.

```
21 // Additional Comments:
22 //
23 ///////////////////////////////////////////////////////////////////,
24
25 module mynand_tb_v;
26
27     // Inputs
28     reg A;
29     reg B;
30
31     // Outputs
32     wire Y;
33
34     // Instantiate the Unit Under Test (UUT)
35     mynand uut (
36         .A(A),
37         .B(B),
38         .Y(Y)
39     );
40
41     initial begin
42         // Initialize Inputs
43         A = 0;
44         B = 0;
45
46         // Wait 100 ns for global reset to finish
47         #100;
48
49         // Add stimulus here
50
51     end
52
53 endmodule
54
```

I'm going to add some slightly more interesting testbench code for this example. I'm going to test all four possible values of the inputs, but I'm also going to use a **\$display** statement to display information about the simulation, and use an **if** statement to have the testbench check if I'm getting the right answers for me. This is known as a **self-checking testbench** and is by far the best way to test things! It's much easier to have the testbench alert you when things are failing than to have to stare at the timing diagram to see if it's behaving properly.

The **\$display** statement works very much like a C printf statement. It simply prints data to the simulation console. The **%b** says to print the Verilog variable as binary data. You could also use **%d** for decimal, **%h** for hex, etc. The portion of the testbench with my new testbench code looks like:

```

40
41 initial begin
42     // Initialize Inputs
43     A = 0;
44     B = 0;
45
46     // Wait 100 ns for global reset to finish
47     #100;
48
49     // Add stimulus here
50     $display("AB = %b%b, Y = %b", A, B, Y);
51     if (Y != 1) $display("ERROR - Y is %b, should be 1", Y);
52
53     A = 1;
54     #20 $display("AB = %b%b, Y = %b", A, B, Y);
55     if (Y != 1) $display("ERROR - Y is %b, should be 1", Y);
56
57     B = 1;
58     A = 0;
59     #20 $display("AB = %b%b, Y = %b", A, B, Y);
60     if (Y != 1) $display("ERROR - Y is %b, should be 1", Y);
61
62     A = 1;
63     #20 $display("AB = %b%b, Y = %b", A, B, Y);
64     if (Y != 0) $display("ERROR - Y is %b, should be 0", Y);
65
66
67 end

```

As an alternative, you could have Verilog generate the expected answer for you. For example, your check statements could be:

```
if (Y != ~(A & B)) $display("ERROR - Y is %b, should be %b", Y, ~(A & B));
```

Another type of testbench that is very useful uses loops to generate the data. Here's another testbench that uses for loops to generate all possible inputs to the circuit. Note that the loop variables have to be type integer. There is an interesting question of what happens when you assign an integer (which by default is a 32-bit integer) to a single-bit Verilog variable. What happens is that Verilog casts the integer into the single-bit type. Essentially it takes the low-order bit of the integer and puts that into the Verilog variable. Here's the code. Note that the integers for the loop variables need to be defined outside of the **initial** block, and that there needs to be a delay inside the loop too so that the changes you make to the inputs have time to propagate to the outputs of the circuit. The delay before the inner loop's **\$display** statement allows the inputs to propagate through the circuit before you display their values.

```

integer i,j;
initial begin

// Initialize Inputs
A = 0;
B = 0;

// Wait 100 ns for global reset to finish
#100;

// Add stimulus here

for (i=0; i<2; i=i+1)
  for (j=0; j<2; j=j+1)
    begin
      A = i;
      B = j;
      #20 $display("AB = %b%b, Y = %b", A, B, Y);
      if (Y != ~(A & B))
        $display("ERROR - Y is %b, should be %b", Y, ~(A&B));
    end

end

```

4. You can **Check Syntax** on the code (whichever version you use) to make sure things are correct Verilog. Once you have something that passes the syntax check you can double-click the **Simulate Behavioral Model** to fire up the simulator. The output will be displayed as waveforms, and the **\$display** data will show up in the console as shown (after zooming out to see all the waveforms). You can see that not only do the waveforms show the results of the simulation, but the **\$display** statements have printed data, and because the circuit is correctly functioning, no error statements were printed.



