Introduction to WebPACK 4.1 for CPLDs

Using Xilinx WebPACK Software to Create CPLD Designs for the XS95 Board

Release date: 10/29/2001

Table of Contents

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 a CPLD project.
- How to target a design to a particular type of CPLD.
- 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 a CPLD.
- How to check device utilization and timing for a CPLD.
- How to generate a bitstream for a CPLD.
- How to download a bitstream to program a CPLD.
- How to test the programmed CPLD.

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 a CPLD.
- It will not teach you what to do if your design doesn't fit in a particular CPLD.
- It will not show you every feature of the WebPACK software and discuss how to set every option and property.

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

1

CPLD Programming

Implementing a logic design with a CPLD 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 a *fitter* program to map the logic gates and interconnections into the CPLD. The CPLD consists of several *function blocks* which can be further decomposed into *macrocells* that can perform logic operations. The function blocks and macrocells are interwoven with various *routing matrices*. The fitter assigns gates from your netlist to various macrocells in the function blocks and opens or closes switches in the routing matrices to connect the gates together.

4. Once the fitting 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 CPLD chip (usually embedded in some larger system). The electronic switches in the CPLD open or close in response to the binary bits in the bitstream. Upon completion of the downloading, the CPLD will perform the operations specified by your HDL code or schematic.

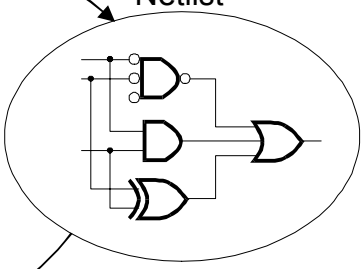
