XESS Corporation

Introduction to WebPACK 6.1

Using Xilinx WebPACK Software to Create FPGA Designs for the XSA Board

Release date: 10/30/2003

Table of Contents

What This Is and *Is Not* 1

FPGA Programming 3

Installing WebPACK 5

- Getting WebPACK 5
- Installing WebPACK 6
- Getting XSTOOLs 6
- Installing XSTOOLs 7

Our First Design 8

- An LED Decoder 8
- Starting WebPACK Project Navigator 10
- Describing Your Design With VHDL 21
- Checking the VHDL Syntax 28
- Fixing VHDL Errors 29
- Synthesizing the Logic circuitry for Your Design 32
- Implementing the Logic Circuitry in the FPGA 33
- Checking the Implementation 35
- Assigning Pins with Constraints 36
- Viewing the Chip 45
- Generating the Bitstream 53
- Downloading the Bitstream 55
- Testing the Circuit 57

Hierarchical Design 59

- A Displayable Counter 59
- Starting a New Design 60
- Adding the LED Decoder ..... **Error! Bookmark not defined.**

Adding a Counter 66
Tying Them Together 72
Constraining the Design 97
Synthesizing the Logic Circuitry for the Design 103
Implementing the Logic Circuitry in the FPGA 104
Checking the Implementation 104
Checking the Timing 106
Generating the Bitstream 107
Downloading the Bitstream 112
Testing the Circuit 120
Going Further… 121

0

What This Is and *Is Not*

There are numerous requests on newgroups that go something like this:

"I am new to using programmable logic like FPGAs and CPLDs. How do I start? Is there a tutorial and some free tools I can use to learn more?"

Xilinx has released their WebPACK on the web so that anyone can download a free set of tools for CPLD and FPGA-based logic designs. And XESS Corp. has written this tutorial that attempts to give you a gentle introduction to using the WebPACK tools. (Other programmable logic manufacturers have also released free toolsets. Someone else will have to write a tutorial for them.)

This tutorial shows the use of the WebPACK tools on two simple design examples: 1) an LED decoder and 2) a counter which displays its current value on a seven-segment LED. Along the way, you will see:

- How to start an FPGA project.
- How to target a design to a particular type of FPGA.
- How to describe a logic circuit using VHDL and/or schematics.
- How to detect and fix VHDL syntactical errors.
- How to synthesize a netlist from a circuit description.
- How to fit the netlist into an FPGA.
- How to check device utilization and timing for an FPGA.
- How to generate a bitstream for an FPGA.
- How to download a bitstream to program an FPGA.
- How to test the programmed FPGA.

That said, it is important to say what this tutorial will not teach you:

- It will not teach you how to design logic with VHDL.
- It will not teach you how to choose the best type of FPGA or CPLD for your design.
- It will not teach you how to arrange your logic for the most efficient use of the resources in an FPGA.
- It will not teach you what to do if your design doesn't fit in a particular FPGA.
- It will not show you every feature of the WebPACK software and discuss how to set every option and property.
- It will not show you how to use the variety of peripheral devices available on the XSA-50 Board.

In short, this is just a tutorial to get you started using the Xilinx WebPACK FPGA tools. After you go through this tutorial you should be able to move on to more advanced topics.

1

FPGA Programming

Implementing a logic design with an FPGA usually consists of the following steps (depicted in the figure which follows):

1. You enter a description of your logic circuit using a *hardware description language* (HDL) such as VHDL or Verilog. You can also draw your design using a schematic editor.

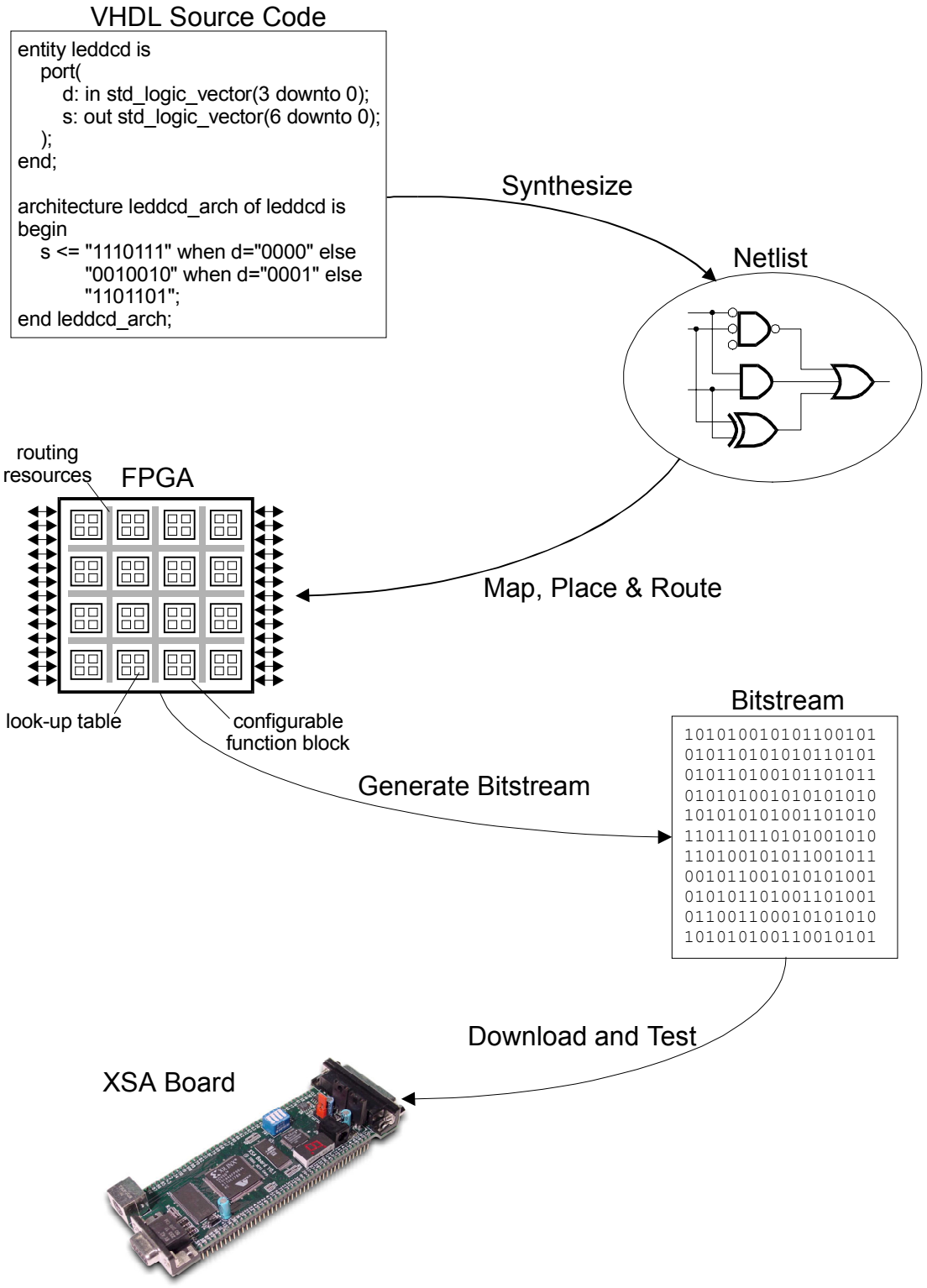
2. You use a *logic synthesizer* program to transform the HDL or schematic into a *netlist*. The netlist is just a description of the various logic gates in your design and how they are interconnected.

3. You use the *implementation tools* to map the logic gates and interconnections into the FPGA. The FPGA consists of many *configurable logic blocks* which can be further decomposed into *look-up tables* that perform logic operations. The CLBs and LUTs are interwoven with various *routing resources*. The mapping tool collects your netlist gates into groups that fit into the LUTs and then the place & route tool assigns the gate collections to specific CLBs while opening or closing the switches in the routing matrices to connect the gates together.

4. Once the implementation phase is complete, a program extracts the state of the switches in the routing matrices and generates a *bitstream* where the ones and zeroes correspond to open or closed switches. (This is a bit of a simplification, but it will serve for the purposes of this tutorial.)

5. The bitstream is *downloaded* into a physical FPGA chip (usually embedded in some larger system). The electronic switches in the FPGA open or close in response to the binary bits in the bitstream. Upon completion of the downloading, the FPGA will perform the operations specified by your HDL code or schematic.

That's really all there is to it. Xilinx WebPACK provides the HDL and schematic editors, logic synthesizer, fitter, and bitstream generator software. The XSTOOLs from XESS provide utilities for downloading the bitstream into an XSA-50 Board containing a Xilinx XC2S50 SpartanIIE FPGA.

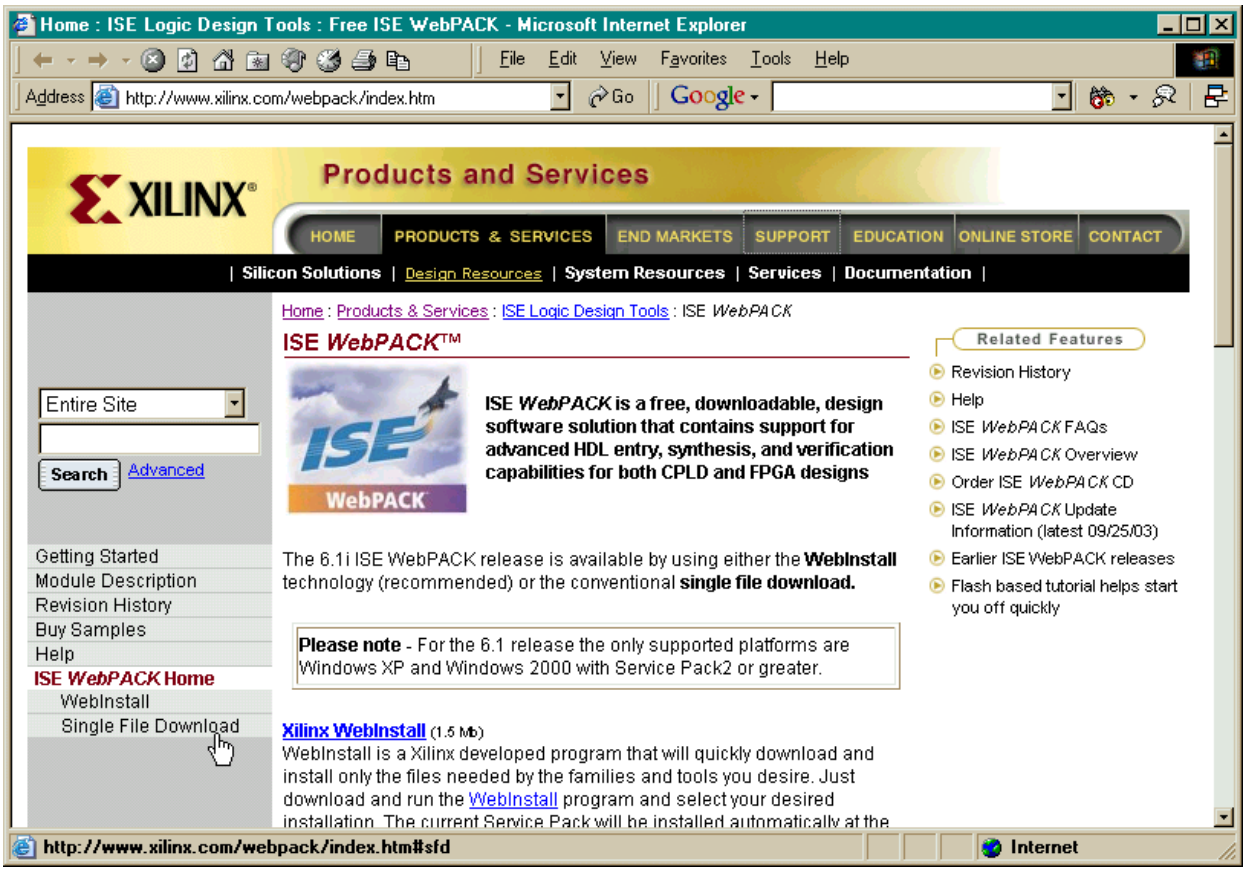

VHDL Source Code
entity leddcd is
port(
d: in std_logic_vector(3 downto 0);
s: out std_logic_vector(6 downto 0);
);
end;
architecture leddcd_arch of leddcd is
begin
s <= "1110111" when d="0000" else
"0010010" when d="0001" else
"1101101";
end leddcd_arch;
Synthesize
Netlist
routing resources
FPGA
Map, Place & Route
look-up table
configurable function block
Bitstream
101010010101100101
010110101010110101
010110100101101011
010101001010101010
101010101001101010
110110110101001010
110100101011001011
001011001010101001
010101101001101001
011001100010101010
101010100110010101
Generate Bitstream
Download and Test
XSA Board

2

Installing WebPACK

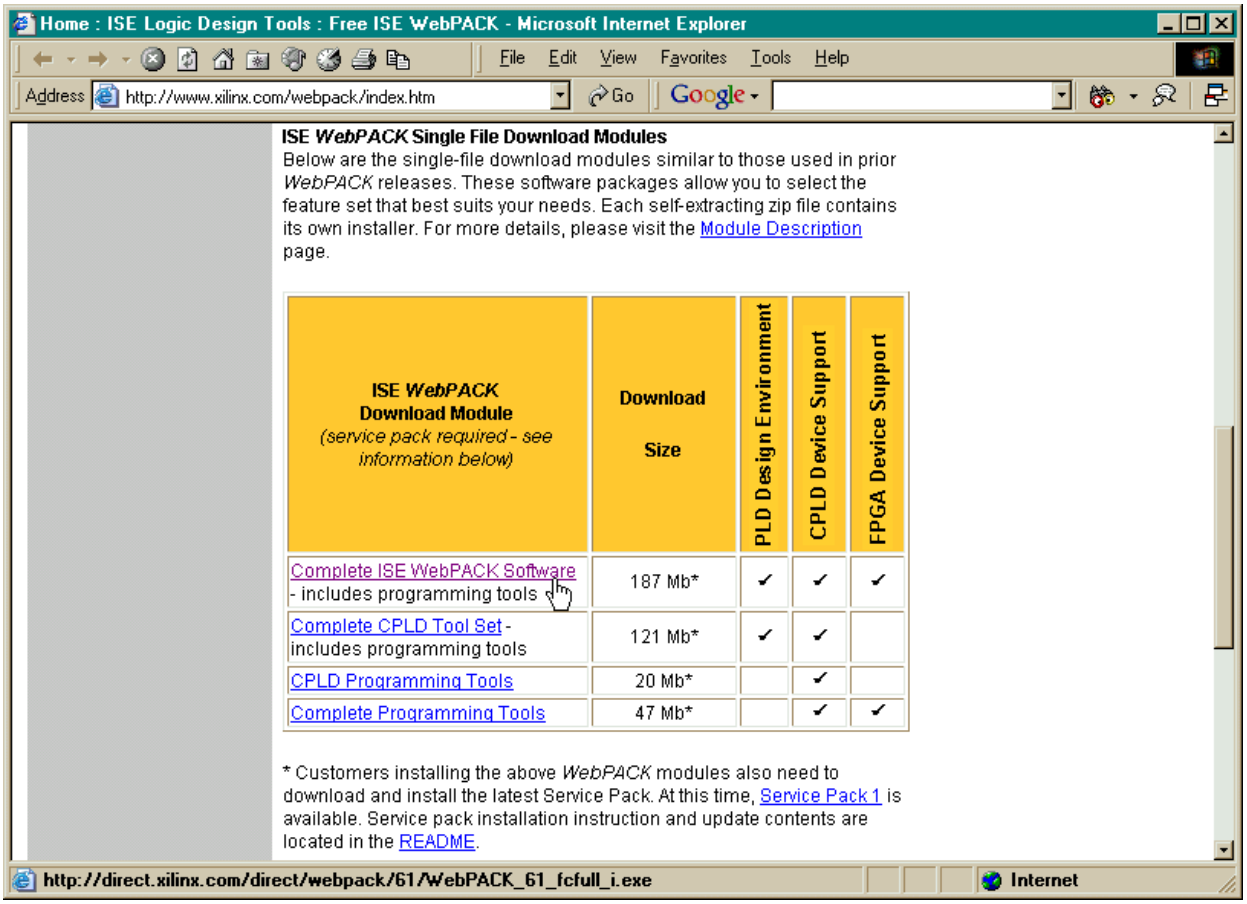
Getting WebPACK

Before downloading the WebPACK software you will have to create an account. You will choose a user ID and password and then you will be allowed to enter the site. Then you can go to http://www.xilinx.com/webpack/index.htm to begin downloading the WebPACK software. After entering the WebPACK homepage, click on the Single File Download link as shown below.



Next, click on the Complete ISE WebPACK Software link. This will initiate the download of all the WebPACK software modules that cover both FPGA and CPLD designs. After this

download completes, you also need to download Service Pack 1 to get the most current WebPACK updates.



Installing WebPACK

After the WebPACK software download completes, double-click the WebPACK_61_fcfull_i.exe file. The installation script will run and install the software. Accept the default settings for everything and you shouldn't have any problems. Then repeat this procedure with the Service Pack 1 install file 6_1_01i_pc.exe.

Getting XSTOOLs

If you are going to download your FPGA bitstreams into an XSA-50 Board, then you will need to get the XSTOOLS software from http://www.xess.com/ho07000.html. Just download the xstools4_0_3.exe file.

Installing XSTOOLs

Double-click the xstools4_0_3.exe file. The installation script will run and install the software. Accept the default settings for everything.

3

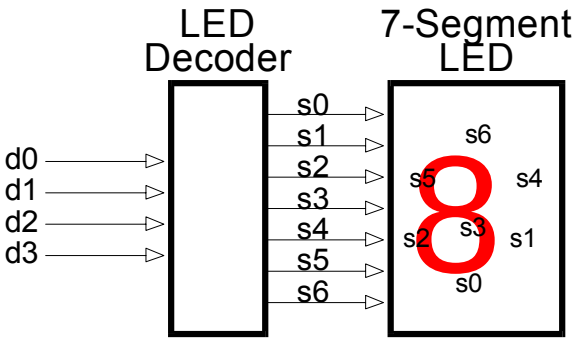
Our First Design

An LED Decoder

The first FPGA design we will try is an LED decoder. An LED decoder takes a four-bit input and outputs seven signals which drive the segments of an LED digit. The LED segments will be driven to display the digit corresponding to the hexadecimal value of the four input bits as follows:

Four-bit Input	Hex Digit	LED Display
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	A
1011	B	B
1100	C	C
1101	D	D
1110	E	E
1111	F	F

A high-level diagram of the LED decoder looks like this:



Starting WebPACK Project Navigator

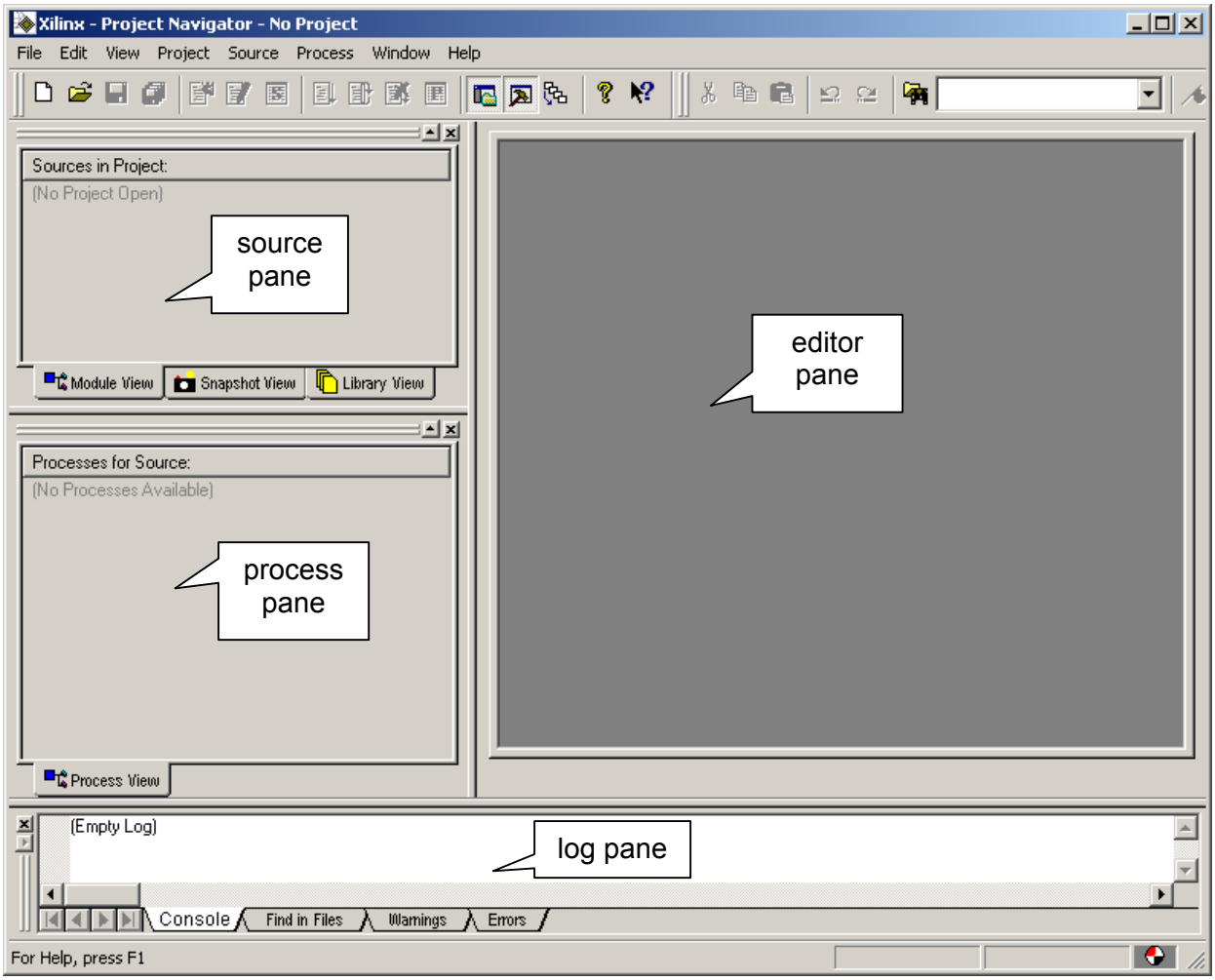
We start WebPACK by double-clicking the icon, ,on the desktop. This will bring up an empty project window as shown below. The window has four panes:

1. A **source pane** that shows the organization of the source files that make up our design. There are four tabs so we can view the source files, functional modules, or HDL libraries for our project or look at various snapshots of the project.

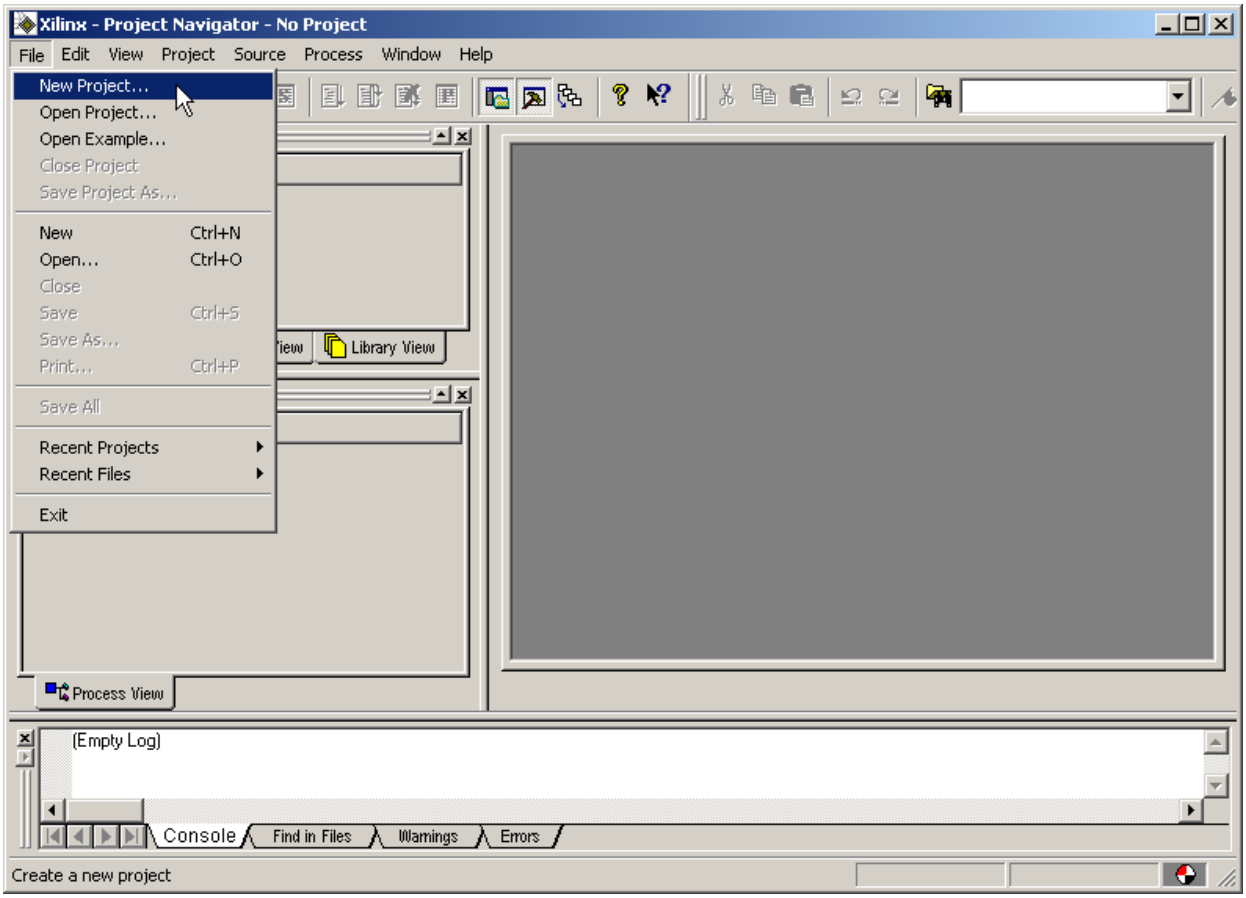
2. A **process pane** that lists the various operations we can perform on a given object in the source pane.

3. A **log pane** that displays the various messages from the currently running process.

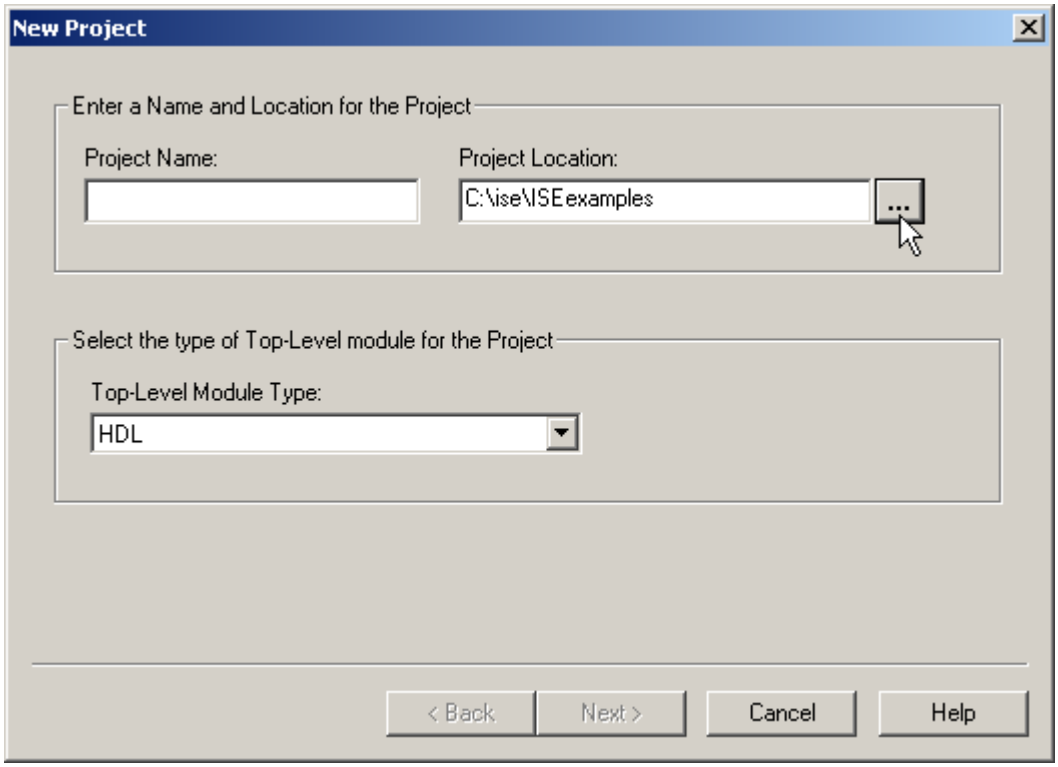
4. An **editor pane** where we can enter HDL code. Schematics are entered in a separate window.



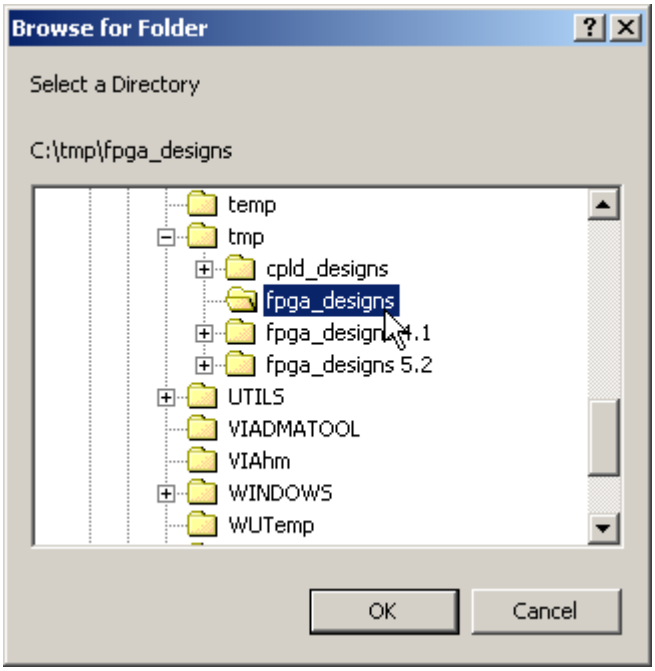
To start our design, we must create a new project by selecting the File➔New Project item from the menu bar.



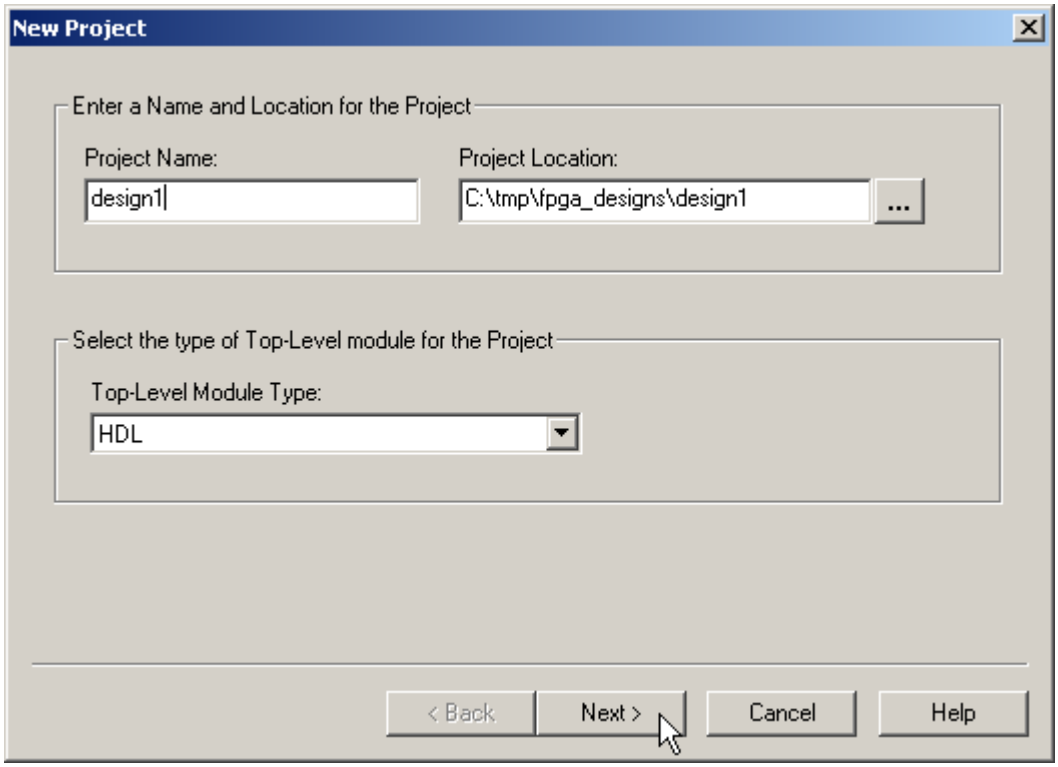
This brings up the **New Project** window where we can enter the location of our project files, project name, the target device for this design, and the tools used to synthesize logic from our source files.



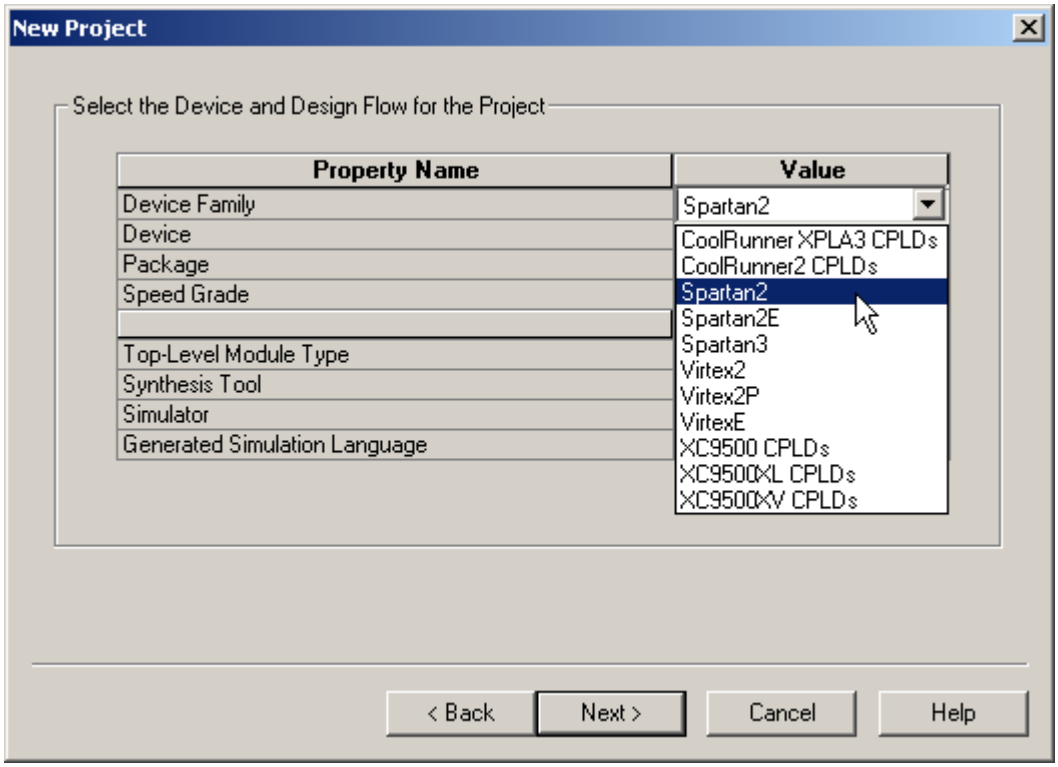
Click on the **…** button next to the Project Location field and use the **Browse for Folder** window to select a folder where our project files will be stored. For our design examples, we will store everything in the C:\tmp\fpga_designs folder. Click on the OK button ater highlighting this folder.



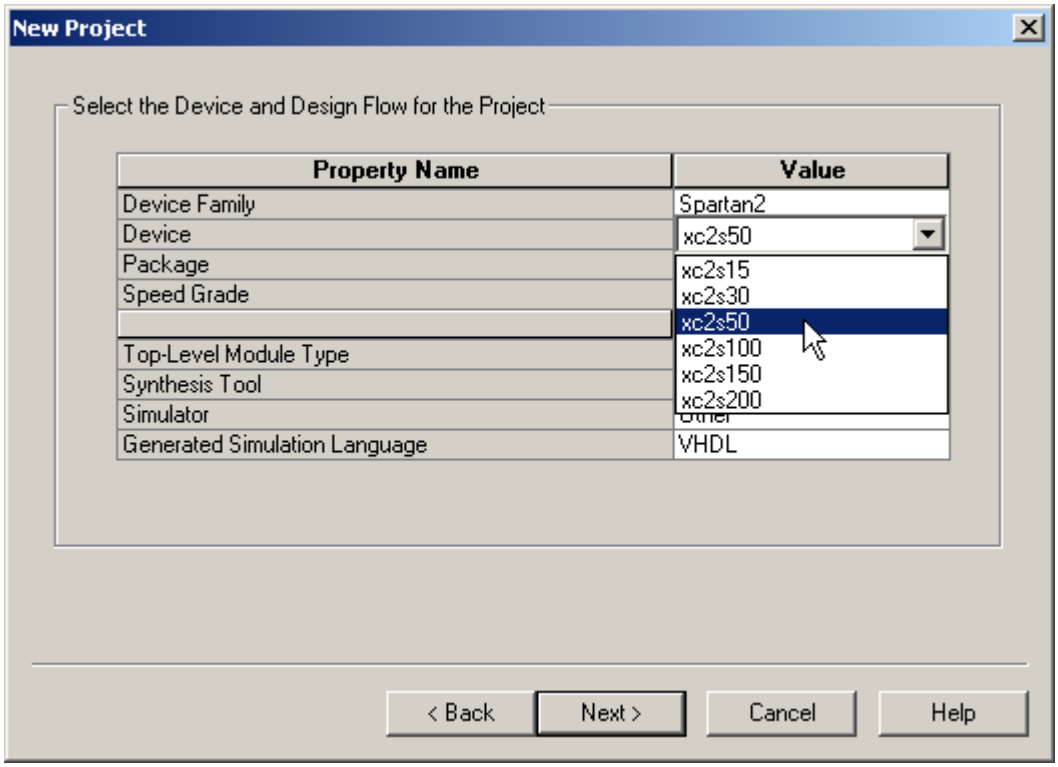
Next we will give our LED decoder design the descriptive title of `design1` by typing it into the Project name field. Then we click on the Next button to continue creating this project.



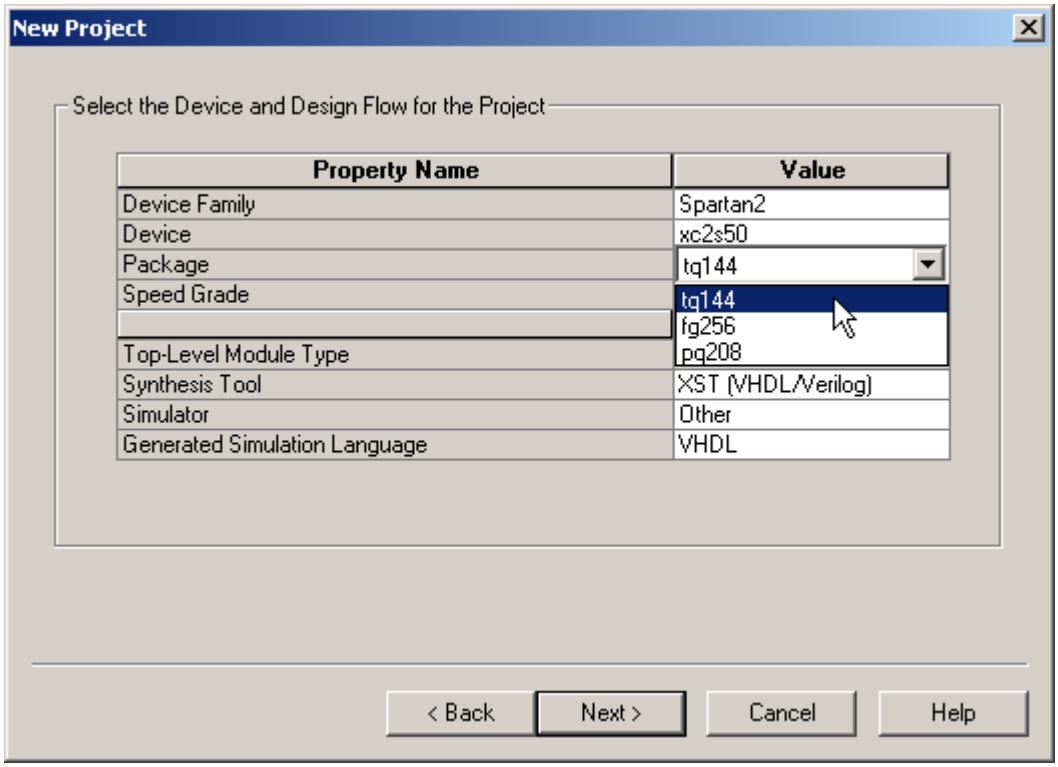
To set the family of FPGA devices we will target with this design, click in the Value field of the Device Family property. Select the Spartan2 entry in the drop-down menu that appears.



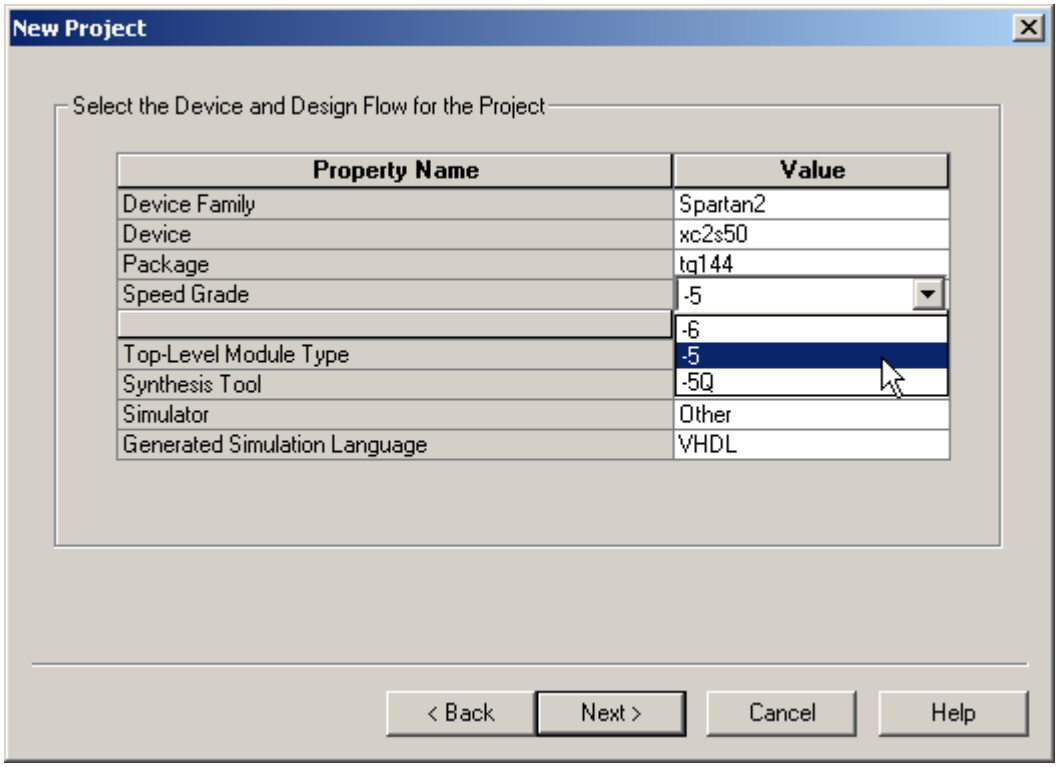
Then click in the Value field of the Device property to select a particular device within the device family. For our designs, we will select the XC2S50 since this is the device used in the XSA-50 Board where we will test our design.



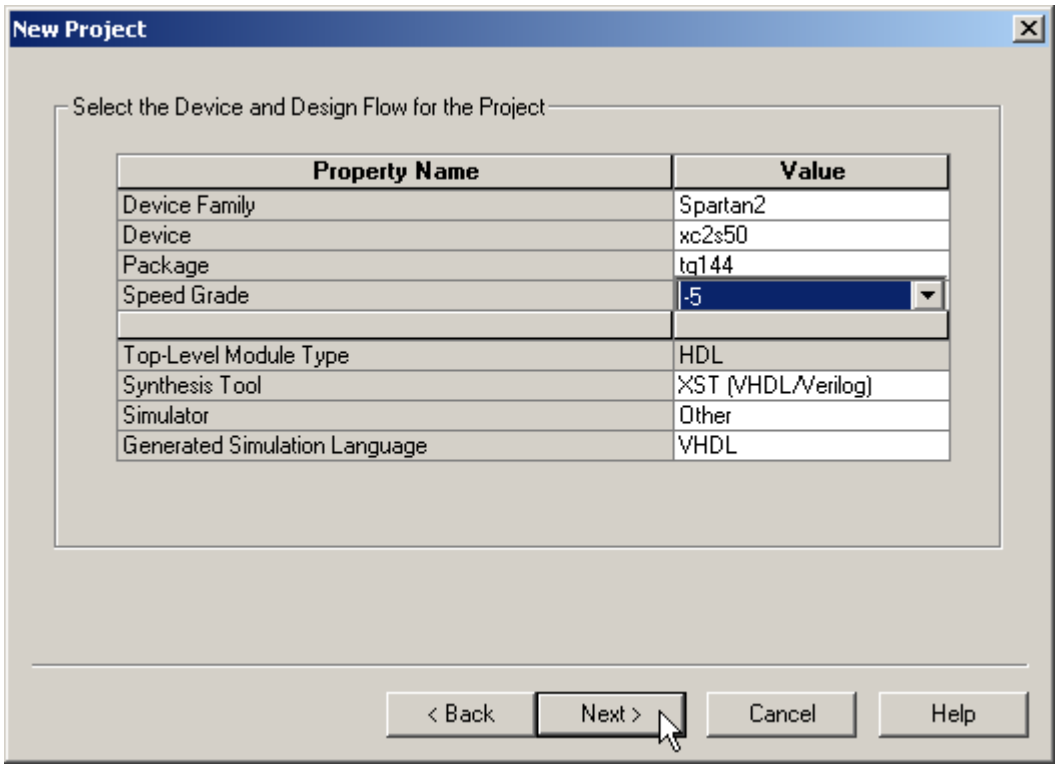
Now click in the Value field of the Package property and choose the TQ144 package style for the FPGA on the XSA-50 Board.



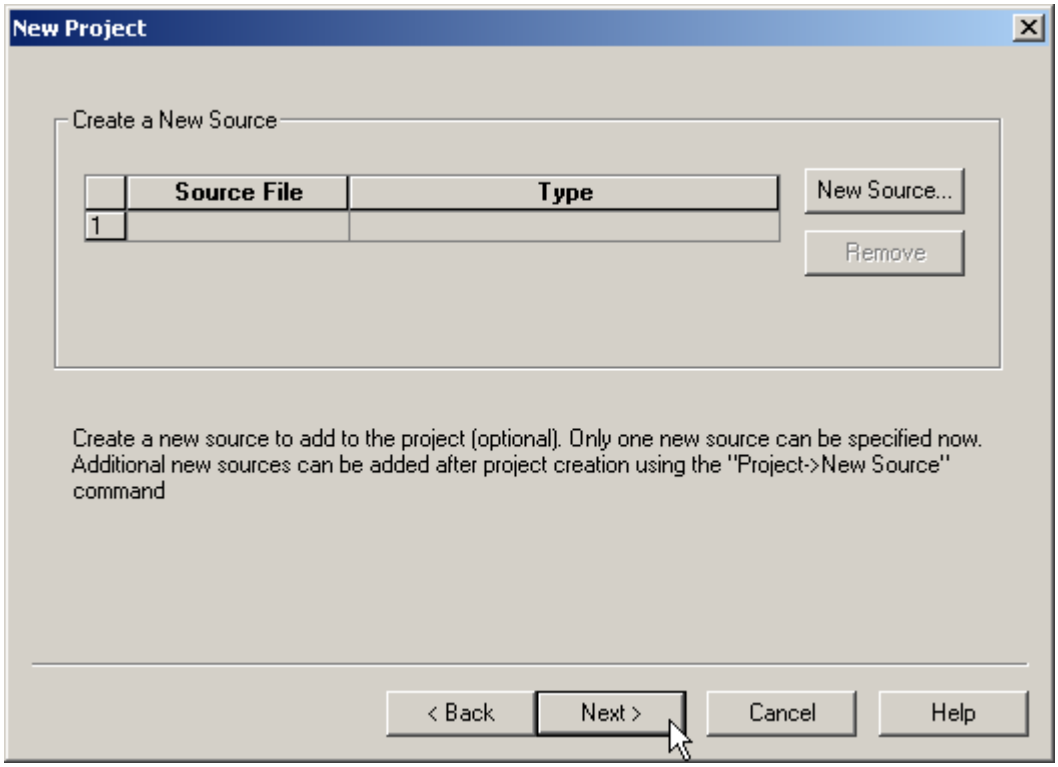
Then set the speed grade property for the FPGA to -5.



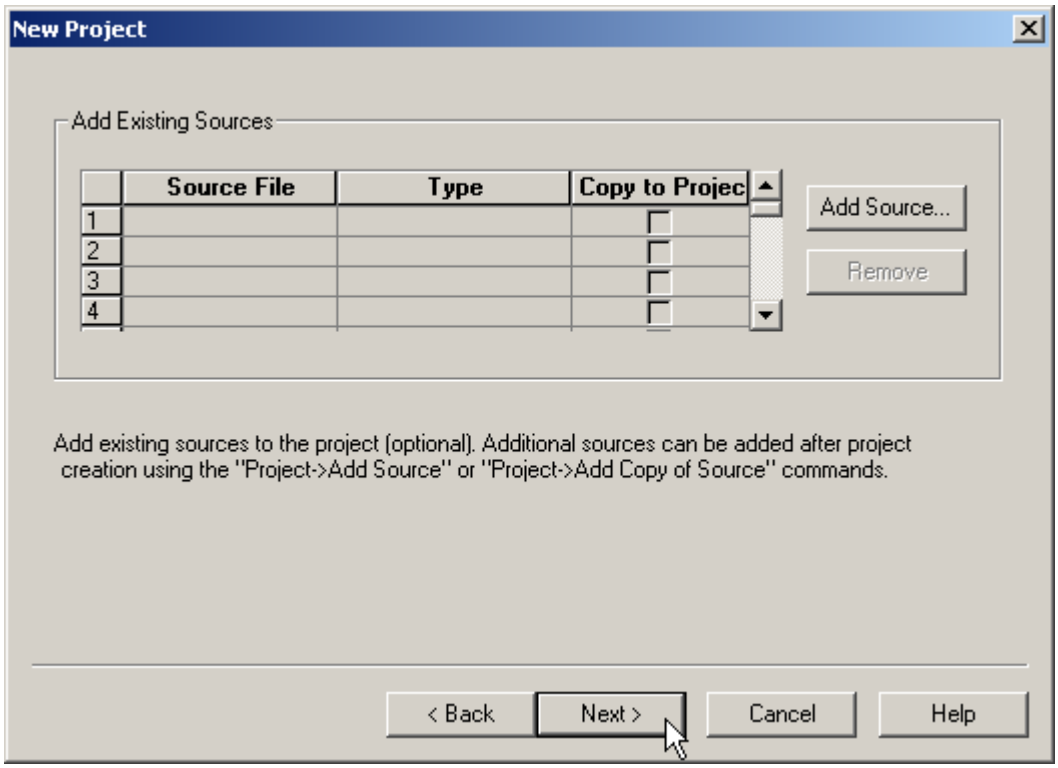
The Top-Level Module Type, Synthesis Tool, Simulator and Generated Simulation Language can all be left at their default values, so we can just click on the Next button to continue creating the project.