Xilinx - ISE - C:\elb\class\cs3700\labs\lab2\lab2.ise - [Simulation]

File Edit View Project Source Process Test Bench Simulation Window Help

Sources for: Behavioral Simulation

- lab2
  - xc2s50-6tq144
    - my NAND tb\_v (my NAND tb.v)

Processes for: my NAND tb\_v

- Add Existing Source
- Create New Source
- Xilinx ISE Simulator
- Check Syntax
- Simulate Behavioral Model

Current Simulation Time: 1000 ns

Signal	Value
Y	0
A	1
B	1

AB = 00, Y = 1  
AB = 10, Y = 1  
AB = 01, Y = 1  
AB = 11, Y = 0  
%

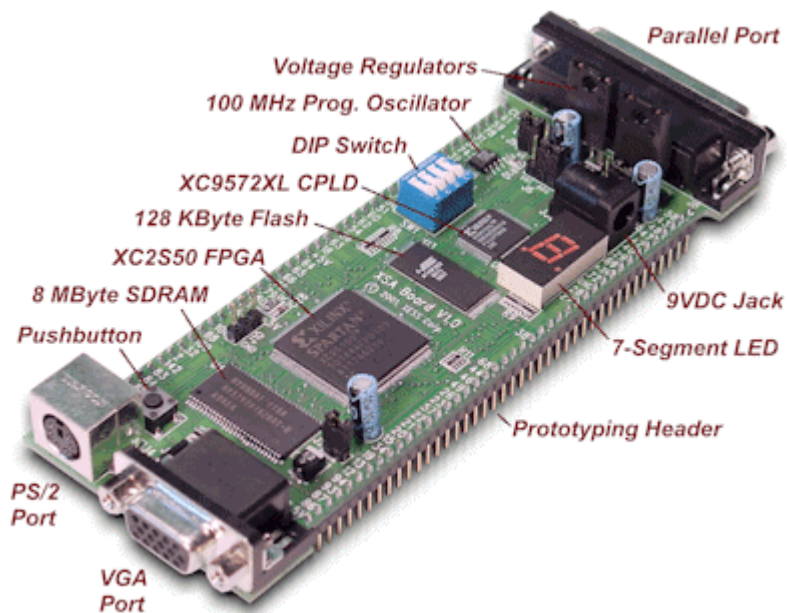
Console Errors Warnings Tcl Shell Find in Files Sim Console - my NAND tb\_v

The screenshot displays the Xilinx ISE simulation environment. The top toolbar includes standard file and simulation controls. The 'Sources' window shows a project named 'lab2' containing a device 'xc2s50-6tq144' and a testbench 'my NAND tb\_v'. The 'Processes' window lists actions like 'Simulate Behavioral Model'. The central window shows a timing diagram for signals Y, A, and B, with a current simulation time of 1000 ns. The console window at the bottom shows the testbench output for various input combinations of A and B.

# Synthesizing your circuit to the Xilinx FPGA

Now that you have a correctly simulating Verilog module, you will have the ISE (webPACK) tool synthesize your Verilog 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.

You will use your XSA-50 board for this part of the tutorial. This is the small board with the parallel port and the FPGA chip, NOT the extender board you used for Lab1. For this tutorial we'll connect the **A** and **B** inputs of the **myand** circuit to two switches on the blue DIP (Dual In-line Package) switch, and send the output **Y** to the decimal point (DP) on the 7-segment display. That way you can switch the DIP switches to provide inputs and see the output as the dot on the display.