That's really all there is to it. Xilinx WebPACK provides the HDL and schematic editors, logic synthesizer, fitter, and bitstream generator software. The XSTOOLs from XESS provide utilities for downloading the bitstream into an XS95 Board containing a Xilinx XC95108 CPLD.

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;
```

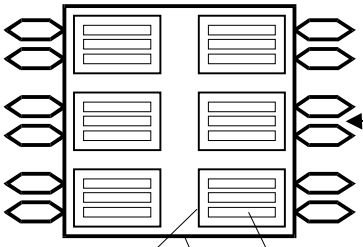
Synthesize

Netlist



Fit

CPLD



function block

macrocell

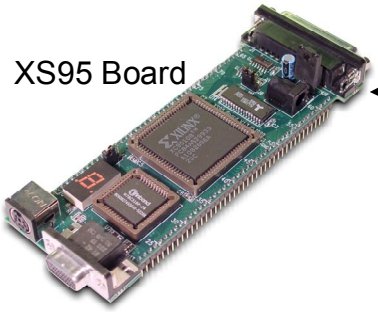
Generate Bitstream

Bitstream

```
101010010101100101
010110101010110101
010110100101101011
01010100101010101
010101010100110101
011011011010100101
011010010101100101
100101100101010100
101010110100110100
101100110001010101
```

Download and Test

XS95 Board

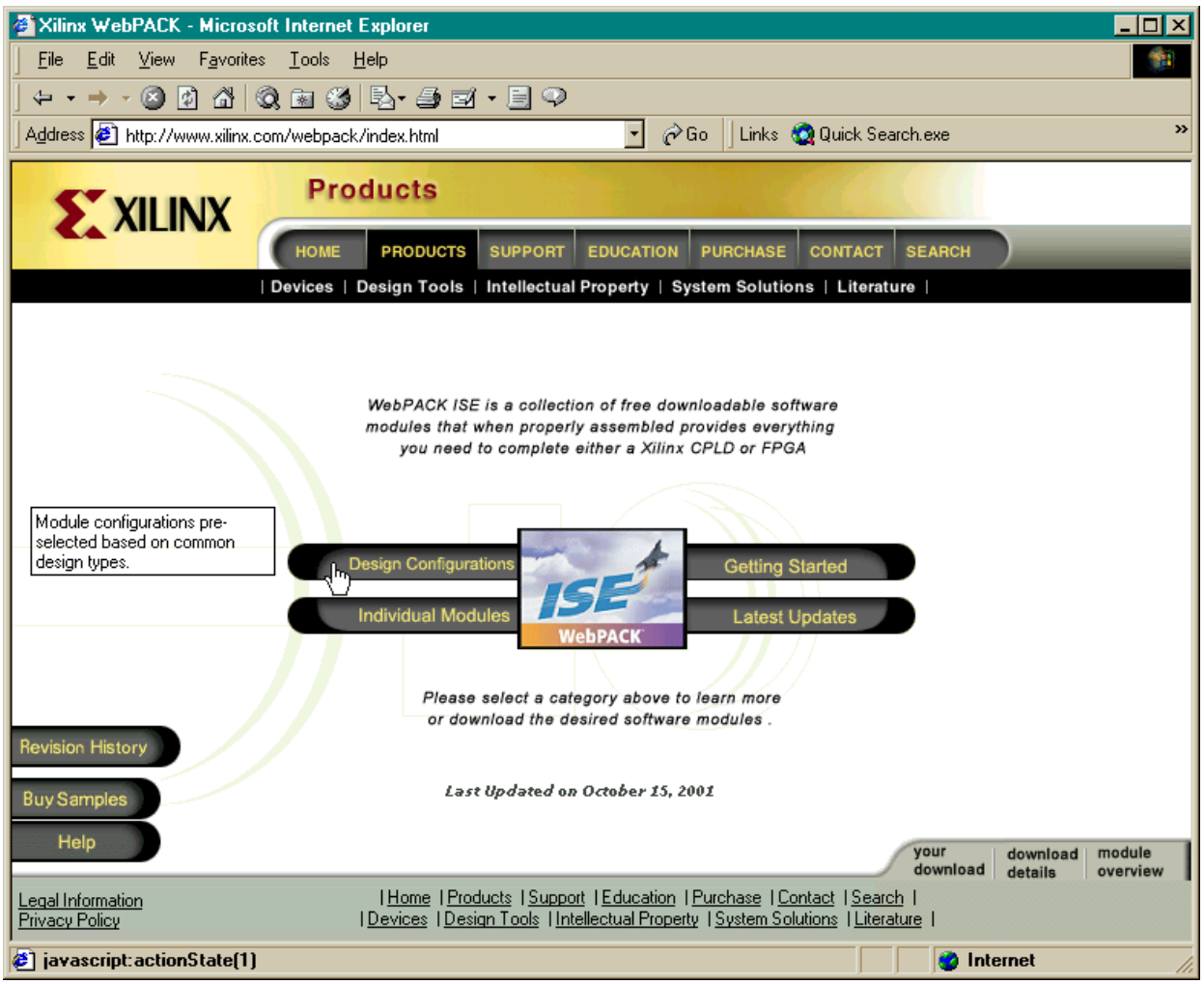


2

Installing WebPACK

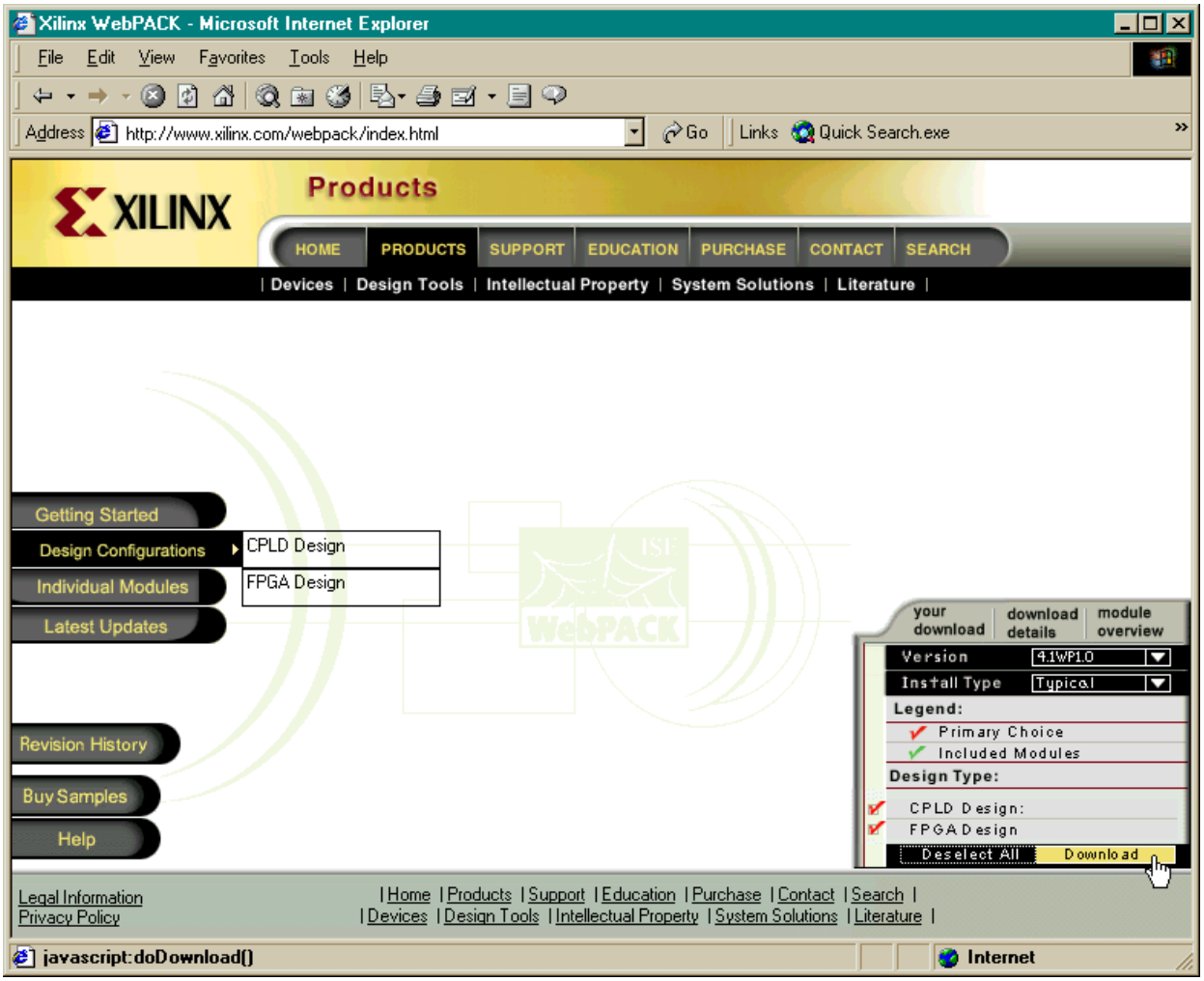
Getting WebPACK