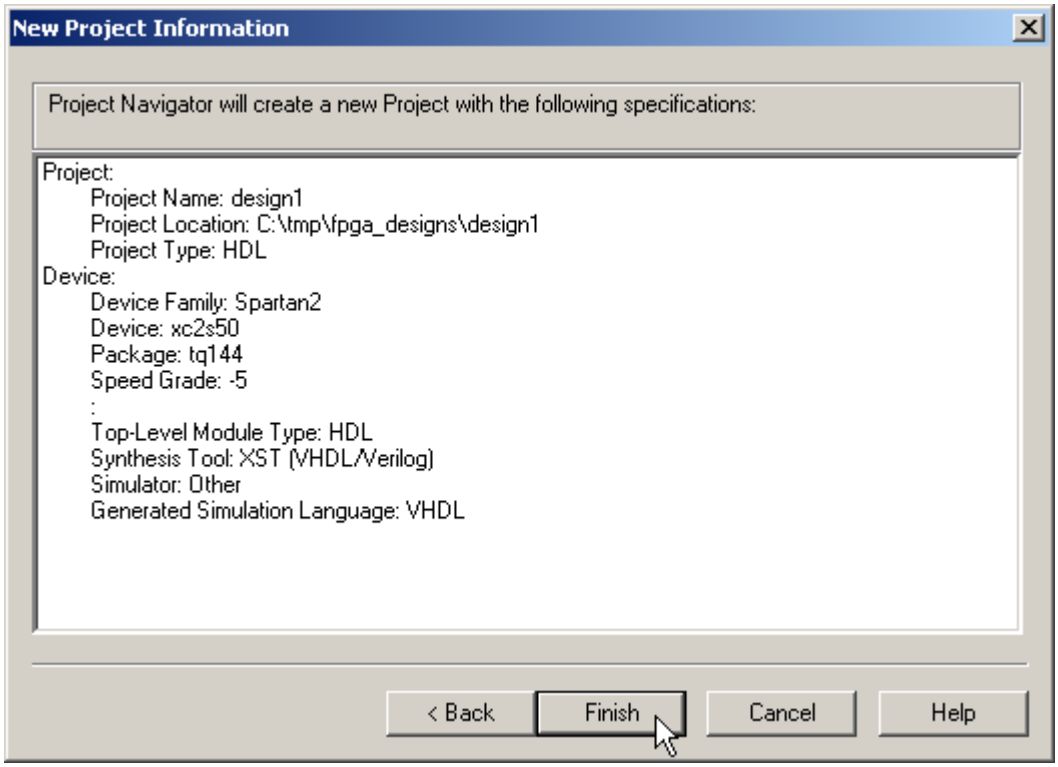
Click the Next button on the next window that appears. (We will create the VHDL source code at a later step.)



We have no existing source files to add to this project, so once again click the Next button.

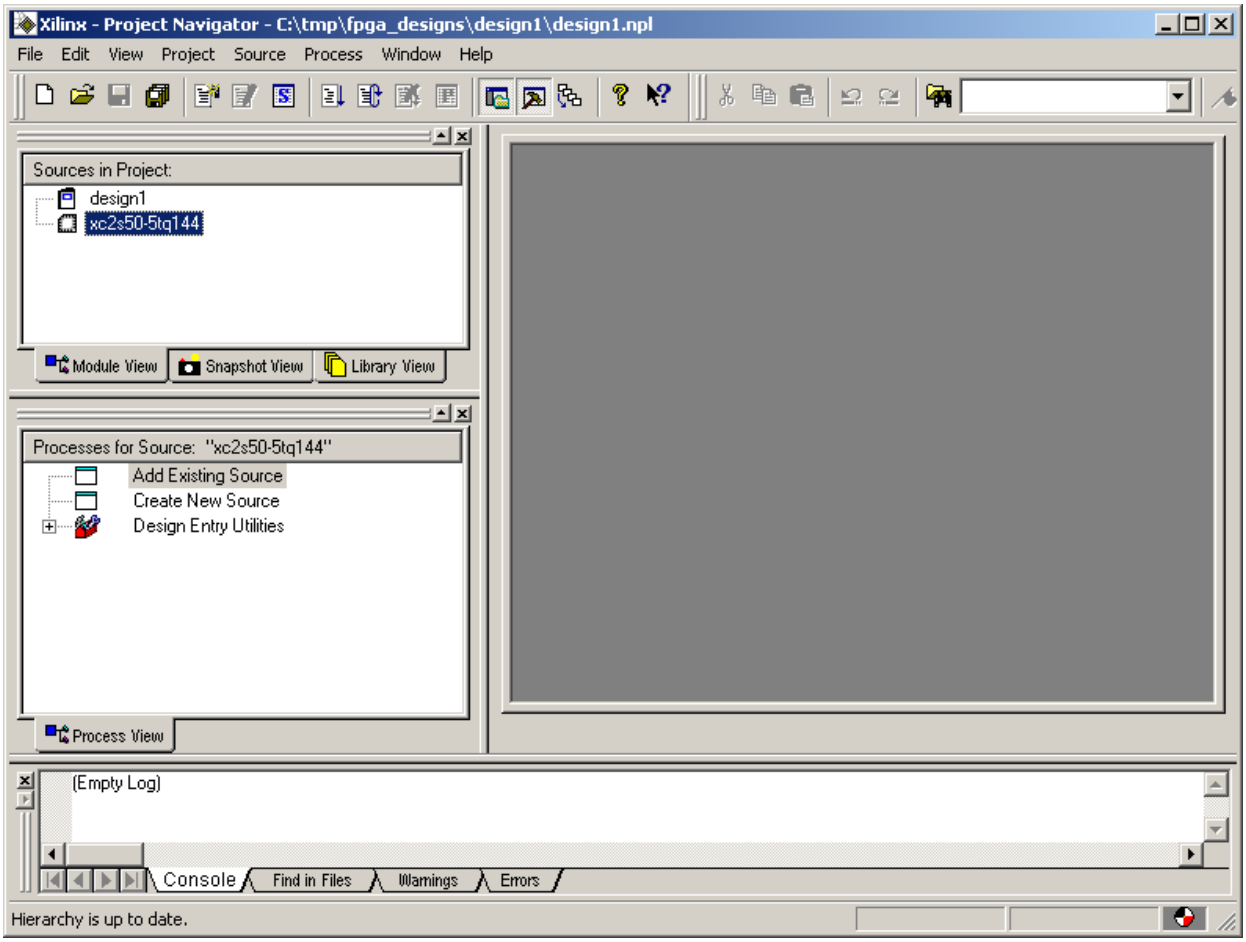


The final screen shows the pertinent information for the new project. Click on the Finish button to complete the creation of the project.



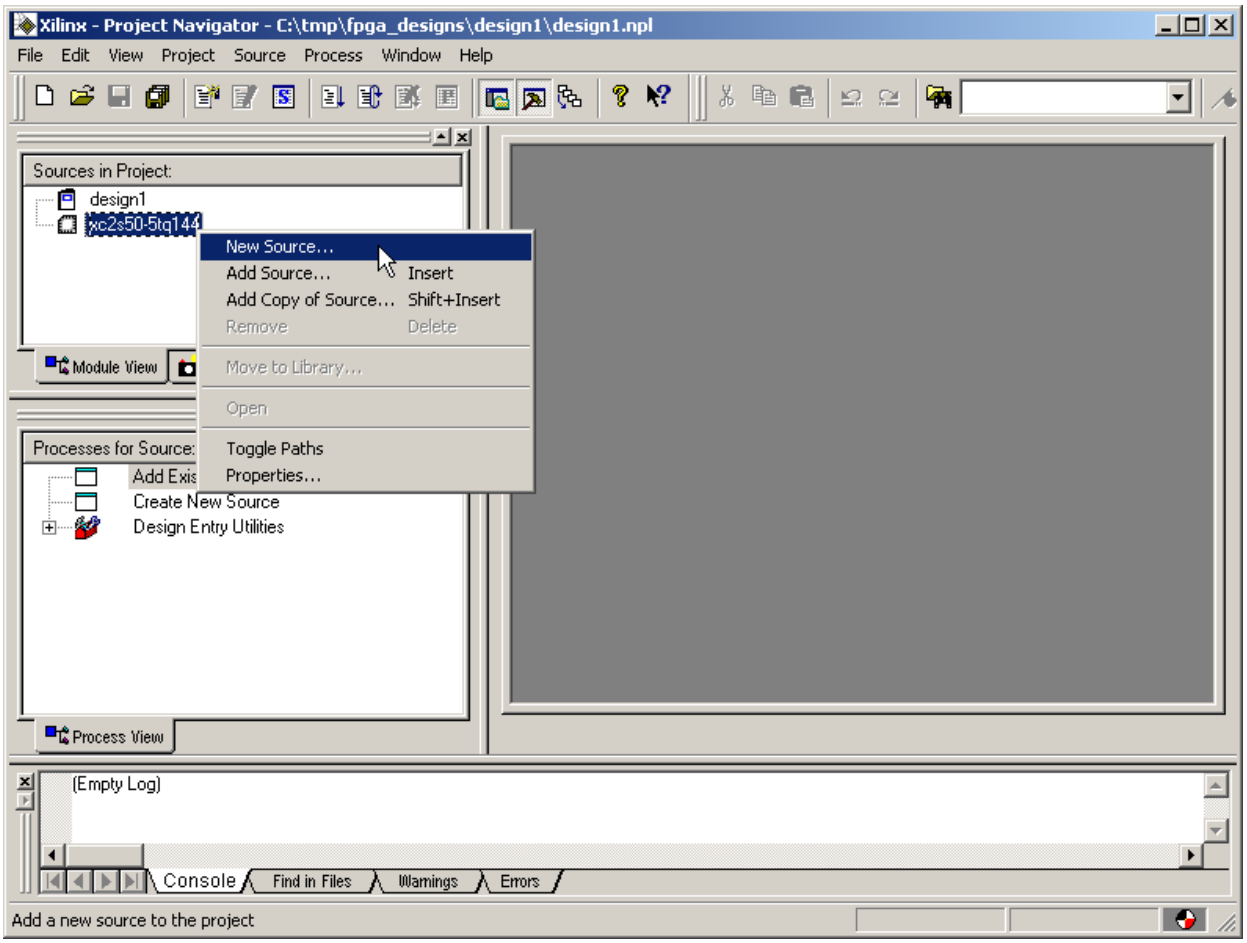
Now the Sources pane in the **Project Navigator** window contains two items:

1. A project object called design1.
2. A chip object called XC2S50-5TQ144.

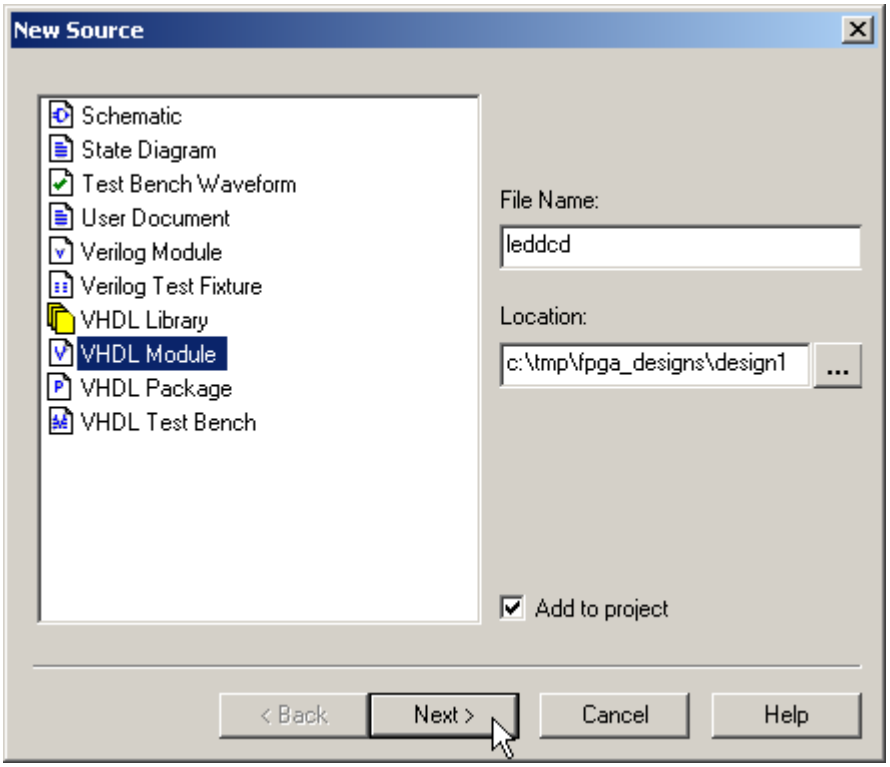


Describing Your Design With VHDL

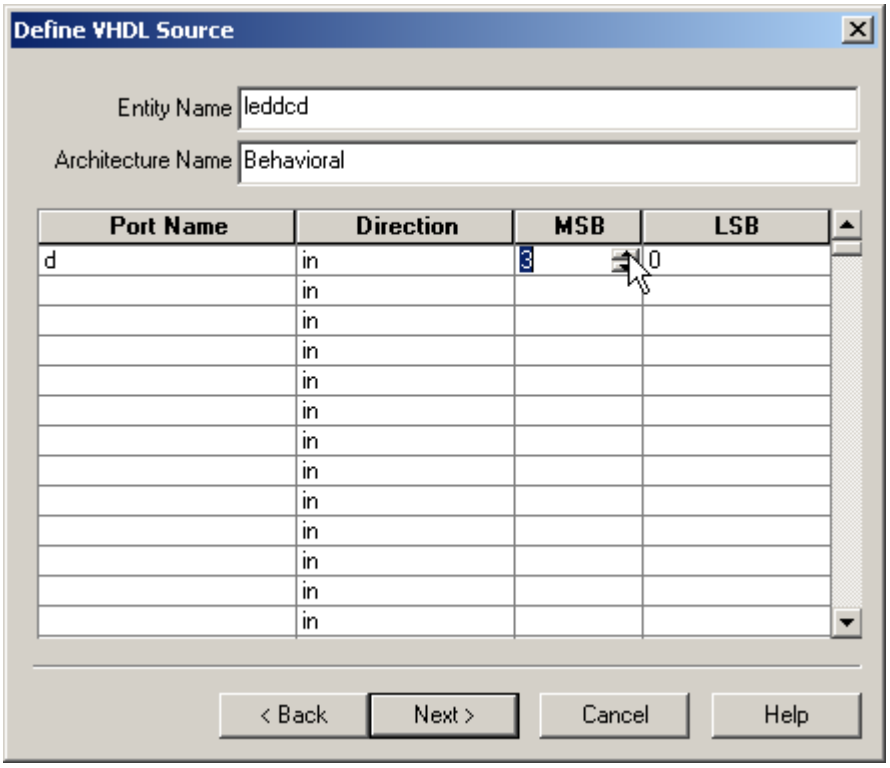
Once all the project set-up is complete, we can begin to actually design our LED decoder circuit. We start by adding a VHDL file to the ***design1*** project. Right-click on the XC2S50-5TQ144 object in the Sources pane and select New Source ... from the pop-up menu as shown below.



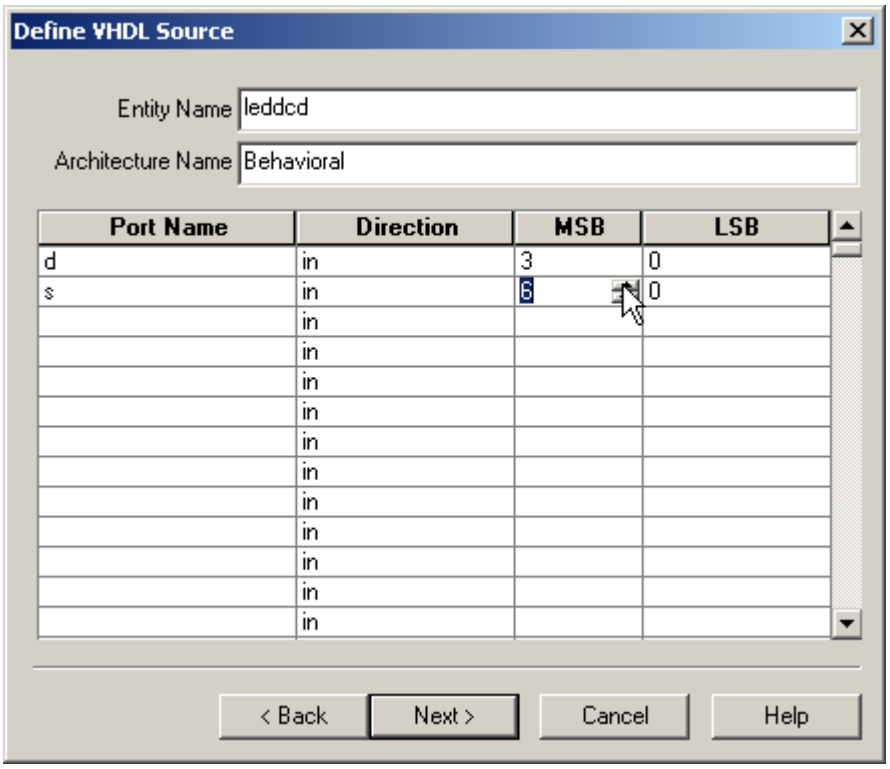
This causes a window to appear where we must select the type of source file we want to add. Since we are describing the LED decoder with VHDL, highlight the VHDL Module item. Then we type the name of the module, `leddcd`, into the File Name field and click on Next.



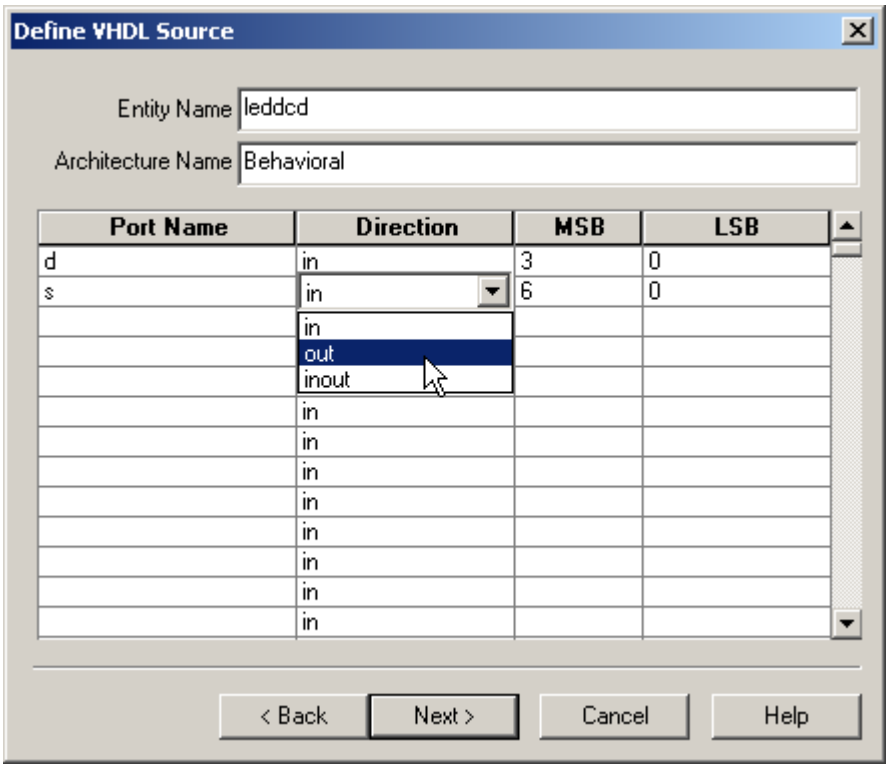
Then the **Define VHDL Source** window appears where we declare the inputs and outputs to the LED decoder circuit. In the first row, click in the Port Name field and type in d (the name of the inputs to the LED decoder). The **d** input bus has a width of four, so click in the MSB field and increment the upper range of the input field to 3 while leaving 0 in the LSB field.



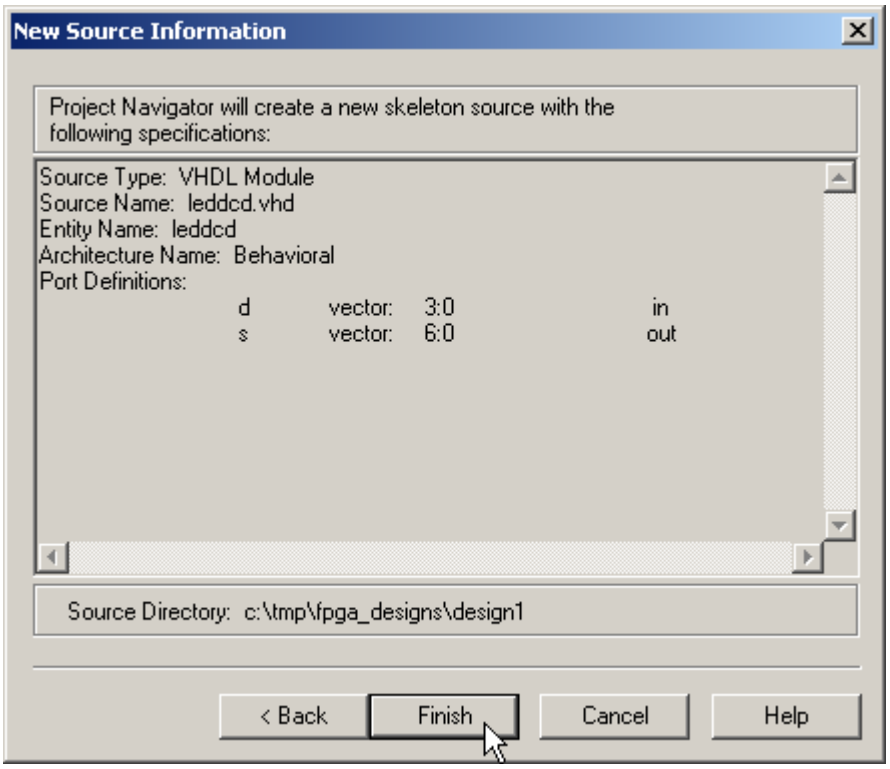
Perform the same operations to create the seven-bit wide **s** bus that drives the LEDs.



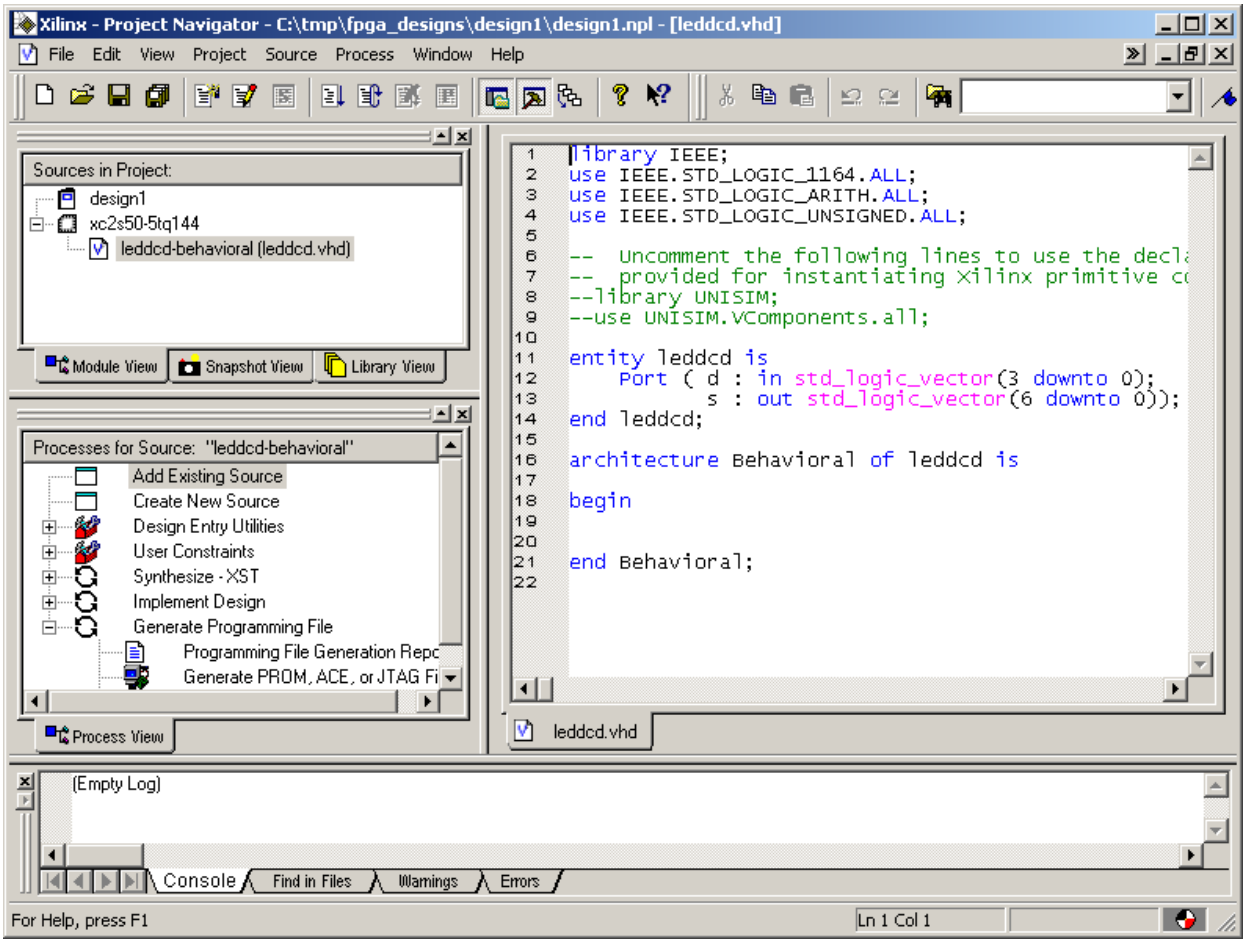
We must also click in the Direction field for the **s** bus and select out from the drop-down menu in order to make the **s** bus signals into outputs.



Click on Next in the **Define VHDL Source** window and we will get a summary of the information we just typed in:



After clicking on Finish, the editor pane of the **Project Navigator** window displays a VHDL skeleton for our LED decoder. (We also see the leddcd.vhd file has been added to the Sources pane.) Lines 1-4 create links to the IEEE library and packages that contain various useful definitions for describing a design. The LED decoder inputs and outputs are declared in the VHDL entity on lines 11-14. We will describe the logic operations of the decoder in the architecture section between lines 18 and 21.



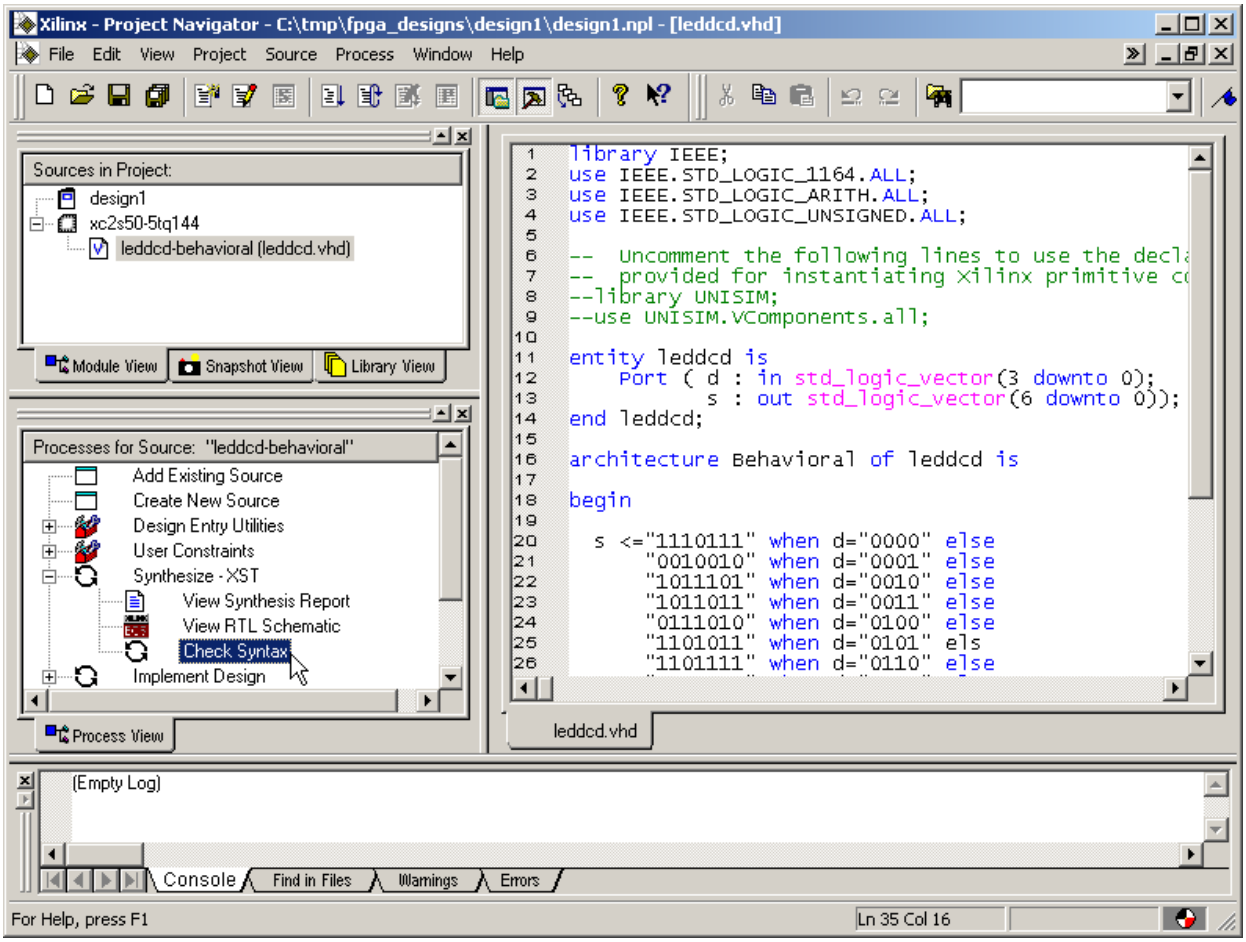
The completed VHDL file for the LED decoder is shown below. The architecture section contains a single statement which assigns a particular seven-bit pattern to the **s** output bus for any given four-bit input on the **d** bus (lines 20-35).

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_ARITH.ALL;
4  use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6  --  Uncomment the following lines to use the declarations that are
7  --  provided for instantiating Xilinx primitive components.
8  --library UNISIM;
9  --use UNISIM.VComponents.all;
10
11 entity leddcd is
12     Port ( d : in std_logic_vector(3 downto 0);
13            s : out std_logic_vector(6 downto 0));
14 end leddcd;
15
16 architecture Behavioral of leddcd is
17
18 begin
19
20   s <="1110111" when d="0000" else
21       "0010010" when d="0001" else
22       "1011101" when d="0010" else
23       "1011011" when d="0011" else
24       "0111010" when d="0100" else
25       "1101011" when d="0101" els
26       "1101111" when d="0110" else
27       "1010010" when d="0111" else
28       "1111111" when d="1000" else
29       "1111011" when d="1001" else
30       "1111110" when d="1010" else
31       "0101111" when d="1011" else
32       "0001101" when d="1100" else
33       "0011111" when d="1101" else
34       "1101101" when d="1110" else
35       "1101100"
36
37 end Behavioral;
38
For Help, press F1
```

Once the VHDL source is entered, we click on the  button to save it in the leddcd.vhd file.

Checking the VHDL Syntax

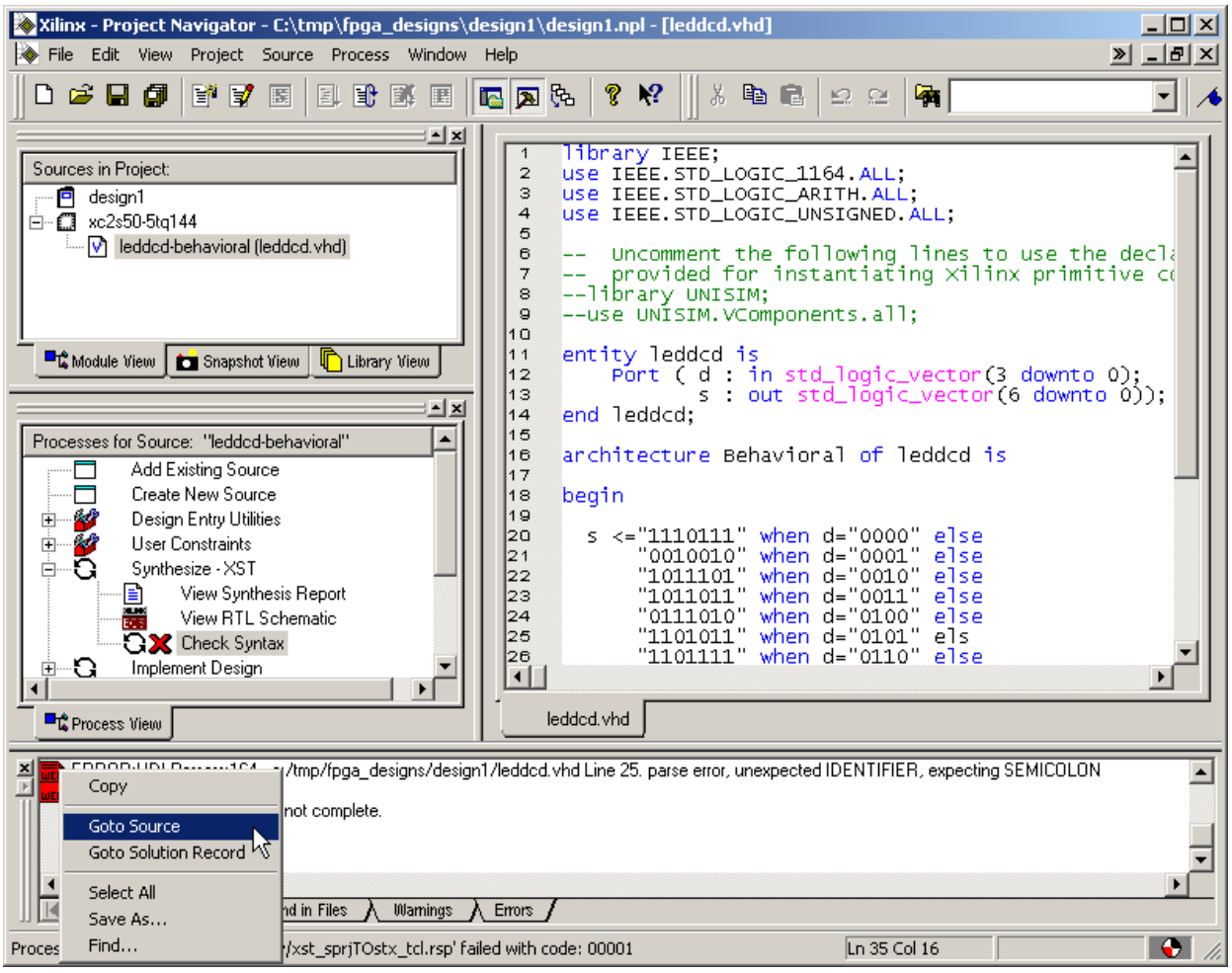
We can check for errors in our VHDL by highlighting the leddcd object in the Sources pane and then double-clicking on Check Syntax in the Process pane as shown below.



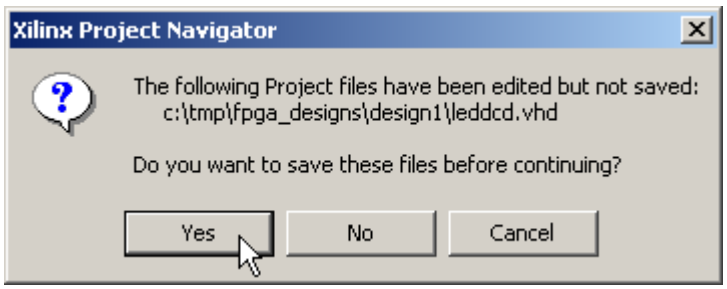
The syntax checking tool grinds away and then displays the result in the process window. In our case, an error was found as indicated by the ✗ next to the Check Syntax process. But what is the error and where is it?

Fixing VHDL Errors

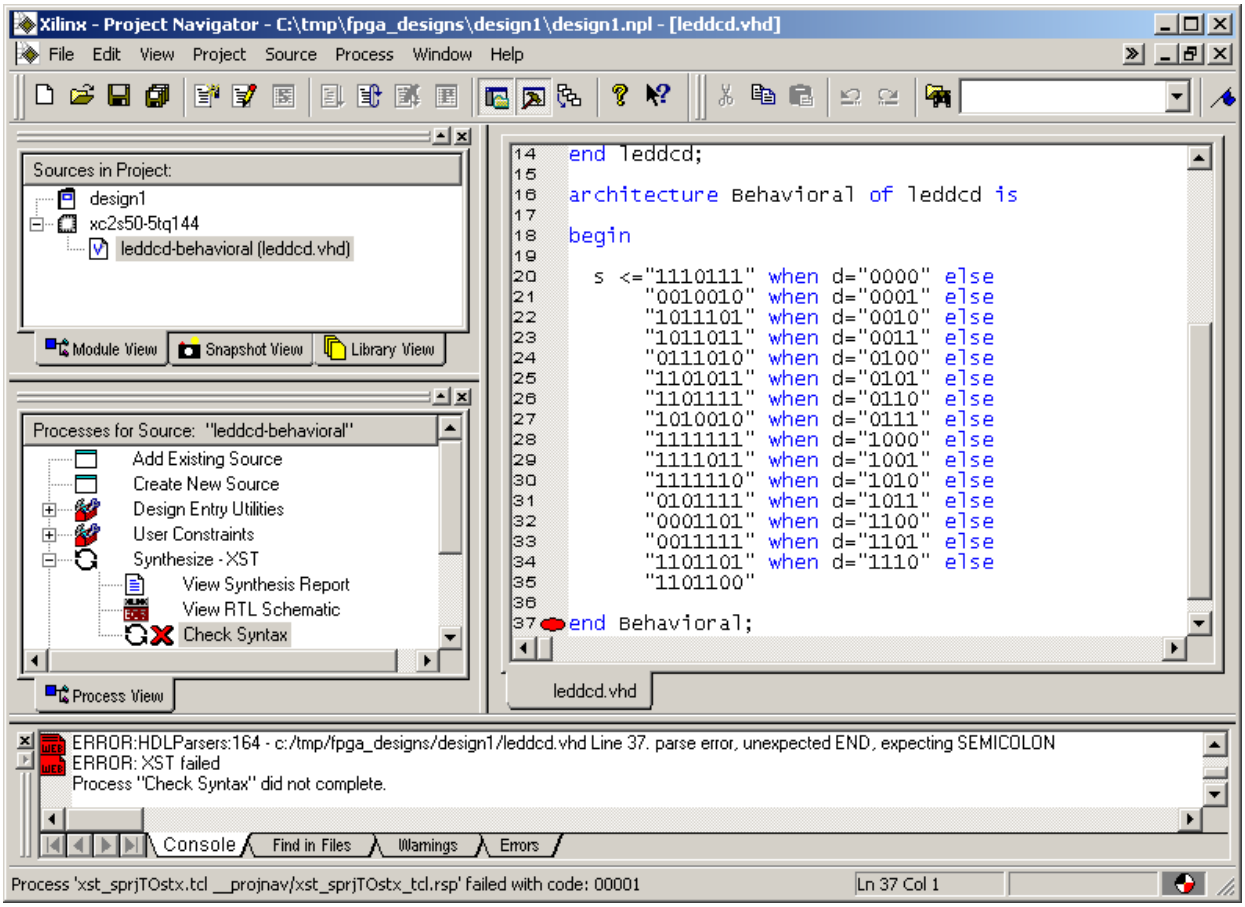
We can find the location of the error by scrolling the log pane at the bottom of the **Project Navigator** window until we find an error message. In this case, the error is located on line 25 and we can manually scroll there. You can also right-click on the error message in the log pane to go directly to the erroneous source. (This is most useful in more complicated projects consisting of multiple source files.)



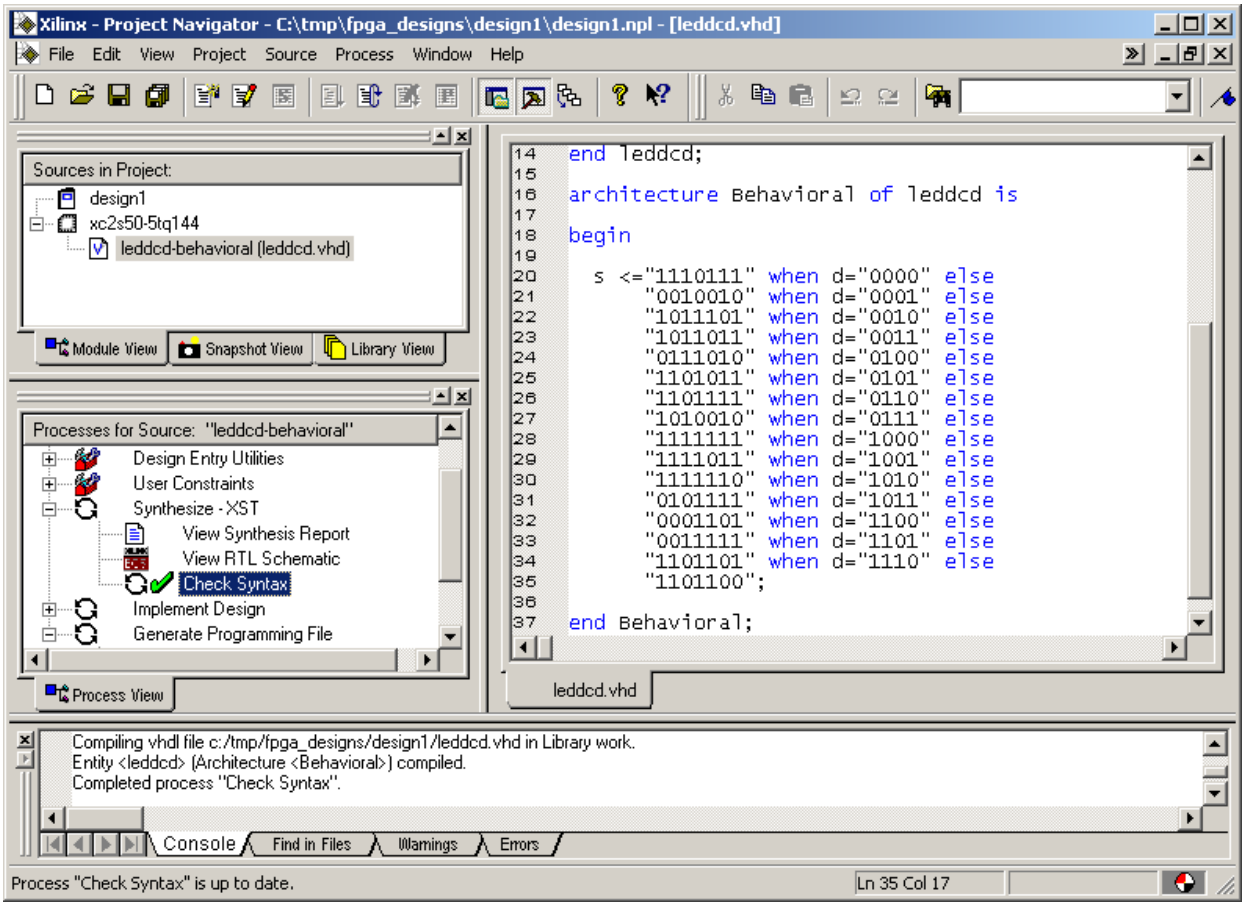
On line 25, we see that we have left the 'e' off the end of the `else` keyword. After correcting this error, we can double-click the on Check Syntax in the Process pane to re-check the VHDL code. We will be asked to save the file before the syntax check proceeds. Click on Yes.



The syntax checker now finds another error on line 37 of the VHDL code.

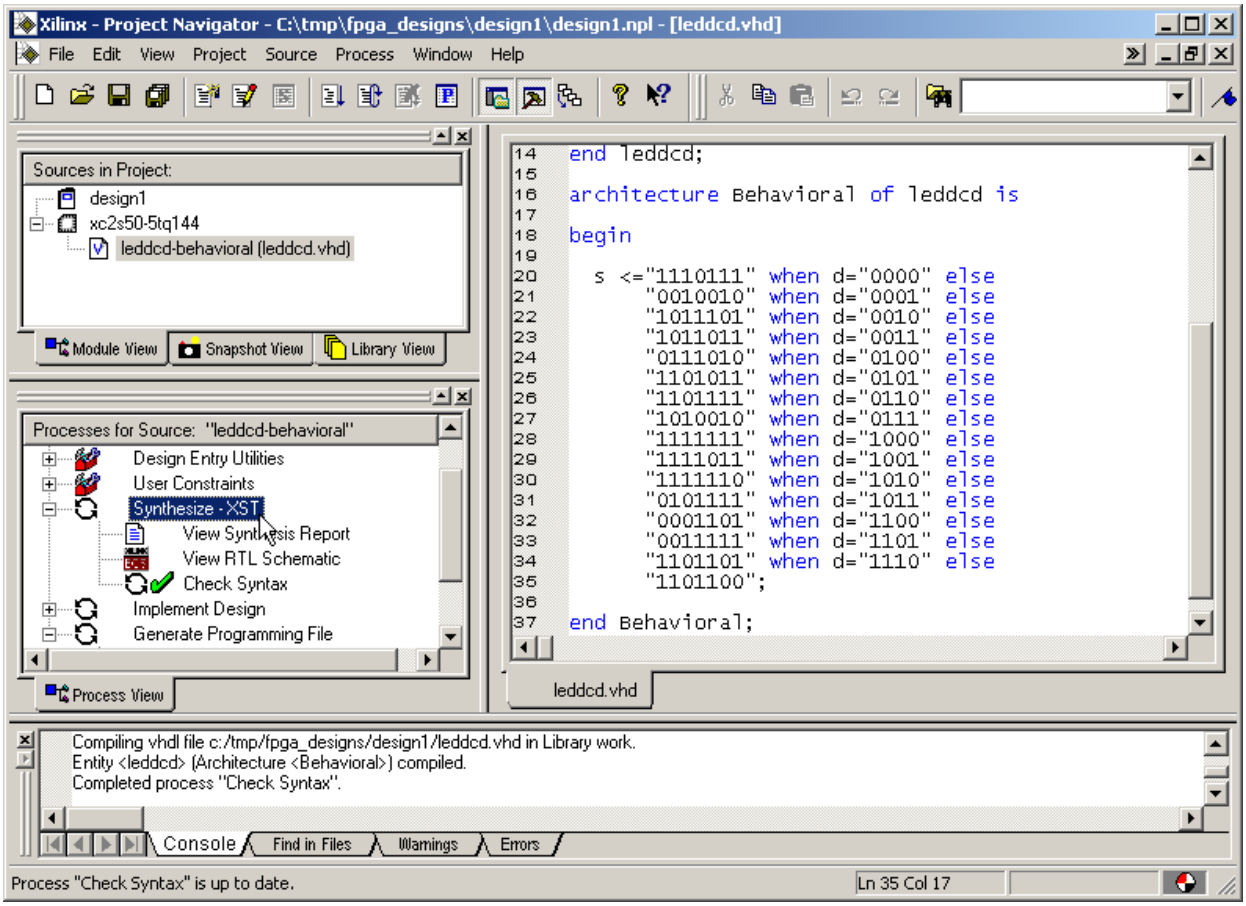


When we look at line 37 we see it is just the end statement for the architecture section. The VHDL syntax checker was expecting to find a ';' and we can see it is missing from the end of line 35. Add the semicolon and save the file. Now when we double-click the Check Syntax process, it runs and then displays a ✔ to indicate there are no more errors.



Synthesizing the Logic circuitry for Your Design

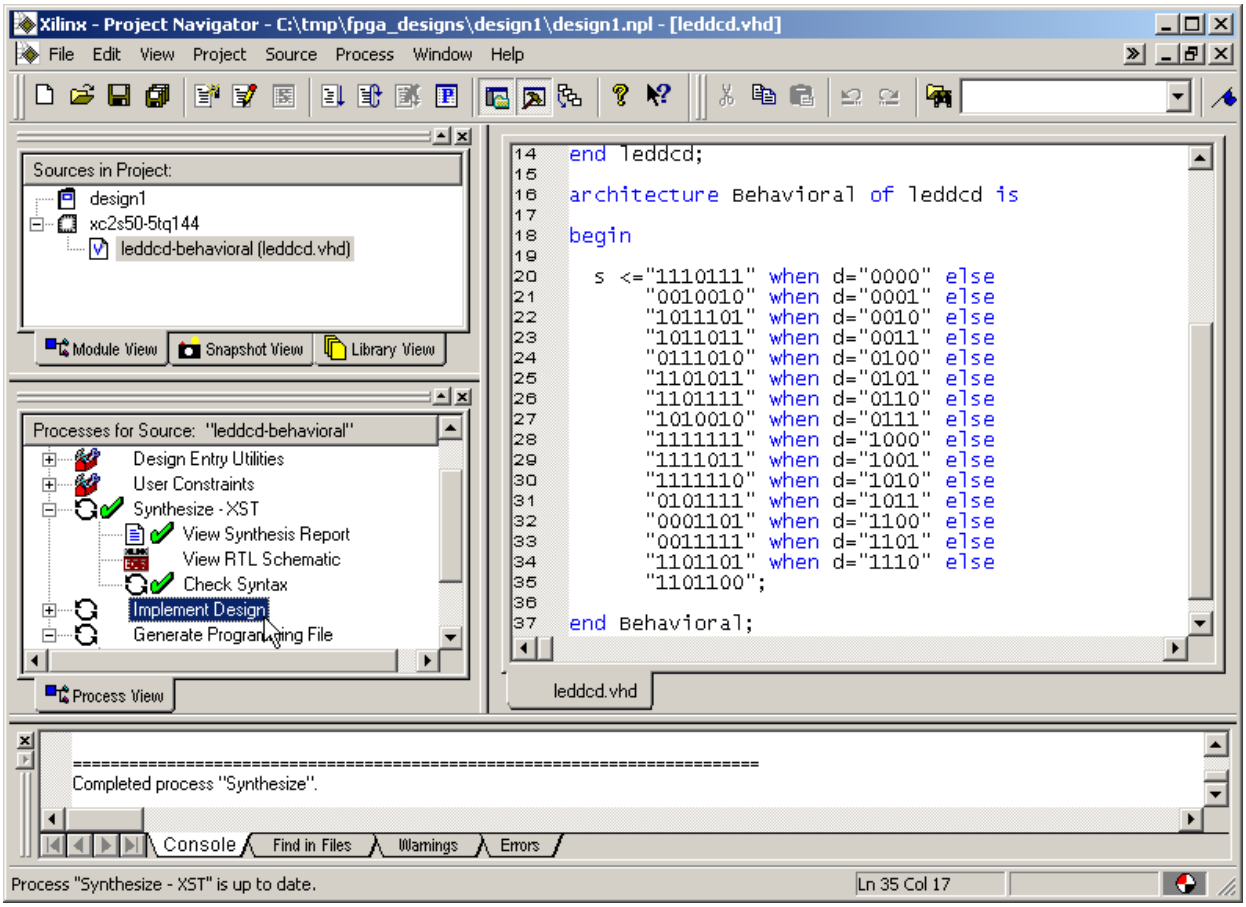
Now that we have valid VHDL for our design, we need to convert it into a logic circuit. This is done by highlighting the leddcd object in the Sources pane and then double-clicking on the Synthesize process as shown below.



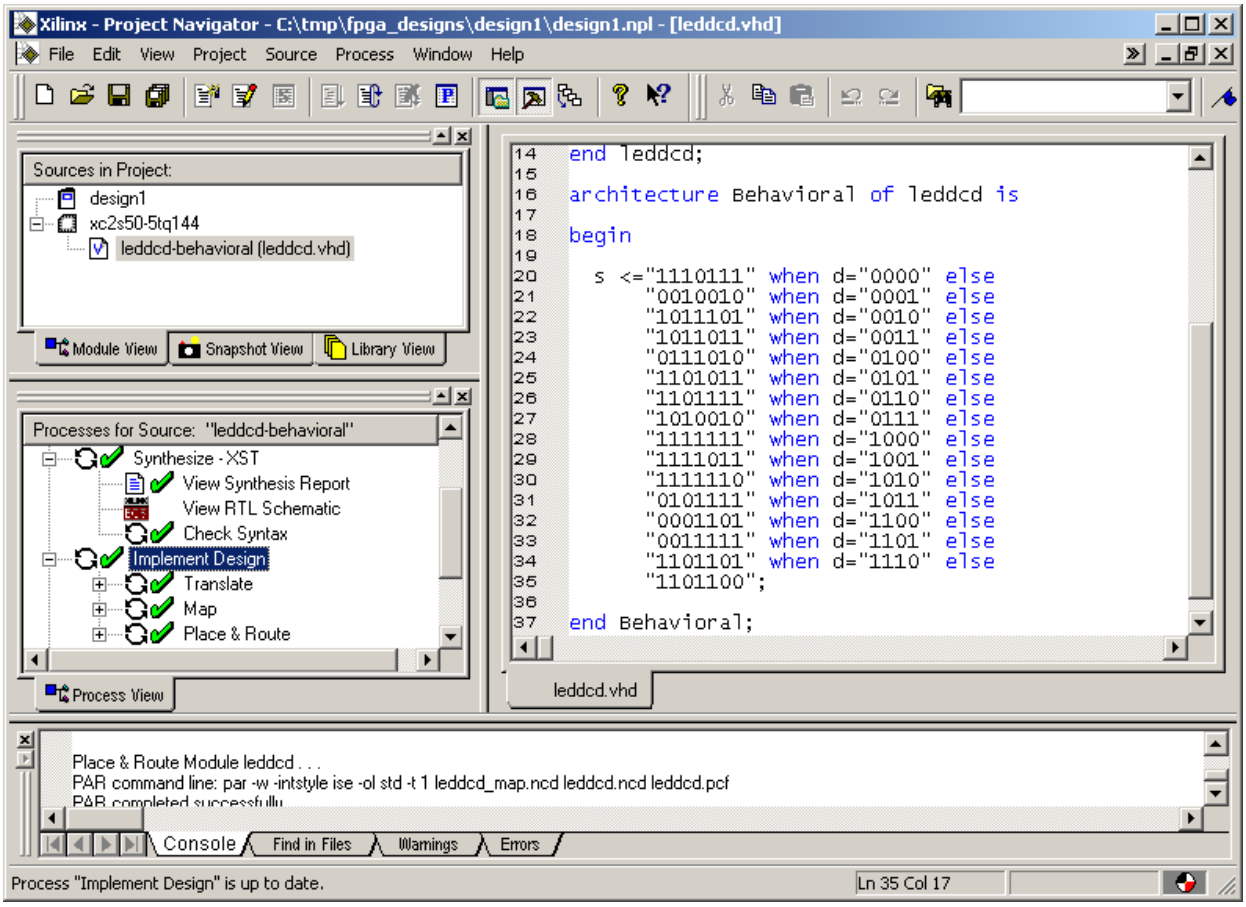
The synthesizer will read the VHDL code and transform it into a netlist of gates. This will take less than a minute. If no problems are detected, a ✔ will appear next to the Synthesize process. You can double-click on the View Synthesis Report to see the various synthesizer options that were enabled and some device utilization and timing statistics for the synthesized design. You can also double-click on View RTL Schematic to see the schematic that was derived from the VHDL source code, but it's not very interesting in this case.

Implementing the Logic Circuitry in the FPGA

We now have a synthesized logic circuit for the LED decoder, but we need to translate, map and place & route it into the logic resources of the FPGA in order to actually use it. We start this process by highlighting the leddcd object in the Sources pane and then double-clicking on the Implement Design process.

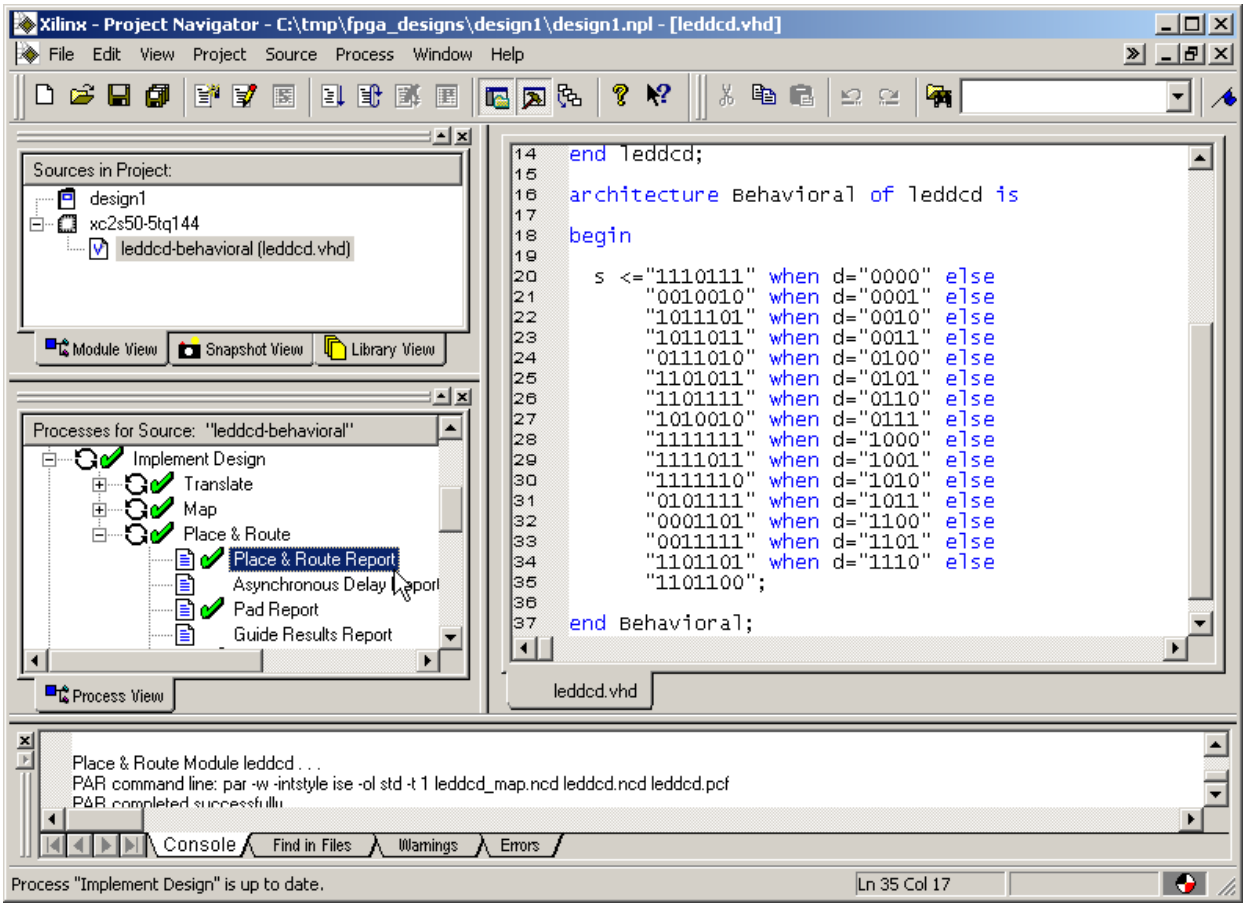


You can watch the progress of the implementation process in the status bar at the bottom of the **Project Navigator** window. For a simple design like the LED decoder, the fitting is completed in seconds (on a 850 MHz Athlon PC with 768 Mbytes or RAM). A successful implementation is indicated by the ✔ next to the Implement Design process. You can expand the Implement Design process to see the subprocesses within it. The Translate process converts the netlist output by the synthesizer into a Xilinx-specific format and annotates it with any design constraints we may specify (more on that later). The Map process decomposes the netlist and rearranges it so it fits nicely into the circuitry elements contained in the FPGA device we selected. Then the Place & Route process assigns the mapped elements to specific locations in the FPGA and sets the switches to route the logic signals between them. If the Implement Design provcess had failed, a ✘ would appear next to the subprocess where the error occured. You may also see a ! to indicate a successful completion but some warnings were issued or not all the subprocesses were enabled.



Checking the Implementation

We have our design fitted into the XC2S50 FPGA, but how much of the chip does it use? Which pins are the inputs and outputs assigned to? We can find answers to these questions by double-clicking on the Place & Route Report and the Pad Report in the Process pane.



The device utilization of the LED decoder circuit can be found near the top of the place & route report. The circuit only uses 4 of the 768 available slices in the XC2S50 FPGA. Each slice contains two CLBs and each CLB can compute the logic function for one LED segment output.