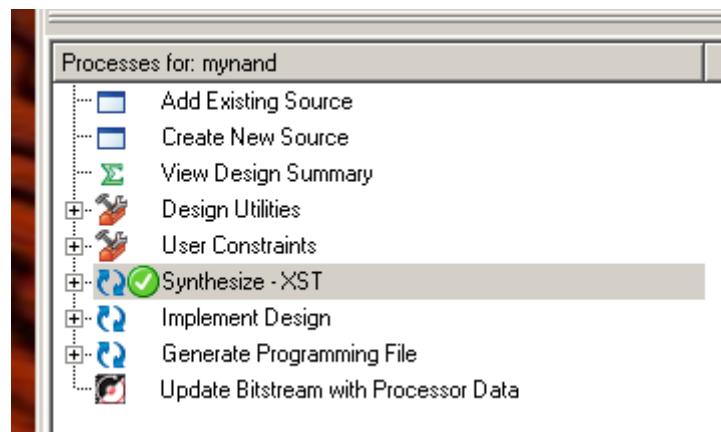
In order to do this we need to:

- Synthesize the Verilog code into FPGA configuration
- Assign A, B, and Y to the correct pins on the FPGA that connect to the switches and LED
- Generate a programming file with all this information (.bit file)
- Use the tools from Xess (the company who makes the XSA board) to use the .bit file to configure the FPGA.

1. Back in the **Source** pane, return to the **Synthesis/Implementation** view and select your **myrand.v** verilog module. Now in the bottom (**Processes**) pane you will see some new options including **Synthesize – XST**. Double click on this to synthesize your circuit. After a while you will (hopefully) get the “**Process ‘Synthesize’ completed successfully**” message in the console. If you’ve already simulated your Verilog and found it to do what you want, there’s every chance that this will synthesize correctly without problems.

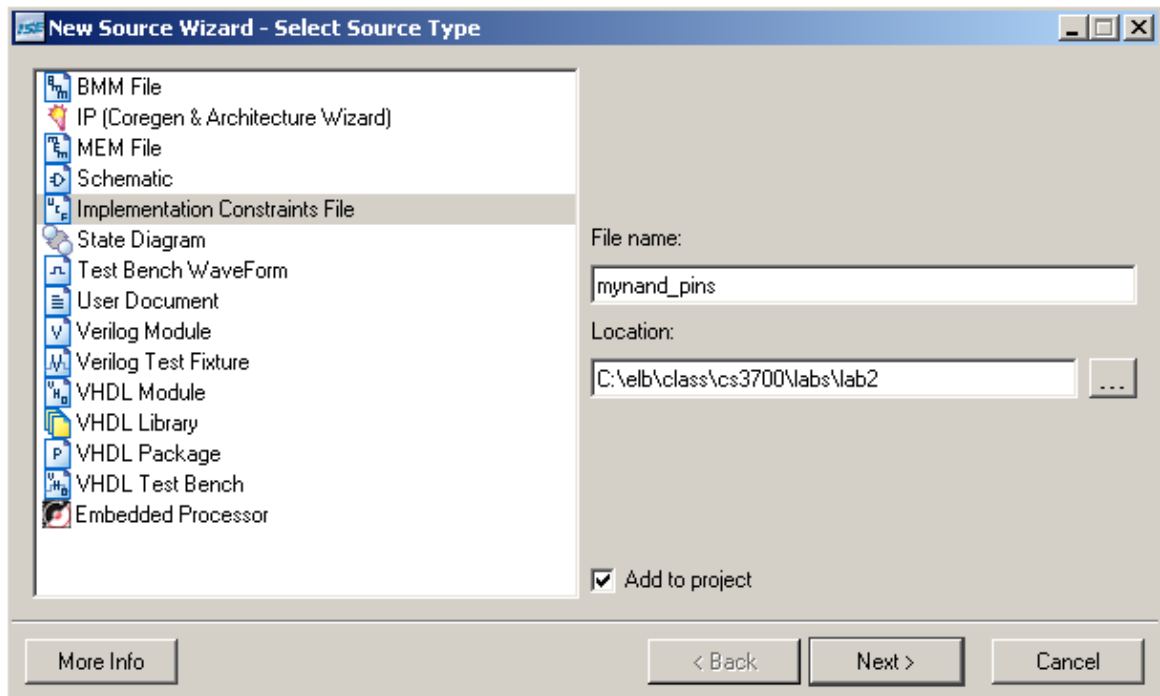
In any case, there is lots of interesting information in the **synthesis report** (the data in the console window). It’s worth looking at, although for this amazingly simple example there isn’t anything that fascinating.