Before downloading the WebPACK software you will have to register at http://www.Xilinx.com/xlnx/xil_entry2.jsp?sMode=login&group=webpack. 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.html to begin downloading the WebPACK software. After entering the WebPACK homepage, click on the Design Configurations button as shown on the next page.

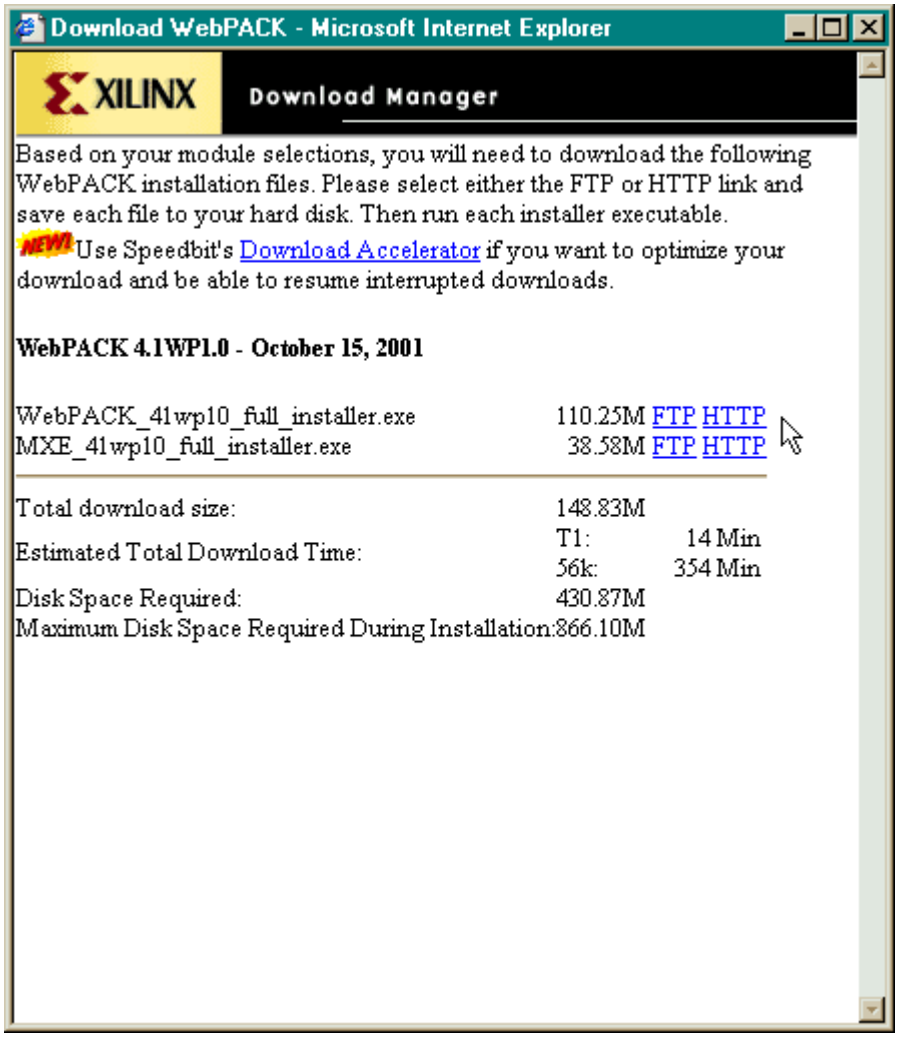


Next, click on the Select All button. This will select all the WebPACK software modules that cover both FPGA and CPLD designs.

Then click on the Download button to begin downloading the WebPACK software.



Click on the link to download the WebPACK software in the Download WebPACK window. You can use either the FTP or the HTTP link. (You can also download the demo version of the ModelSim HDL simulator but we will not discuss the operation of that software in this tutorial.)



Installing WebPACK