```
Device utilization summary:

   Number of External IOBs           11 out of 92     11%
      Number of LOCed External IOBs   0 out of 11      0%

   Number of SLICEs                   4 out of 768     1%
```

The pad report shows what pins the inputs and outputs use. The **d** inputs have been assigned to pins 124, 121, 130 and 118. The **s** outputs which drive the LED segments have been routed through pins 126, 116, 129, 122, 120, 123 and 117. (The pad report was edited to remove unused pins and fields so it would fit into this document.)

```
----------|------------|----------|----------|------------|
Pin Number|Signal Name |Pin Usage |Direction |IO Standard |
----------|------------|----------|----------|------------|
P116      |s<1>        |IOB       |OUTPUT    |LVTTL       |
P117      |s<6>        |IOB       |OUTPUT    |LVTTL       |
P118      |d<3>        |IOB       |INPUT     |LVTTL       |
P120      |s<4>        |IOB       |OUTPUT    |LVTTL       |
P121      |d<1>        |IOB       |INPUT     |LVTTL       |
P122      |s<3>        |IOB       |OUTPUT    |LVTTL       |
P123      |s<5>        |IOB       |OUTPUT    |LVTTL       |
P124      |d<0>        |IOB       |INPUT     |LVTTL       |
P126      |s<0>        |IOB       |OUTPUT    |LVTTL       |
P129      |s<2>        |IOB       |OUTPUT    |LVTTL       |
P130      |d<2>        |IOB       |INPUT     |LVTTL       |
```

Assigning Pins with Constraints

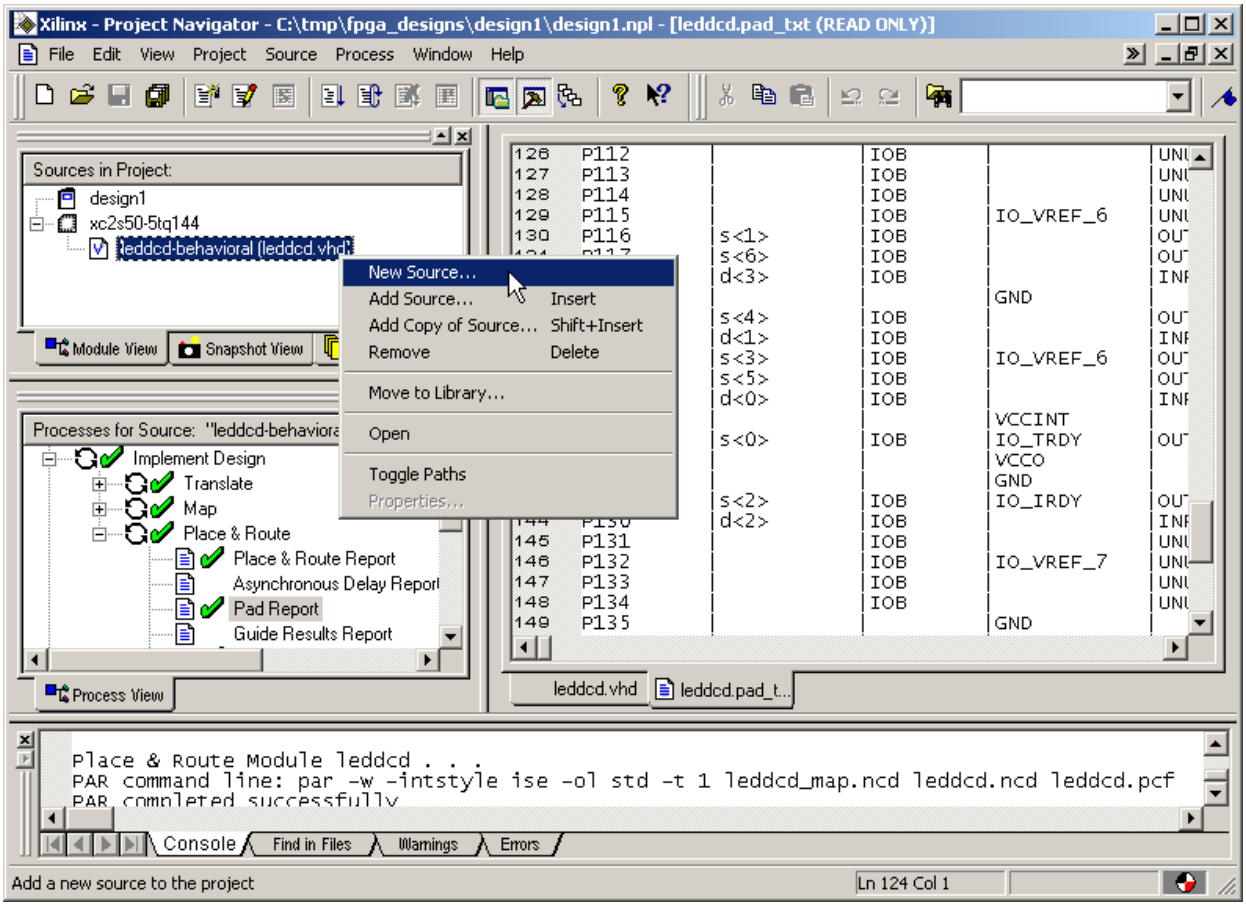
The problem we have now is that the inputs and outputs for the LED decoder were assigned to pins picked by the implementation process, but these are not the pins we actually want to use on the XSA-50 Board. The FPGA on the XSA-50 Board has eight inputs which are driven by the PC parallel port and we would like to assign the LED decoder inputs to four of these as follows:

LED Decoder Input	XSA-50 XC2S50 FPGA Pin
d0	P50
d1	P48
d2	P42
d3	P47

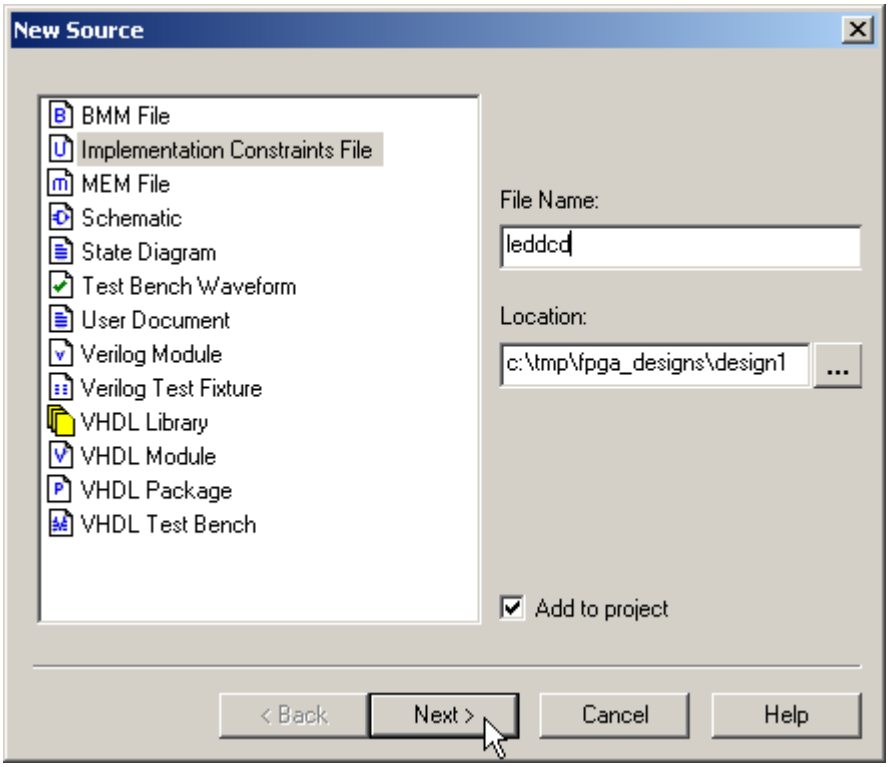
Likewise, the XSA-50 Board has a seven-segment LED attached to the following pins of the FPGA:

LED Decoder Output	XSA-50 XC2S50 FPGA Pin
s0	P67
s1	P39
s2	P62
s3	P60
s4	P46
s5	P57
s6	P49

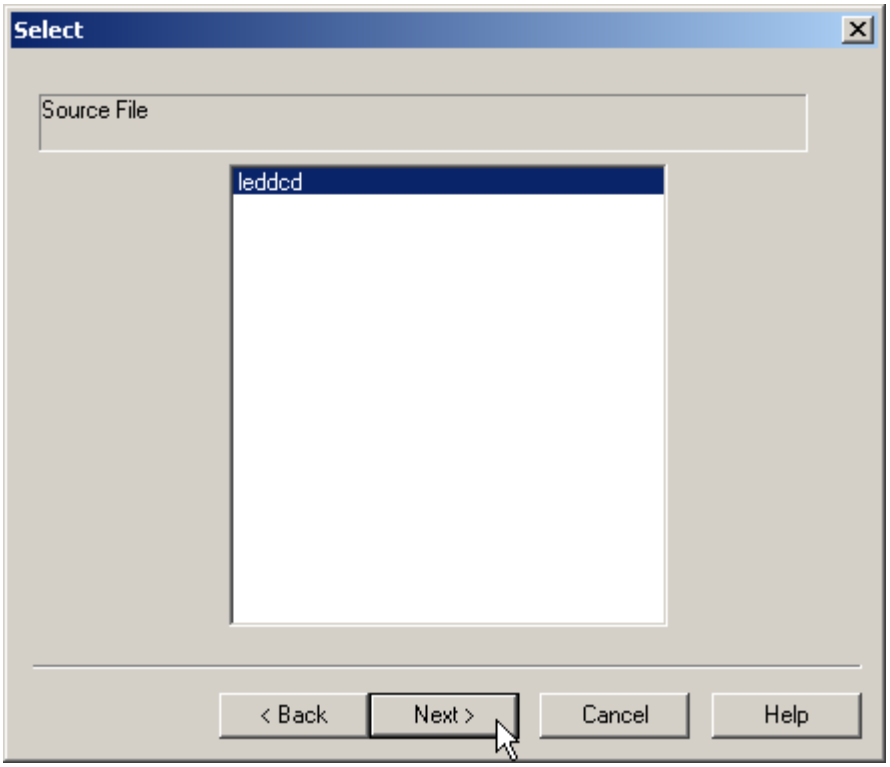
How do we constrain the implementation process so it assigns the inputs and outputs to the pins we want to use? We start by right-clicking the leddcd object in the Sources pane and selecting New Source... from the pop-up menu.



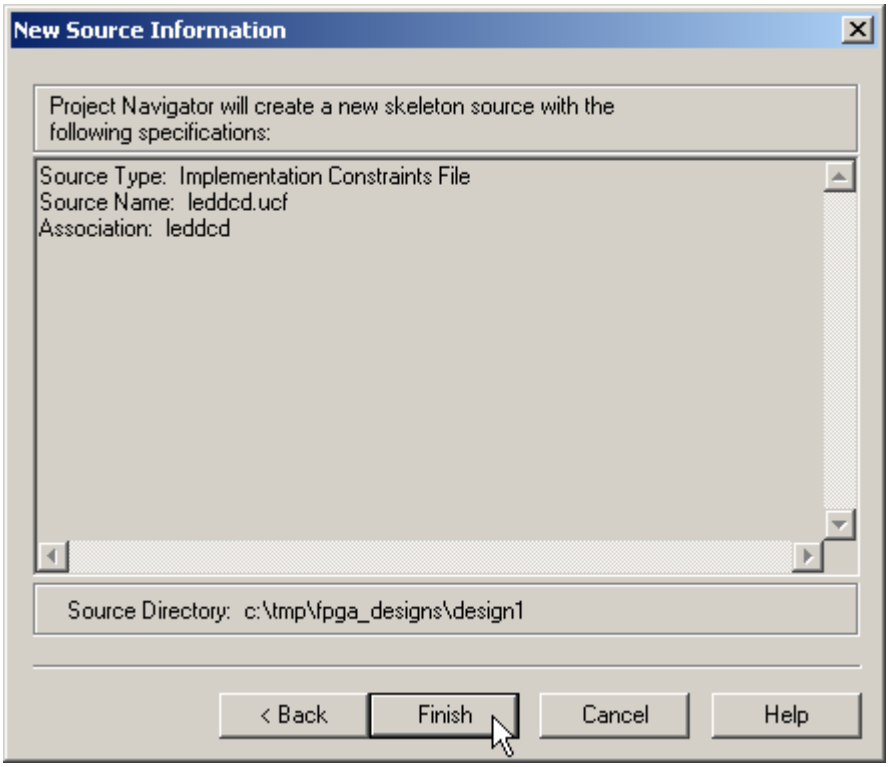
Select Implementation Constraints File as the type of source file we want to add and type `leddcd` in the File Name field. Then click on the Next button.



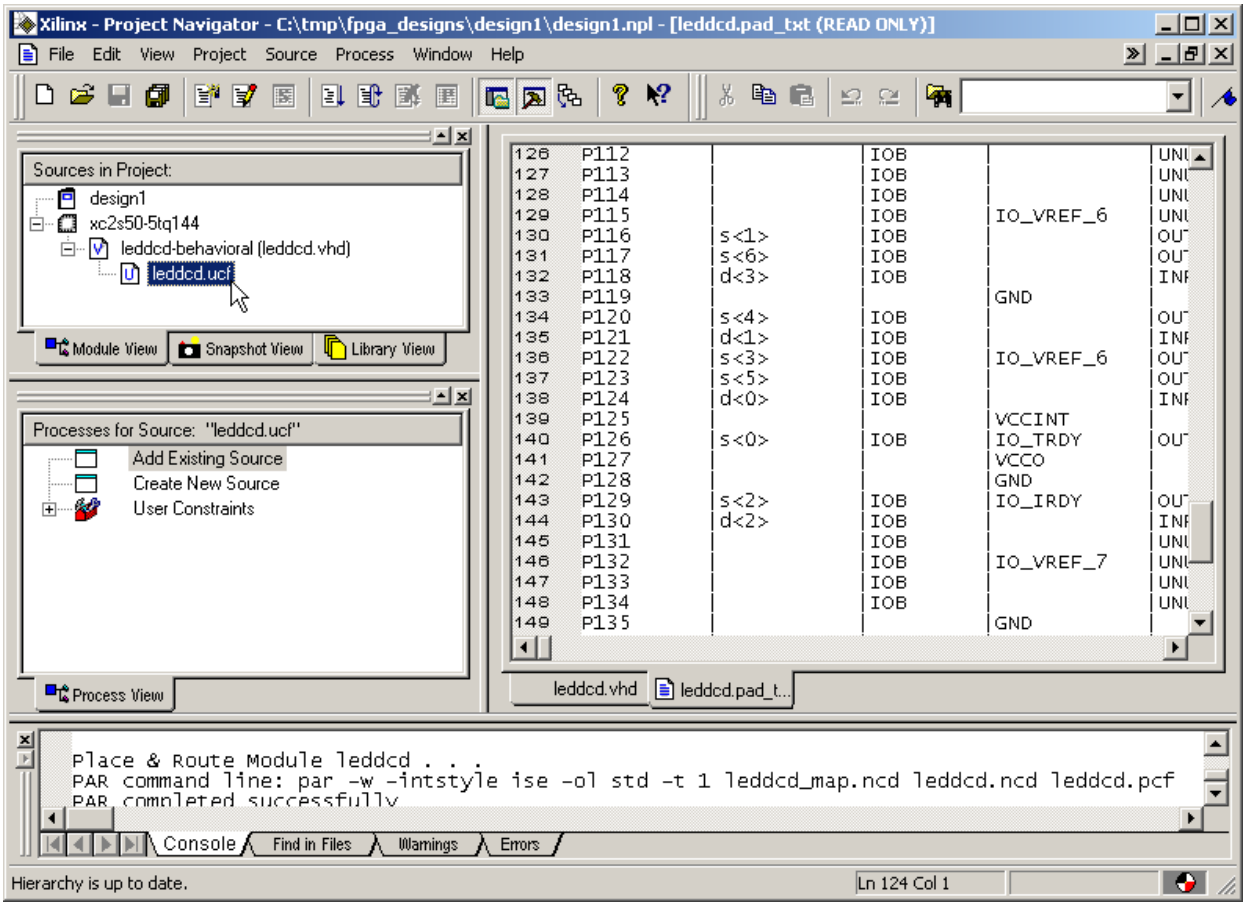
Then we are asked to pick the file with which to associate the constraints. For this design there is only one choice, so click on the Next button and proceed.



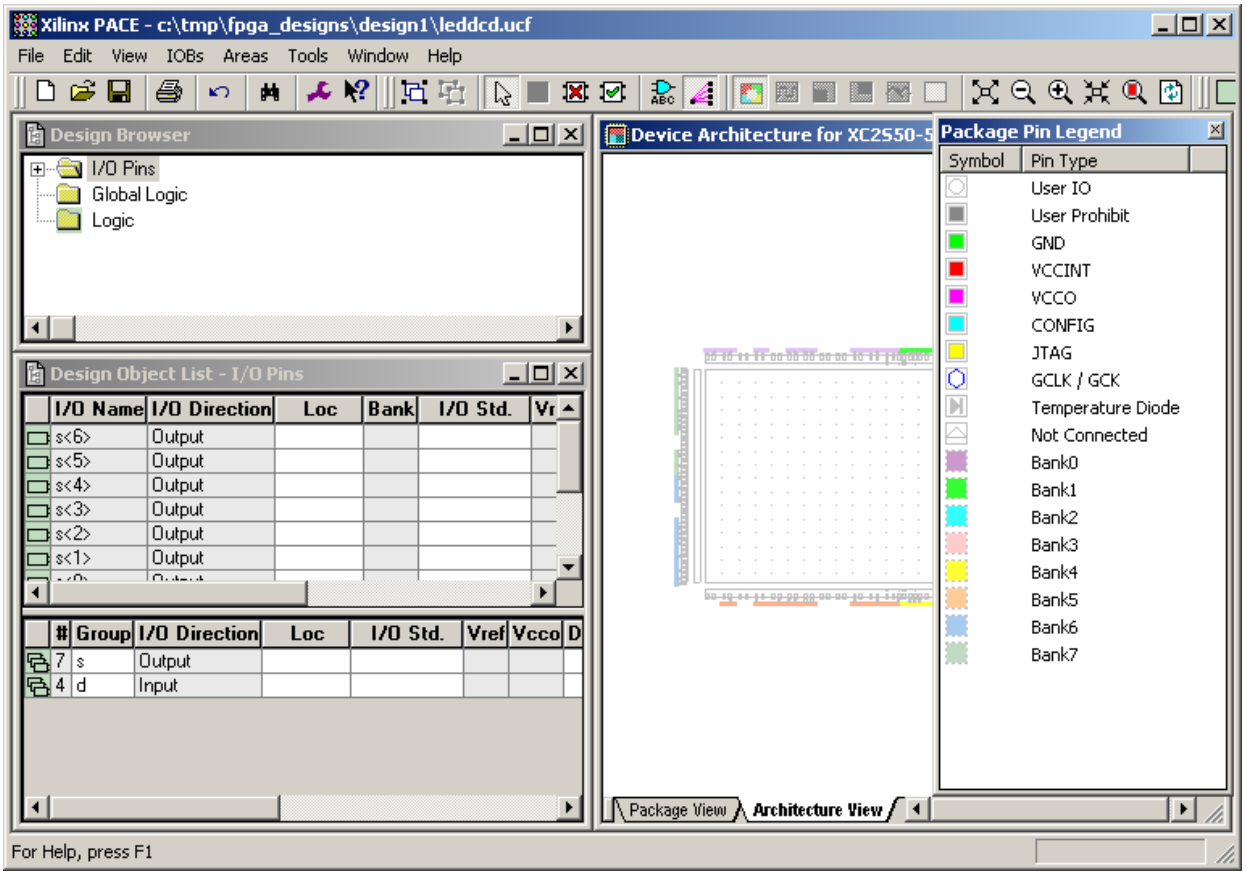
We will receive a feedback window that shows the name and type of the file we created and the file to which it is associated. Click on the Finish button to complete the addition of the leddcd.ucf file to this project.



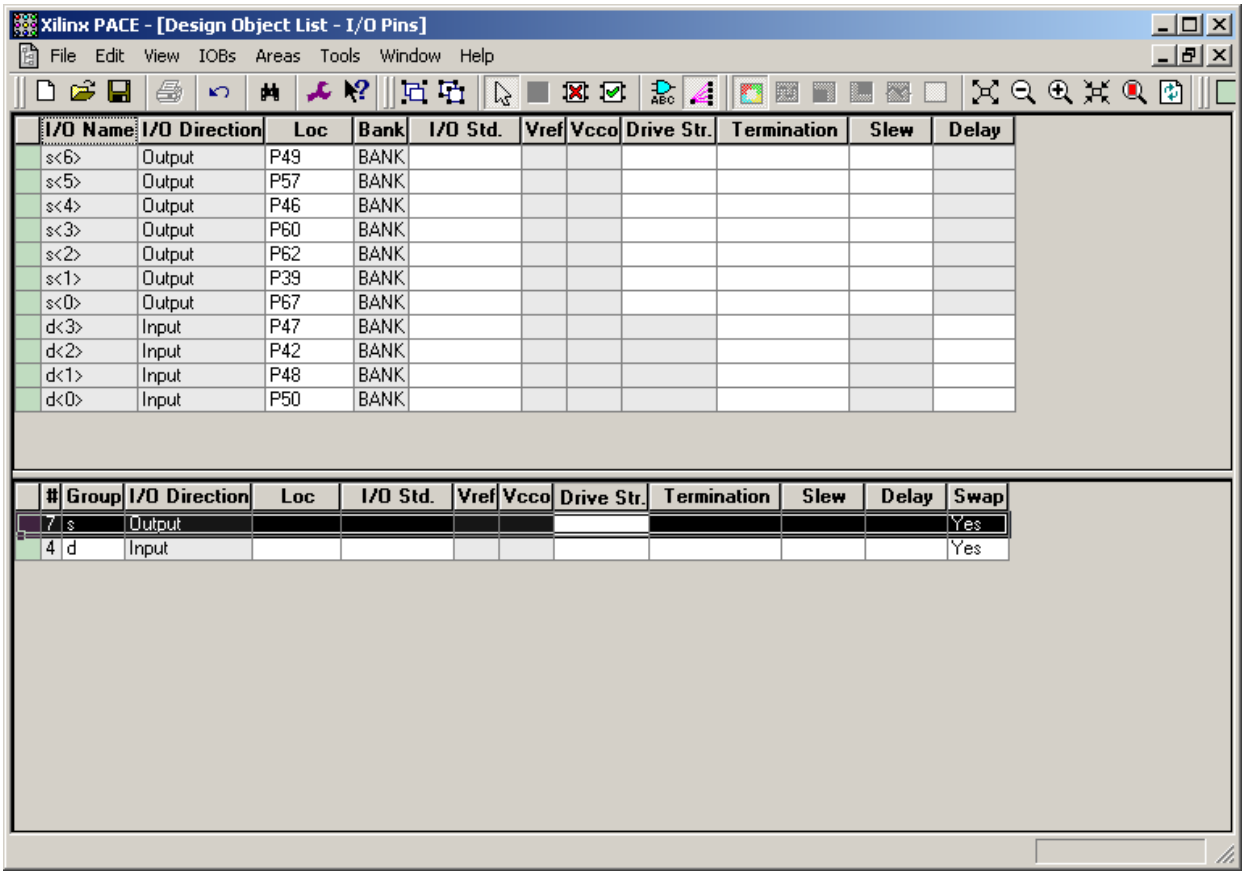
Now double-click the leddcd.ucf object in the Sources pane to begin adding pin assignments to the design.



The **Xilinx PACE** window now appears. Click on the Ports tab in the upper pane. A list of the current inputs and outputs for the LED decoder will appear in the **Design Object List – I/O Pins** pane. We can change our pin assigments here.

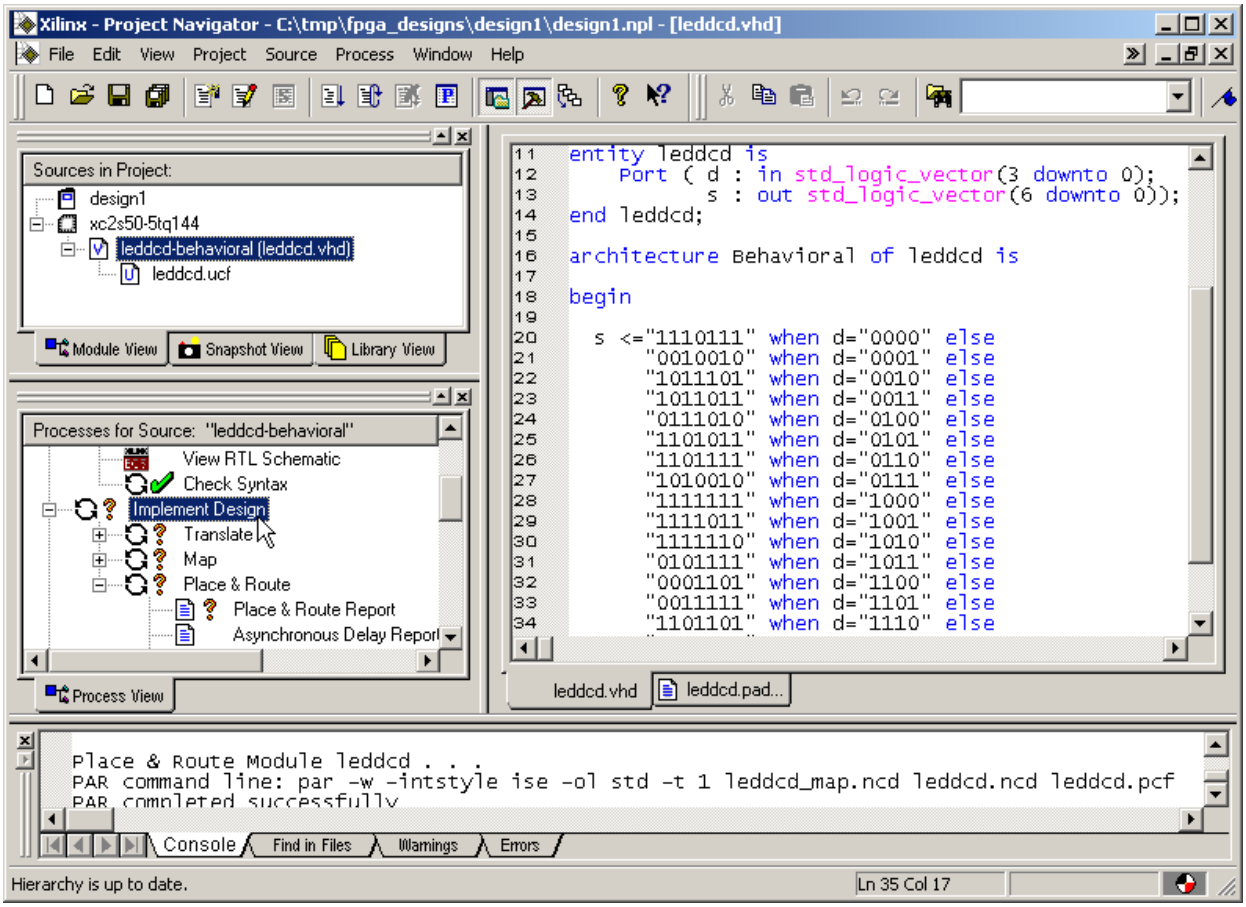


We start by clicking in the Location field for the **d<0>** input. Then just type in the pin assignment for this input: `P50`. Do this for each of the inputs and outputs using the pin assignments from the tables shown previously. After doing this, the **Xilinx PACE** window appears as follows.



After the pin assignments are entered, click on the  button to save the pin assignment constraints. Then select File➔Exit to close the **Xilinx PACE** window.

Now we can re-implement our design by highlighting the leddcd object in the Sources pane and double-clicking on the Implement Design process.



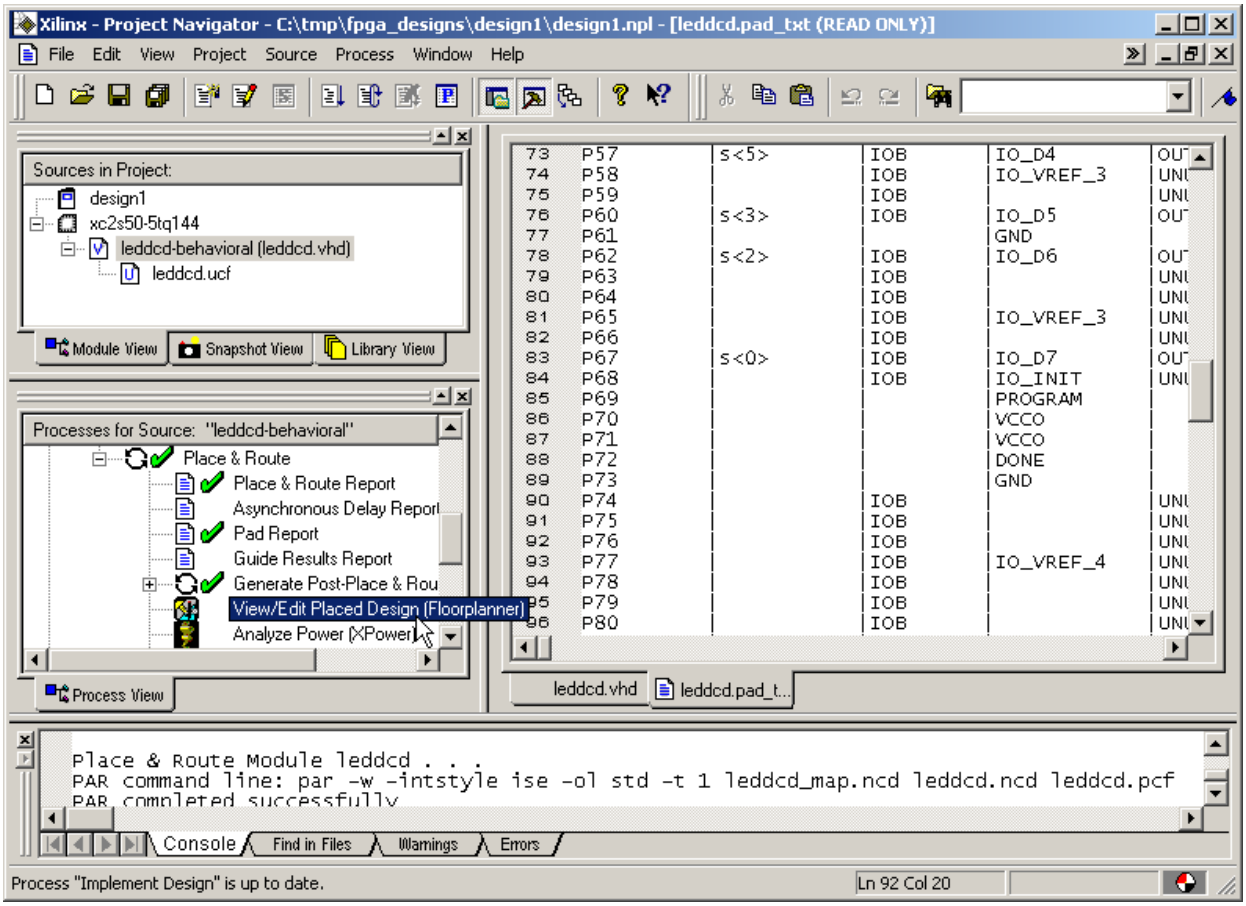
Next double-click on Pad Report to view the pin assignments made during the implementation process. Now the pad report contains the following pin assignments:

```
----------|------------|----------|----------|------------|
Pin Number|Signal Name |Pin Usage |Direction |IO Standard |
----------|------------|----------|----------|------------|
P39       |s<1>        |IOB       |OUTPUT    |LVTTL       |
P42       |d<2>        |IOB       |INPUT     |LVTTL       |
P46       |s<4>        |IOB       |OUTPUT    |LVTTL       |
P47       |d<3>        |IOB       |INPUT     |LVTTL       |
P48       |d<1>        |IOB       |INPUT     |LVTTL       |
P49       |s<6>        |IOB       |OUTPUT    |LVTTL       |
P50       |d<0>        |IOB       |INPUT     |LVTTL       |
P57       |s<5>        |IOB       |OUTPUT    |LVTTL       |
P60       |s<3>        |IOB       |OUTPUT    |LVTTL       |
P62       |s<2>        |IOB       |OUTPUT    |LVTTL       |
P67       |s<0>        |IOB       |OUTPUT    |LVTTL       |
```

The reported pin assignments match the assignments we made in the **Constraints Editor** window so it appears we accomplished what we wanted.

Viewing the Chip

After the implementation process completes, we can get a graphical depiction of how the logic circuitry and I/O are assigned to the FPGA CLBs and pins. Just highlight the leddcd object in the Sources pane and then double-click the View/Edit Placed Design (FloorPlanner) process.

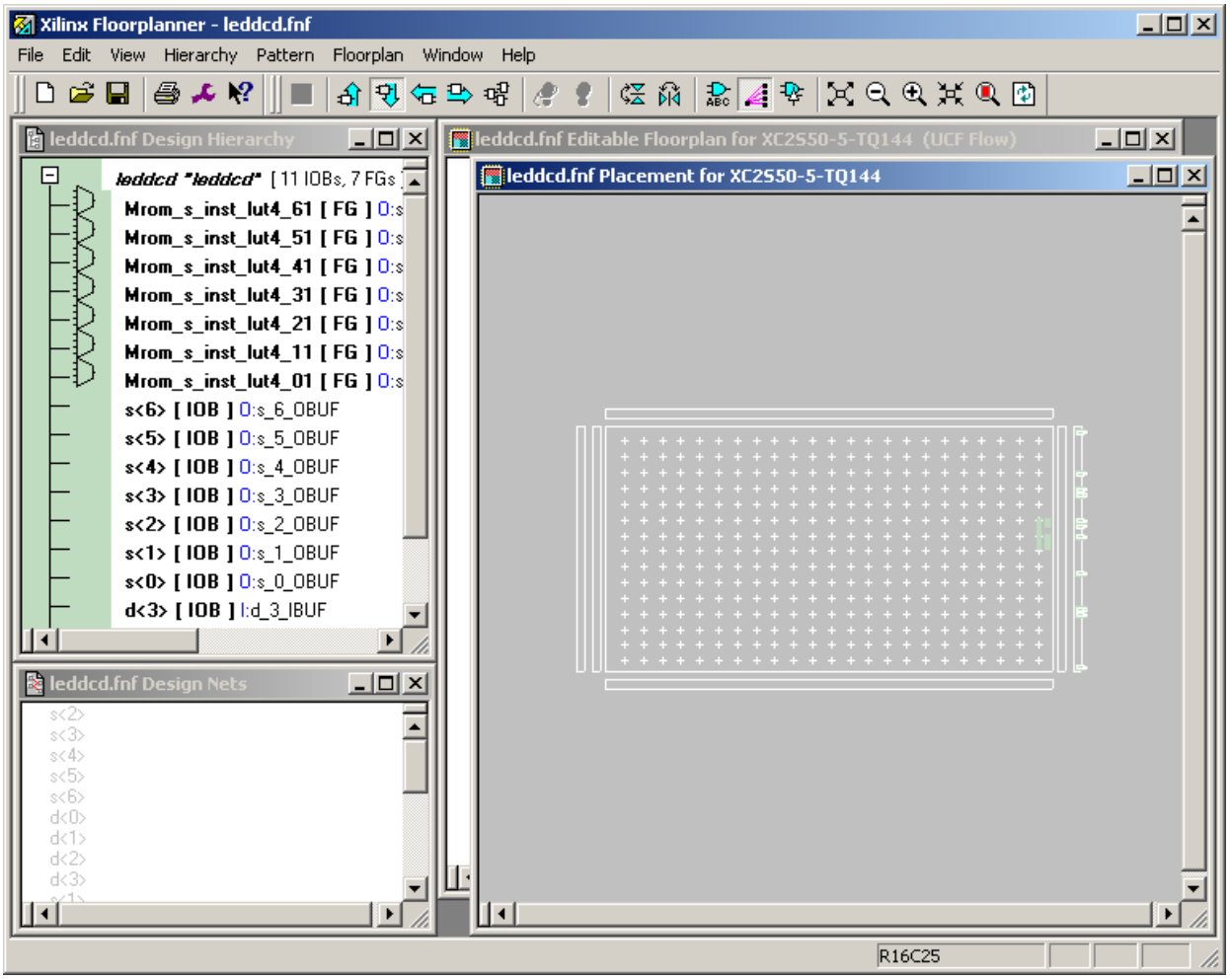


The **FloorPlanner** window will appear containing three panes:

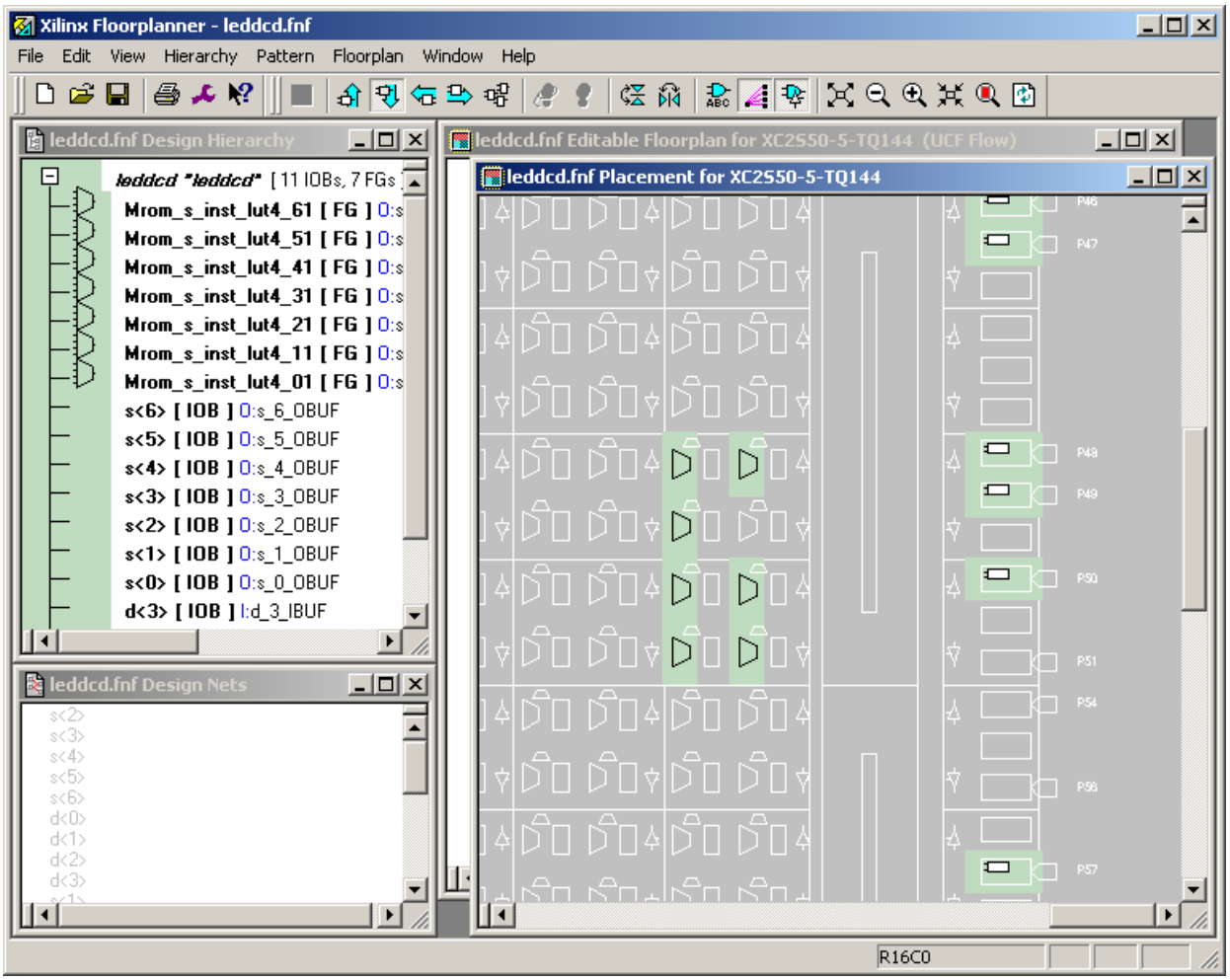
1. The **Design Hierarchy** pane lists the LED decoder inputs, outputs and LUTs assigned to the various CLBs in the FPGA.

2. The **Design Nets** pane lists the various signal nets in the LED decoder.

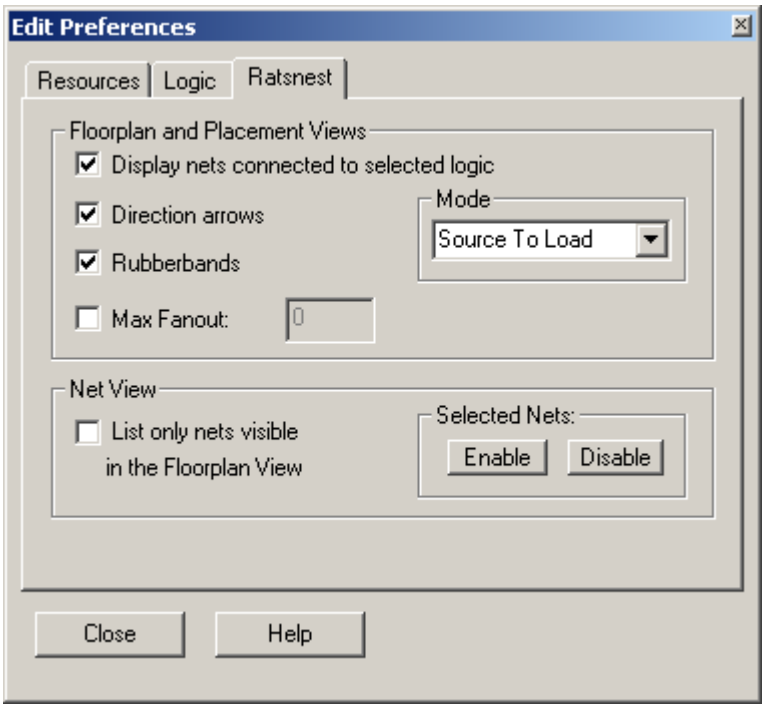
3. The **Placement** pane shows the 32 × 48 array of slices in the FPGA. The I/O pins are also shown around the periphery. (The pins used for Vcc, GND, and programming are not shown.)



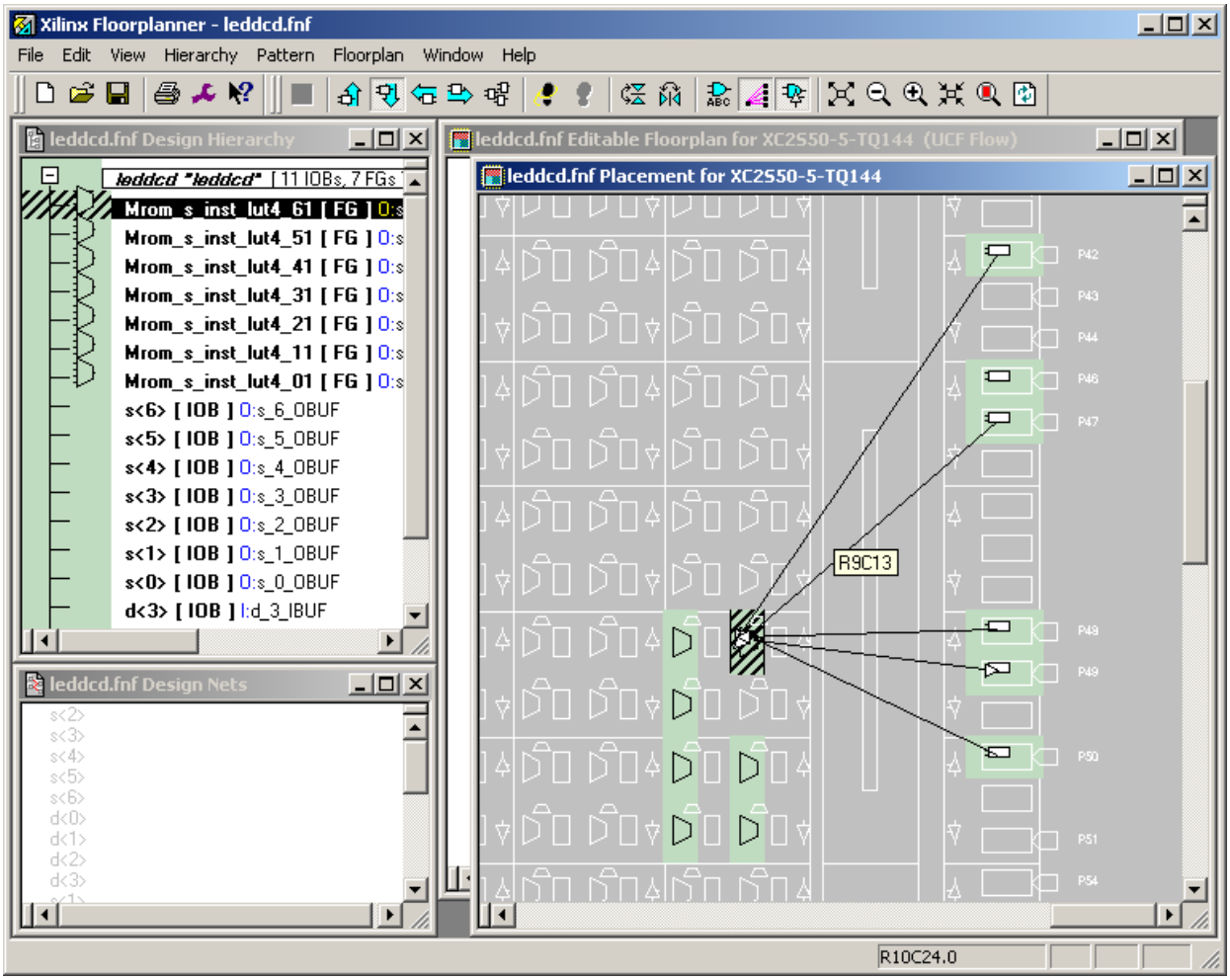
The CLBs used by the LED decoder circuit are highlighted in light-green and are clustered near the right-hand edge of the CLB array. To enlarge this region of the array, click on the  button and then draw a rectangle around the highlighted CLBs in the **Placement** pane. The enlarged view of the CLBs used by the LED decoder will appear as shown below.



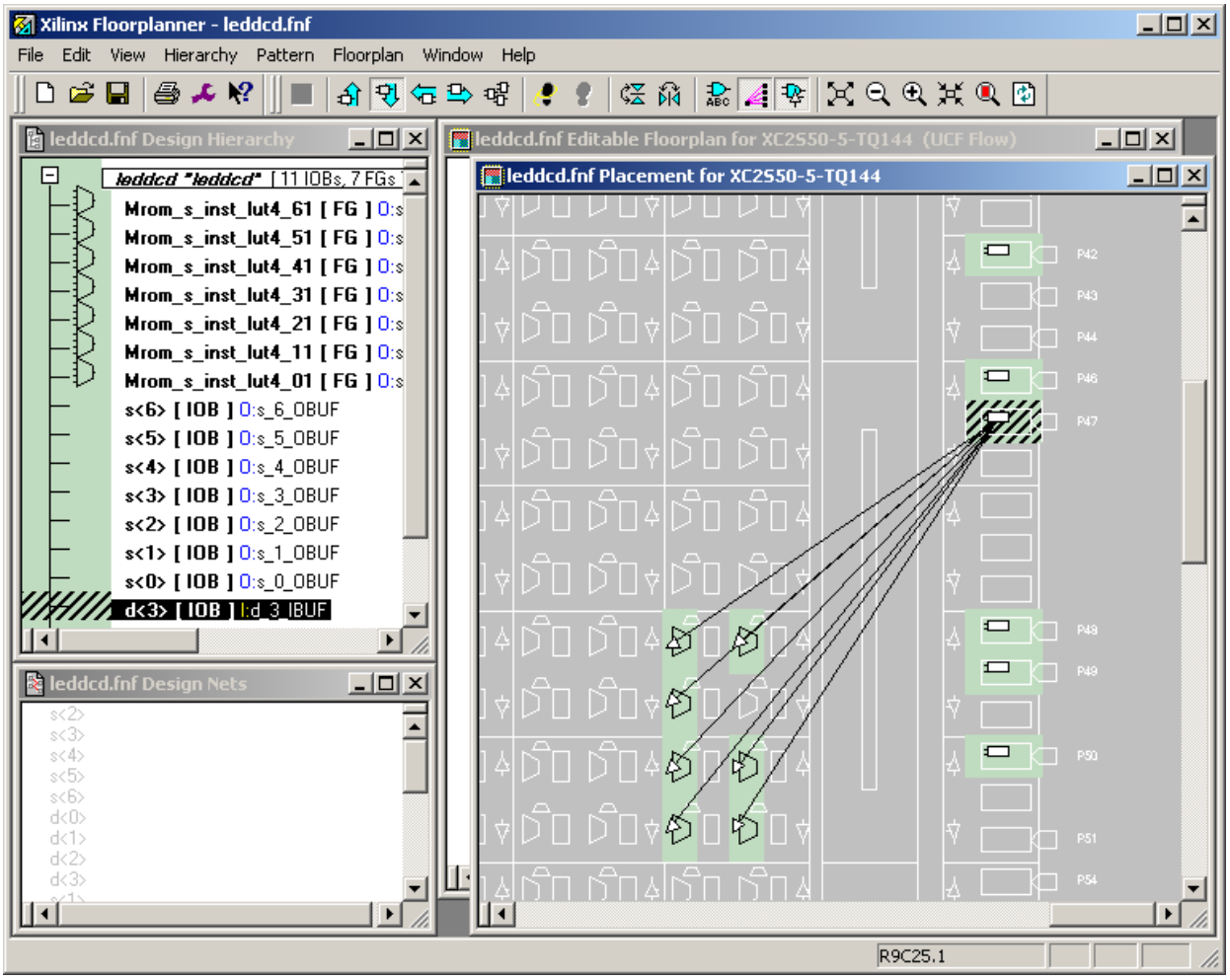
We can enable the display of the connections between I/O pins and CLBs by selecting the Edit➔Preferences menu item and then checking the boxes in the Ratsnest tab of the **Edit Preferences** window as shown below.



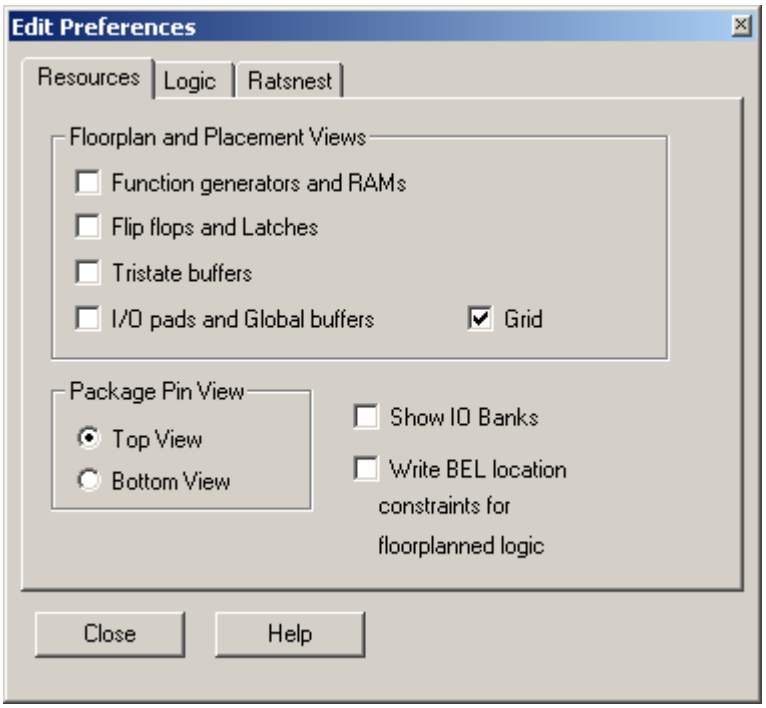
Now clicking on a CLB will highlight the nets connecting the inputs and output to the CLB.



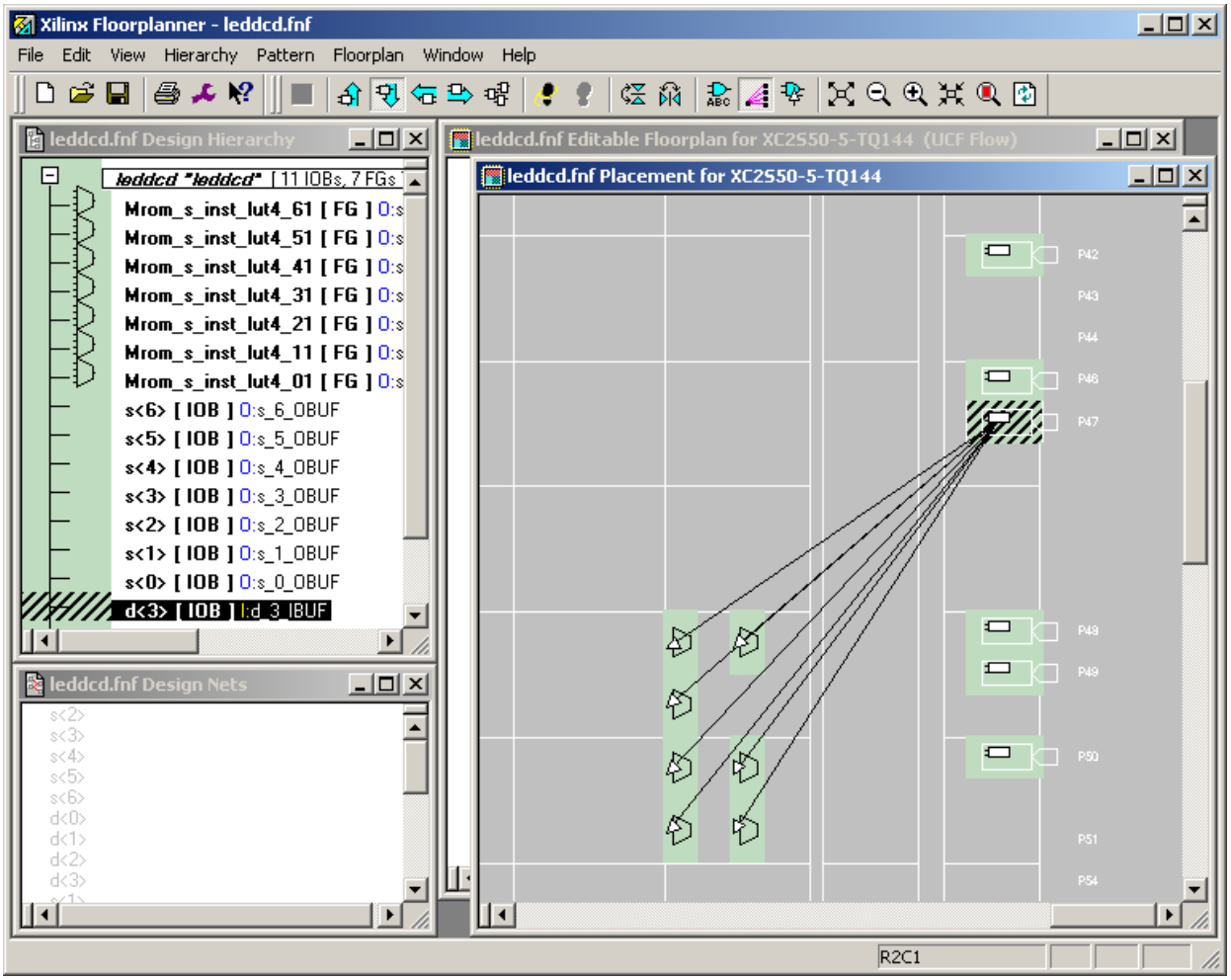
In an analogous manner, we can click on an input pin to highlight which CLBs are dependent on that input.



The **Placement** pane may be showing too many details of the FPGA internal structure. To view only the FPGA resources that are used by the design, select Edit➔Preferences and uncheck all the boxes in the Resources tab as shown below.



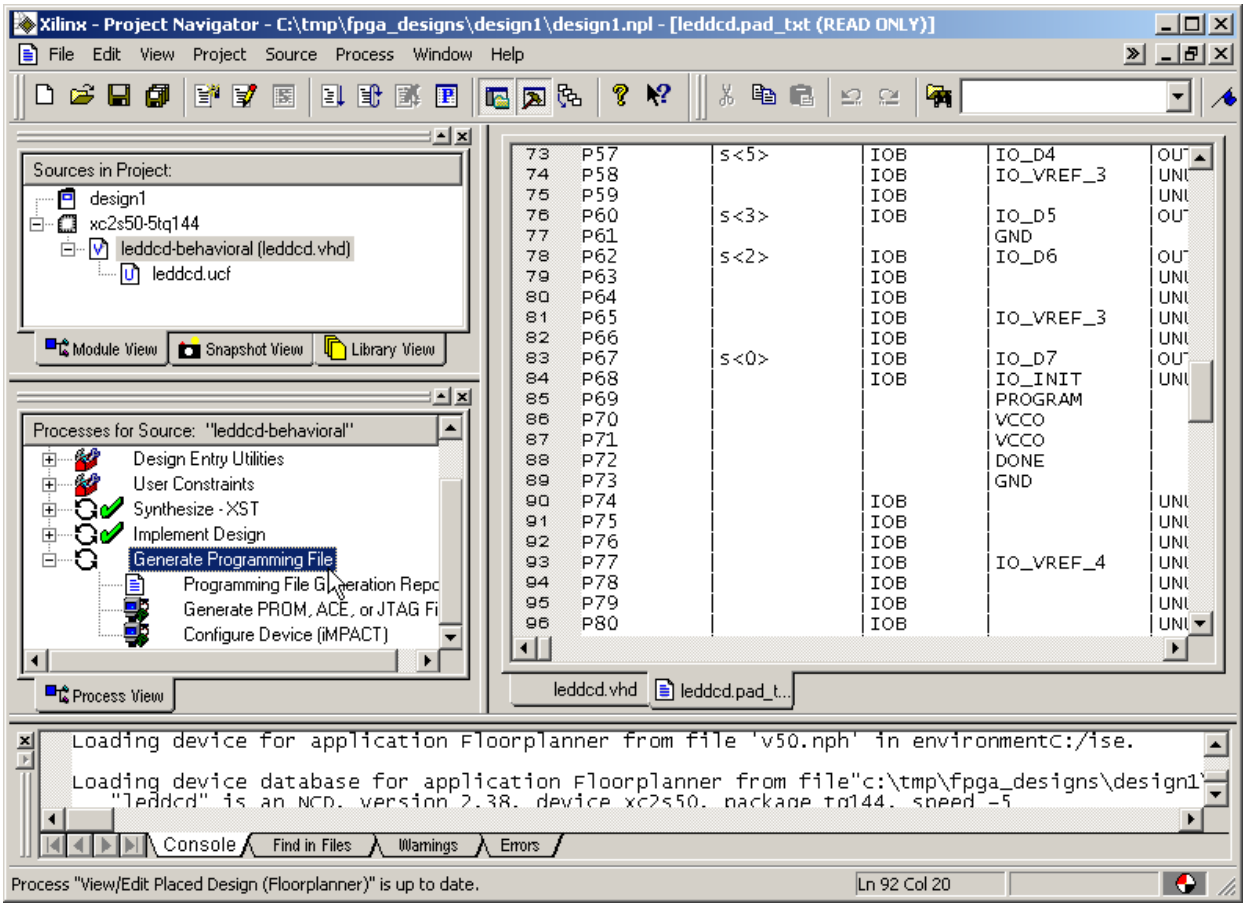
Now the **Placement** pane shows only the LUTs and I/O pins that are used in this design.



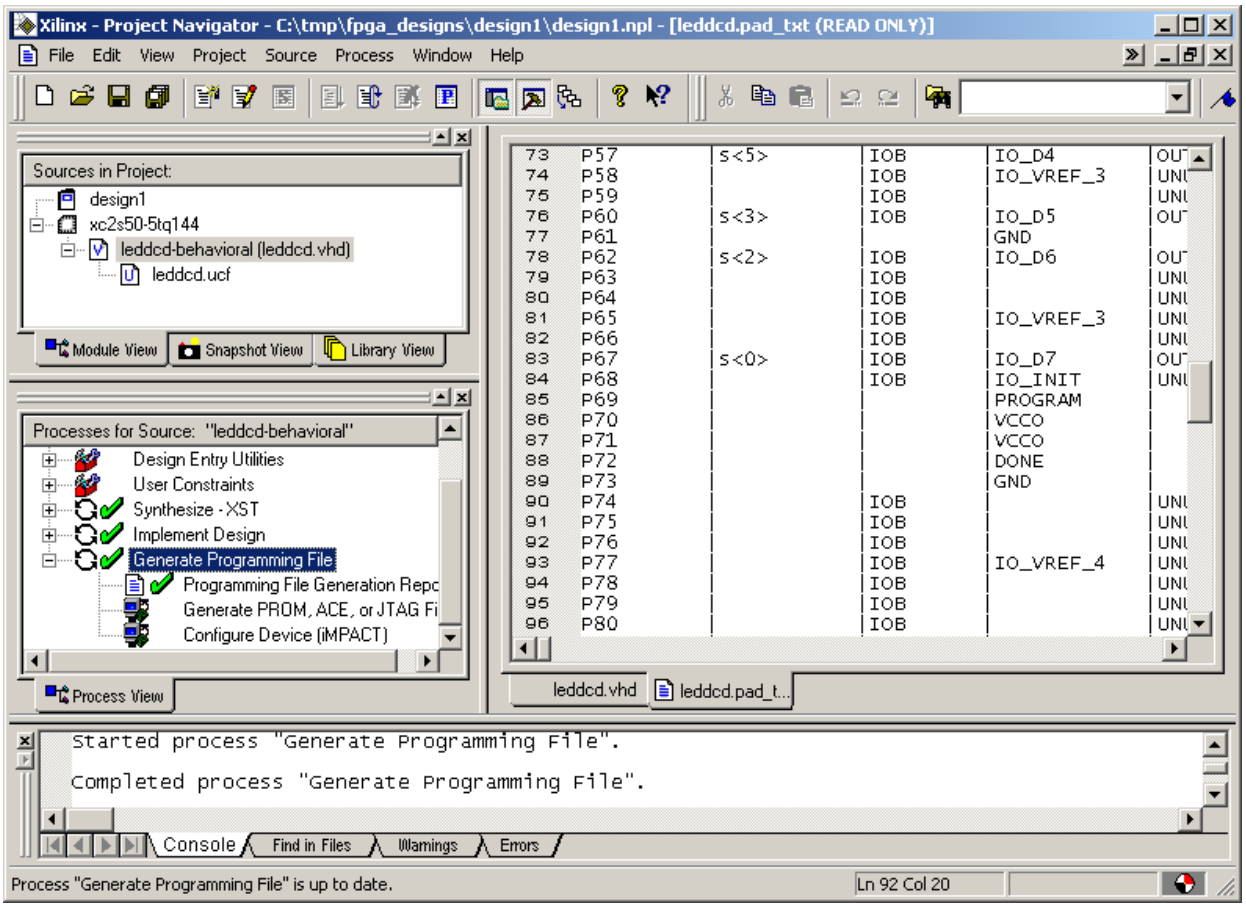
Viewing the placement of circuit elements after the place & route process can give you insights into the resource usage of certain VHDL language constructs. In addition to viewing the placement of the design, the Floorplanner can be used to re-arrange and optimize the placement. This is akin to the software technique of hand-optimizing assembly code output by a compiler. We won't do this here, but it is an option for designs which push at the extremes of the capabilities of FPGAs.

Generating the Bitstream

Now that we have synthesized our design and mapped it to the FPGA with the correct pin assignments, we are ready to generate the bitstream that is used to program the actual chip. To initiate the programmer, we highlight the leddcd object in the Sources pane and double-click on the Generate Programming File process.

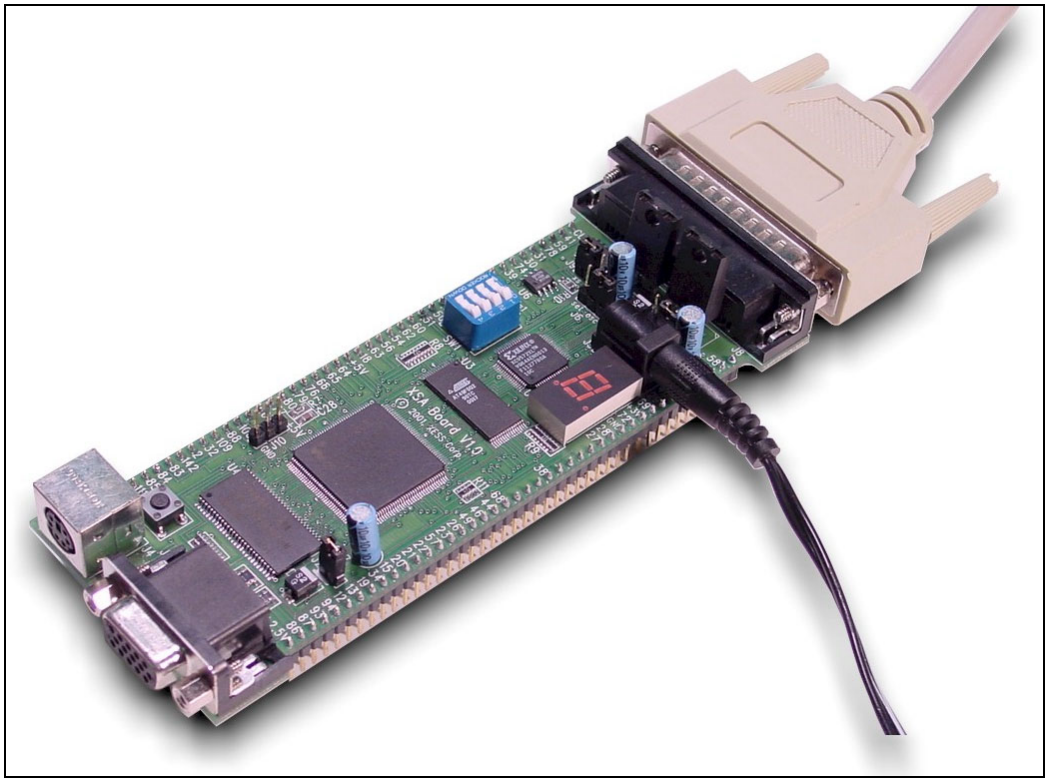


Within a few seconds, a ✔ will appear next to the Generate Programming File process and a file detailing the bitstream generation process will be created. A bitstream file named leddcd.bit can now be found in the design1 folder.



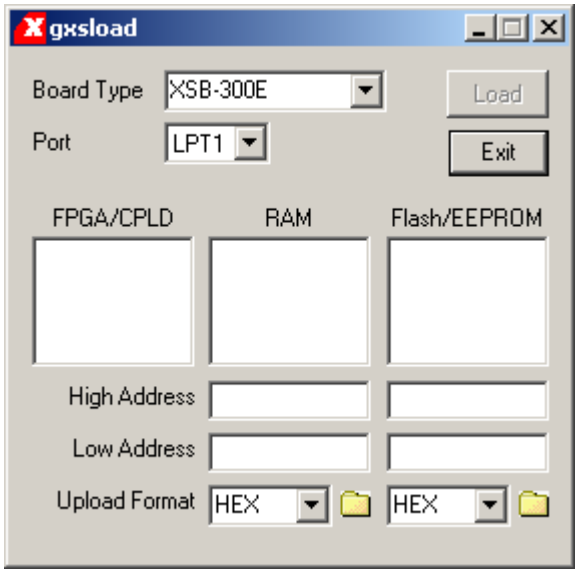
Downloading the Bitstream

Now we have to get the bitstream file programmed into the FPGA on the XSA Board. The XSA Board is powered with a 9 VDC power supply and is attached to the PC parallel port with a standard 25-wire cable as shown below.

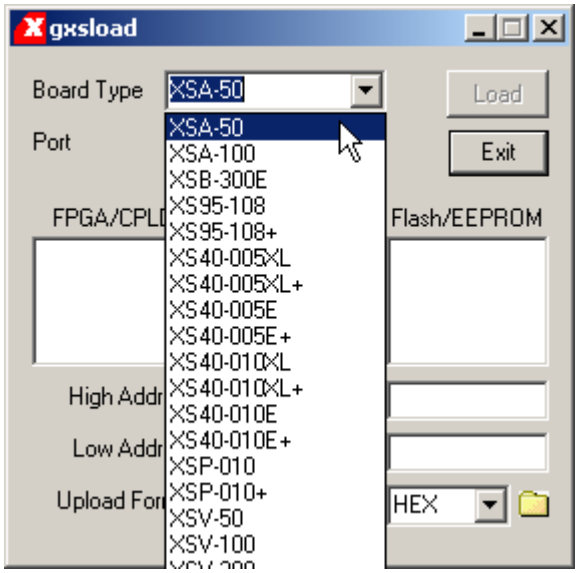


The XSA-50 Board is programmed using the gxsload utility. We double click the 

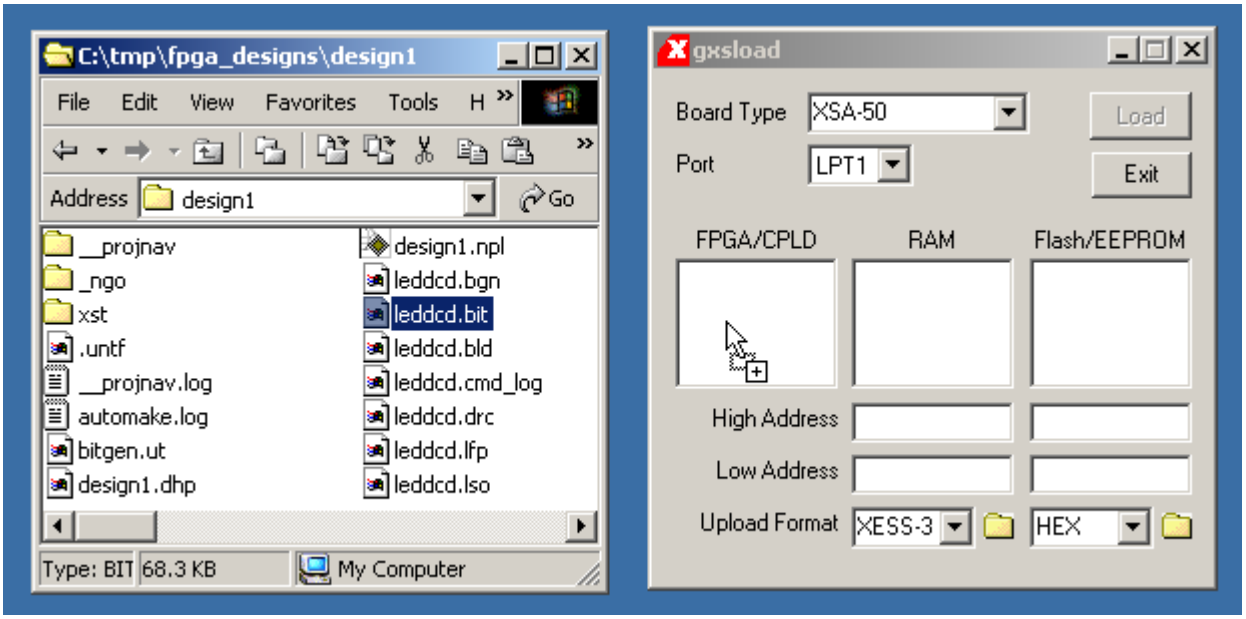
icon to bring up the **gxsload** window:



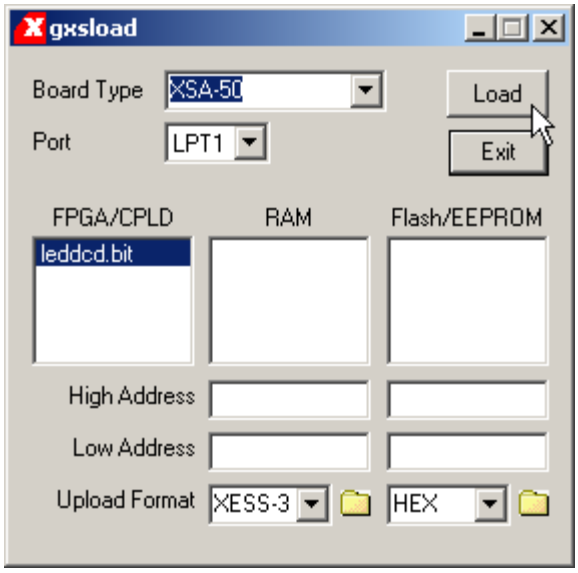
Then we click in the Board Type field and select XSA-50 from the drop-down menu since this is the board we are going to load with the bitstream.



Then we open a window that shows the contents of the folder where we have stored our LED decoder design (C:\tmp\fpga_designs\design1 in this case). We just drag-and-drop the leddcd.bit file from the **design1** window into the **gxsload** window.



Then we click on the Load button to initiate the programming of the FPGA. Downloading the leddcd.bit file to the XSA-50 takes only a few seconds.



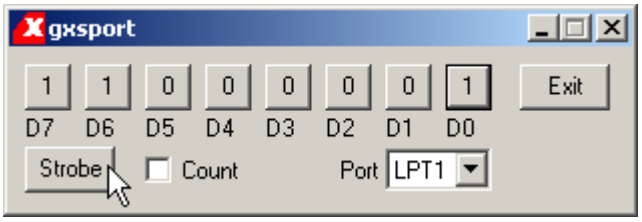
Testing the Circuit

Once the FPGA on the XSA Board is programmed, we can begin testing the LED decoder. The eight data pins of the PC parallel port connect to the FPGA through the downloading cable. We have assigned the inputs of the LED decoder to pins which are connected to the parallel

port data pins. The gxsport utility lets us control the logic values on these pins. By placing different bit patterns on the pins, we can observe the outputs of the LED decoder through the seven-segment LED on the XSA Board.

Double-clicking the 

icon initiates the gxsport utility. The **d0**, **d1**, **d2**, and **d3** inputs of the LED decoder are assigned to the pins controlled by the D0, D1, D2, and D3 buttons of the **gxsport** window. To apply a given input bit pattern to the LED decoder, click on the D buttons to toggle their values. Then click on the Strobe button to send the new bit pattern to the pins of the parallel port and on to the FPGA. For example, setting (D3,D2,D1,D0) = (1,1,1,0) will cause E to appear on the seven-segment LED of the XSA Board.



If you check the Count box in the **gxsport** window, then each click on the Strobe button increments the eight-bit value represented by D7-D0. This makes it easy to check all sixteen input combinations.

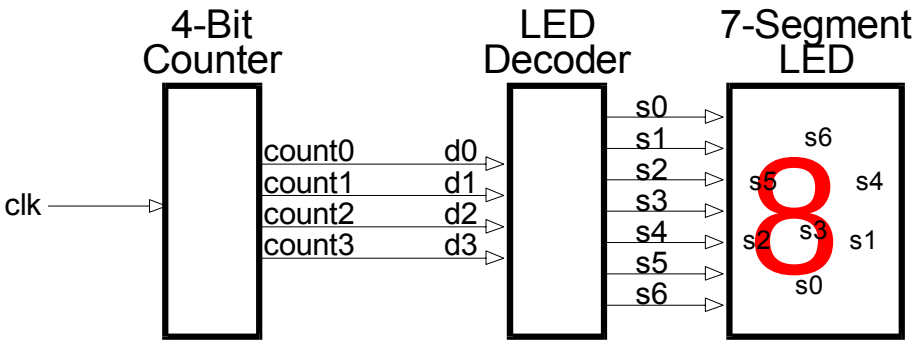
NOTE: Bit D7 of the parallel port controls the /PROGRAM pin of the FPGA. Do not set D7 to 0 or you will erase the configuration of the FPGA. Then you will have to download the bitstream again to continue testing your design.

4

Hierarchical Design

A Displayable Counter

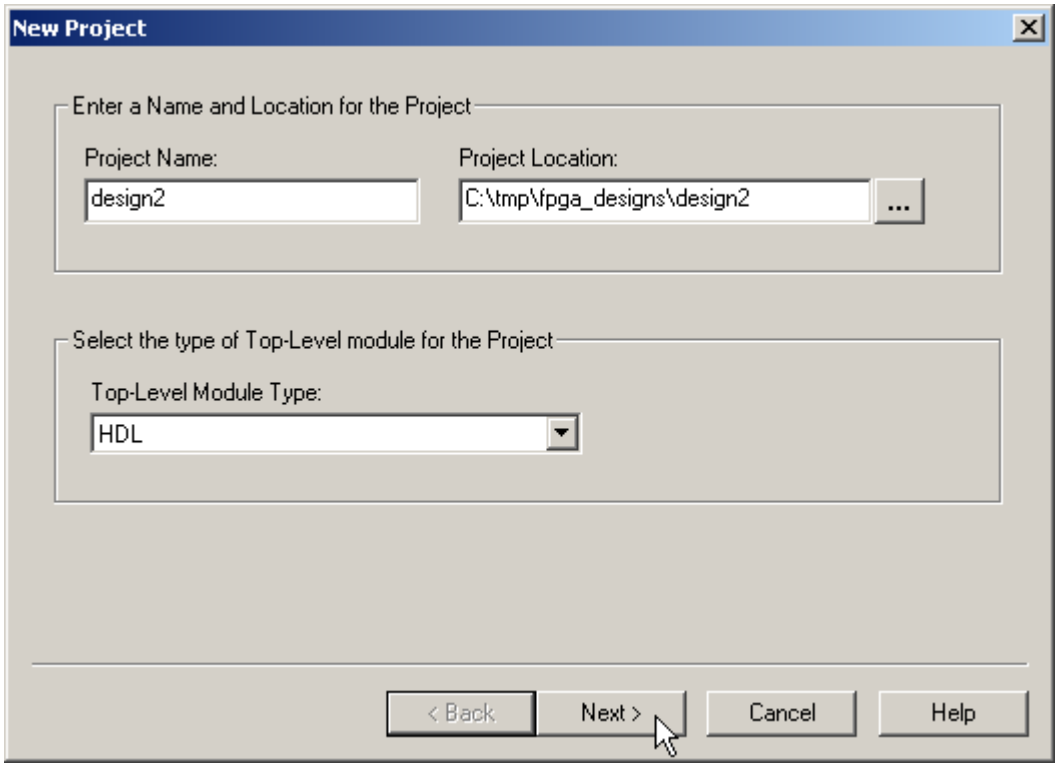
We went through a lot of work for our first FPGA design, so we will reuse it in this design: a four-bit counter whose value is displayed on a seven-segment display. The counter will increment on a rising edge of the clock. The four-bit output from the counter enters the LED decoder whereupon the counter value is displayed on the seven-segment LED. A high-level diagram of the displayable counter looks like this:



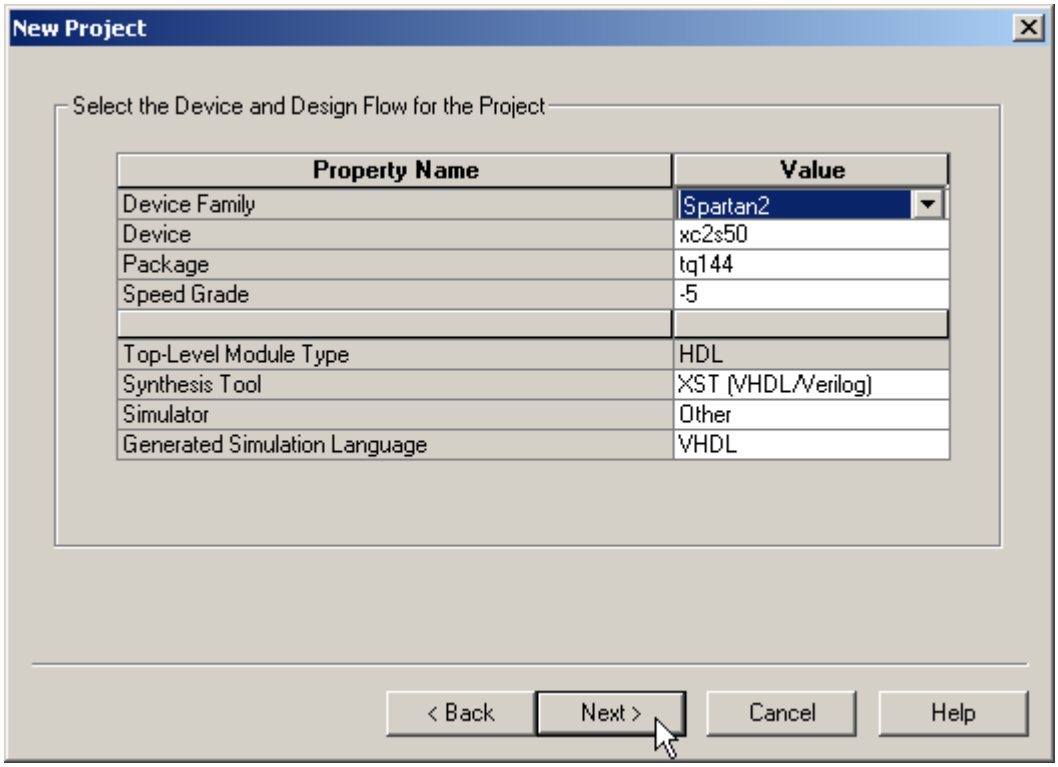
This design is hierarchical in nature. The LED decoder and counter are modules which are interconnected within a top-level module.

Starting a New Design

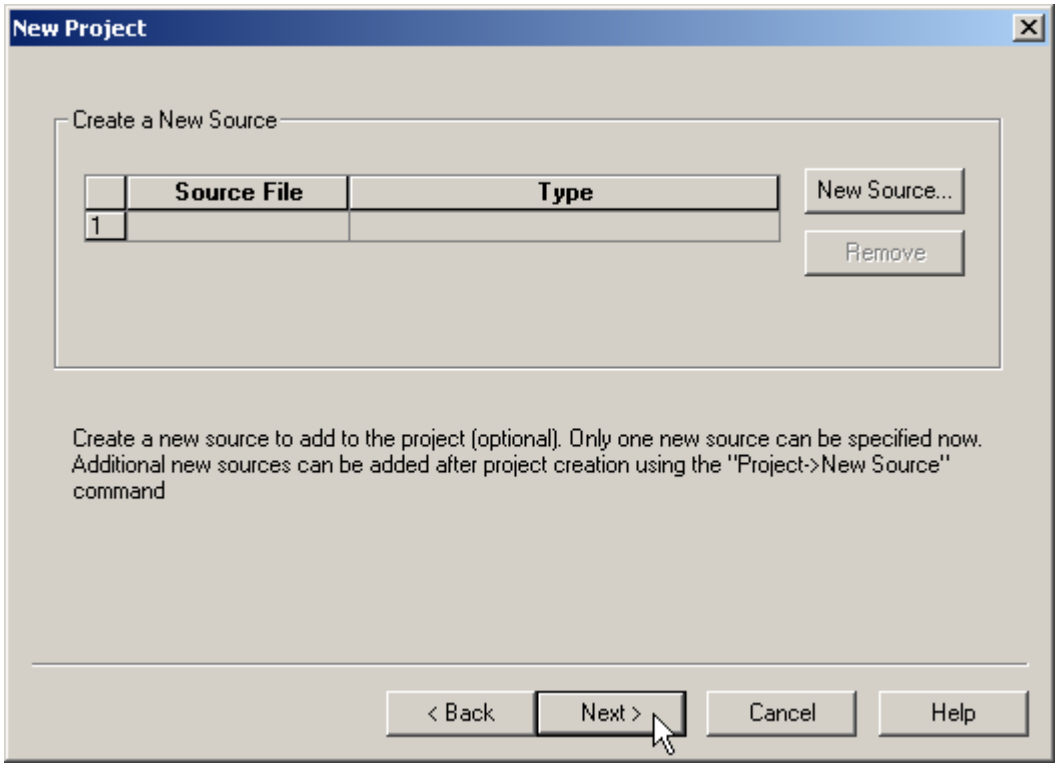
We can start a new project using the File➔New Project... menu item. We name the project ***design2*** and store it in the same folder as the previous design: C:\tmp\fpga_designs. Then we click on the Next button.



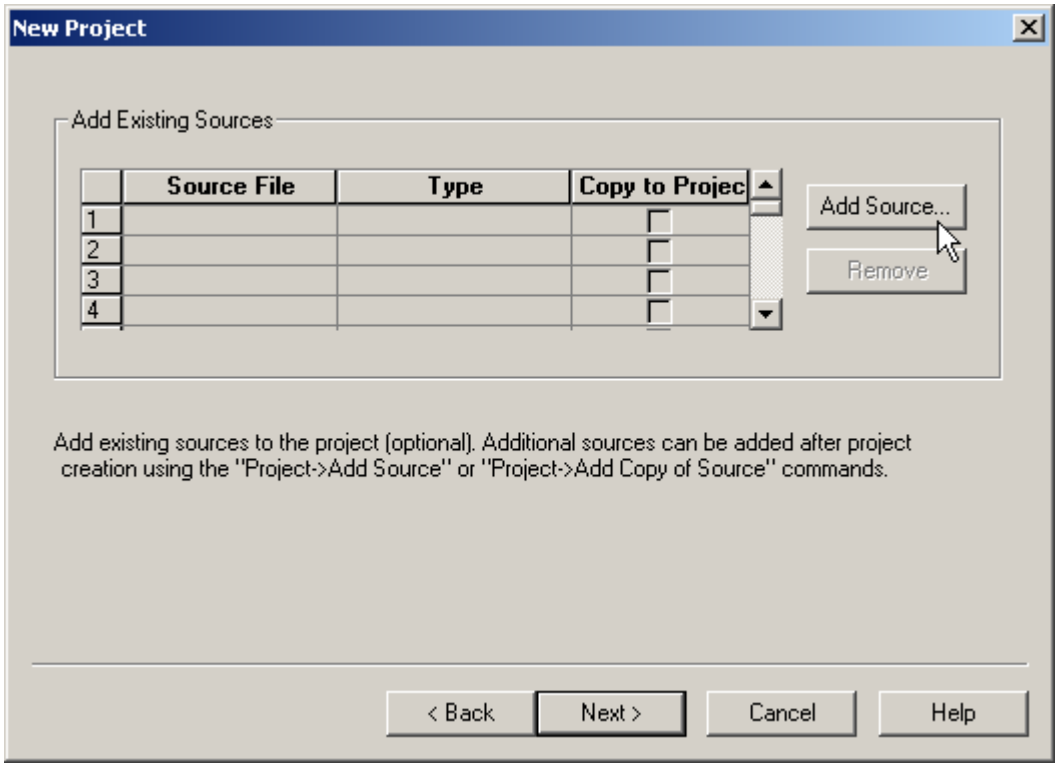
We are still targeting the same FPGA on the XSA-50 Board, so the other properties in the **New Project** window retain the same values we set in the previous project.



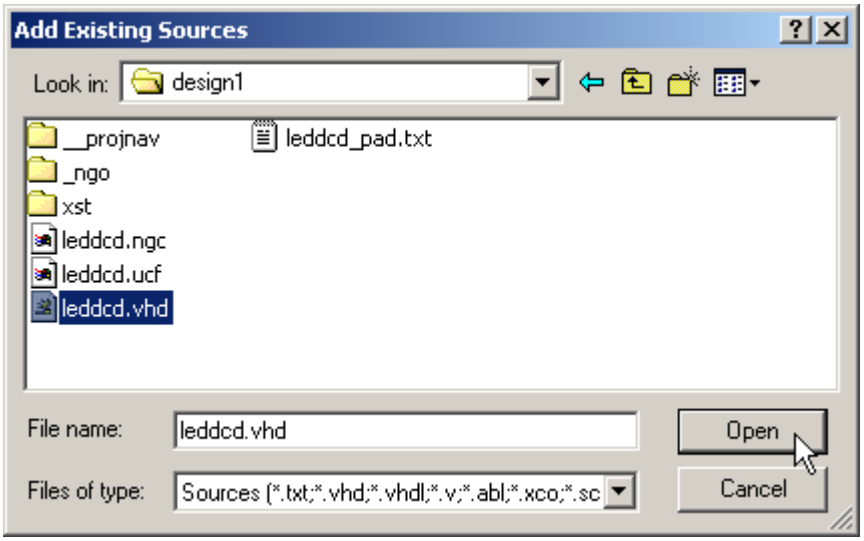
We won't add any new source files at the moment, so just click on the Next button.



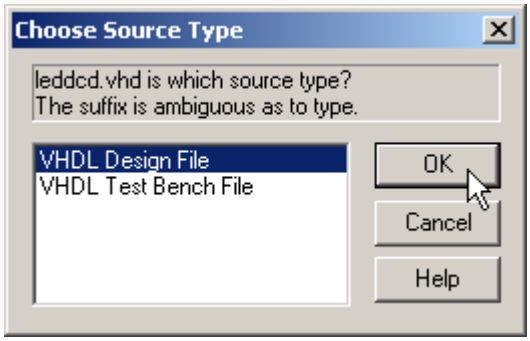
The next window allows us to add existing source files to our new project. We want to re-use the LED decoder from our previous design, so click on the Add Source... button.



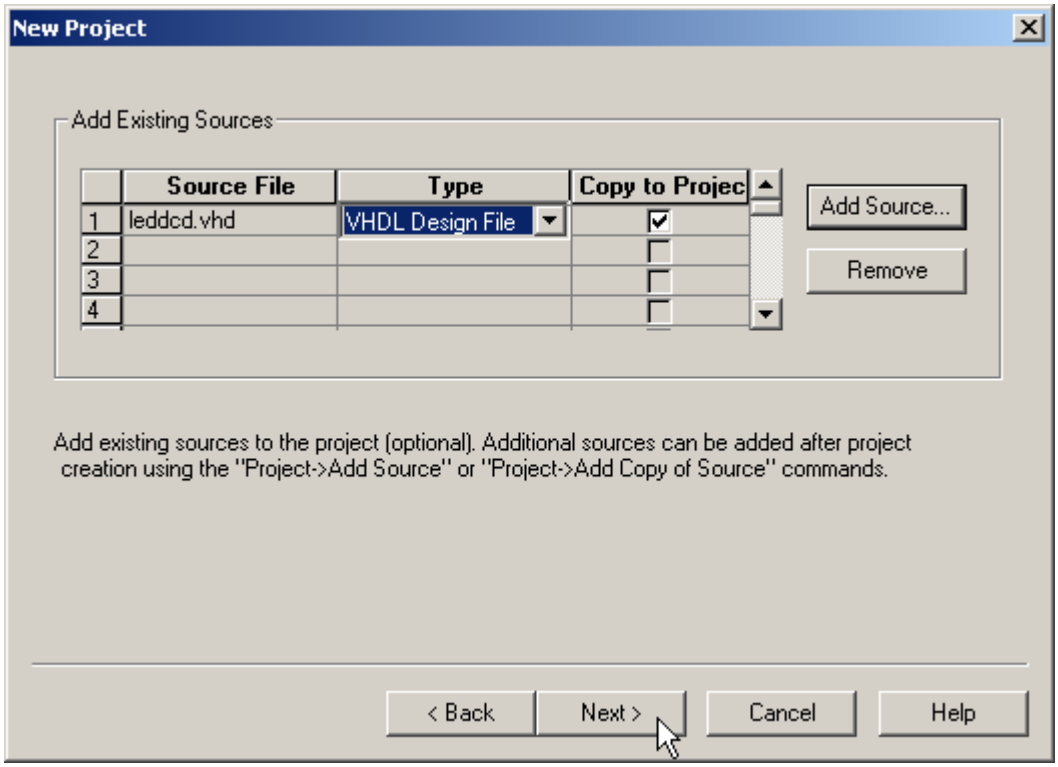
The **Add Existing Sources** window appears and we move to the C:\tmp\fpga_designs\design1 folder. Then we highlight the leddcd.vhd file that contains the VHDL source code for the LED decoder.



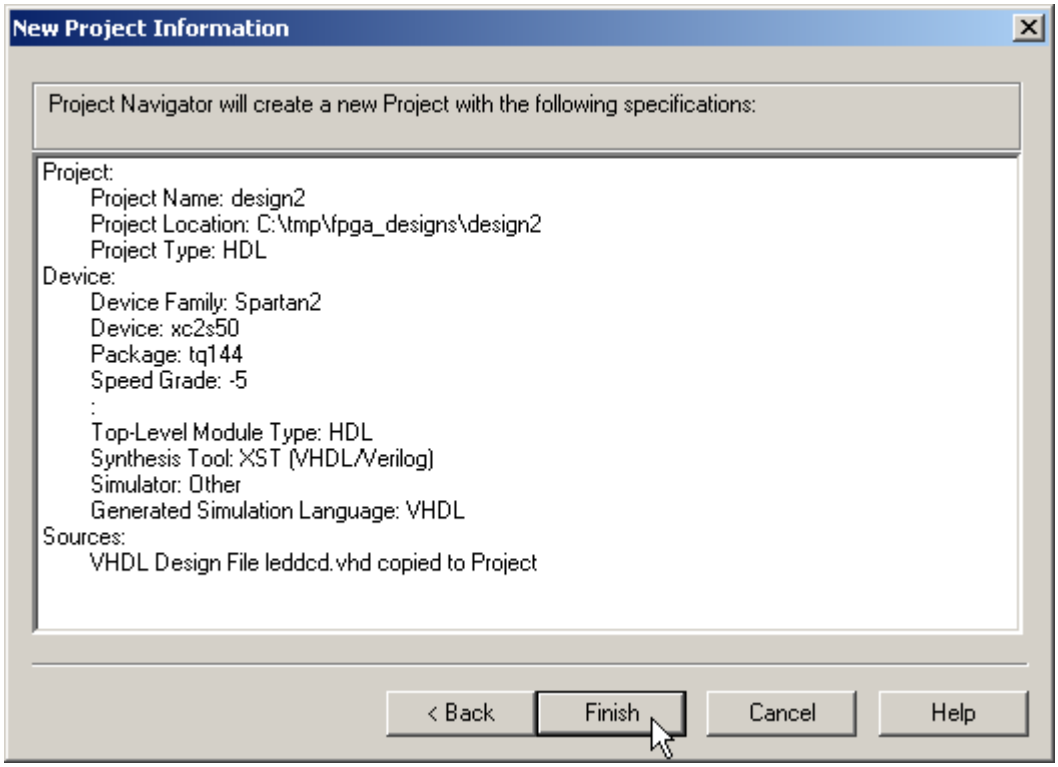
After clicking on Open, a window appears that asks us the type of file we are adding to the project. Select VHDL Design File and click the OK button.



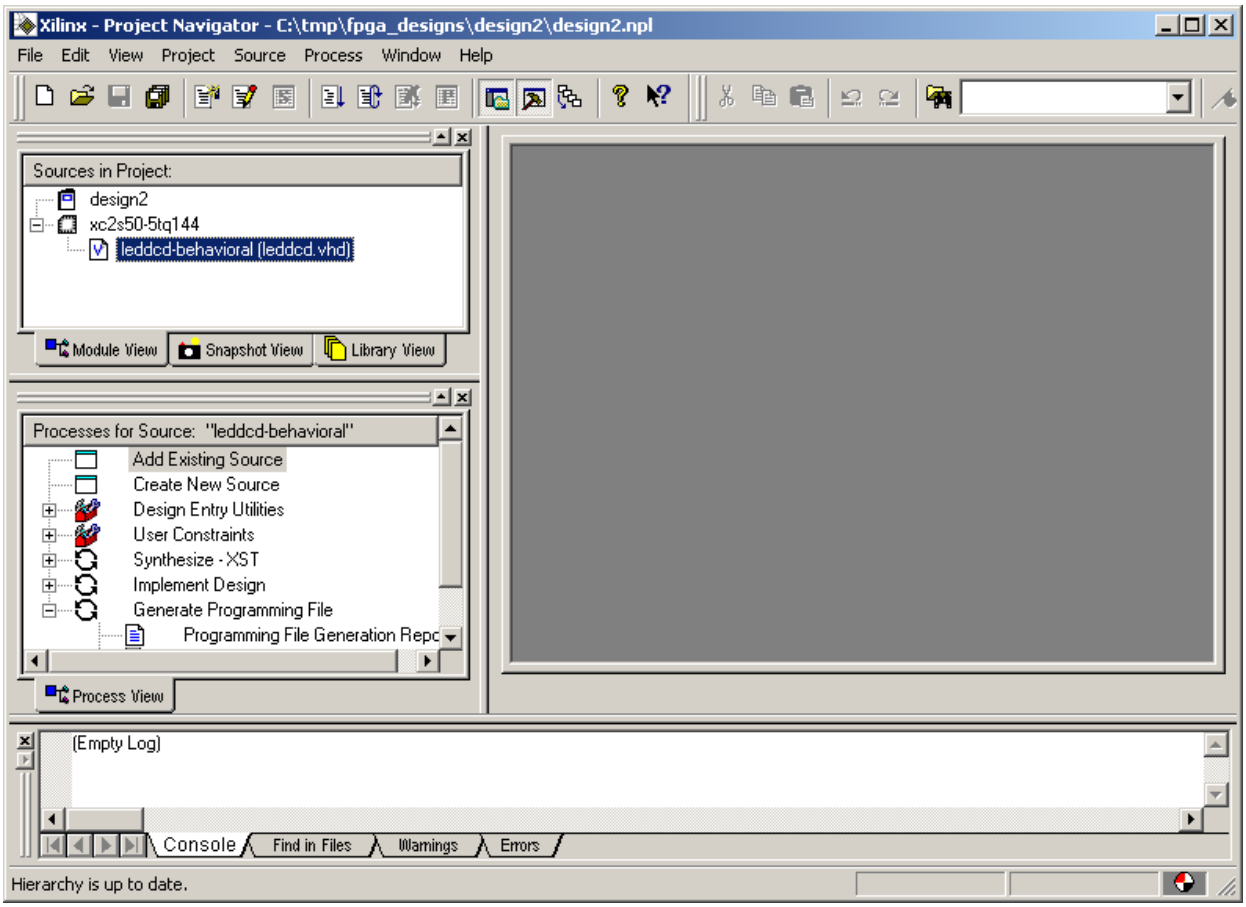
The **New Project** window now shows that a copy of the leddcd.vhd file will be added to the project. This is the only existing file we want to add, so click on the Next button to move on to the next window.



The final screen shows the pertinent information for the new project. Click on the Finish button to complete the creation of the project.

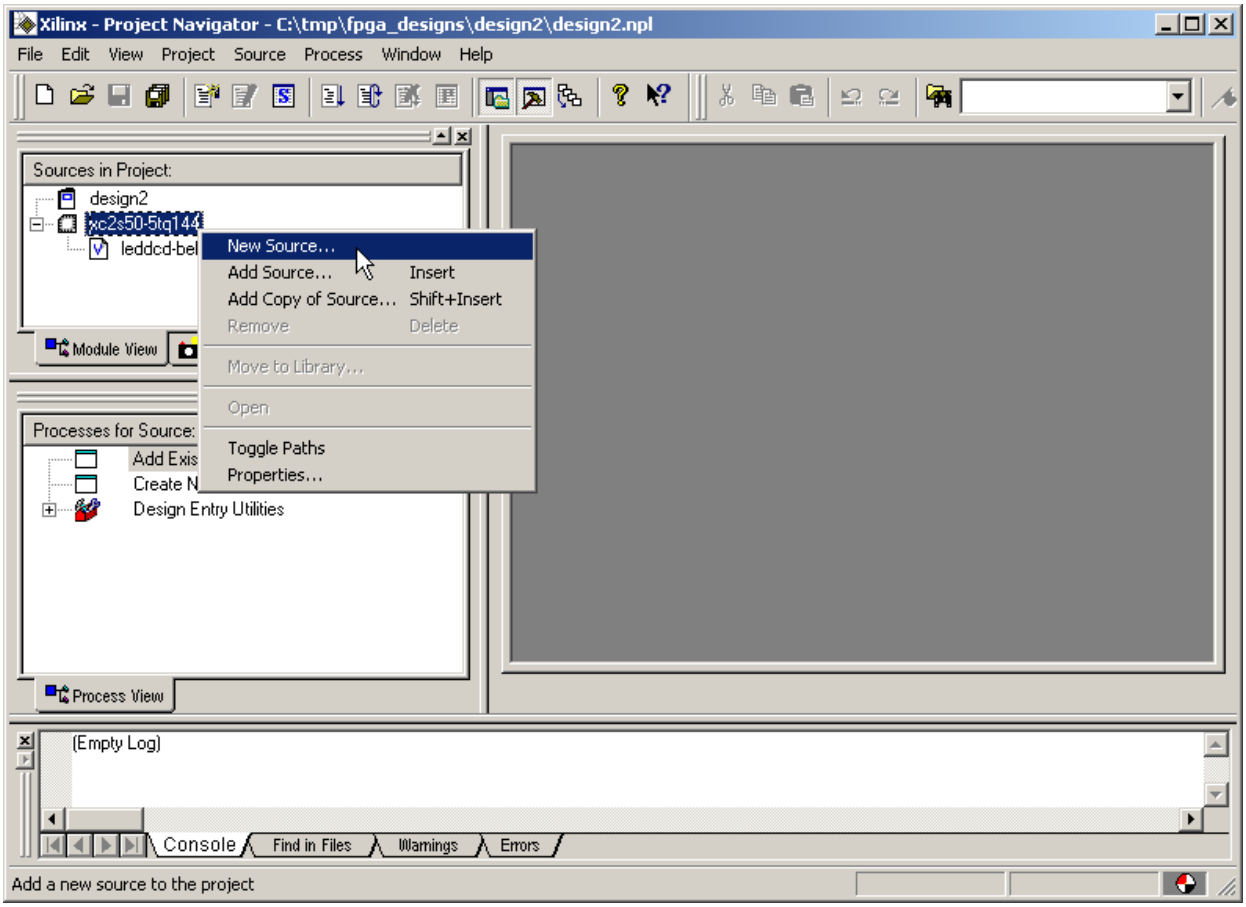


Once we click on Finish in the **New Project** window, the **Project Navigator** window appears as shown below. The project currently contains only the single leddcd.vhd source file.

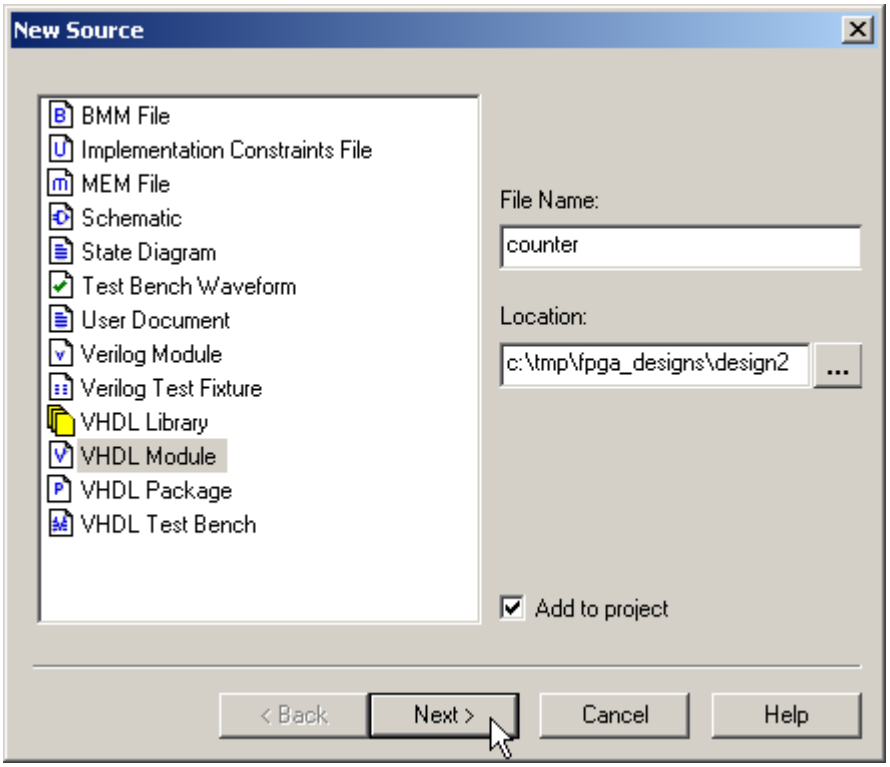


Adding a Counter

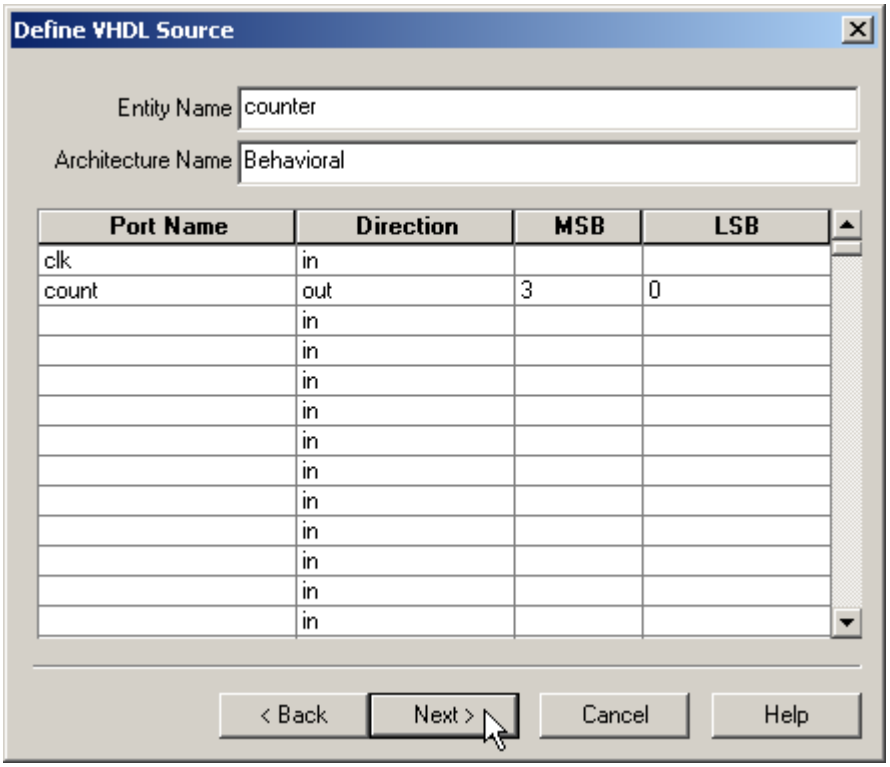
Now we have to add the counter to our design. We don't have a counter module yet, so we have to build one with VHDL. Right-click on the XC2S50-5TQ144 object and select New Source... from the pop-up menu.



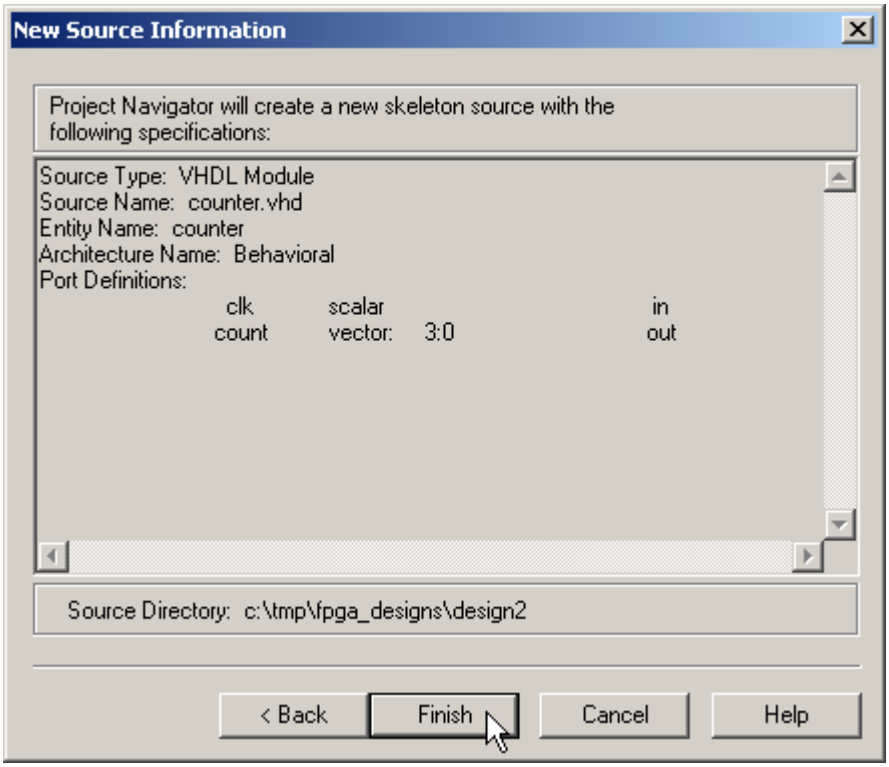
As in the previous example, we are prompted for the type of file we want to add to the project. Once again, we select the VHDL Module menu item. Then we type `counter` into the File Name field and click on the Next button.



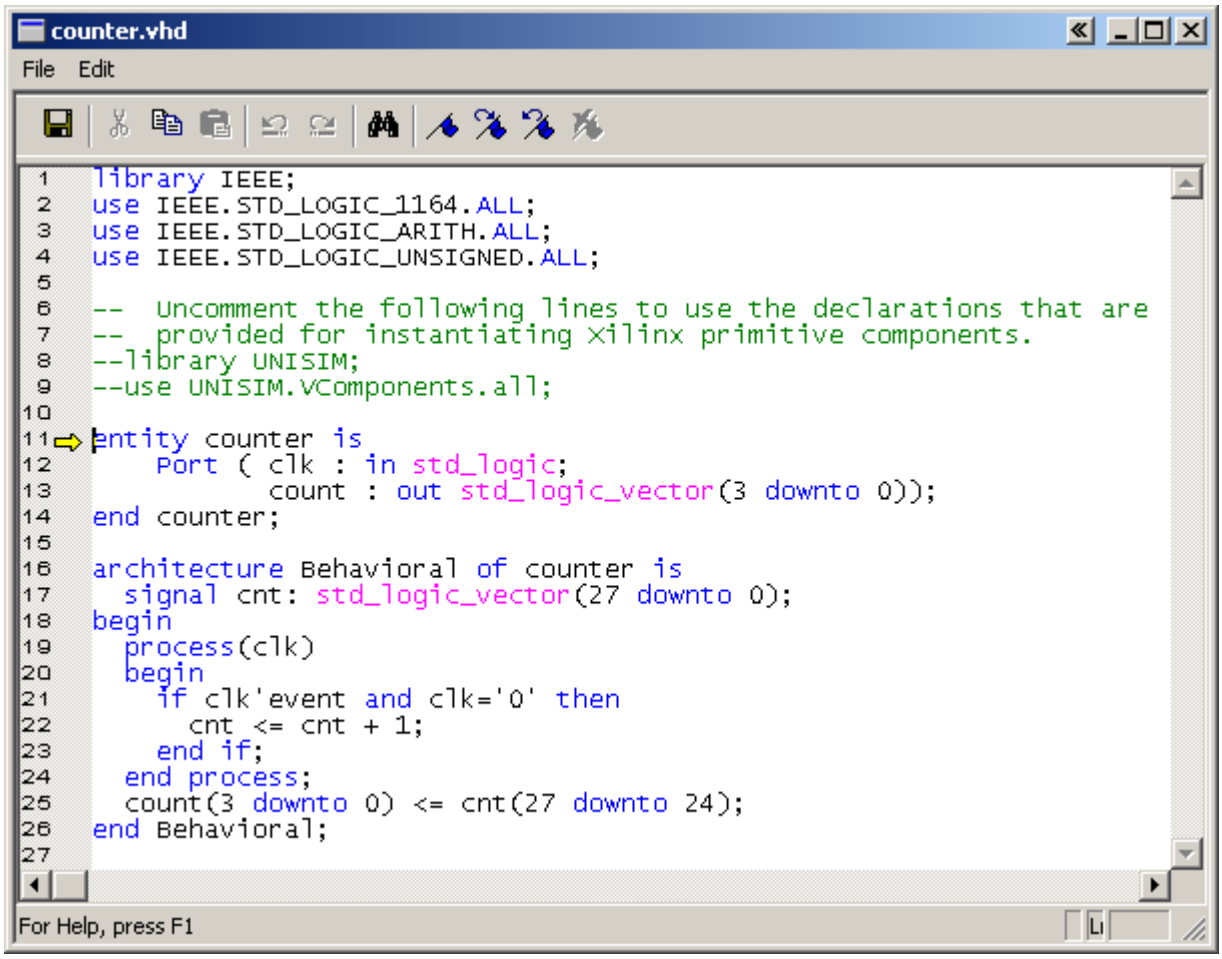
Then we declare the inputs and outputs for the counter in the **Define VHDL Source** window as shown below. The **counter** module receives a single input, **clk**, and has a four-bit output bus, **count**, which outputs the current counter value.



Click on Next and check the information about the module.



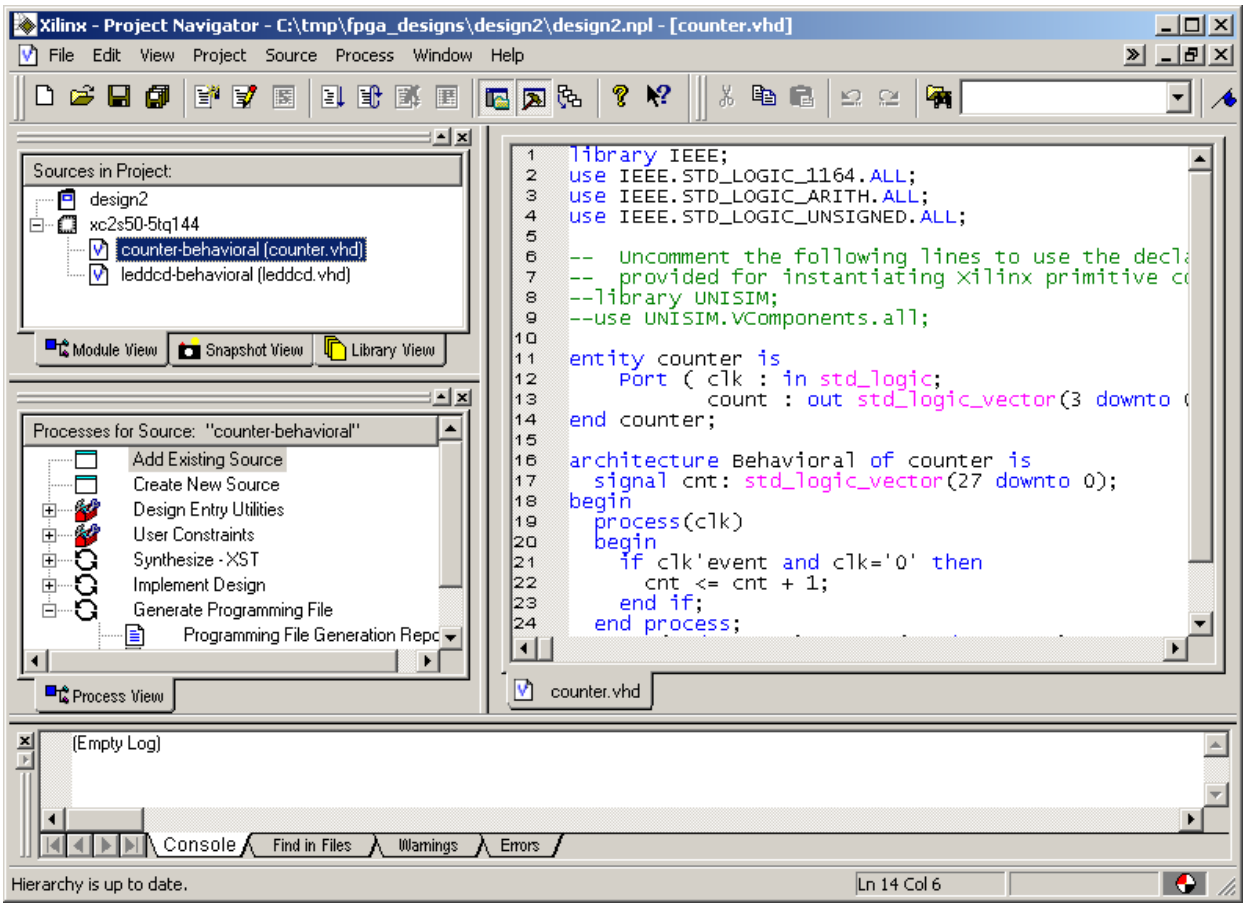
After clicking Finish in the **New Source Information** window, we are presented with a VHDL skeleton for the counter. We flesh-out the skeleton as follows:



Line 17 declares a 28-bit signal, **cnt**, that is the current value of the counter. The process on lines 19-24 controls when the counter increments. The condition clause of line 21 is only true when the value on the **clk** input goes from 1 to 0. Then the statement on line 22 replaces the value in **cnt** with its incremented value. (We can use the high-level addition operator instead of having to describe a 28-bit adder because on line 4 we have linked into the ieee.std_logic_unsigned.all package that supports unsigned arithmetic.) Finally, line 25 places the upper four bits of the current counter value onto the outputs of the module.

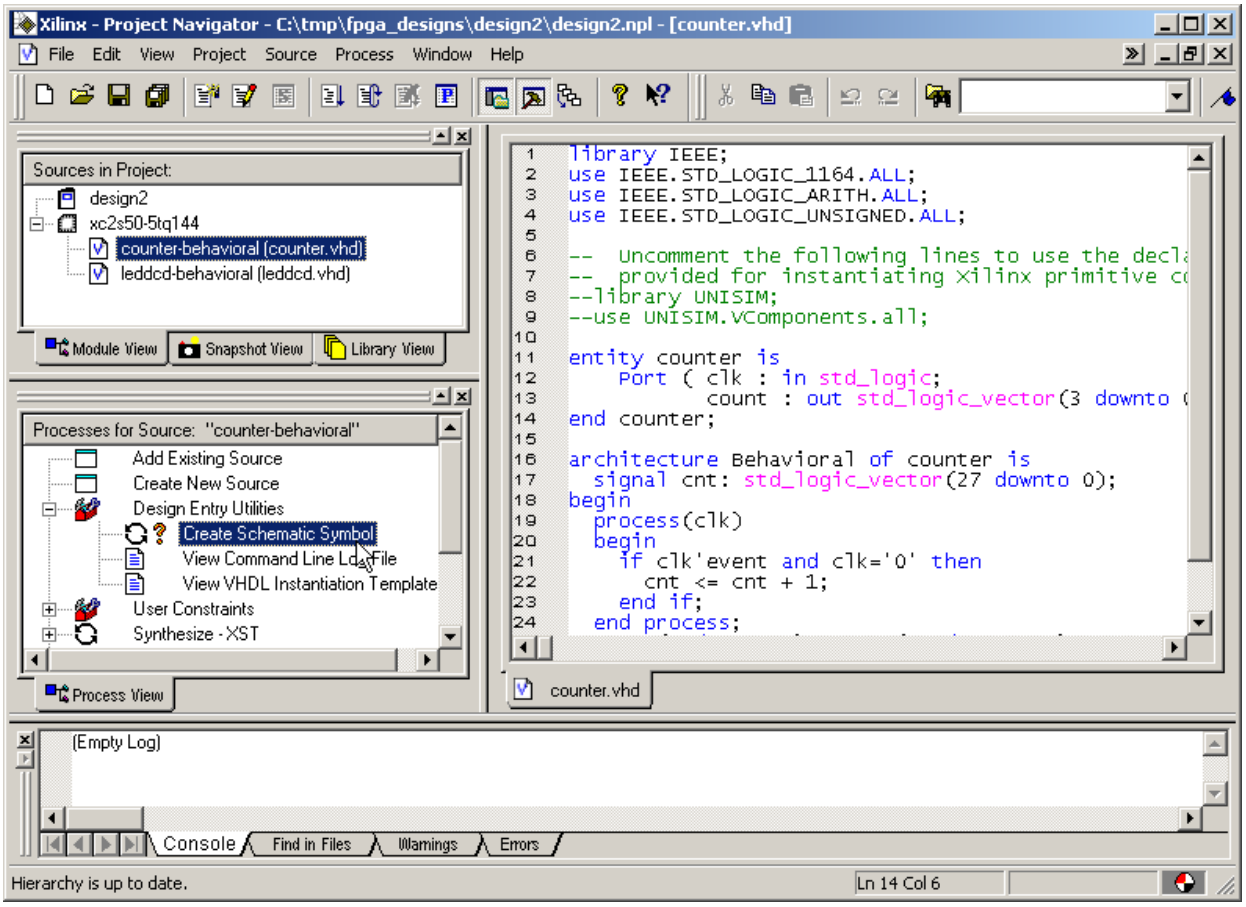
Why are we building a 28-bit counter and using only the upper four bits? The counter will be driven by the oscillator on the XSA-50 Board which has a default frequency of 50 MHz. The LED display would be changing much too quickly to see at this frequency. By connecting the LED decoder to the upper four bits of the 28-bit counter, the display will only change once in every 2^{24} clock cycles. So the LED display will change every 2^{24} / (50 x 10^6) = 0.336 seconds which is slow enough to be read.

After entering the VHDL shown above and saving it, we see that the counter module has been added to the Sources pane of the **Project Navigator** window.

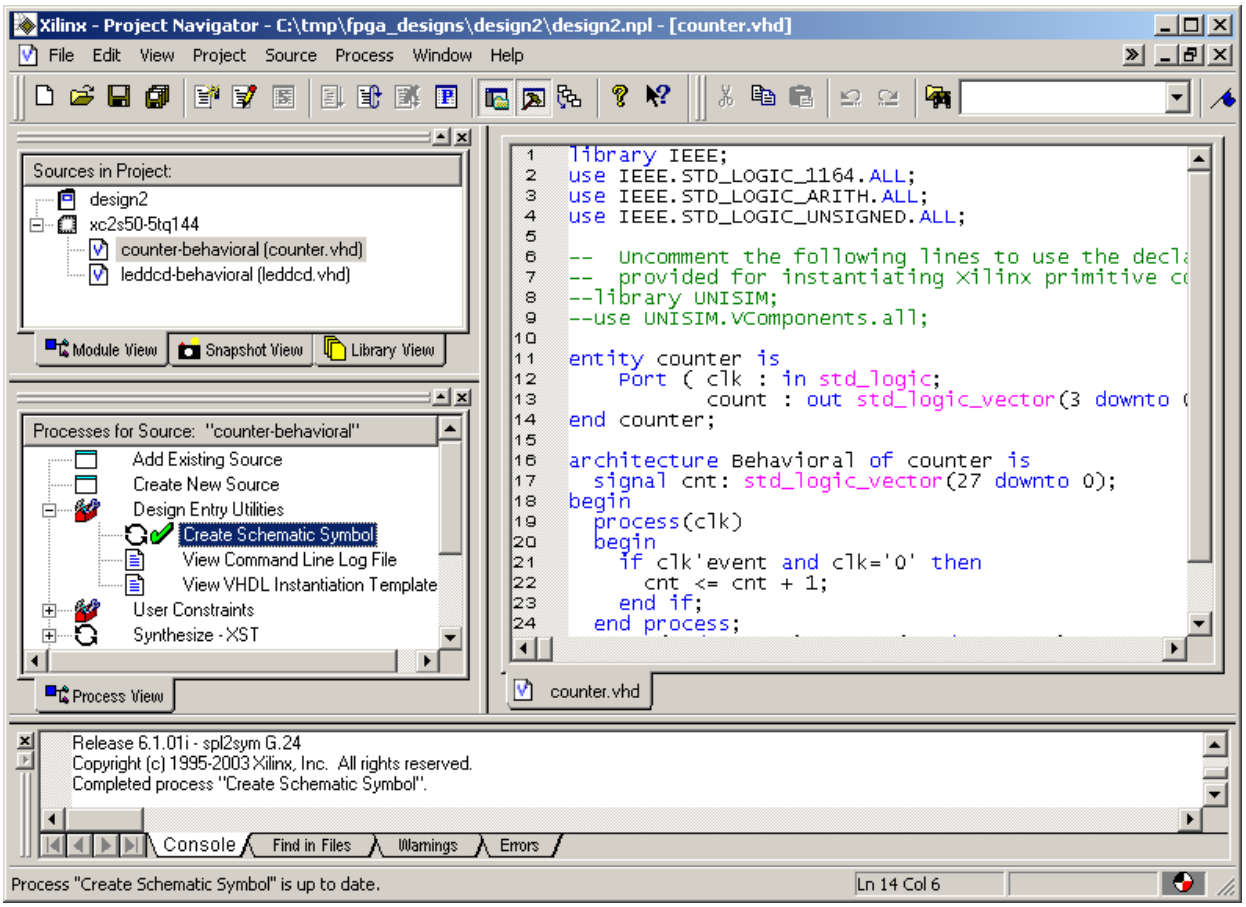


Tying Them Together

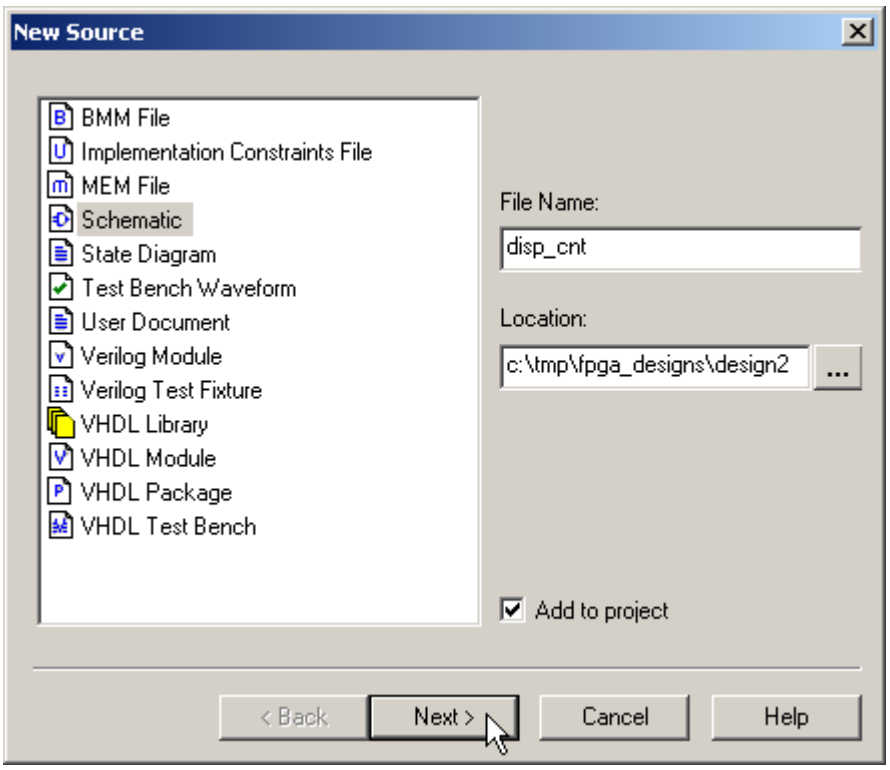
We have the LED decoder and the counter, but now we need to tie them together to build the displayable counter. We will do this by connecting the counter to the LED decoder in a top-level schematic. Before we can do this, we have to create schematic symbols for both the counter and LED decoder VHDL modules. To create the counter schematic symbol, highlight the counter object in the Sources pane and then double-click the Create Schematic Symbol process.



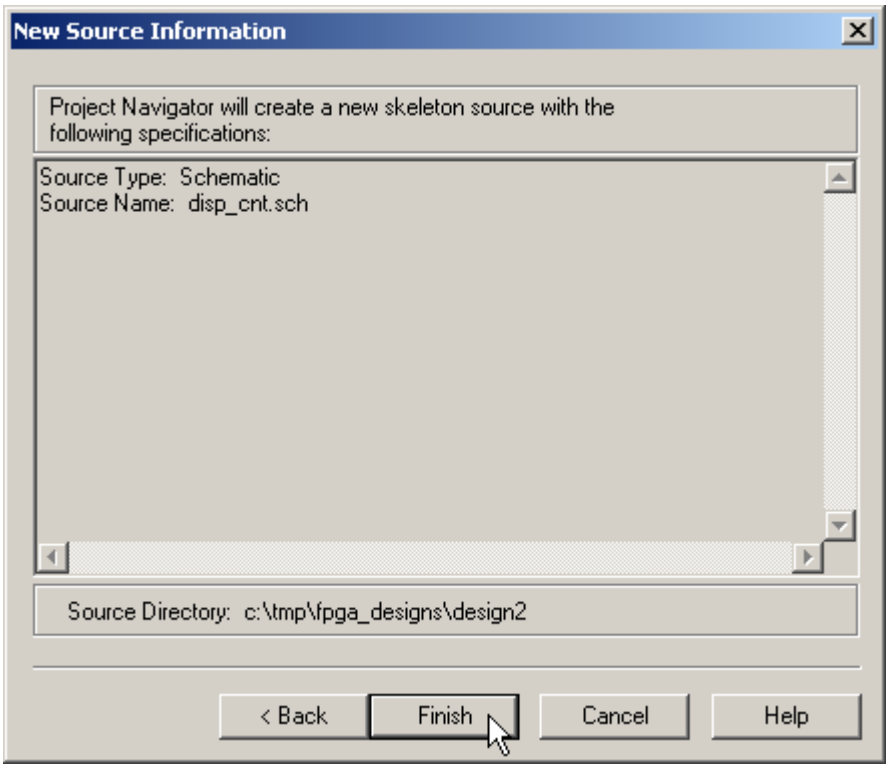
A ✔ will appear next to the Create Schematic Symbol process after the counter symbol is created. Repeat this procedure to create the schematic symbol for the LED decoder.



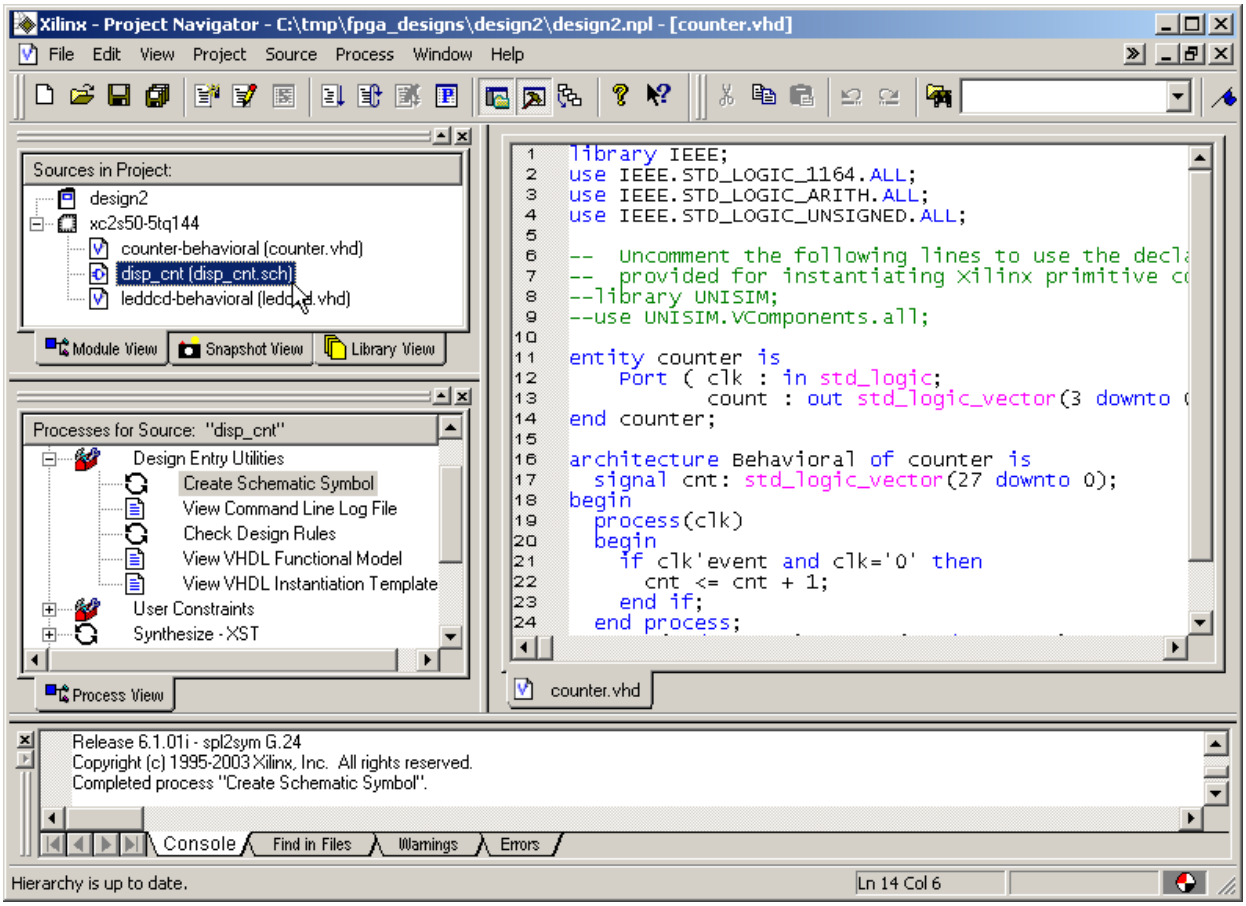
Once the schematic symbols for the lower-level modules are built, we can add the top-level schematic to the project. Right-click on the XC2S50-5TQ144 object and select New Source... from the pop-up menu. Then highlight the Schematic entry in the **New Source** window and name the schematic **disp_cnt**. Then click on Next.



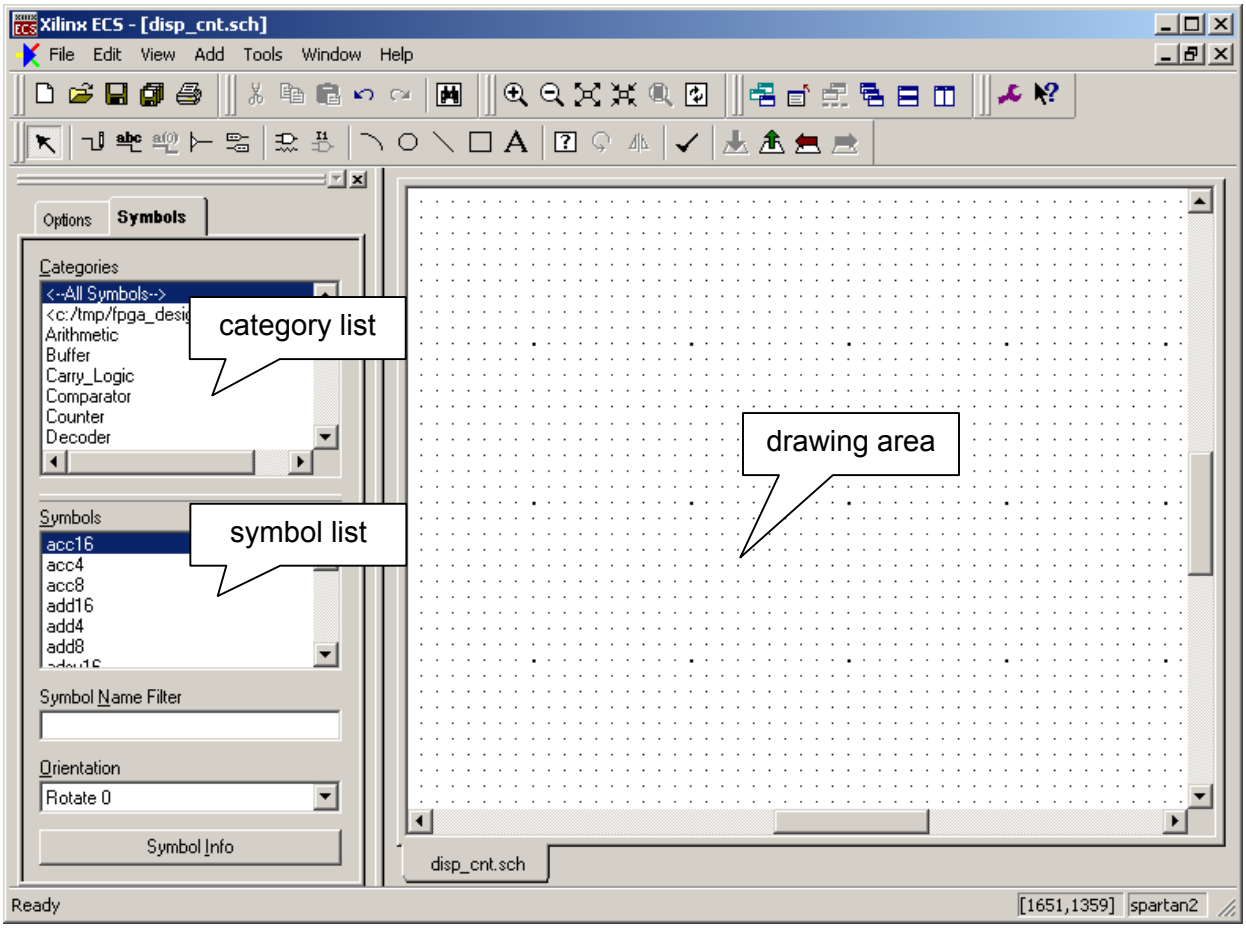
There is very little to do when setting-up a schematic, so just click on the Finish button in the **New Source Information** window that appears.



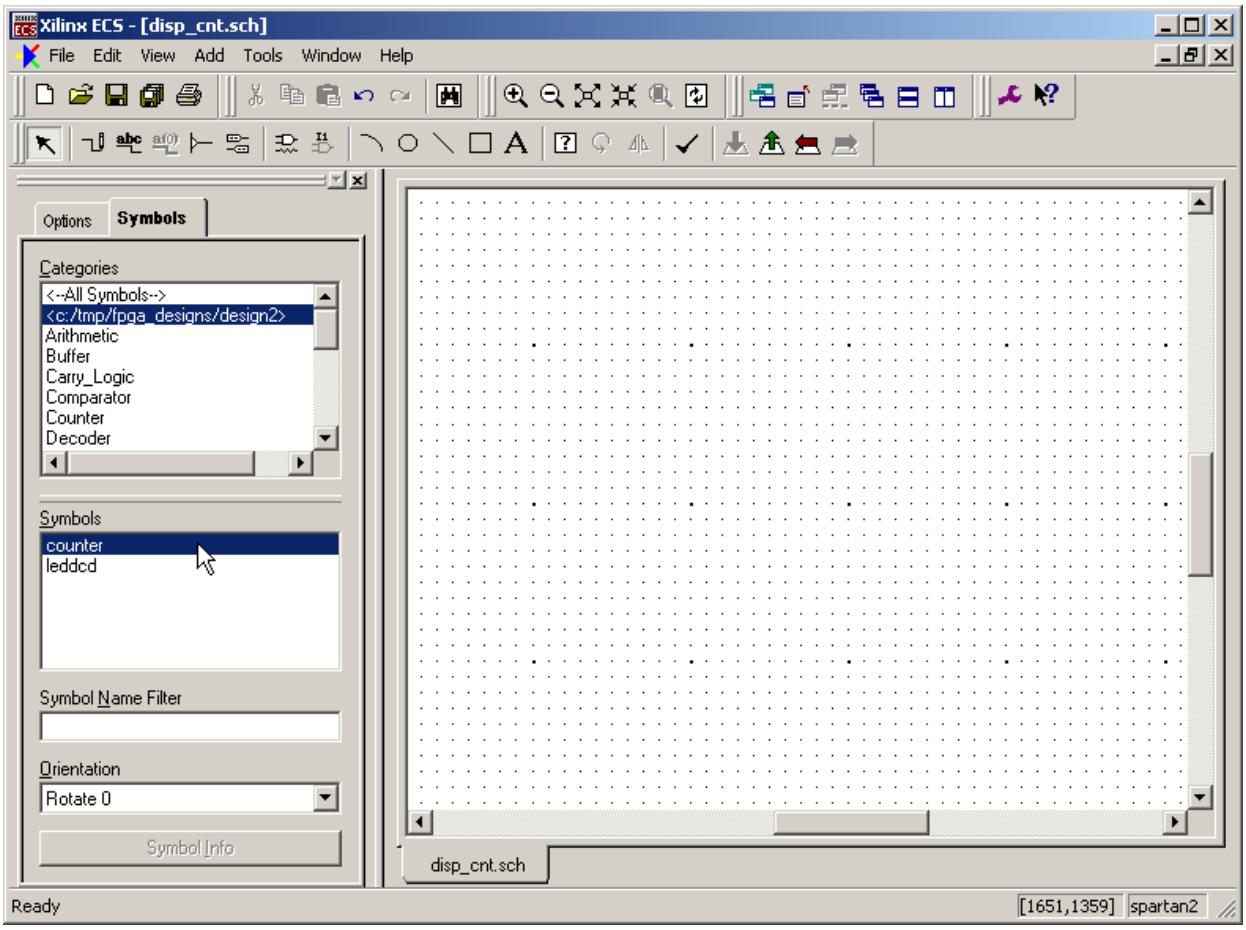
Now the disp_cnt schematic object has been added to the Sources pane. We can double-click it to begin creating the schematic, but a schematic editor window should open automatically once the file is created.



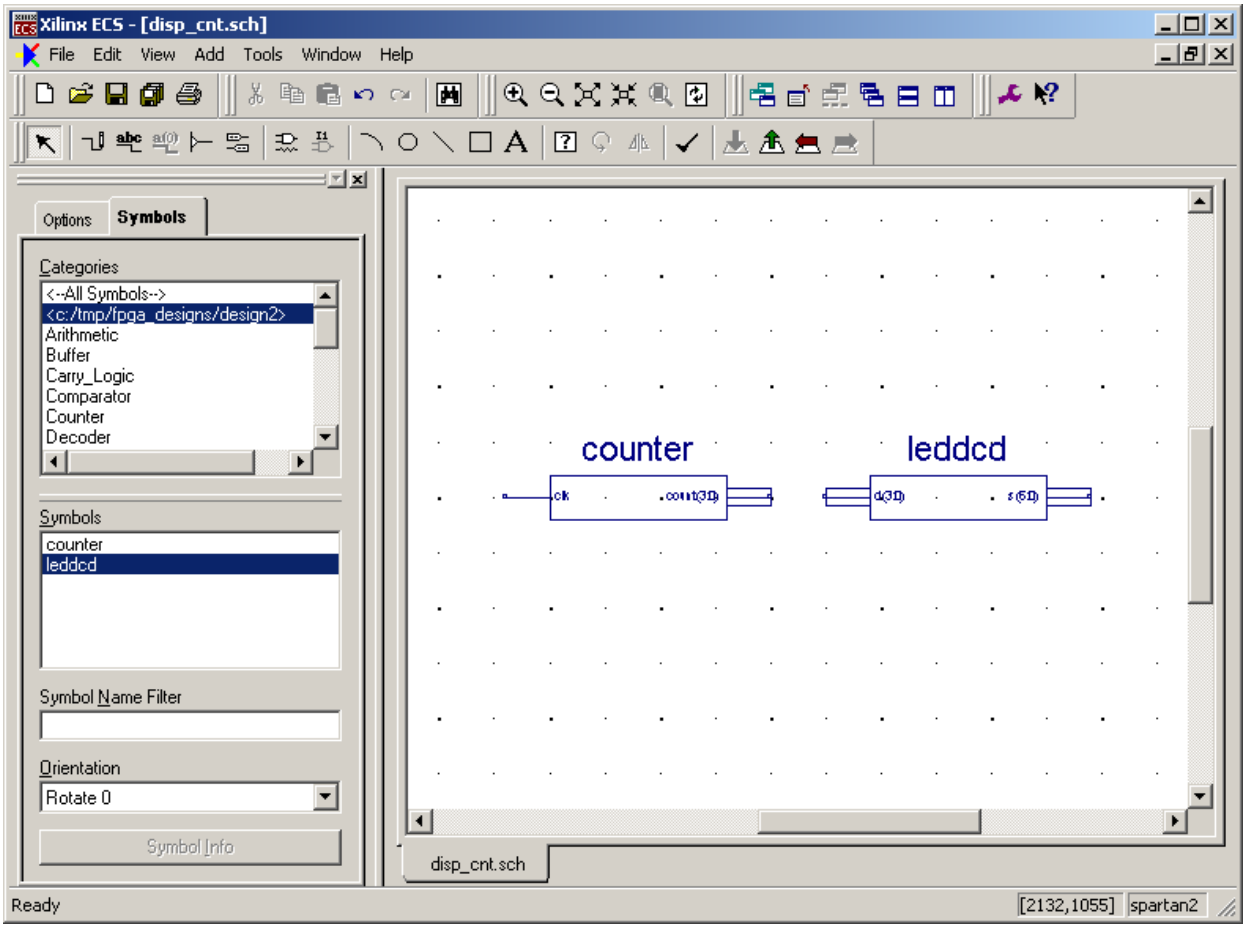
The schematic editor window has a drawing area and a list of categories for various logic circuit elements that can be used in a schematic. Below that is the list of symbols for circuit elements in a highlighted category.



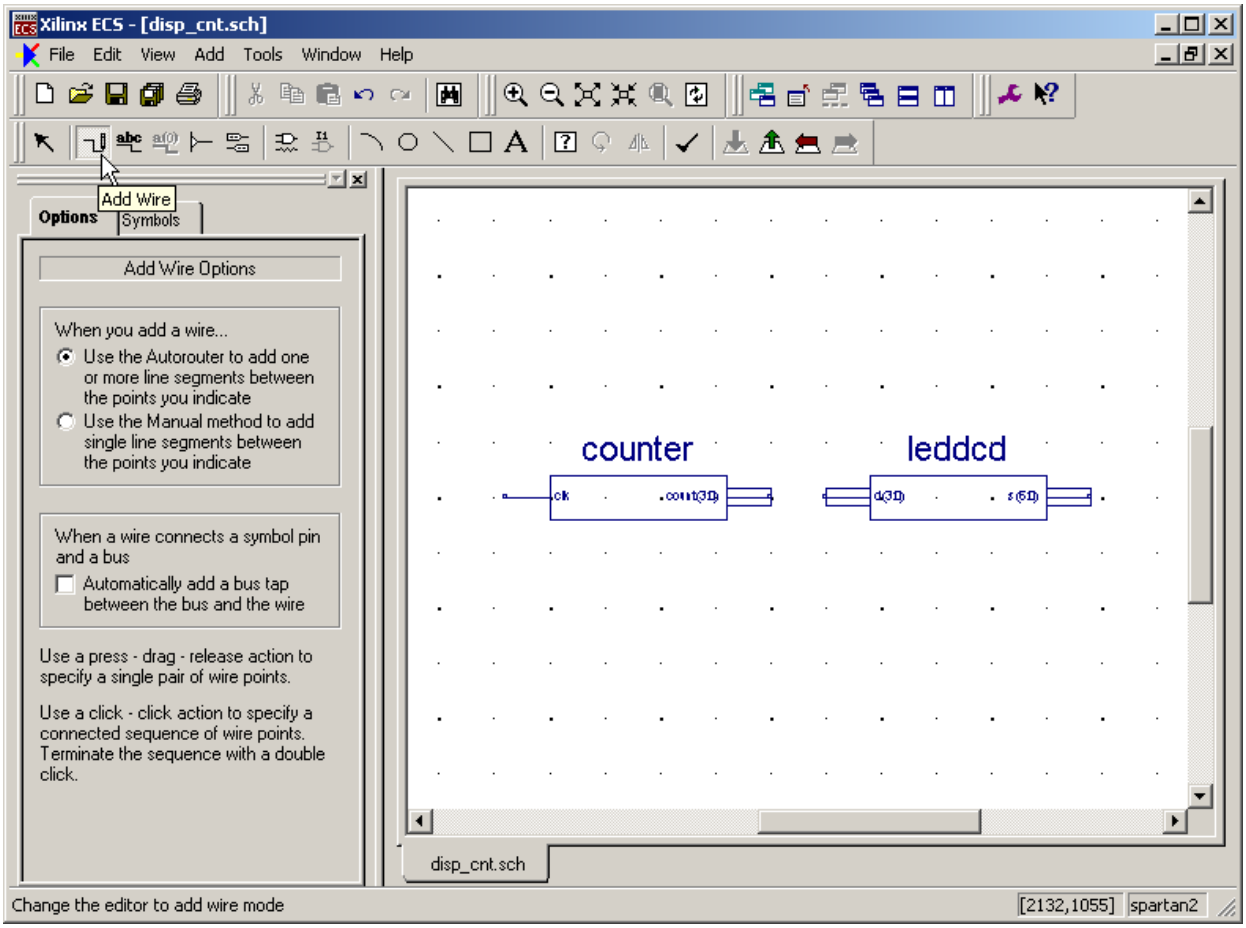
To start creating the top-level schematic, highlight the second entry in the category list. The c:/tmp/fpga_designs/design2 category contains the schematic symbols for the ***design2*** project's counter and LED decoder modules. We can see the names of these modules in the symbol list.



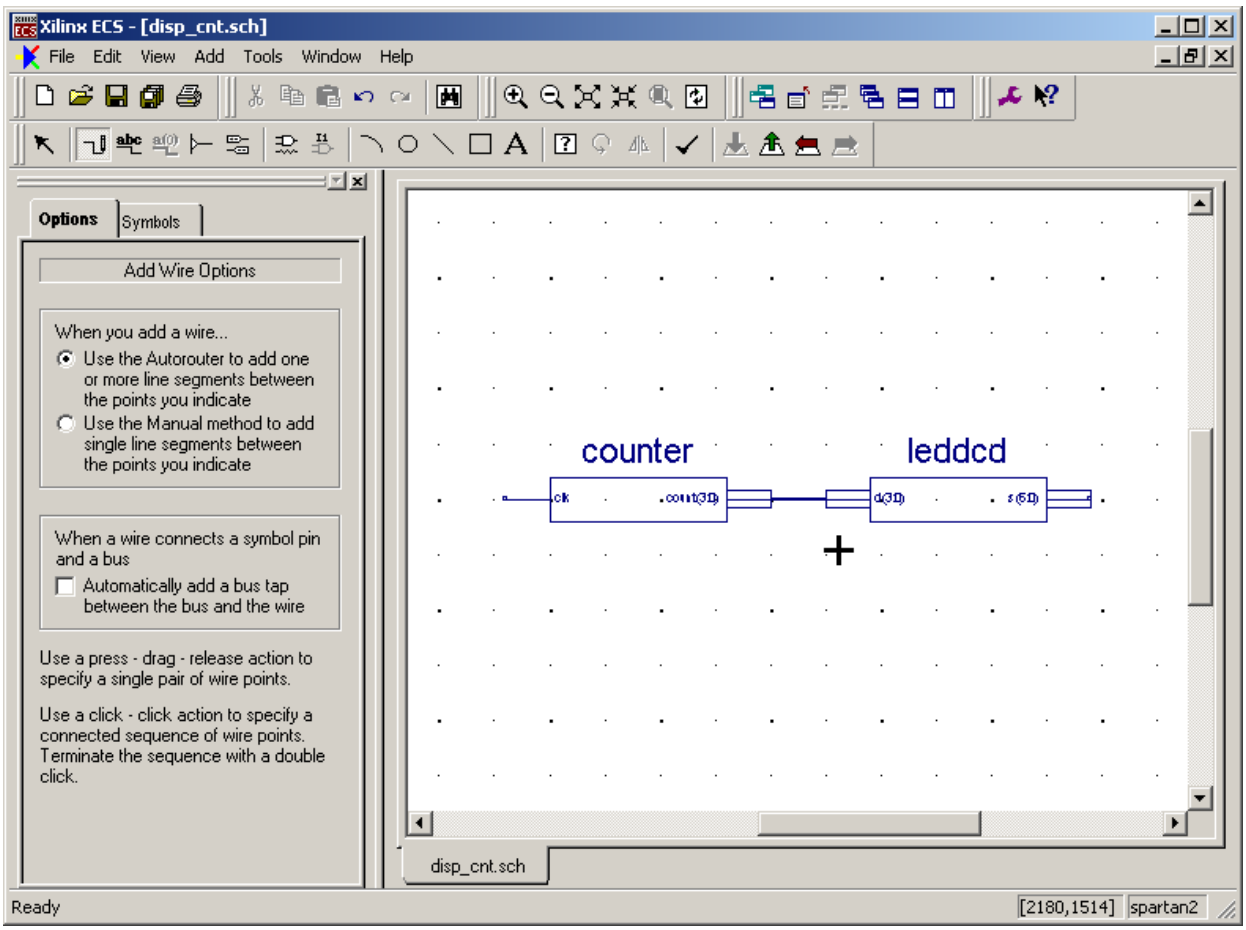
Click on the counter entry in the Symbols list. Then move the mouse cursor into the drawing area and left-click to place an instance of the counter into the schematic. Repeat this process with the leddcd module to arrive at the arrangement shown below.



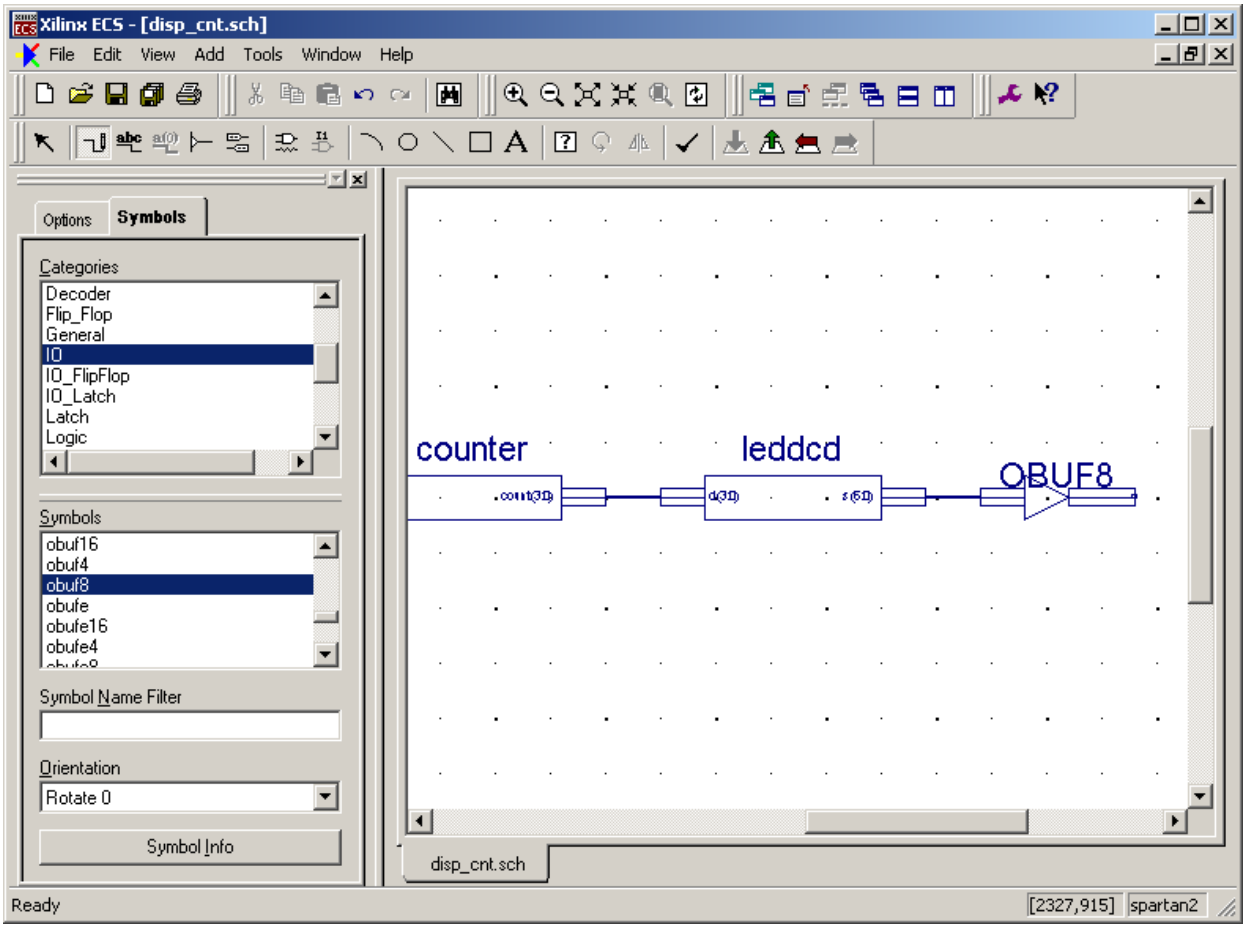
Next, click on the  button to begin adding wires to the schematic.



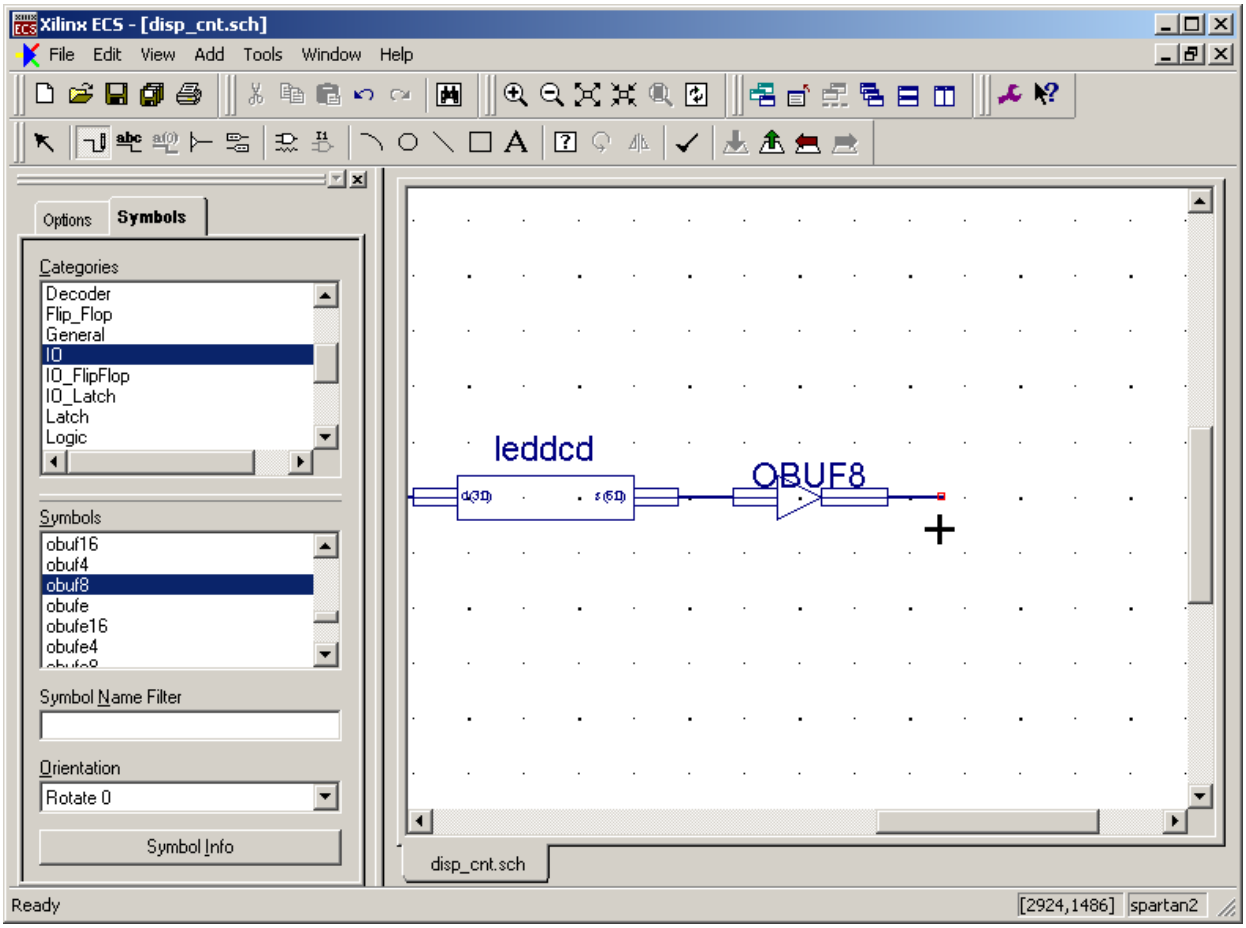
Left-click the mouse on the **count(3:0)** bus on the right-hand edge of the **counter** module. Then left-click on the **d(3:0)** bus on the left-hand edge of the **leddcd** module. As a result of this procedure, a four-bit bus is created between the output of the counter module and the input of the LED decoder module.