Make sure that you end the process with a green check for this process. If you get something else, especially a red X, you’ll need to fix errors and re-synthesize.

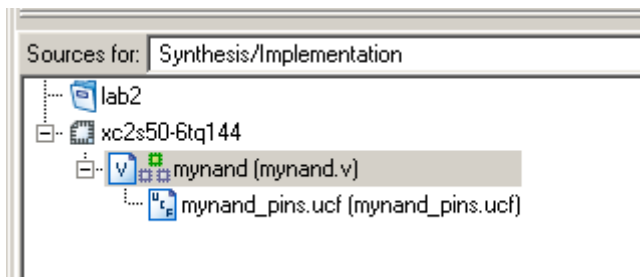


2. Now, because we’re headed towards putting this on the Xilinx FPGA on the XSA board, we need to set some constraints. In particular, we need to tell ISE which pins on the Xilinx chip we want **A**, **B**, and **Y** assigned to so that we can access those from switches and LEDs on the XSA board.

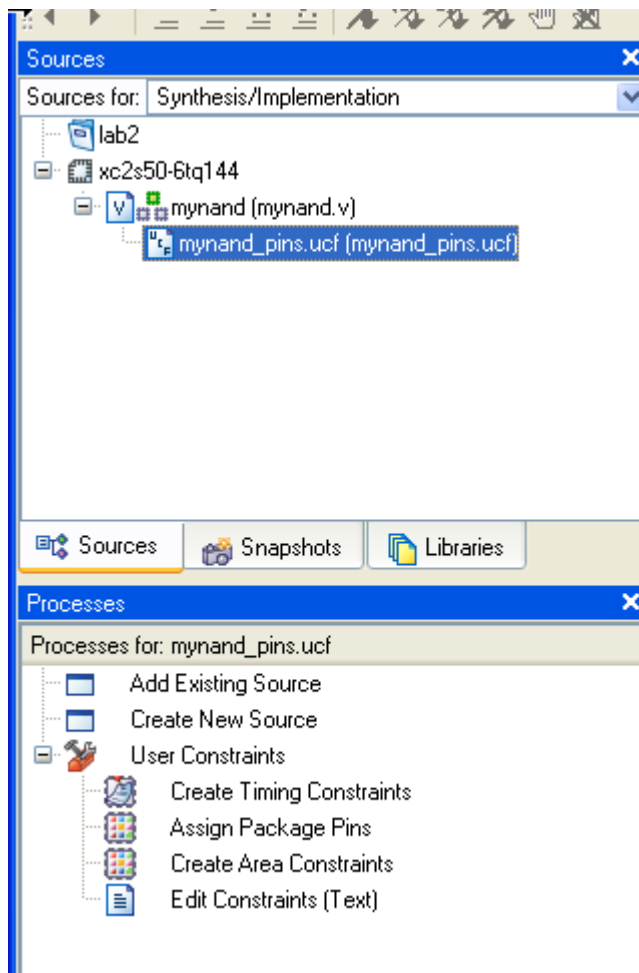
Create a **New Source**, but this time choose **Implementation Constraints File** as the type. I’m calling mine **myrand\_pins**.