After the WebPACK software download completes, double-click the .EXE file. The installation script will run and install the software. Accept the default settings for everything and you shouldn't have any problems.

Getting XSTOOLs

If you are going to download your CPLD bitstreams into an XS95 Board, then you will need to get the XSTOOLS software from http://www.xess.com/ho07000.html. Just download the xstools4.exe file.

Installing XSTOOLs

Double-click the xstools4.exe file. The installation script will run and install the software. Accept the default settings for everything.

3

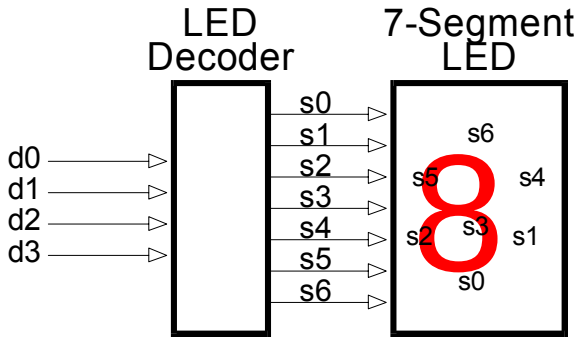
Our First Design

An LED Decoder

The first CPLD 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 

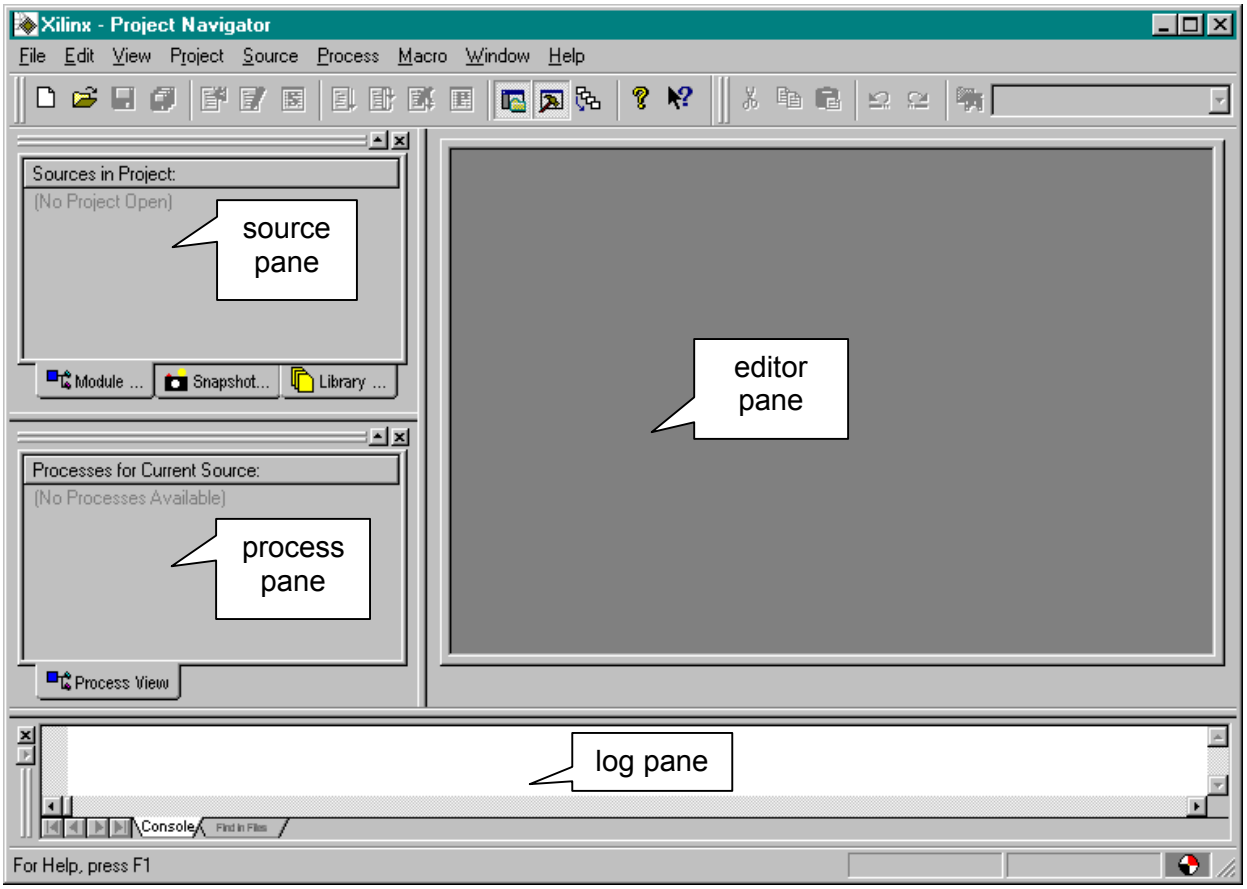
icon, ,on the desktop. This will bring up an empty project window as shown below. The window has four panes:

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.