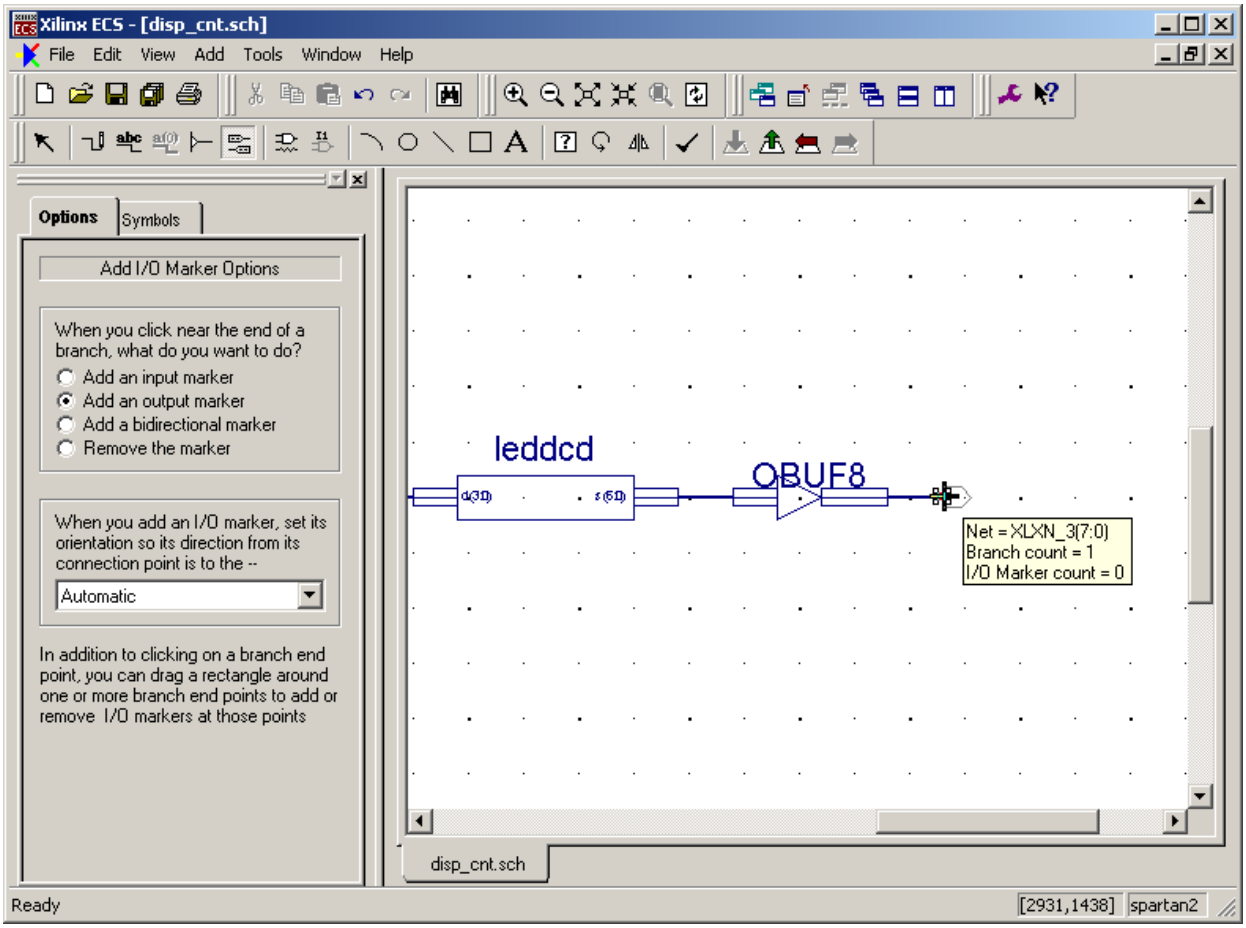
Now highlight the IO category and select a byte-wide output buffer (OBUF8) from the list of symbols. Attach the output buffer to the output of the LED decoder as shown below.



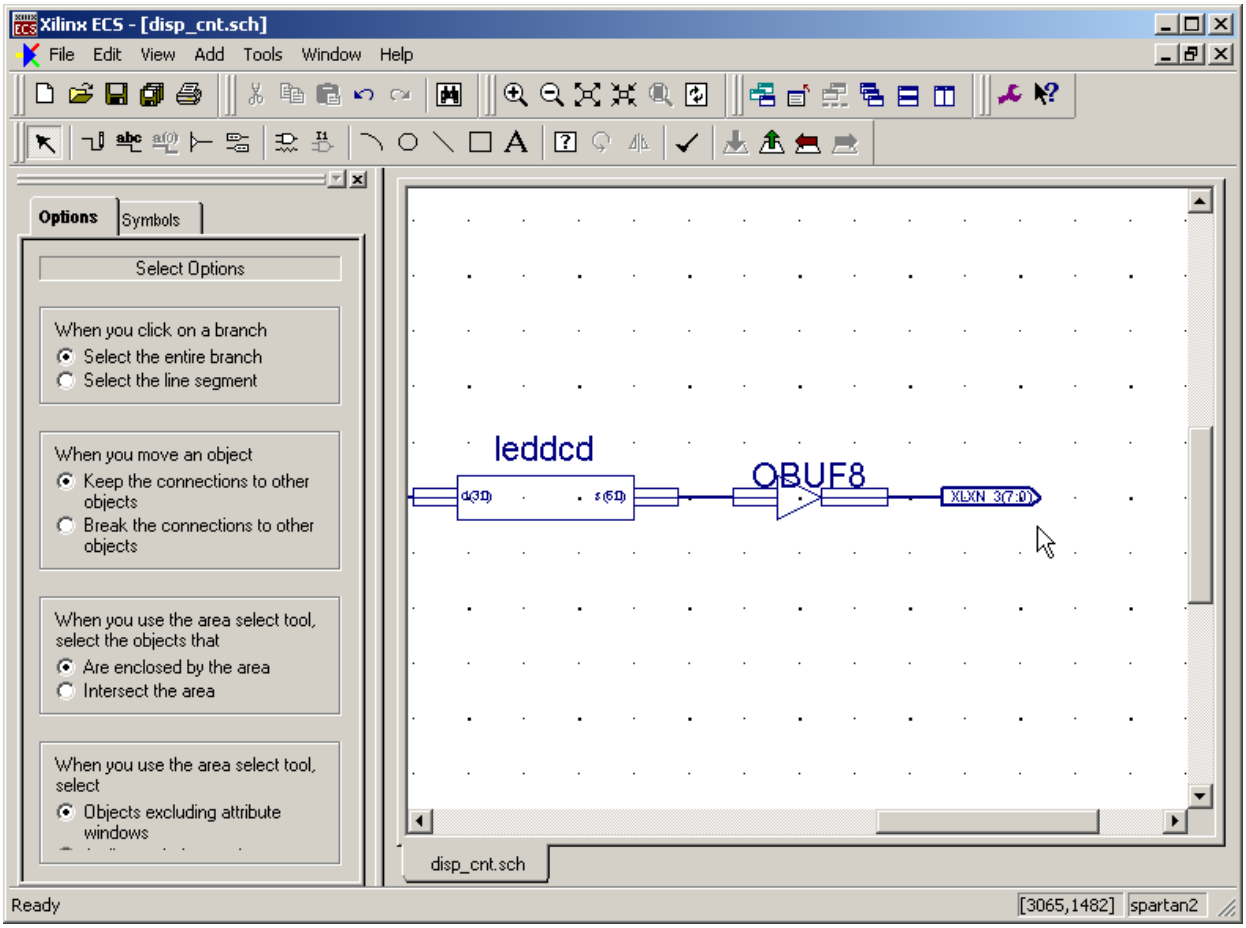
Next attach a short bus segment to the output of the byte-wide buffer. We do this by selecting the  wiring tool, clicking on the output of the byte-wide buffer, and then moving the cursor slightly to the right and double-clicking to create a stub.



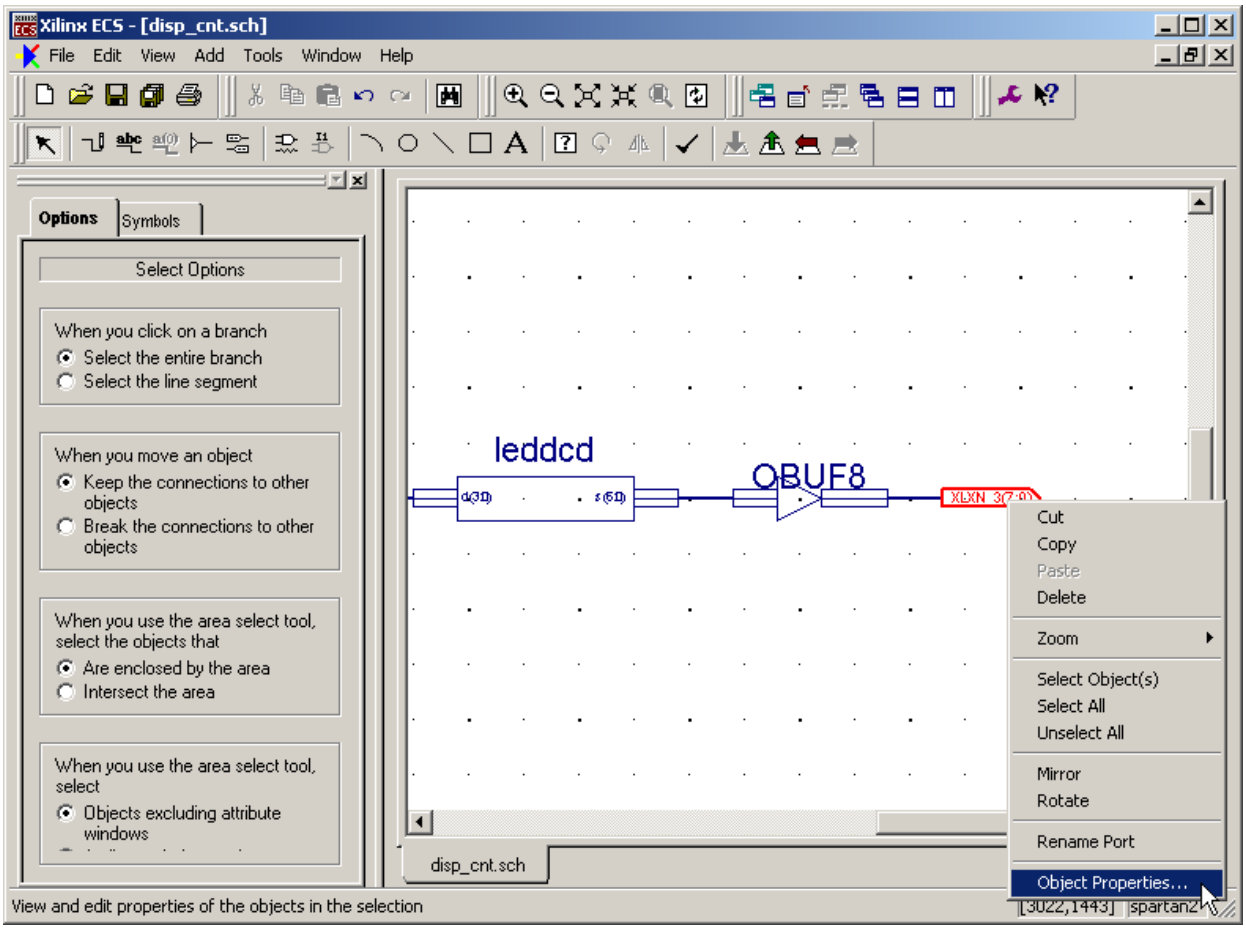
Now click on the  button for adding I/O markers. Click on the Add an output marker button in the Options tab and then click on the free end of the wire segment that we just added.



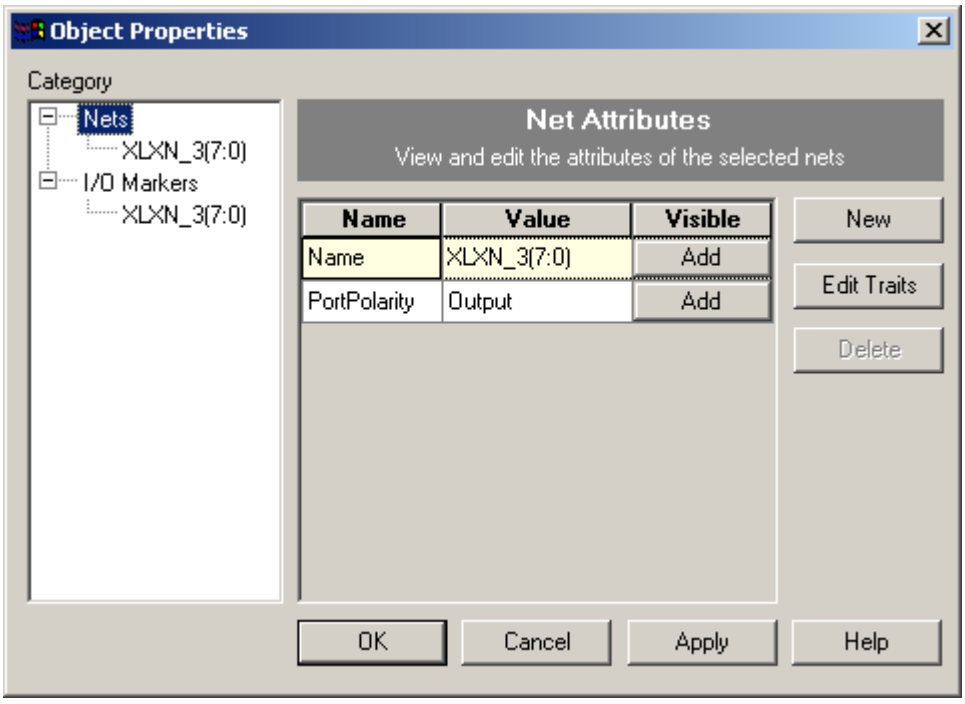
Clicking on the end of the wire creates a byte-wide set of output pins.



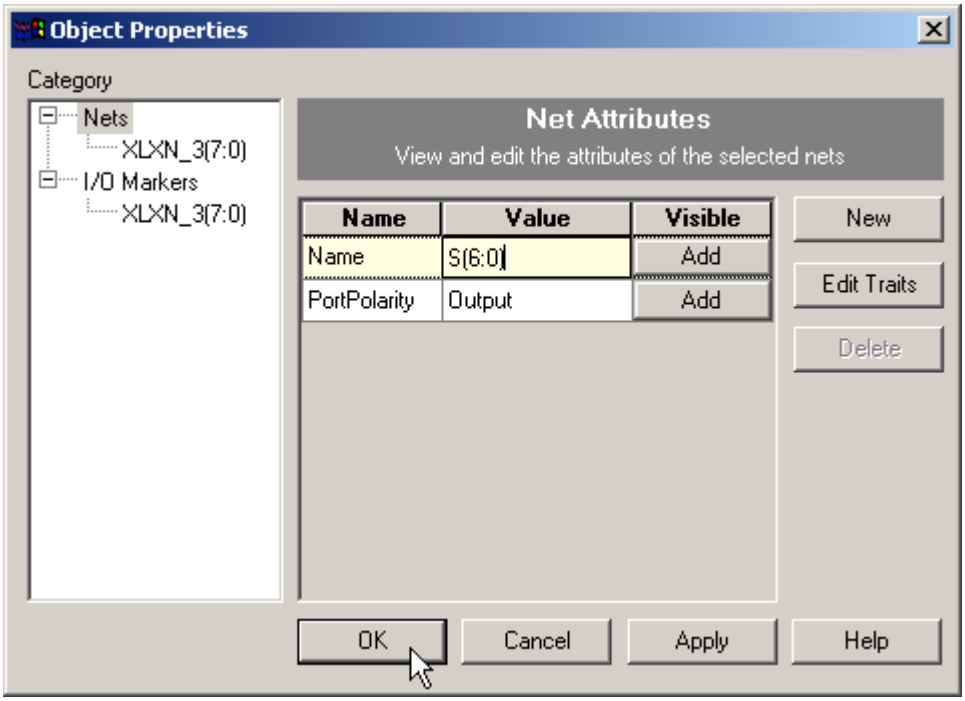
The output pins automatically assume the same name as the bus to which they are attached but this name was automatically generated and doesn't carry a lot of meaning. To change the name of the outputs (and the associated bus), right-click on the I/O marker and select Object Properties... from the pop-up menu.



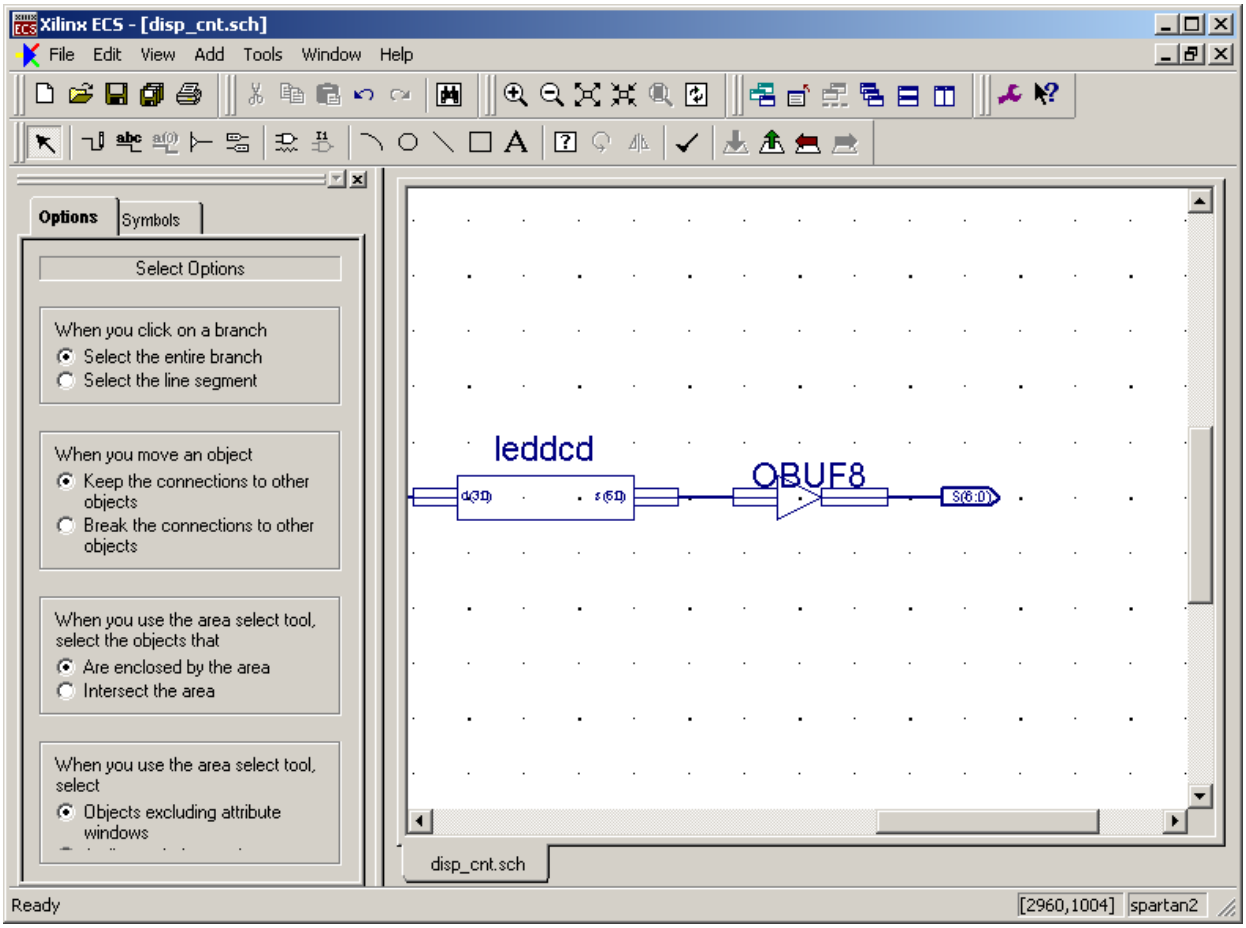
The **Object Properties** window allows us to set the name and direction of the pins.



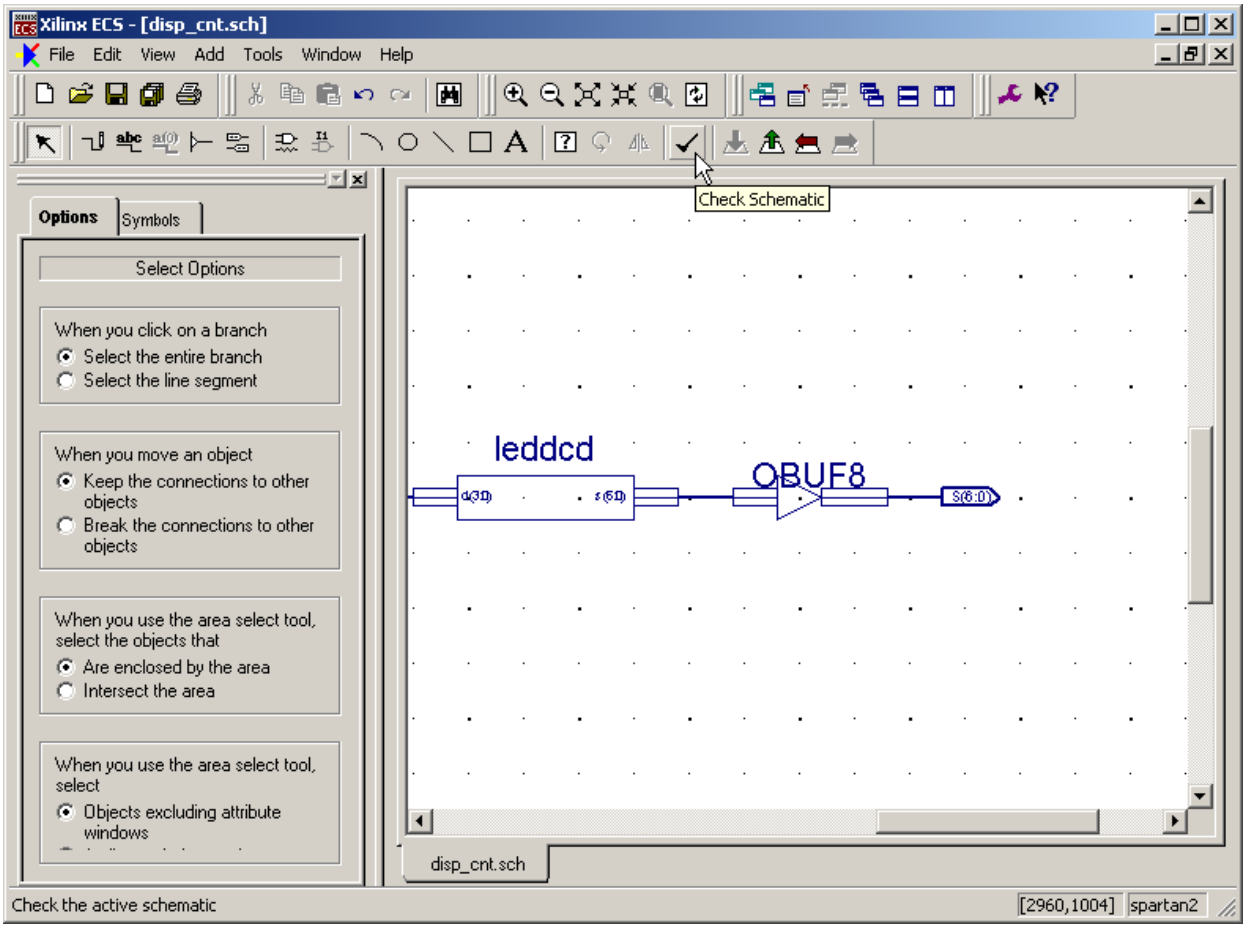
Replace the existing bus name with a seven-bit bus for driving the LED segments: **S(6:0)**. The direction of the bus pins is already set to Output so we can finish by clicking on the OK button.



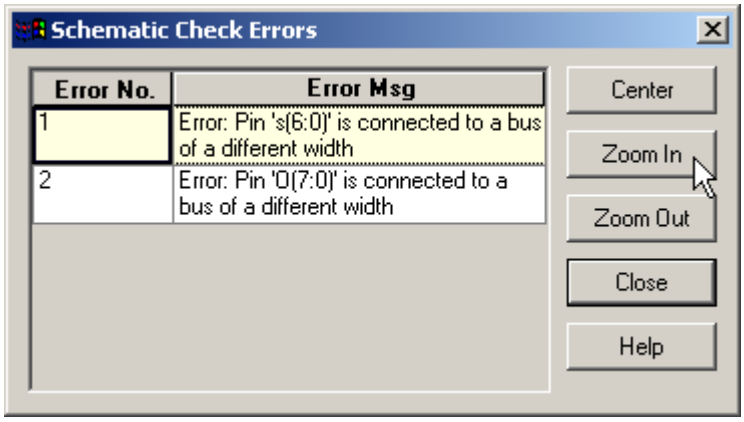
The output pins now appear with their new name, width, and direction.



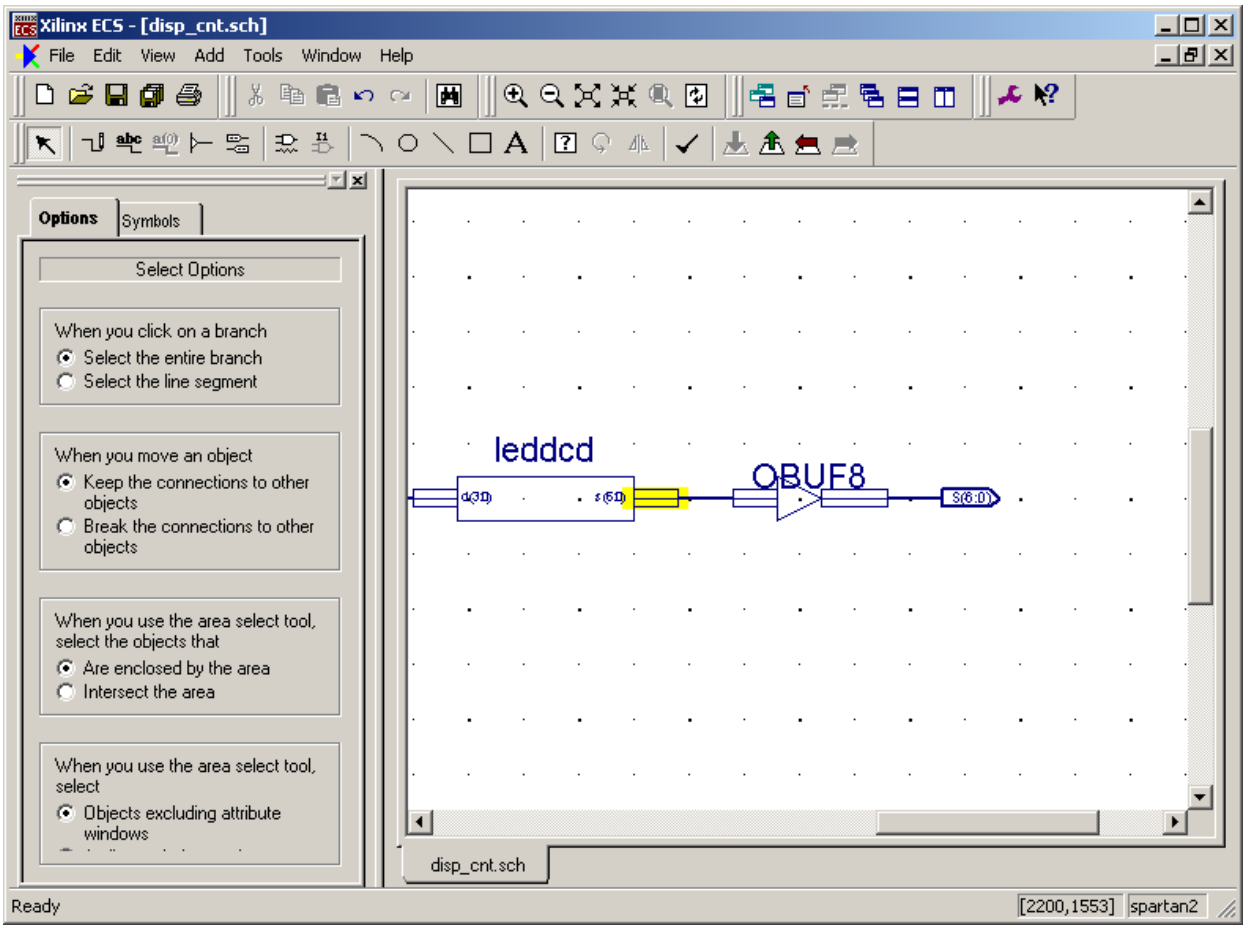
At this point it makes sense to check the schematic to see if there are any errors such as unterminated wire stubs or mismatched bus widths. Click on the  button to perform a schematic check.



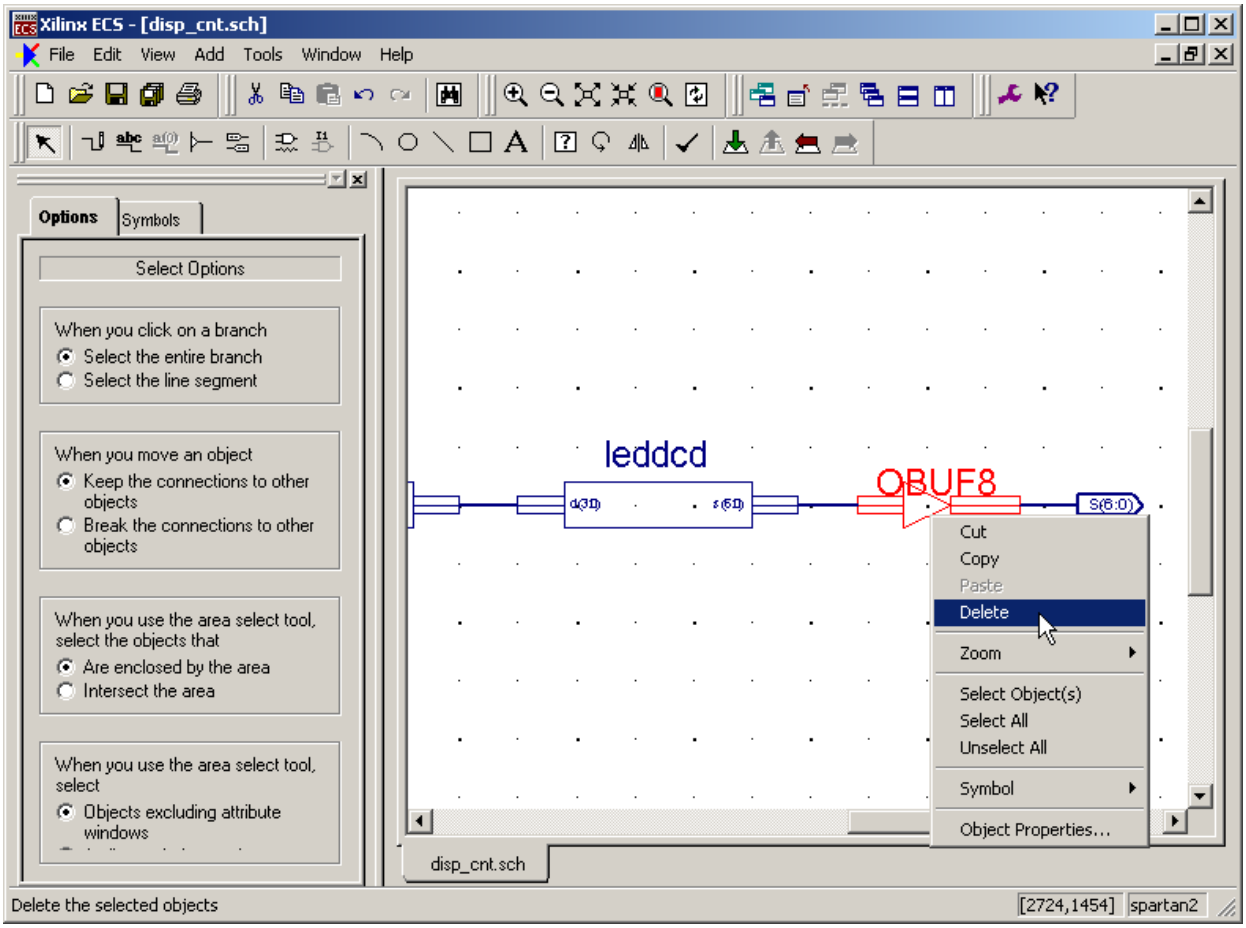
The **Schematic Check Errors** window will appear showing two errors. We can find the place in the schematic where the error occurs by clicking on the associated error message. Then click on the Zoom In button to see an enlarged view of the area where the error lies.



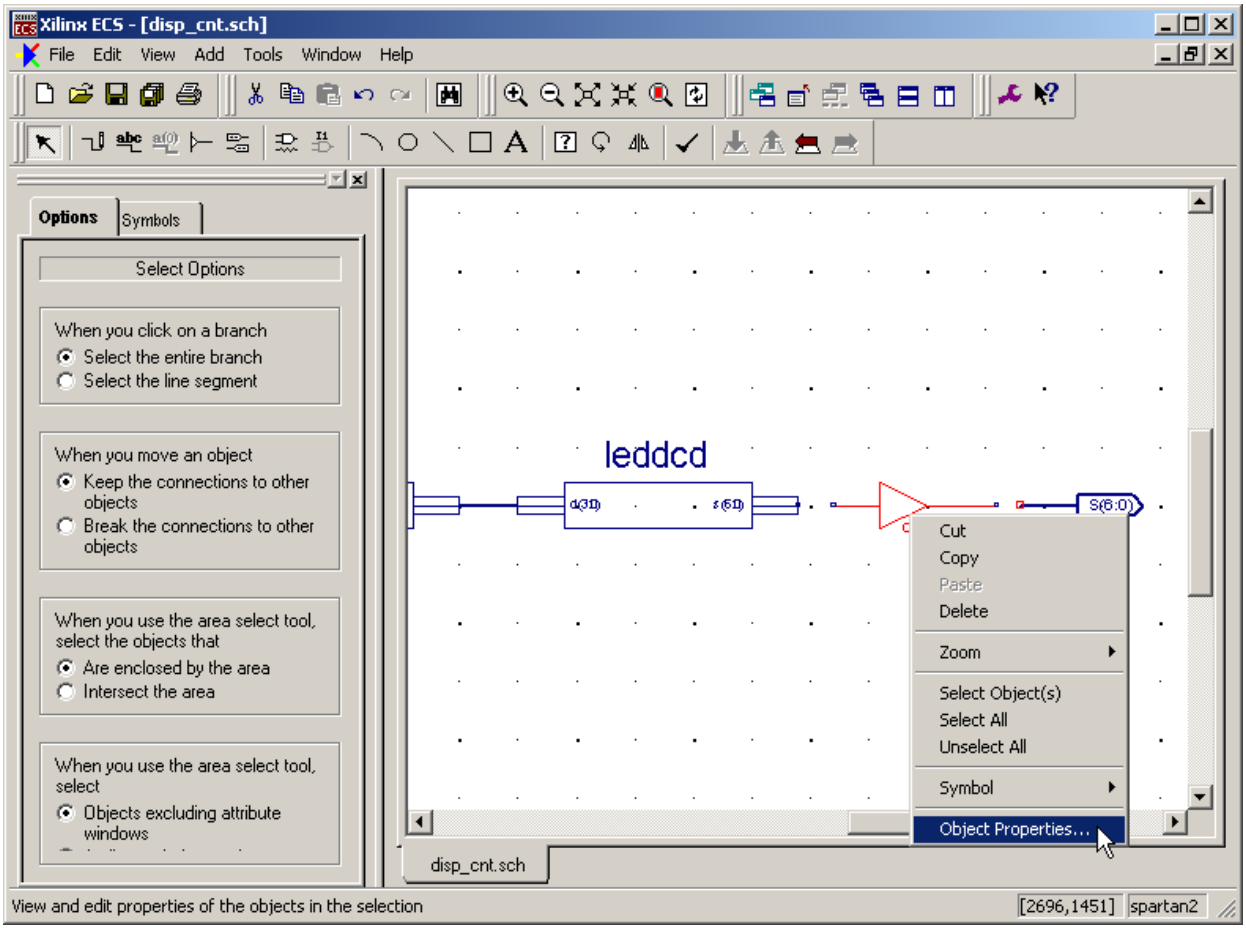
The first error indicates that the seven-bit output of the LED decoder does not match with the byte-wide input of the output buffer symbol. Note how the output of the leddcd symbol is highlighted to indicate the error. The second error is similar to the first in that the byte-wide output of the OBUF8 symbol does not match the width of the seven-bit output pin marker.



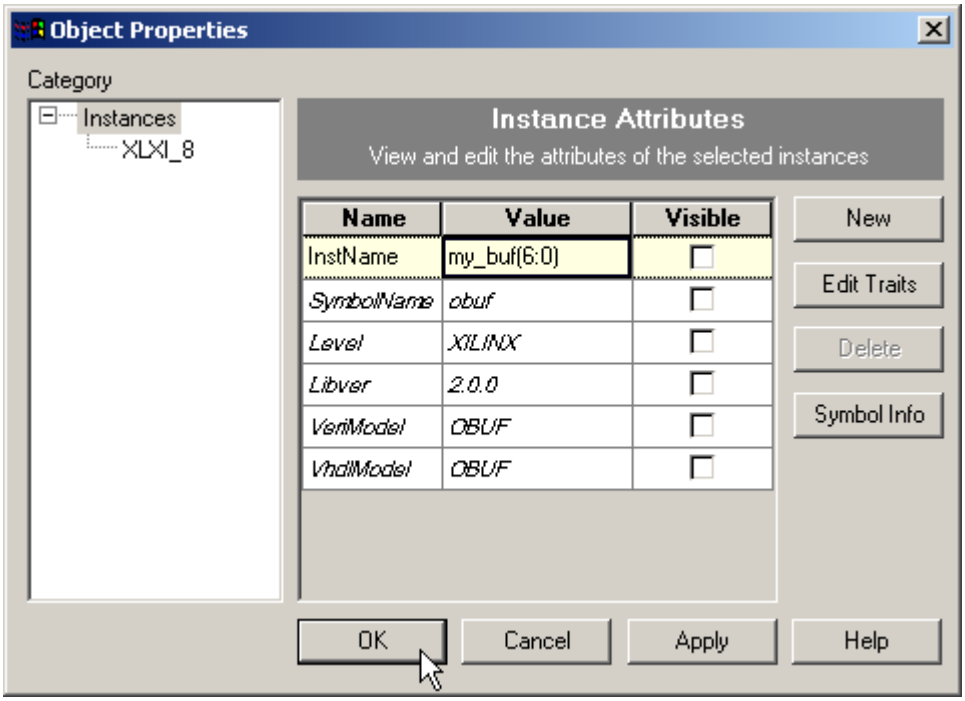
The simplest way remove these errors is to replace the byte-wide output buffer with a seven-bit wide version. To remove the byte-wide buffer, right-click on the OBUF8 symbol and select delete from the pop-up menu. Do the same for the bus that connected to the input of the byte-wide buffer.



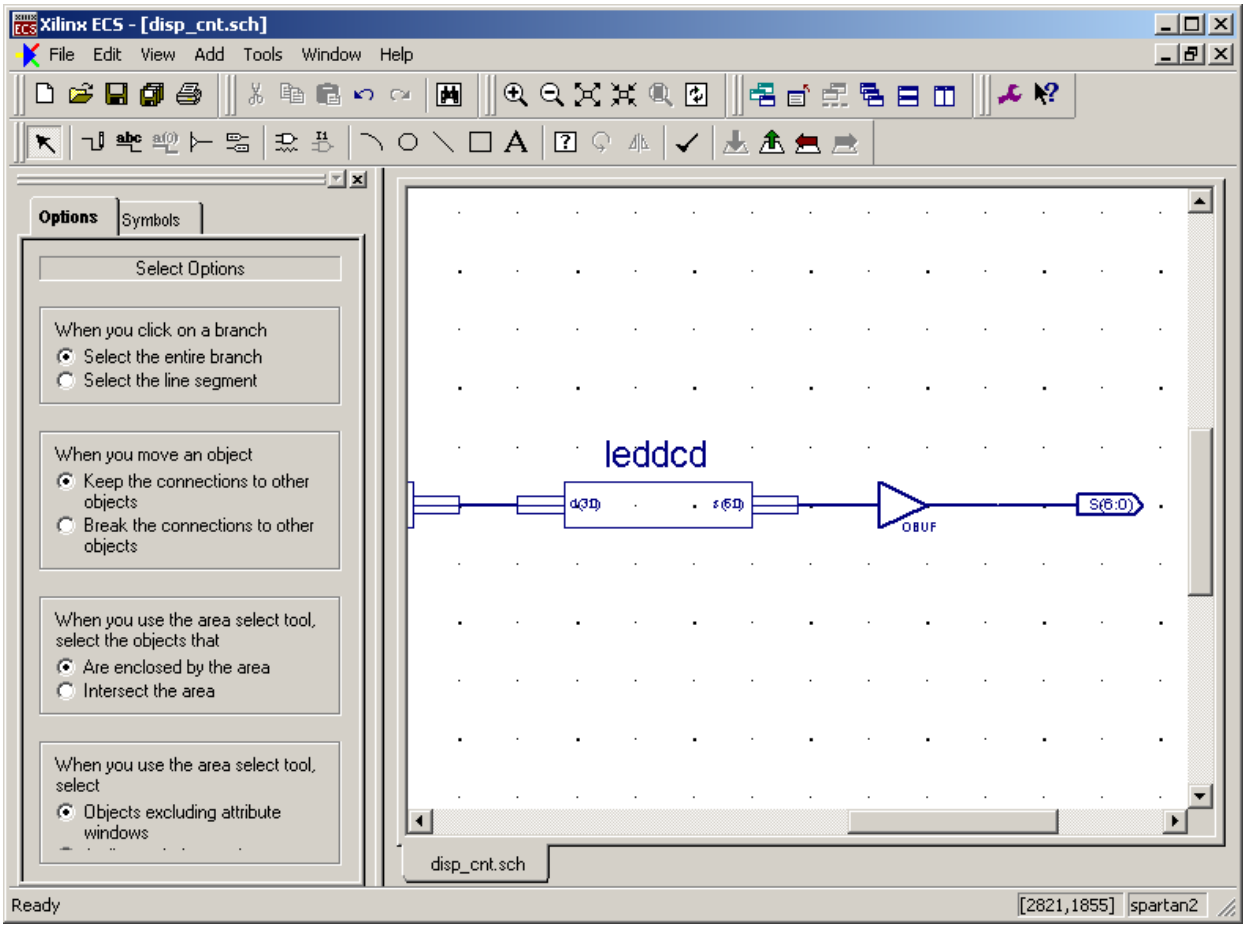
Now add a single OBUF symbol to the schematic. Then right-click on it and select Object Properties... from the pop-up menu.



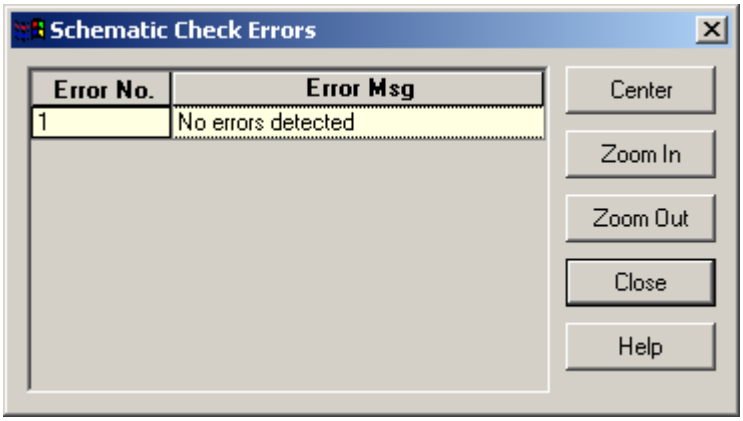
In the **Object Properties** window, highlight the instance name for the buffer and change it to `mybuf(6:0)` and then click on the OK button. This will change the single-bit output buffer to an array of seven output buffers.



After changing the width of the output buffer, reconnect it to the LED decoder and the output terminals as shown below.

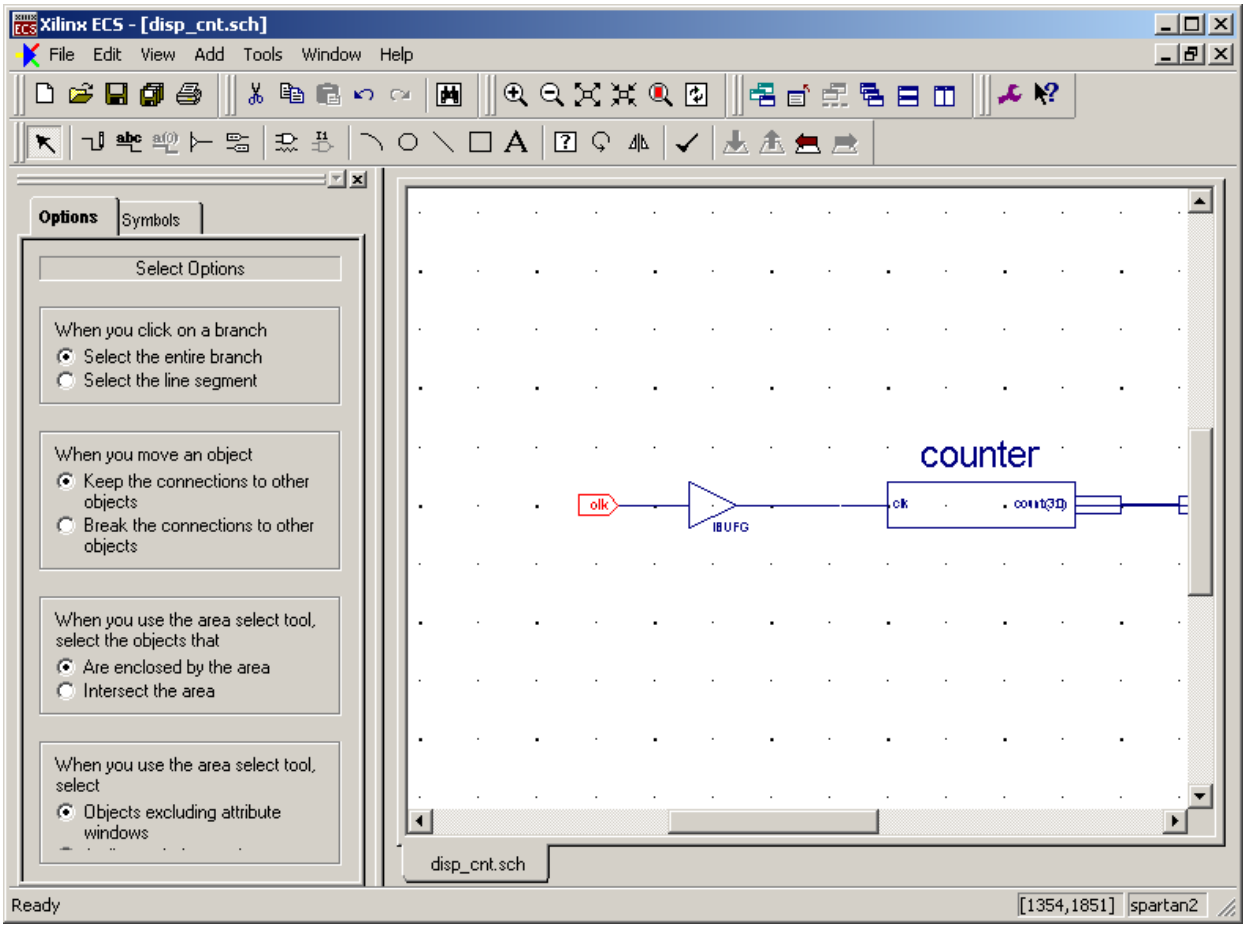


Now when we click on the schematic check button,  , we see the errors have been corrected.

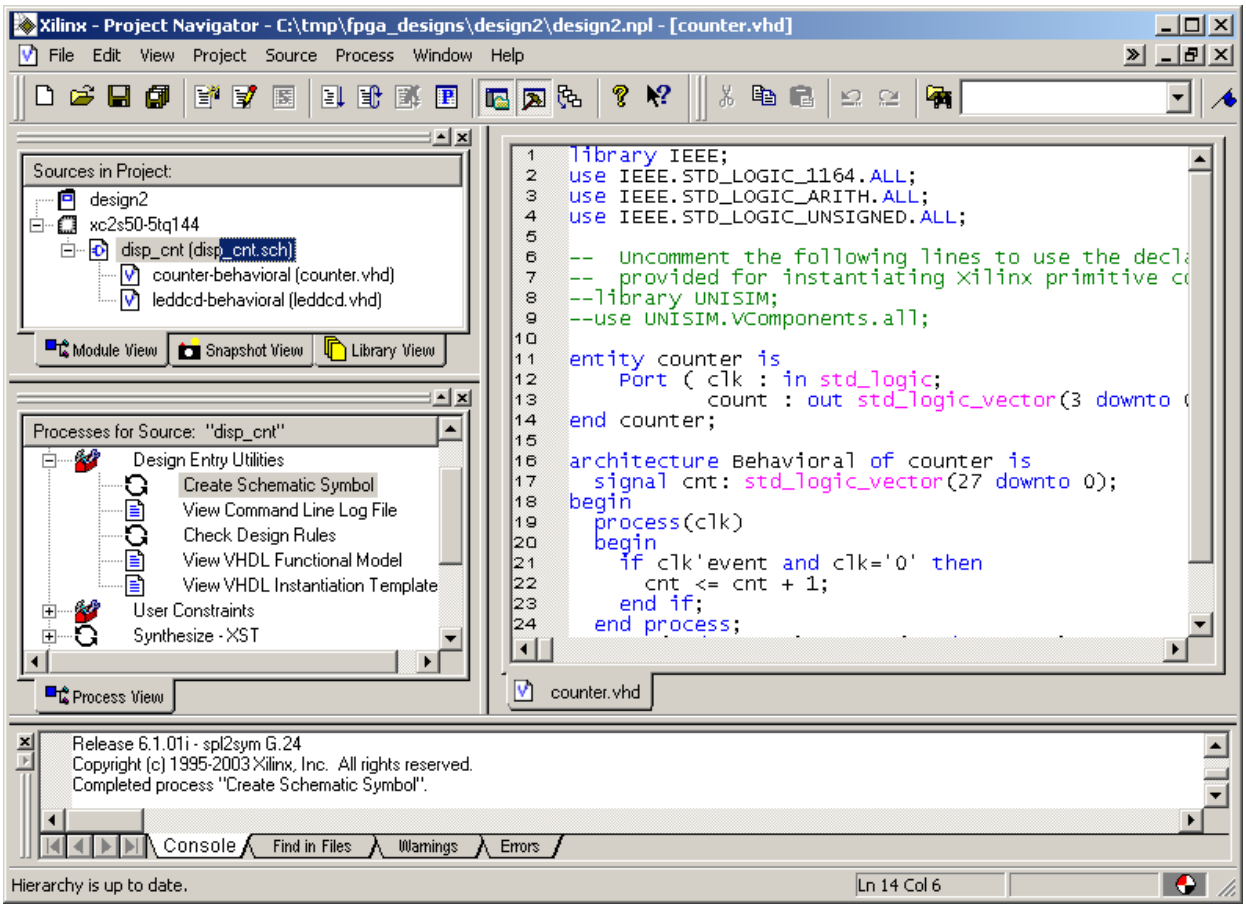


Once the outputs from the circuit are in place, we can create the analogous circuitry for the input. We connect a single, low-skew input buffer module to the clock input of the counter and then we connect a single input I/O marker to the IBUFG symbol. Right-click on the I/O marker

and rename it to `clk`. After this, perform another schematic check to detect any errors, save the schematic using the File➔Save command and then close the schematic editor.

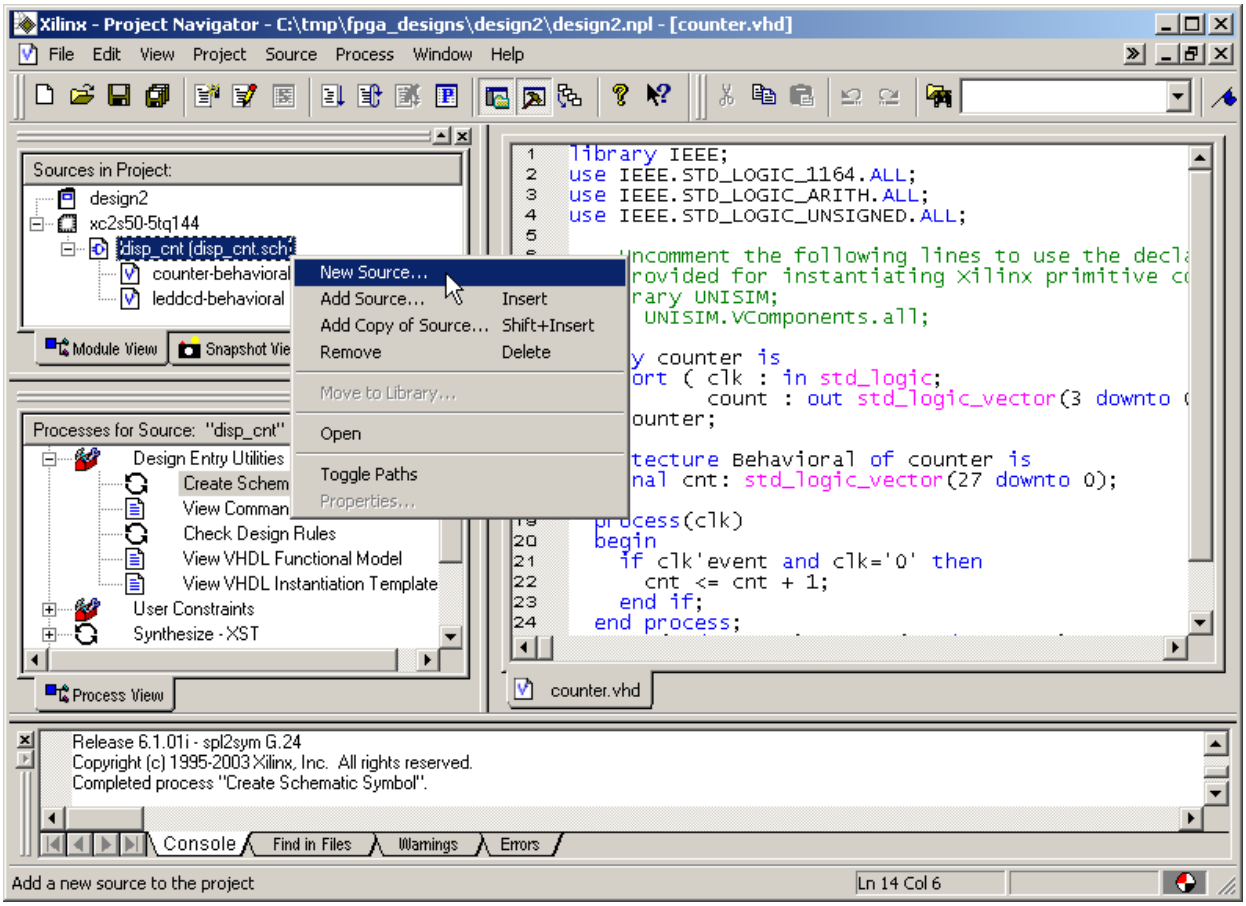


Once we save the schematic for the top-level module, we see the updated hierarchy in the Sources pane of the **Project Navigator** window. Now the **counter** and **leddcd** modules are shown as lower-level modules that are included within the top-level **disp_cnt** module.

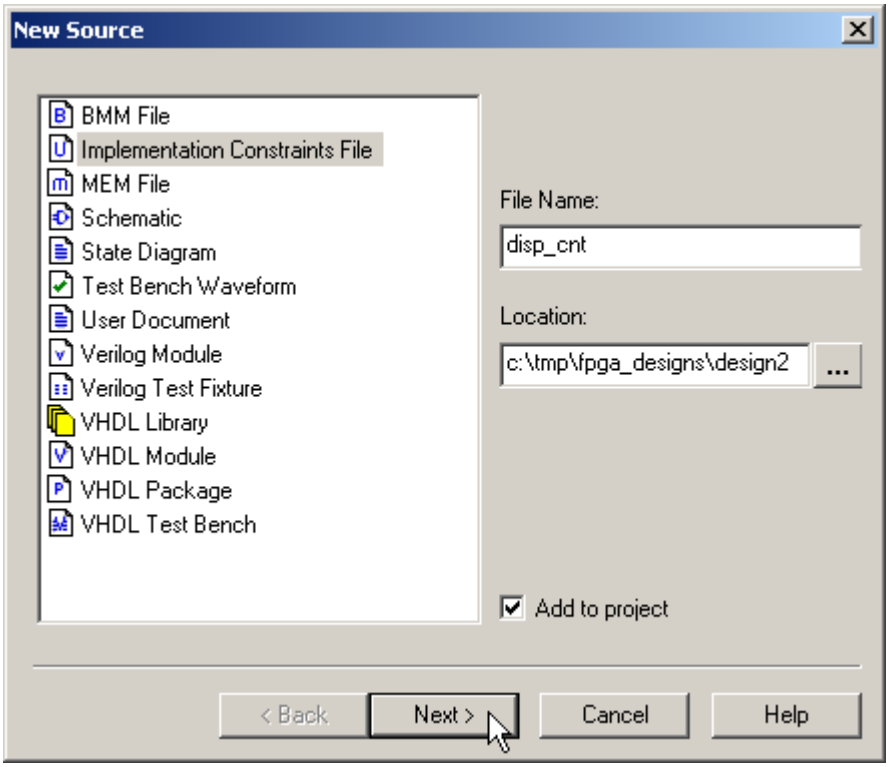


Constraining the Design

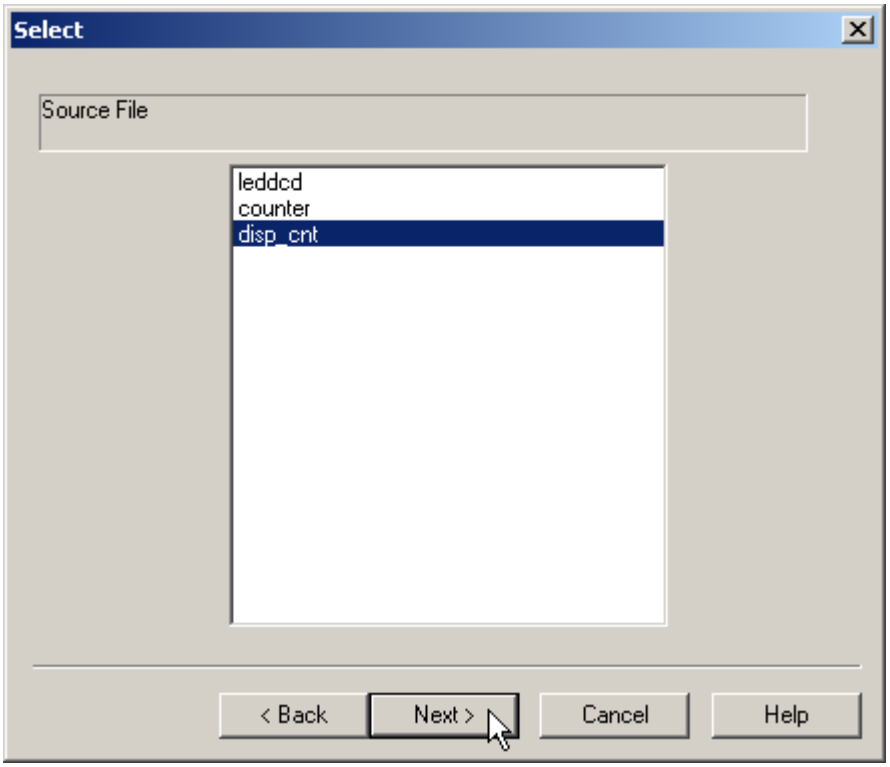
Before synthesizing the displayable counter, we need to assign the pins which the inputs and outputs will use. We start by right-clicking the disp_cnt object in the Sources pane and selecting New Source... from the pop-up menu.



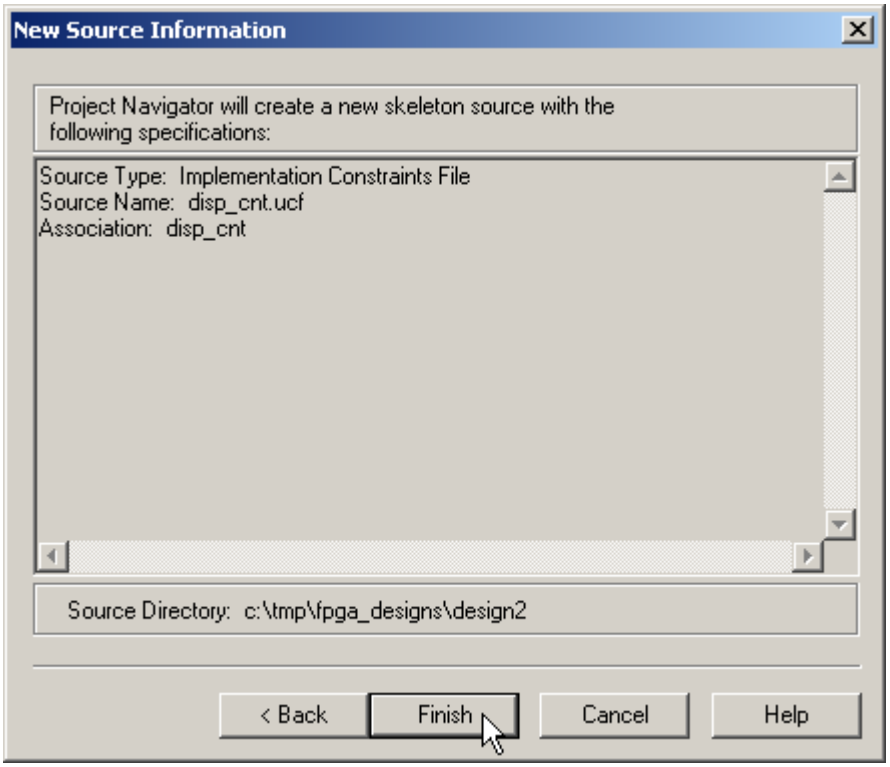
Select Implementation Constraints File as the type of source file we want to add and type `disp_cnt` in the File Name field. Then click on the Next button.



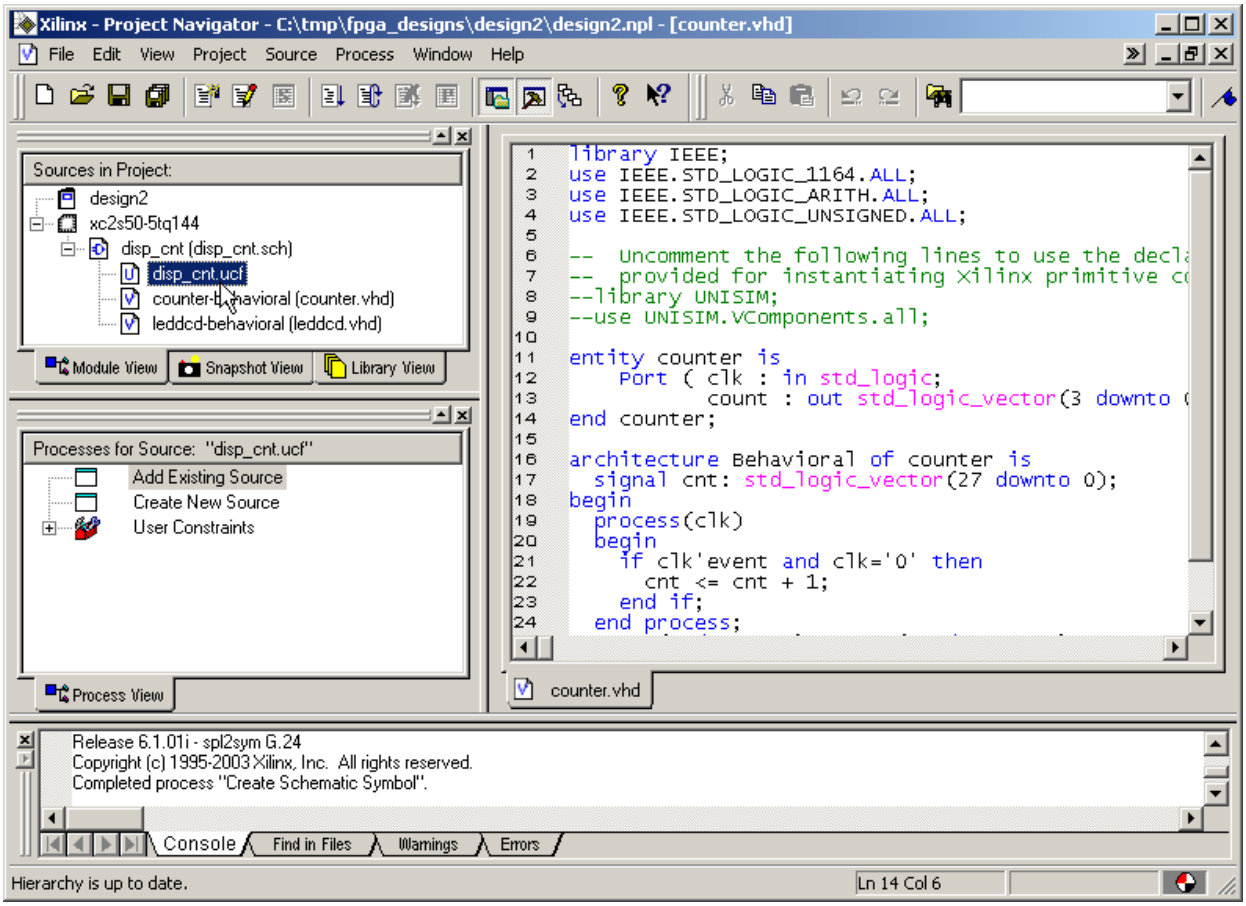
Then we are asked to pick the file with which to associate the constraints. The pin assignments will be made for the top-level module in the design hierarchy, so highlight the disp_cnt item in the list. Then click on the Next button and proceed.



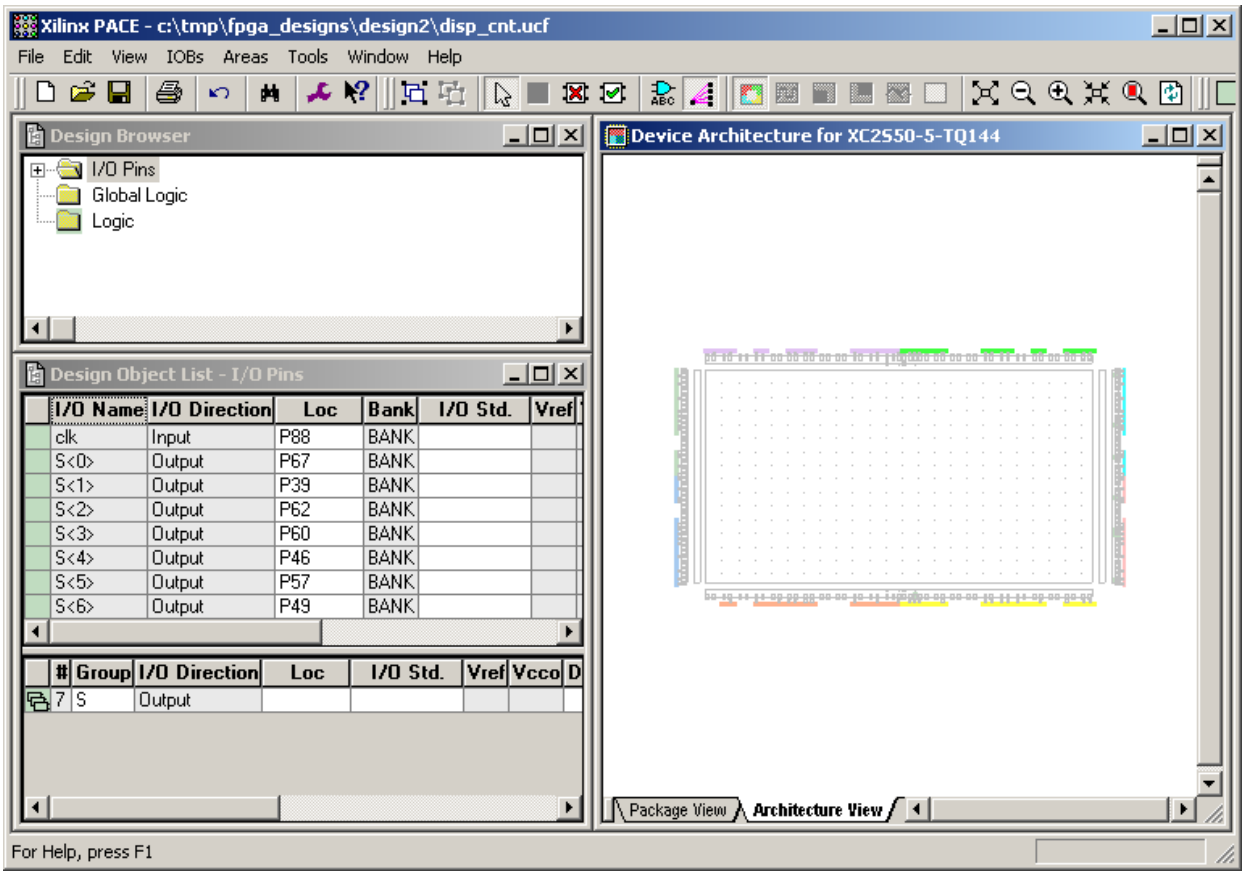
You will receive a feedback window that shows the name and type of the file you created and the file to which it is associated. Click on the Finish button to complete the addition of the disp_cnt.ucf file to this project.



Now double-click the disp_cnt.ucf object in the Sources pane to begin adding pin assignments to the design.



In the **Design Object List – I/O Pins** pane of the **Xilinx PACE** window that appears, set the pin assignments for the clock input and LED segment drivers as follows:

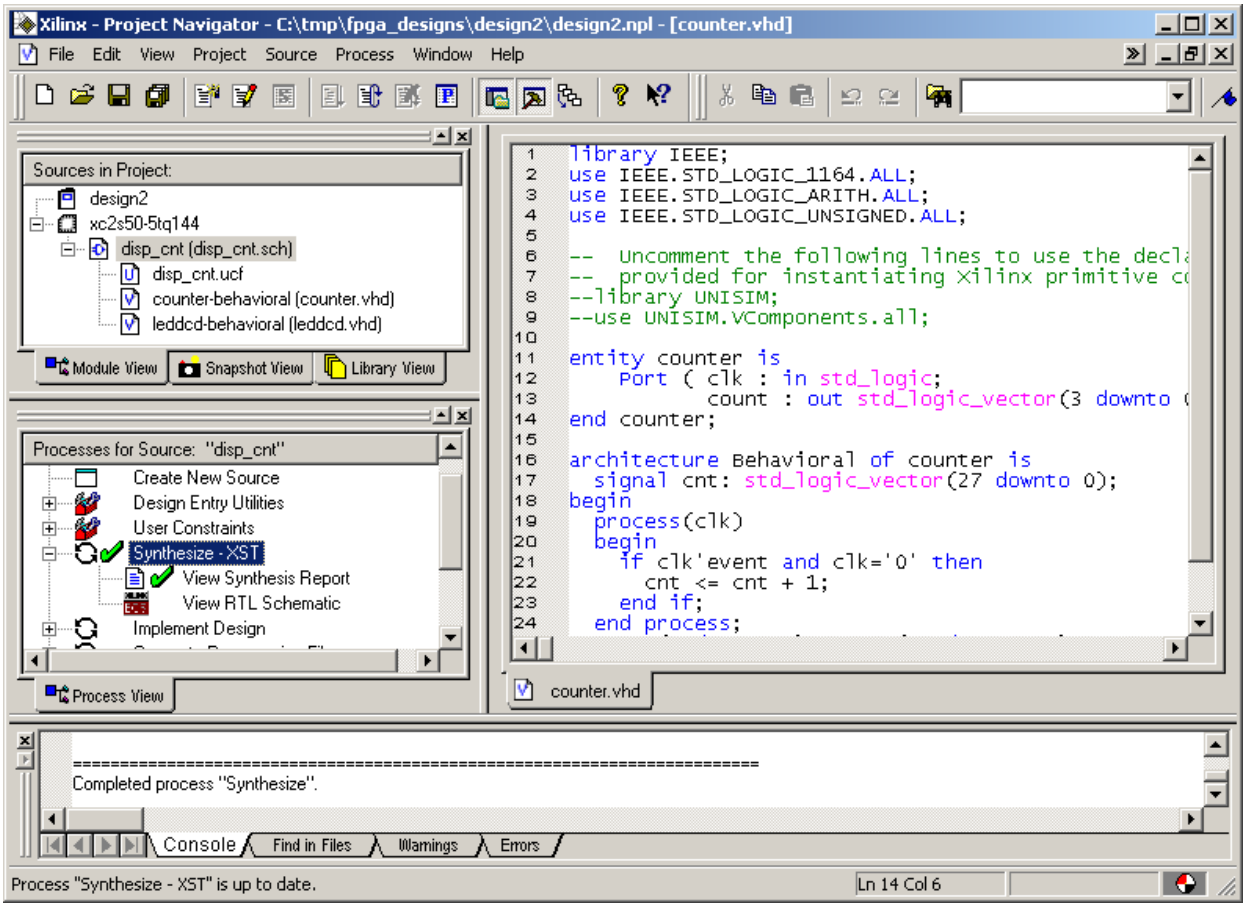


Assigning the **clk** input to pin P88 lets us use the onboard oscillator of the XSA-50 Board to drive the counter. The output assignments connect the displayable counter to the seven-segment LED on the XSA-50 Board as in the previous design example.

After the pin assignments are entered, click on the  button to save the pin assignment constraints. Then select File➔Exit to close the **Xilinx PACE** window.

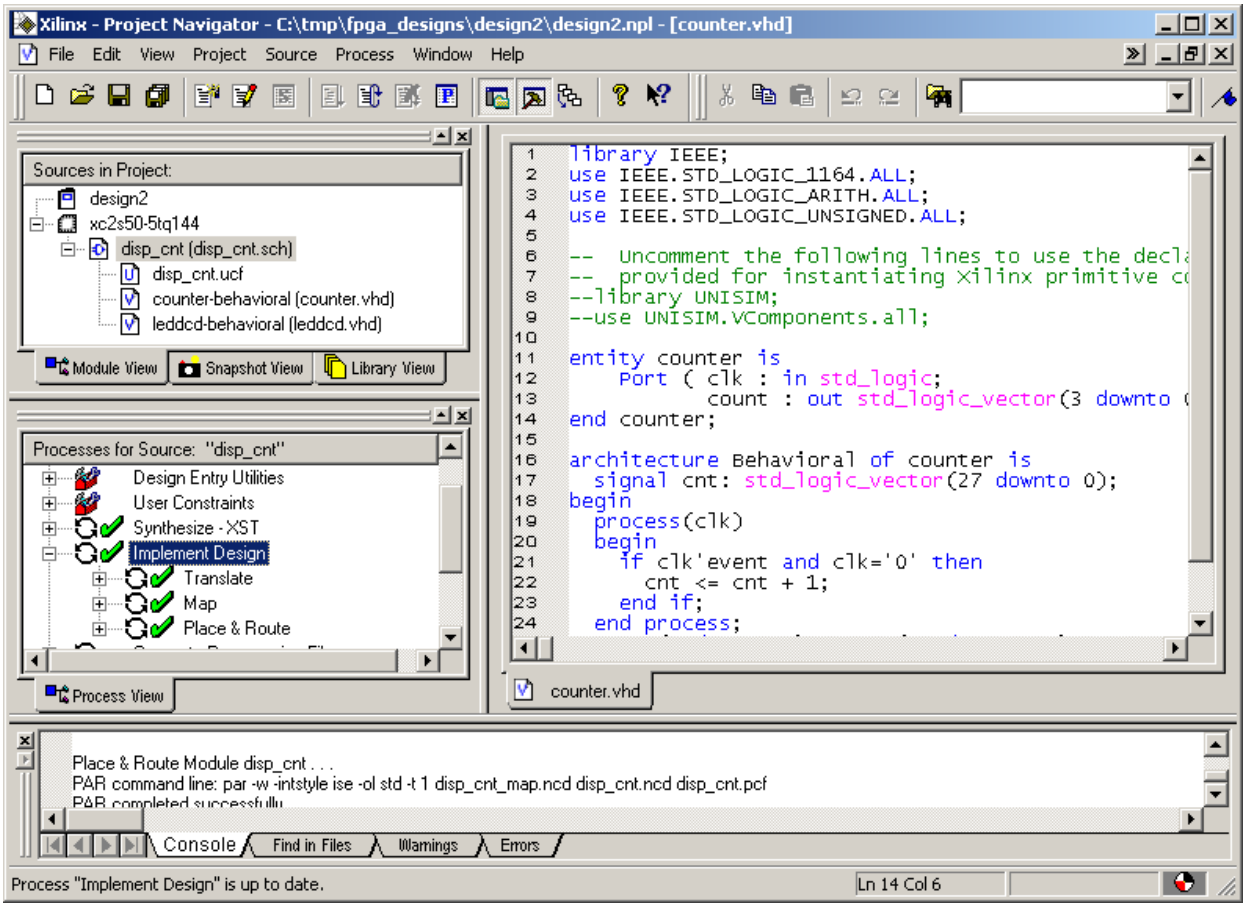
Synthesizing the Logic Circuitry for the Design

Now we can synthesize the logic circuit netlist by highlighting the top-level **disp_cnt** module in the Sources pane and double-clicking the Synthesize process. There should be no problems synthesizing the netlist from the combined VHDL and schematic files.



Implementing the Logic Circuitry in the FPGA

Once the netlist is synthesized, we can begin the process of translating, mapping and placing & routing it into the FPGA. Highlight the disp_cnt object in the Sources pane and then double-click the Implement Design process. There should be no problems implementing the design in the FPGA.



Checking the Implementation

After the implementation process is done, we can check the logic utilization by double-clicking on the Place & Route Report process. Near the top of the file we find:

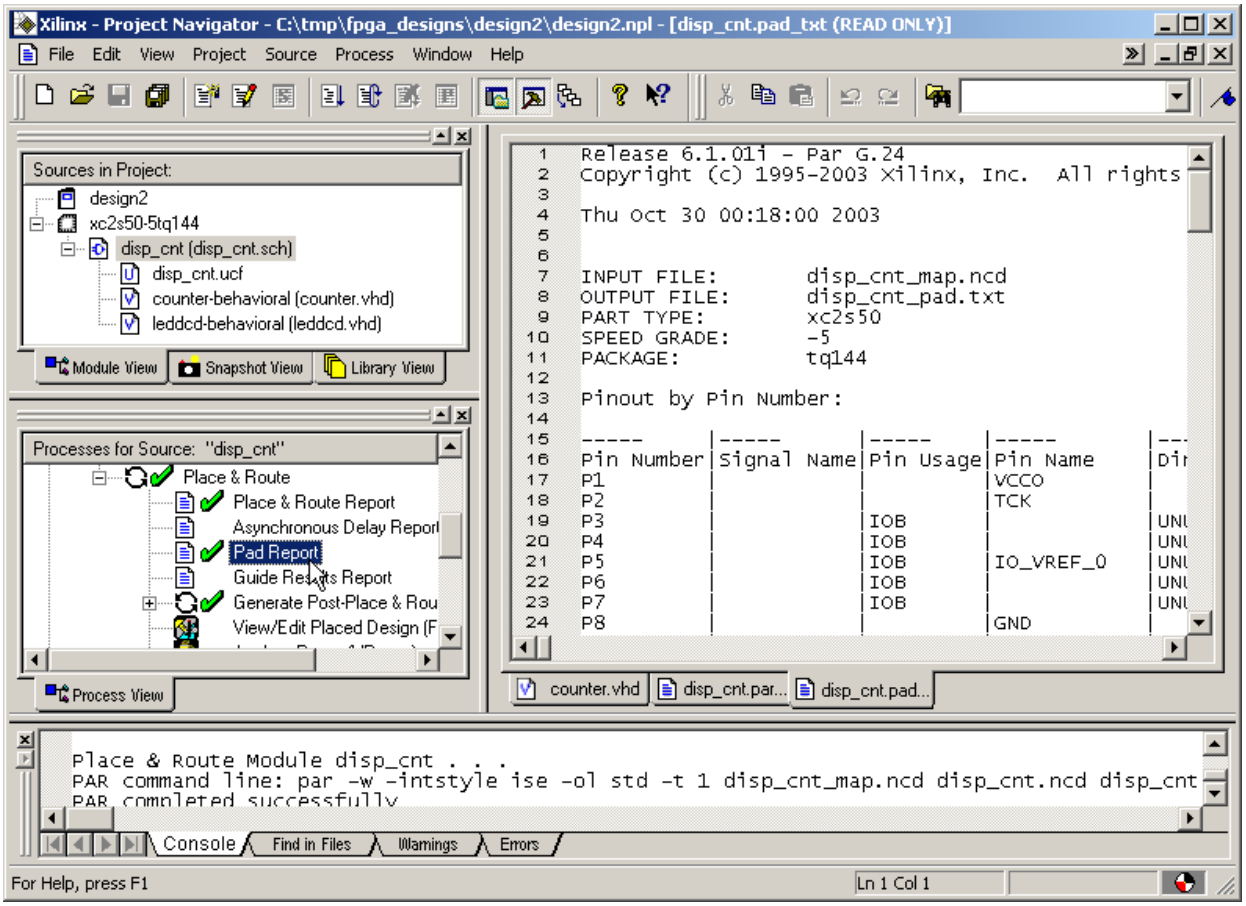
```
Device utilization summary:

   Number of External GCLKIOBs          1 out of 4      25%
   Number of External IOBs              7 out of 92      7%
      Number of LOCed External IOBs     7 out of 7     100%

   Number of SLICEs                    18 out of 768     2%
```

The displayable counter consumes 18 of the 3072 slices in the FPGA. Each slice contains two CLBs, so the displayable counter uses a maximum of 36 CLBs. The 28-bit counter requires at least 28 CLBs and the LED decoder requires 7 CLBs so this totals to 35 CLBs.

As a precaution, we should also double-click the Pads Report and check that the pin assignments for the clock input and LED decoder outputs match the assignments we made with PACE:

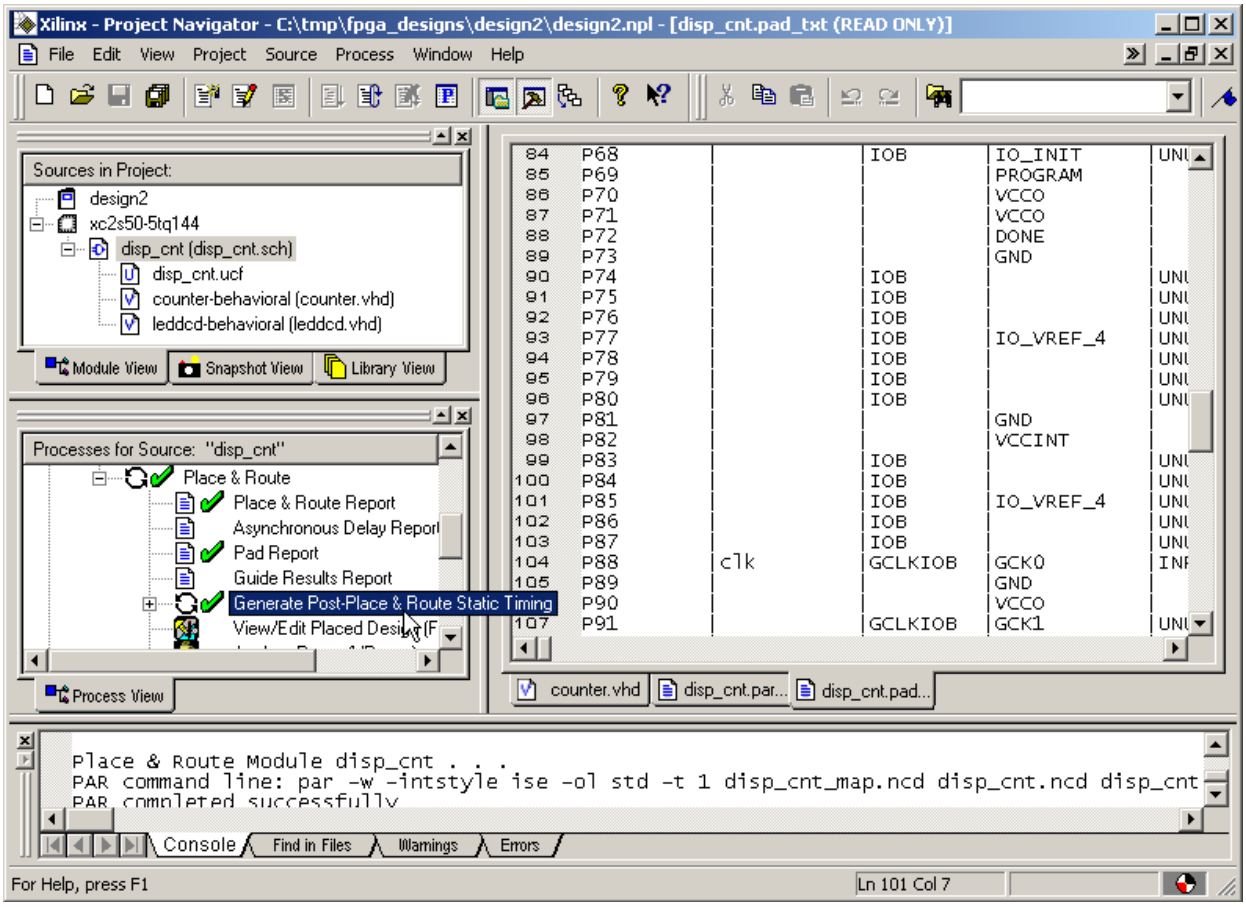


We can see that the pin assignments for the clock input and the LED decoder outputs agree with the constraints we placed in the UCF file.

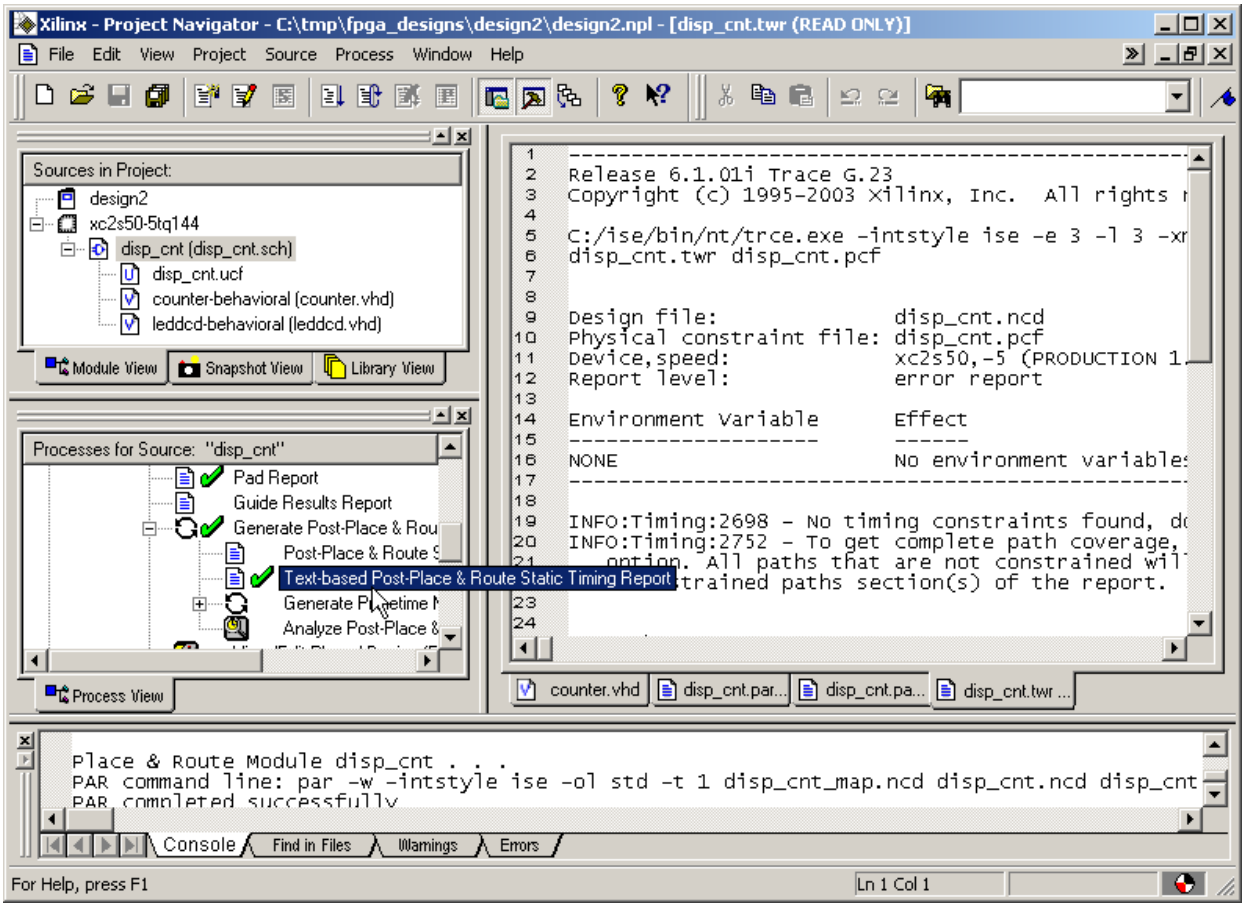
Pin Number	Signal Name	Pin Usage	Direction	IO Standard
P39	S<1>	IOB	OUTPUT	LVTTL
P46	S<4>	IOB	OUTPUT	LVTTL
P49	S<6>	IOB	OUTPUT	LVTTL
P57	S<5>	IOB	OUTPUT	LVTTL
P60	S<3>	IOB	OUTPUT	LVTTL
P62	S<2>	IOB	OUTPUT	LVTTL
P67	S<0>	IOB	OUTPUT	LVTTL
P88	clk	GCLKIOB	INPUT	LVTTL

Checking the Timing

We have the displayable counter synthesized and implemented in the XC2S50 FPGA with the correct pin assignments. But how fast can we run the counter? To find out, double-click on the Generate Post-Place & Route Static Timing process. This will determine the maximum delays between logic elements in the design taking into account logic and wiring delays for the routed circuit.



After the static timing delays are calculated, double-click the Text-based Post-Place & Route Static Timing Report to view the results of the analysis.



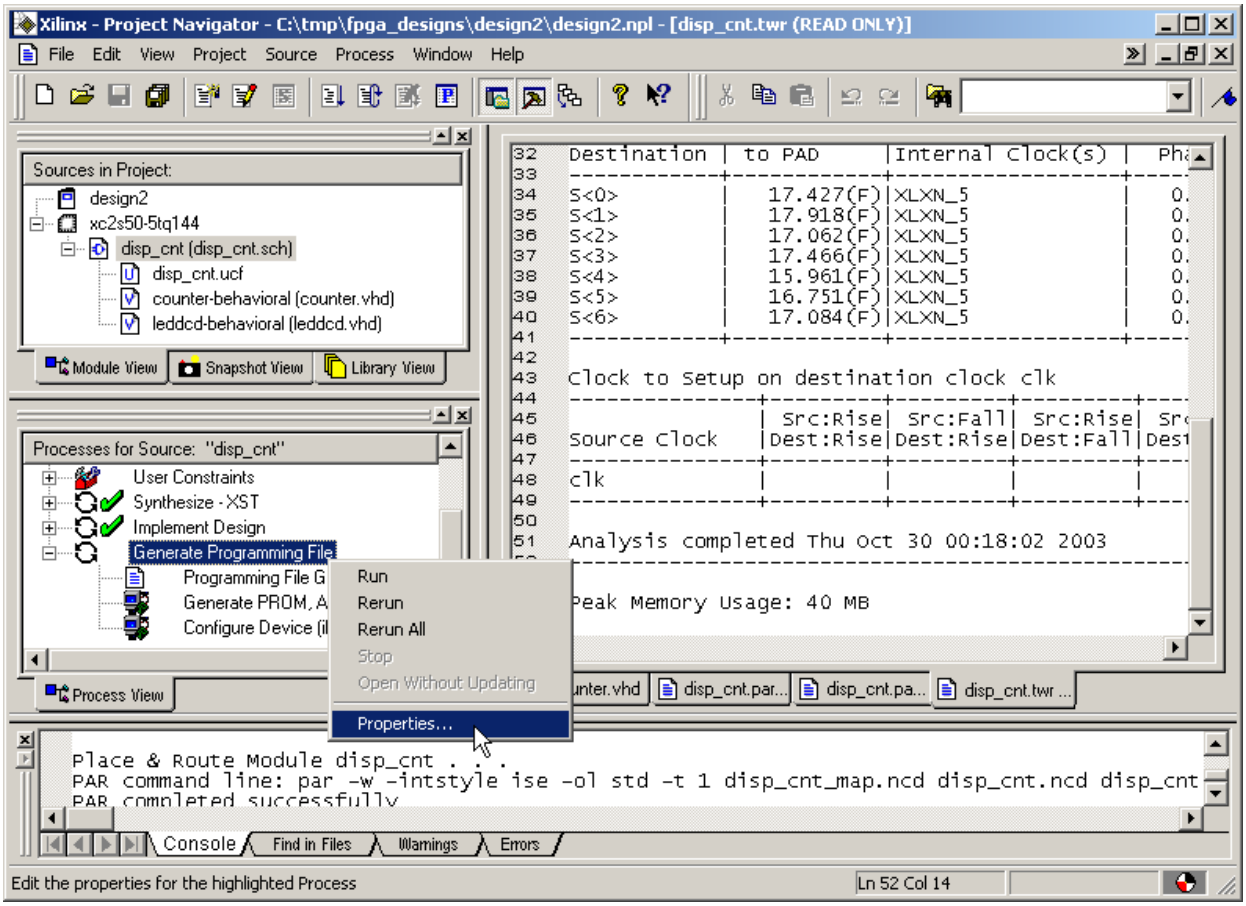
From the information shown in the timing report, we see the minimum clock period for this design is 6.188 ns which means the maximum clock frequency is 161.6 MHz. The default clock frequency on the XSA-50 Board is 50 MHz which is well below the maximum allowable frequency for this design.