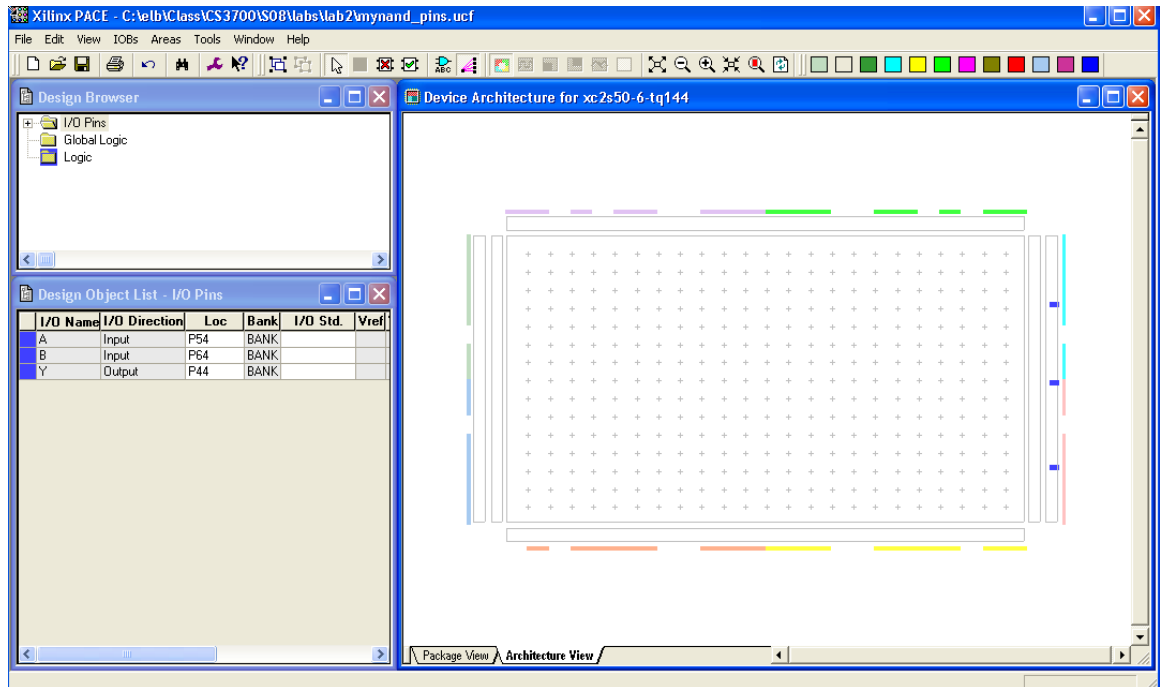
3. After you **Next** and **Finish** (make sure it's associated with the **mynand.v** Verilog file), the **mynand\_pins.ucf** file should show up nested under the **mynand.v** file in the **Sources** pane. The ucf stands for Universal Constraints File.



4. Select the ucf file. This changes the available processes in the **Processes** tab. There are all sorts of constraints (timing, placement, pins, etc.) that can be specified here. For now, we're just going to bind the **A**, **B**, and **Y** signals to specific pins on the Xilinx part. Expand the **User Constraints** tab and double click **Assign Package Pins**.



5. This will open up the Xilinx **PACE** tool which will let you assign which pins on the Xilinx FPGA each of your I/O signals should be connected to. But which pins? For that you need to look at the XSA board documentation on the class web site, or the XSA pin spreadsheet also on the class web site. There are four DIP switches, one pushbutton switch, and a 7-segment LED on the XSA-50 board that we can use for inputs and outputs .For now, I'll tell you that the pins we're interested in are:
  - a. **DIPSW 1A**: Pin 54 on the Xilinx part (DIP switch #1)
  - b. **DIPSW 1B**: Pin 64 on the Xilinx part (DIP switch #2)
  - c. **LED-DP** (decimal point on the 7-segment display): Pin 44 on Xilinx
  
6. In the **PACE** editor, enter **P54**, **P64**, and **P44** in the **Loc** field for the I/O signals as shown. This will tell the implementation process to make sure that those signals are connected to those pins. The XSA-50 documentation tells us that those pins are switches and LEDs. Save the ucf file in **PACE** using the **XST default convention** for bus signaling (if that dialog box comes up). Then exit and go back to the ISE window.