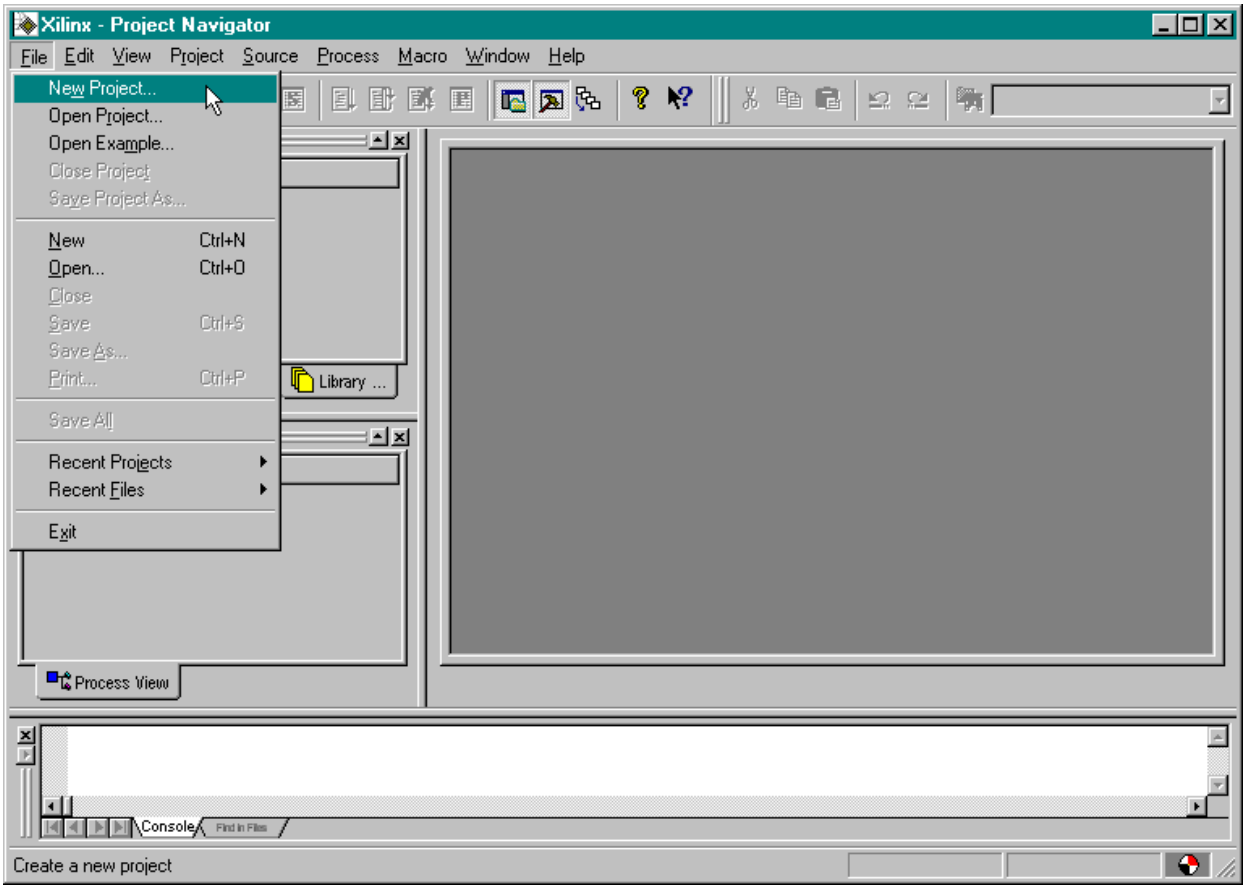
2. A **process pane** that lists the various operations we can perform on a given object in the source pane.

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