```
Clock to Setup on destination clock clk
---------------+---------+---------+---------+---------+
               | Src:Rise| Src:Fall| Src:Rise| Src:Fall|
Source Clock   |Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
---------------+---------+---------+---------+---------+
clk            |         |         |         |    6.188|
---------------+---------+---------+---------+---------+
```

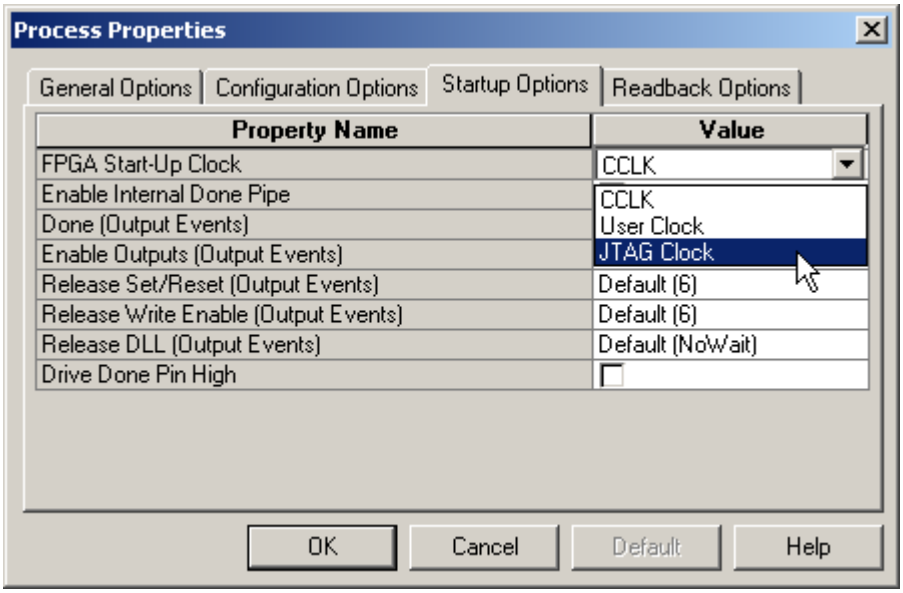
Generating the Bitstream

Now that we have synthesized our design and mapped it to the FPGA with the correct pin assignments, we are ready to generate the bitstream that is used to program the actual chip. In this example, rather than use the gxsload utility we will employ the downloading utilities built into WebPACK. The iMPACT programming tool downloads the bitstream through the JTAG interface of the FPGA so we need to adjust the way the bitstream is generated to account for

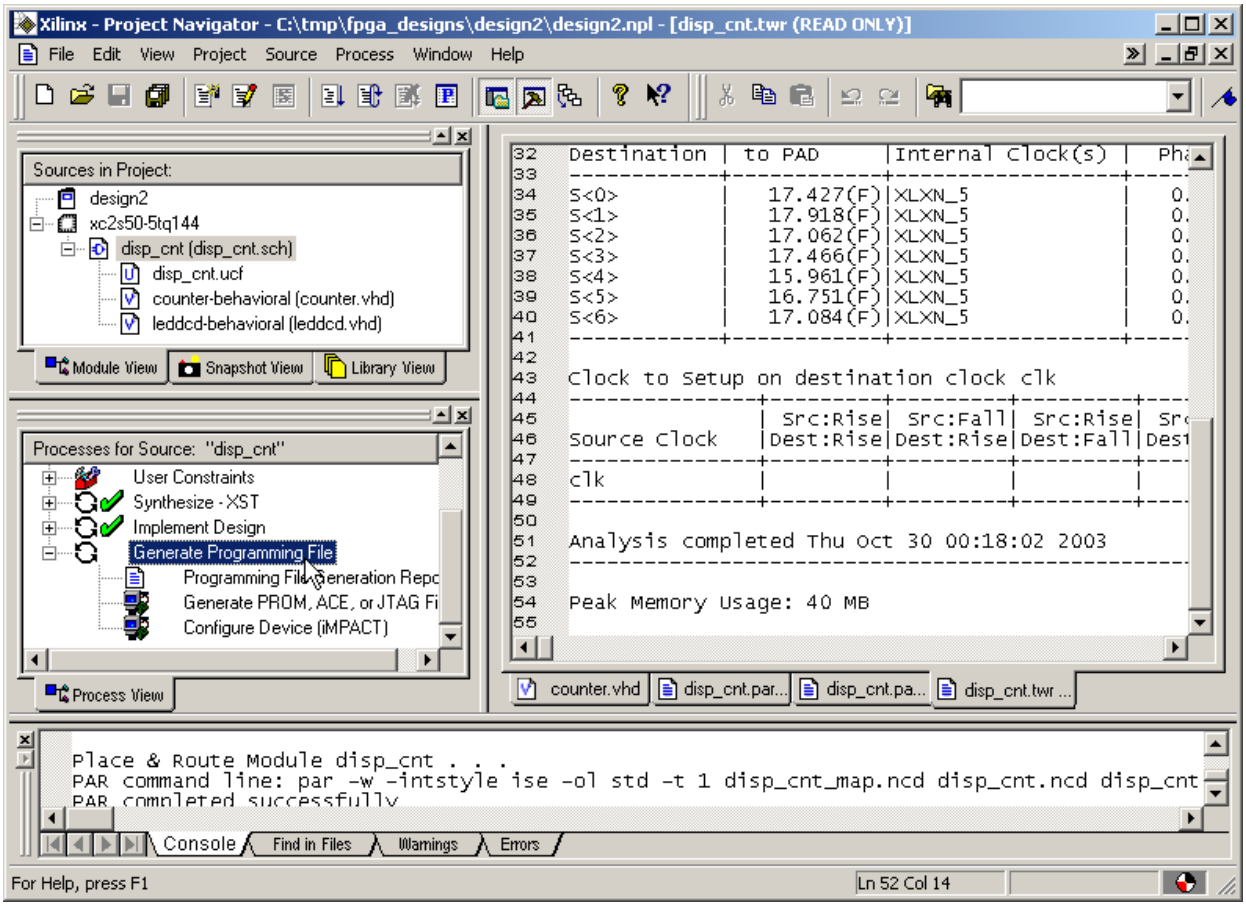
this. Right click on the Generate Programming File process and select the Properties... entry from the pop-up menu.



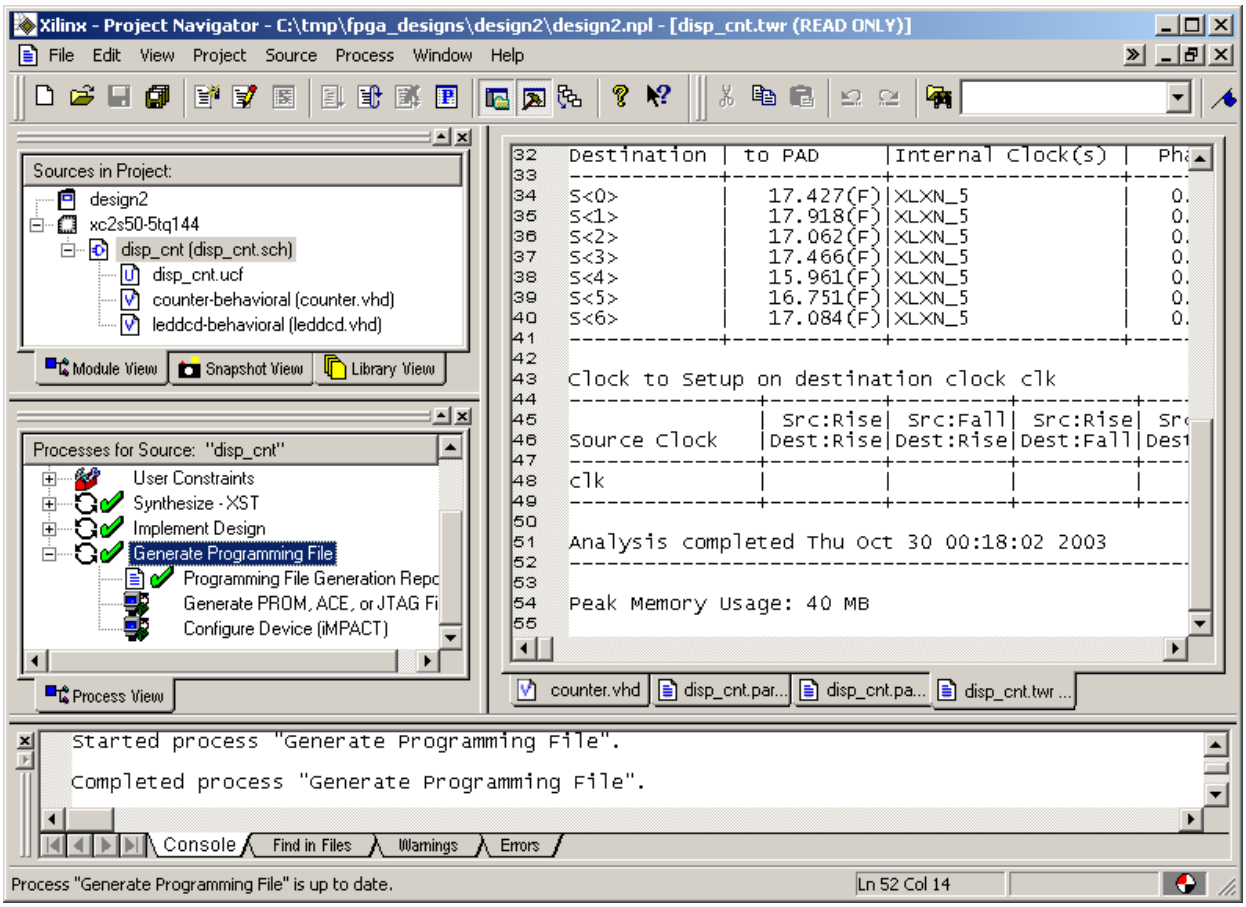
Select the Startup options tab of the Process Properties window. Change the Start-Up Clock property to JTAG Clock so the FPGA will react to the clock pulses put out by the iMPACT tool during the final phase of the downloading process. If this option is not selected, the FPGA will not finish its configuration process and it will fail to operate after the downloading completes. Note that the startup clock is only used to complete the configuration process; it has no affect on the clock that is used to drive the actual circuit after the FPGA is configured.



After setting the Start-Up Clock option, click on the OK button. Then highlight the disp_cnt object in the Sources pane and double-click on the Generate Programming File process to create the bitstream file.



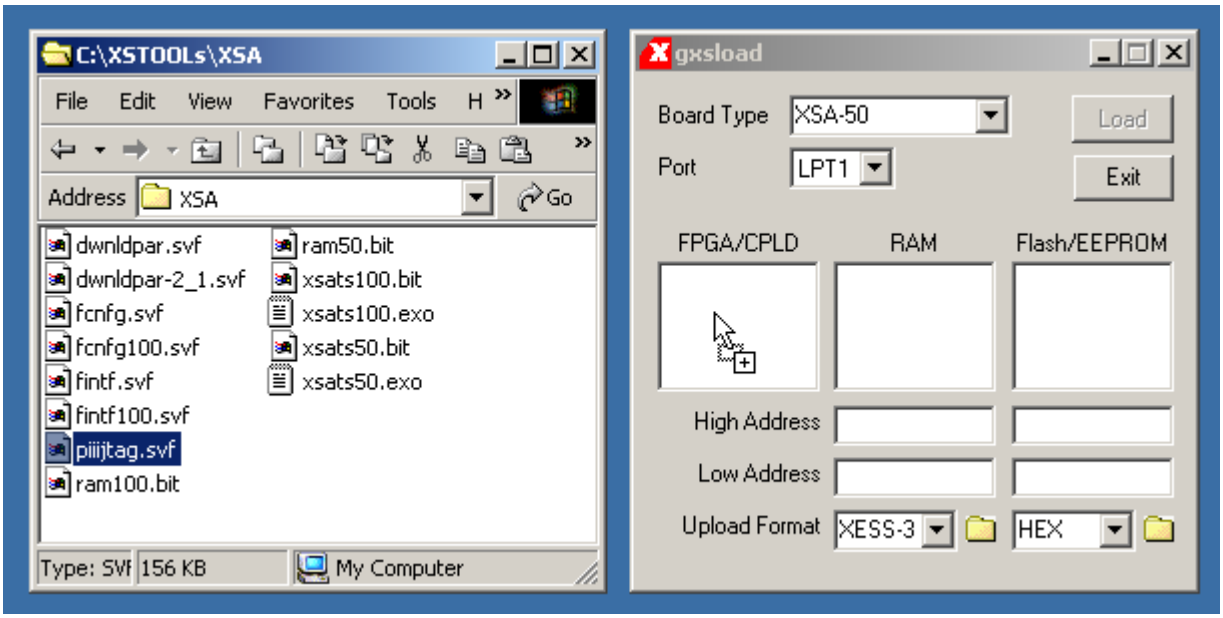
Within a few seconds, a ✔ will appear next to the Generate Programming File process and a file detailing the bitstream generation process will be created. A bitstream file named disp_cnt.bit can now be found in the design2 folder.



Downloading the Bitstream

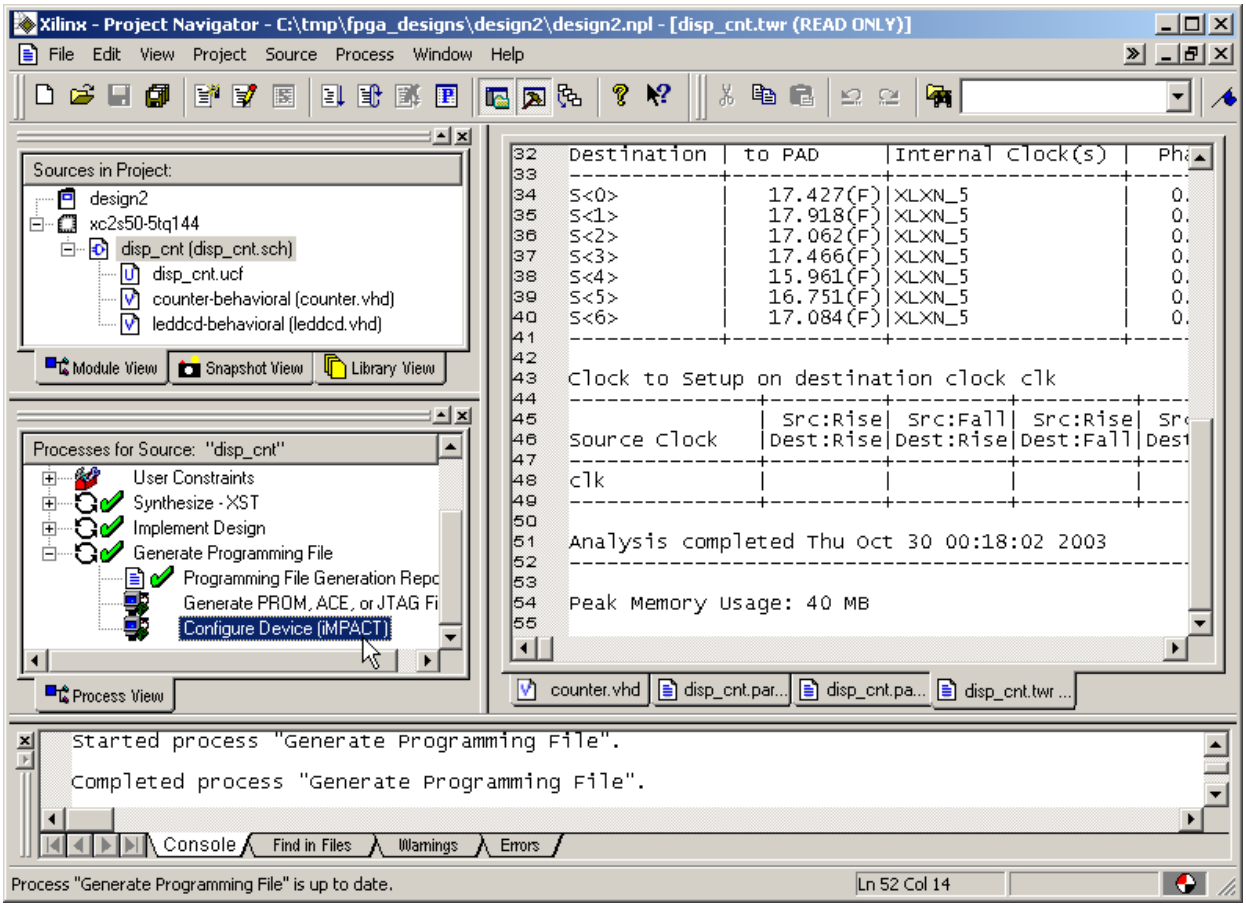
Before downloading the disp_cnt.bit file, we must configure the interface CPLD on the XSA-50 board so it will work with the iMPACT programming tool. Double click the 

icon and then drag & drop the piiijtag.svf file from the C:\XSTOOLS4\XSA folder into the **gxsload** window. Then click on the Load button and the CPLD will be reprogrammed in less than a minute.

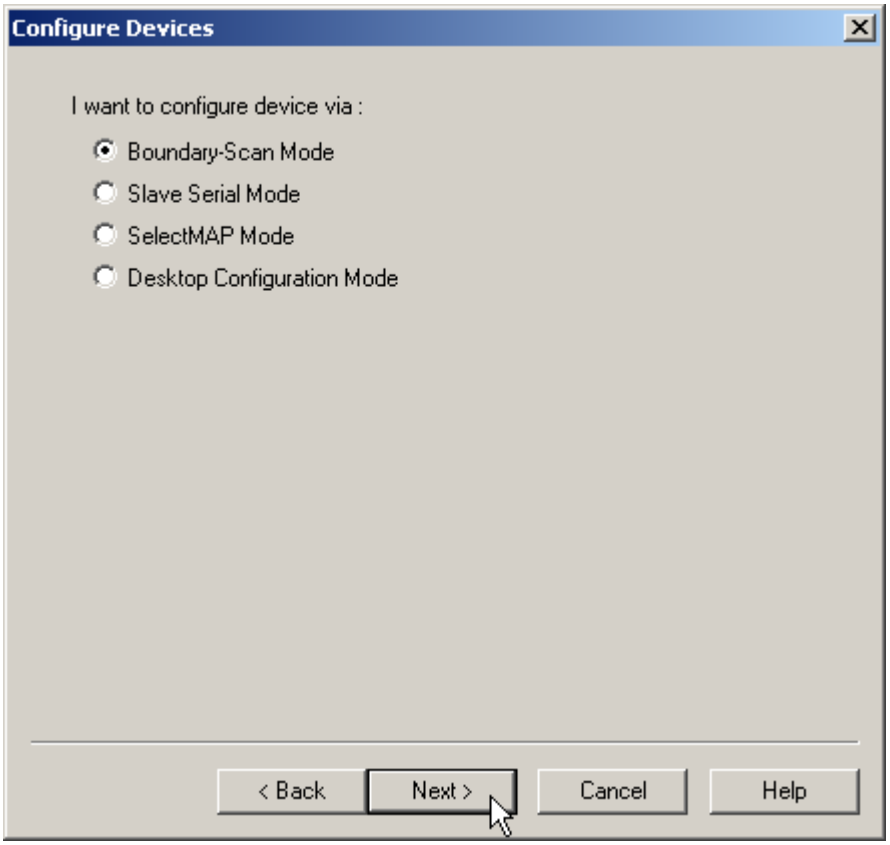


After the piiijtag.svf file is loaded into the XSA-50 Board, move the shunt on jumper J9 from the **xs** to the **xi** position. The XSA-50 Board is now setup so the FPGA can be configured through its boundary-scan pins with the iMPACT programming tool. Note that this process only needs to be done once because the CPLD on the XSA-50 Board will retain its configuration even when power is removed from the board. (If we want to go back to using the gxsload programming utility, we must move the shunt on J9 back to the **xs** position and download the dwnldpar.svf file into the CPLD.)

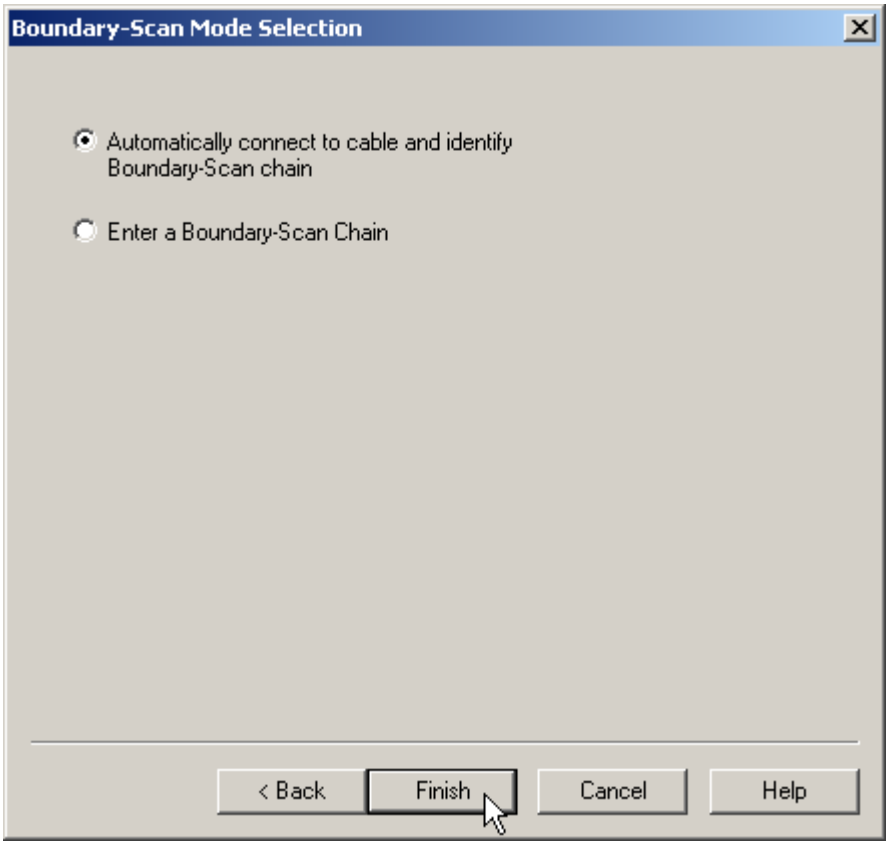
Now double-click on the Configure Device (iMPACT) process.



The **Configure Devices** window now appears. Previously, we programmed the CPLD on the XSA-50 Board so it would support programming of the FPGA through its boundary-scan pins. So select the Boundary-Scan Mode option and click on the Next button.

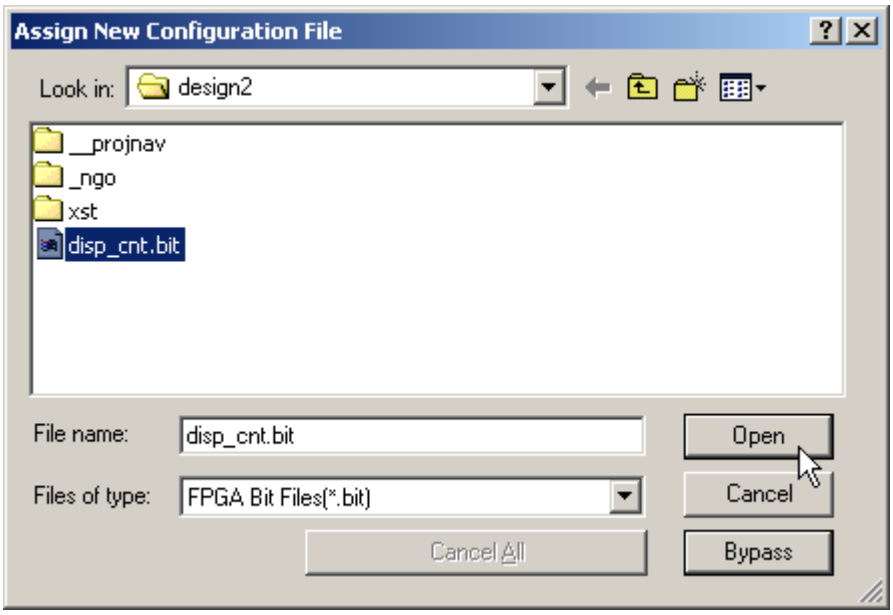


Boundary-scan mode allows the configuration of multiple FPGAs connected together in a chain. To accomplish this, the iMPACT software needs to know the types of the FPGAs in the chain. We know there is just a single FPGA on the XSA-50 Board and we could easily describe this to iMPACT. But iMPACT can also probe the boundary-scan chain and automatically identify the types of the FPGAs. This is even easier, so we select the automatic identification option and click on the Finish button.

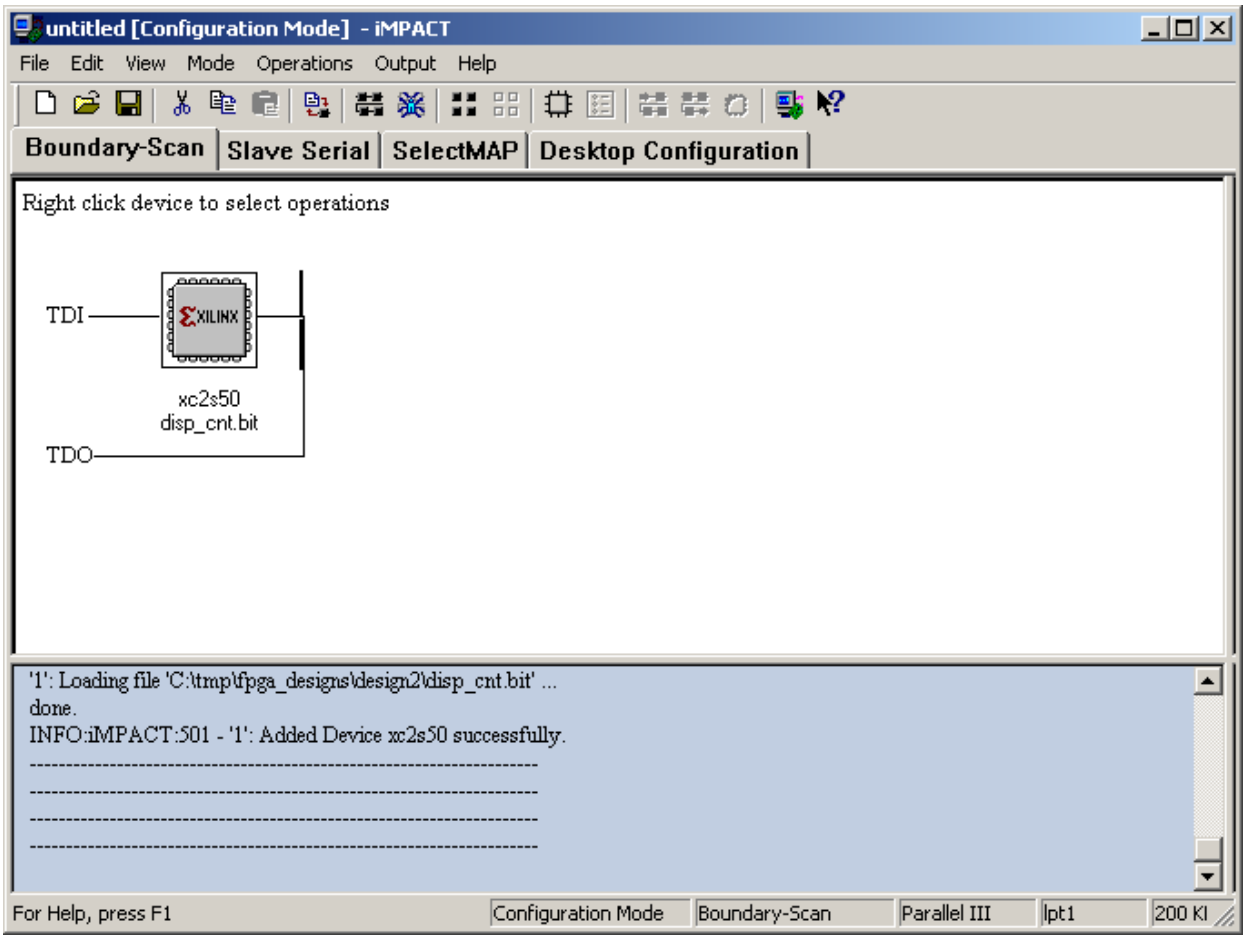


The iMPACT software will probe the boundary-scan chain and find there is a single FPGA in it. Now we need to tell iMPACT what bitstream file we want to download into this FPGA. Click on the OK button to proceed.

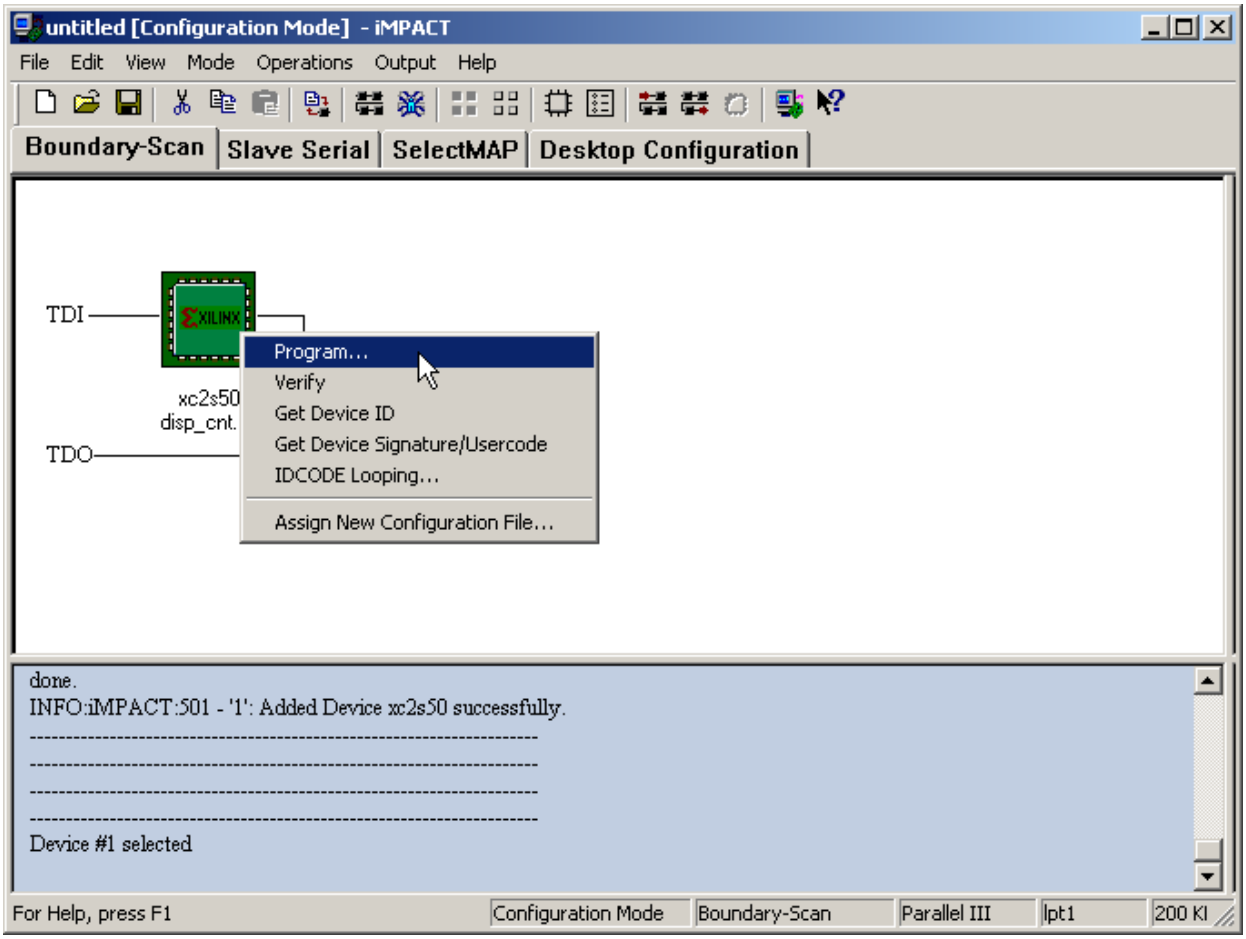
The **Assign New Configuration File** window now appears. Go to the tmp\fpga_designs\design2 folder and highlight the disp_cnt.bit file. Then click on the Open button.



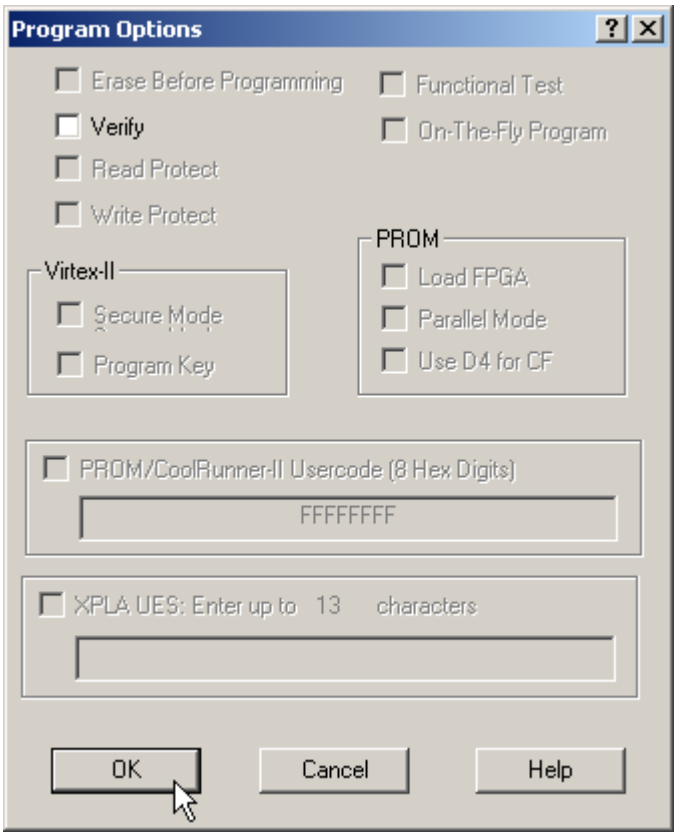
The main **iMPACT** window appears with a boundary-scan chain consisting of a single XC2S50 FPGA.



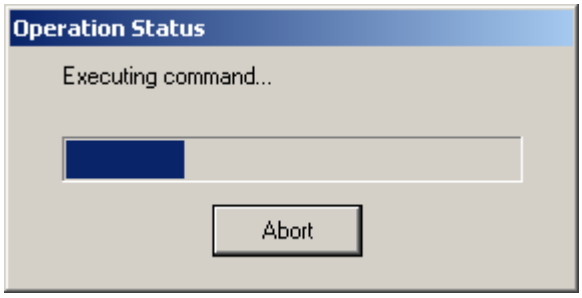
Now right-click on the XC2S50 icon and select the Program... item on the pop-up menu.



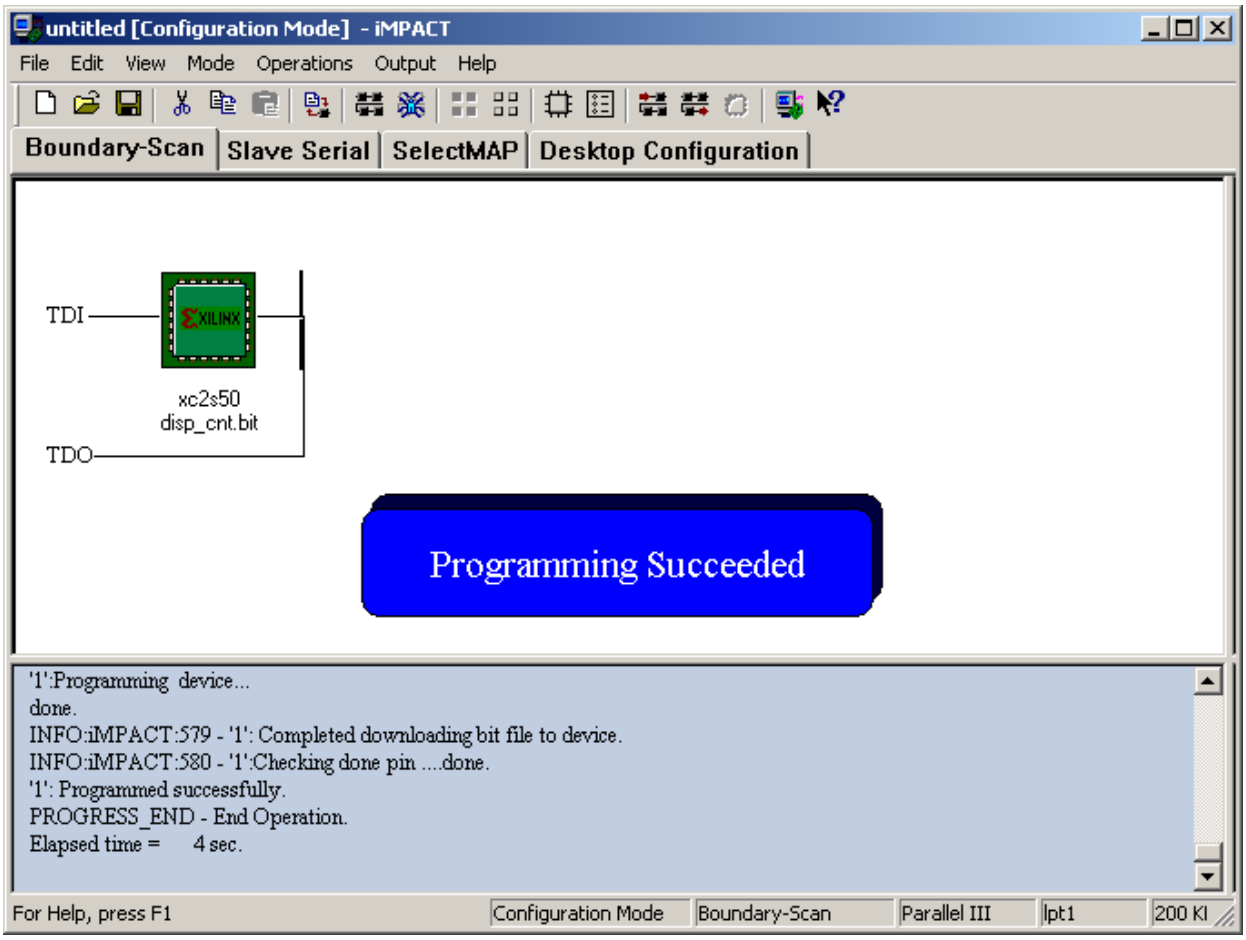
The **Program Options** window will appear. All we need to do at this point is click on the OK button to begin loading the disp_cnt.bit file into the FPGA.



The progress of the bitstream download will be displayed. The download operation should complete within twenty seconds.



After the download operation completes, we can check the status messages in the bottom pane of the **iMPACT** window to see the FPGA was configured successfully.



Testing the Circuit

Once the XC2S50 FPGA on the XSA-50 Board is programmed, the circuit will begin operating without any further action from us. The LED display should repeatedly count through the sequence 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F with a complete cycle taking 5.4 seconds.

5

Going Further…

OK! You made it to the end! You have scratched the surface of programmable logic design, but how do you learn even more? Here are a few easy things to do:

- In the Project Navigator window, select Help➔ISE Help Contents. You will be presented with a browser window containing topics that will let you learn more about the WebPACK software.
- Get *Essential VHDL* (ISBN:0-9669590-0-0) or *The Designer's Guide to VHDL* (ISBN:1-55860-270-4) to learn more about VHDL for logic design.
- Go to the Xilinx web site and read their application notes and data sheets.
- Read the *comp.arch.fpga* newsgroup for helpful questions and answers about programmable logic design.