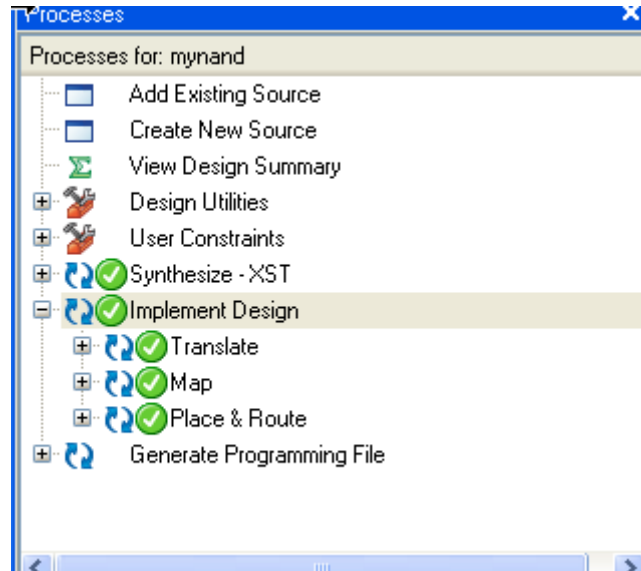


7. With your source file selected (**myrand.v** in this case), double click the **Implement Design** process in the **Processes** tab. This will translate the design to something that can physically be mapped to the particular FPGA that's on our board (the xc2S50-6tq144). You should see a green check 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.

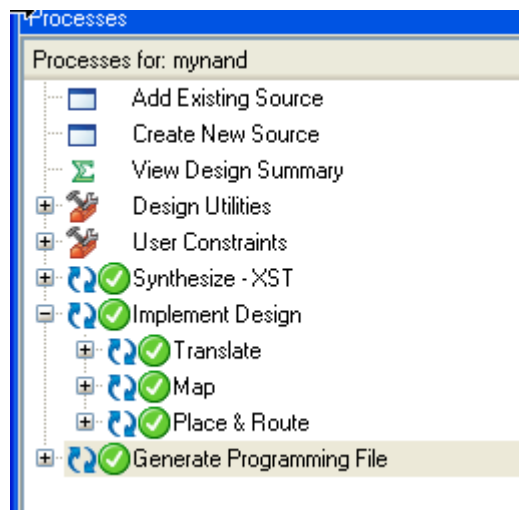
If you expand this **Implement Design** tab (which is not necessary) you will see that the **Implement Design** process actually consists of three parts:

- 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.



8. At this point you can look at the **Design Summary** to find out all sorts of things about your circuit. One thing that you might want to check is to click on the **Pinout Report** and check that your signals were correctly assigned to the pins you wanted them to be assigned to.
9. Now double click the final process: **Generate Programming File**. This will generate the actual configuration bits into a .bit file that you can use to program your XSA-50 board to behave like your circuit (in this case a simple 2-input NAND).



10. If everything has worked and shows green checks to this point, you can quit ISE. You now have a **.bit** file (**myrand.bit** in this case) in your lab folder that you can use to program your XSA-50. But, in order to program the board you need to use tools from **Xess**, the maker of the board.

# Configuring the XSA-50 Board

To configure the XSA-50 board with your **.bit** file you need the **XSTOOLS** software from **Xess** loaded on your computer (it's already loaded on the DSL computers). You can get the software from the Xess web site. The direct link is:

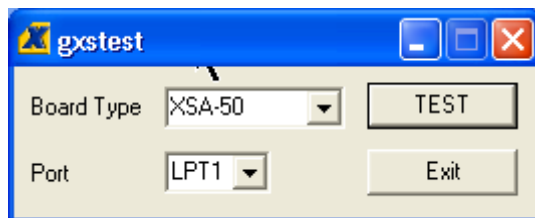
[http://www.xess.com/downloads/setup-XSTOOLS-5\\_1\\_0.exe](http://www.xess.com/downloads/setup-XSTOOLS-5_1_0.exe)

The XSTOOLS package contains the following tools:

- **GXSTEST**: This utility lets you test an XSA Board for proper functioning.
- **GXSSETCLK**: This utility lets you set the clock frequency of the programmable oscillator on an XSA Board.
- **GXSLOAD**: This utility lets you download FPGA and CPLD configuration files and upload/download data files to the RAM and/or Flash on an XSA Board.
- **GXSPORT**: This utility lets you send logic inputs to an XSA Board by toggling the data pins of the PC parallel port.