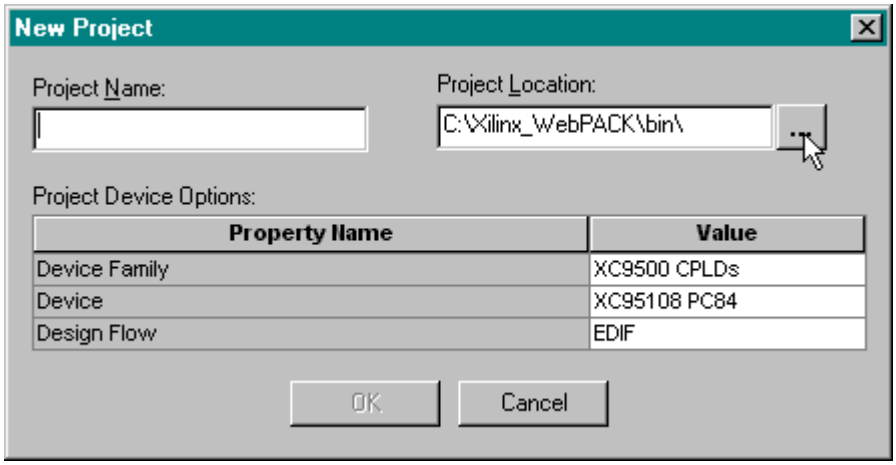
4. An **editor pane** where we can enter HDL code. Schematics are entered in a separate window.



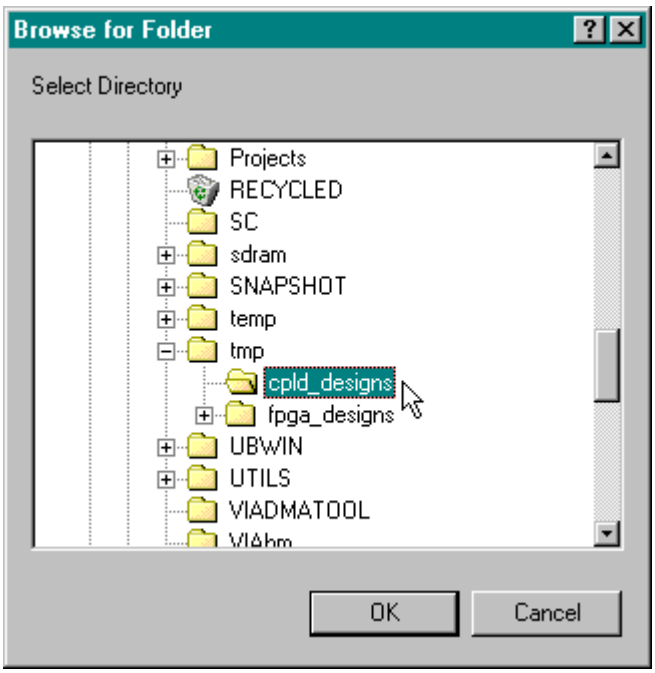
To start our design, we must create a new project by selecting the File➔New Project item from the menu bar.



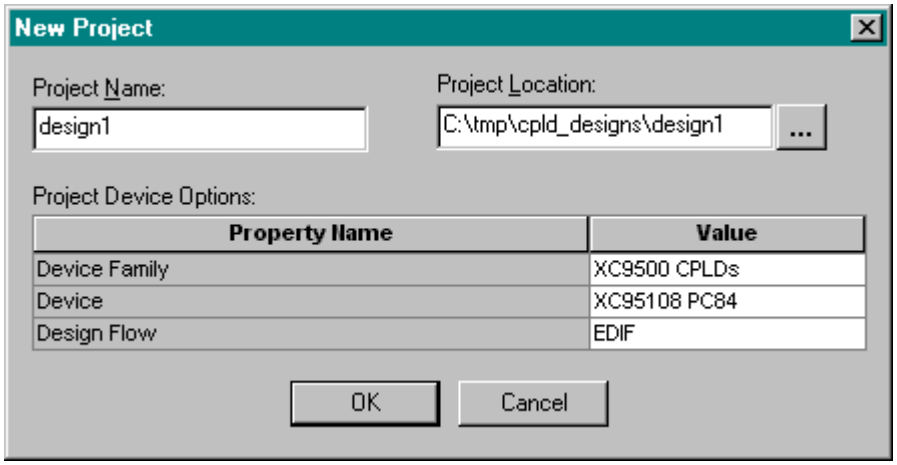
This brings up the **New Project** window where we can enter the location of our project files, project name, the target device for this design, and the tools used to synthesize logic from our source files.



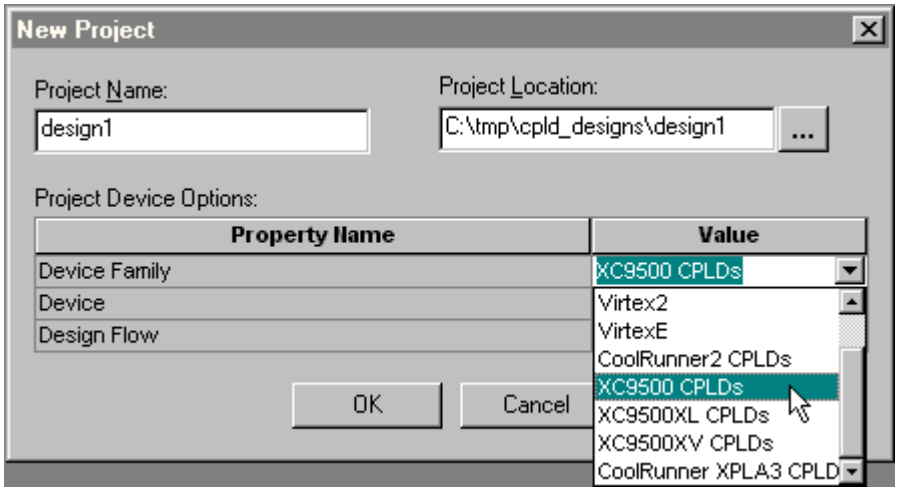
Click on the **…** button next to the Project Location field and use the **Browse for Folder** window to select a folder where our project files will be stored. For our design examples, we will store everything in the C:\tmp\cpld_designs folder.