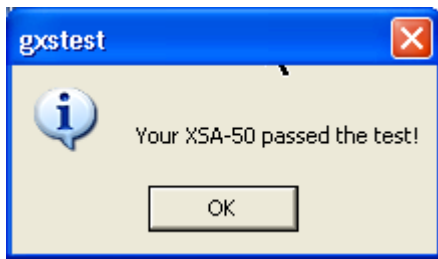
To program your circuit to the XSA-50 board take the following steps:

1. First run **GXSTEST** to make sure your board is working. After installing the GXSTOOLS, start the **GXSTEST** program. Make sure to select the **XSA-50** as your board type. Also make sure your board is connected to the PC through the parallel cable, and that you have connected power to your board using the power supply in your lab kit. Make sure that you don't set your board down on a conductive surface when you plug in the power. You don't want to short out any of the pins on the bottom.

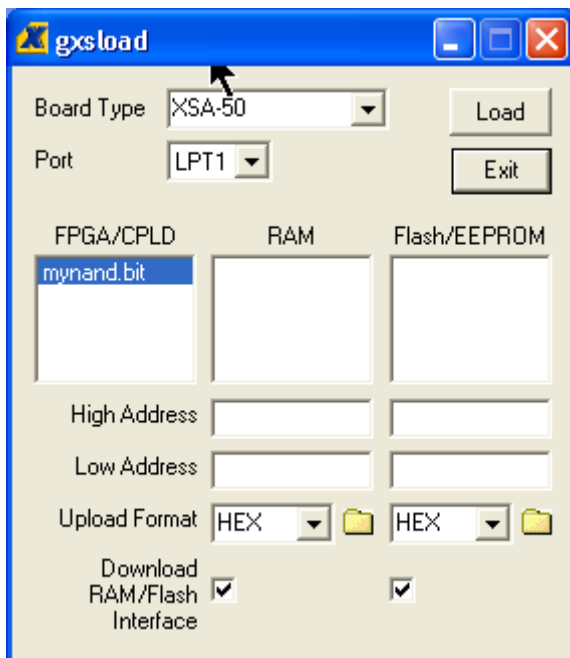


2. After running this program (it takes a minute or so as it downloads the test configuration to the board), you should see a 0 on the 7-segment display and a message that the board has passed the test. If your board doesn't pass the test, please check with Travis in the DSL and have him run further tests.





- Once you've verified that your board is working, you can load your **.bit** file to the XSA-50 board that will configure the FPGA to behave as your circuit. Use the **GXSLOAD** program. Once the window opens, click and drag your **.bit** file (**mynand.bit** in this case) to the **FPGA/CPLD** window as shown, and press **Load** to upload your **.bit** file to the XSA-50 board.



- Assuming you don't get any errors in the Load process, your XSA-50 board is now configured according to your **.bit** file. In this example we used a simple NAND function with the **A** and **B** inputs assigned to DIP switches 1 and 2. The output **Y** was assigned to the decimal point (DP) on the 7 segment display. So, if everything is working properly, the DP should be lit when either or both of switches 1 and 2 are ON, and should be off only when both switch 1 and 2 are OFF. This is, of course, just the truth table for a 2-input NAND:

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

# Overview of the Procedure

1. **Design** the circuit that you would like to map to the Xilinx part on the FPGA. You can use schematics, or Verilog, or a mixture of both (we'll see details of that later).
2. **Simulate** your circuit using the ISE Simulator and a Verilog testbench to provide inputs to the circuit. Use "if" statements in your testbench to make it self-checking.
3. Generate a **UCF** file to hold constraints such as pin assignments (later we'll use the UCF file for other constraints like timing and speed).
4. **Assign** the I/O pins in your design to the pins on the FPGA that you want them connected to.
5. **Synthesize** the design for the FPGA using the XST synthesis tool.
6. **Implement** the design to map it to the specific FPGA on the XSA-50 board
7. Generate the **programming .bit file** that has the bitstream that configures the FPGA.
8. **Test** your XSA-50 board using GXSTEST.
9. **Upload** your .bit file to the XSA-50 board using GXSLOAD.