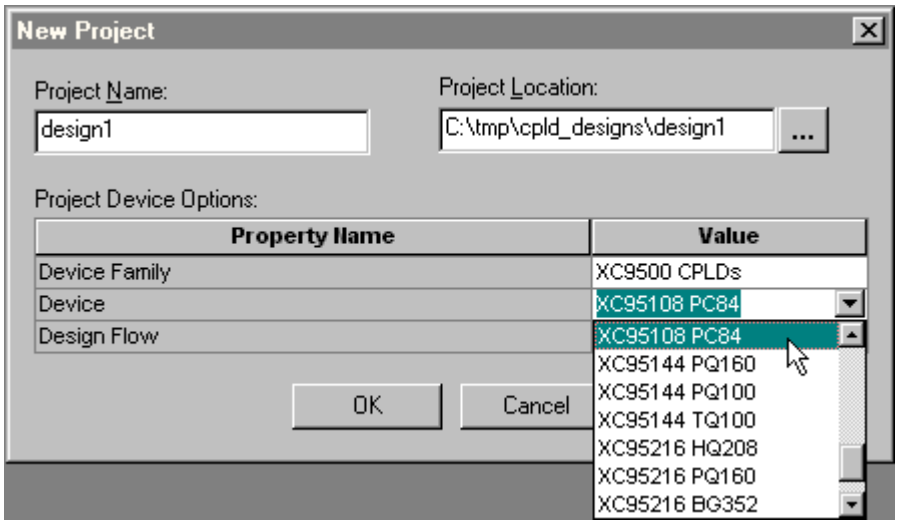
Next we will give our LED decoder design the descriptive title of `design1` by typing it into the Project name field.



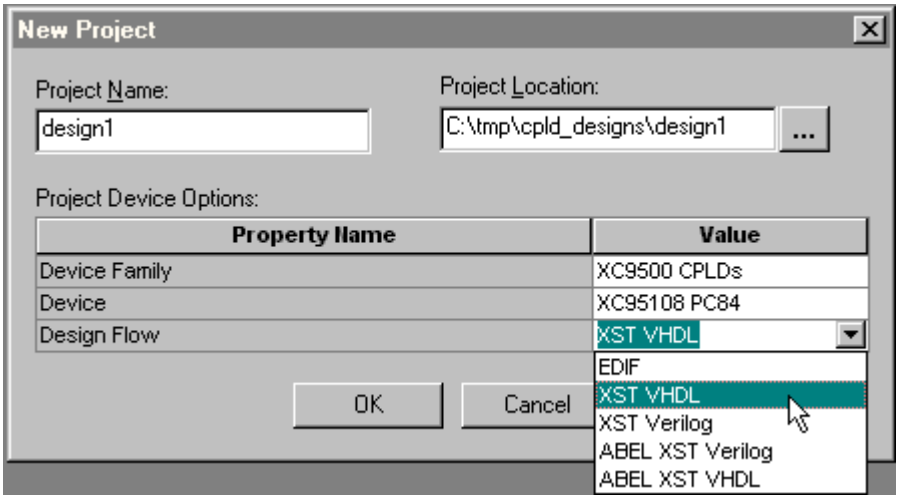
To set the family of CPLD devices we will target with this design, click in the Value field of the Device Family property. Select the XC9500 CPLDs entry in the pop-up menu that appears.



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 XC95108 PC84 since this is the device used in the XS95 Board where we will test our design.

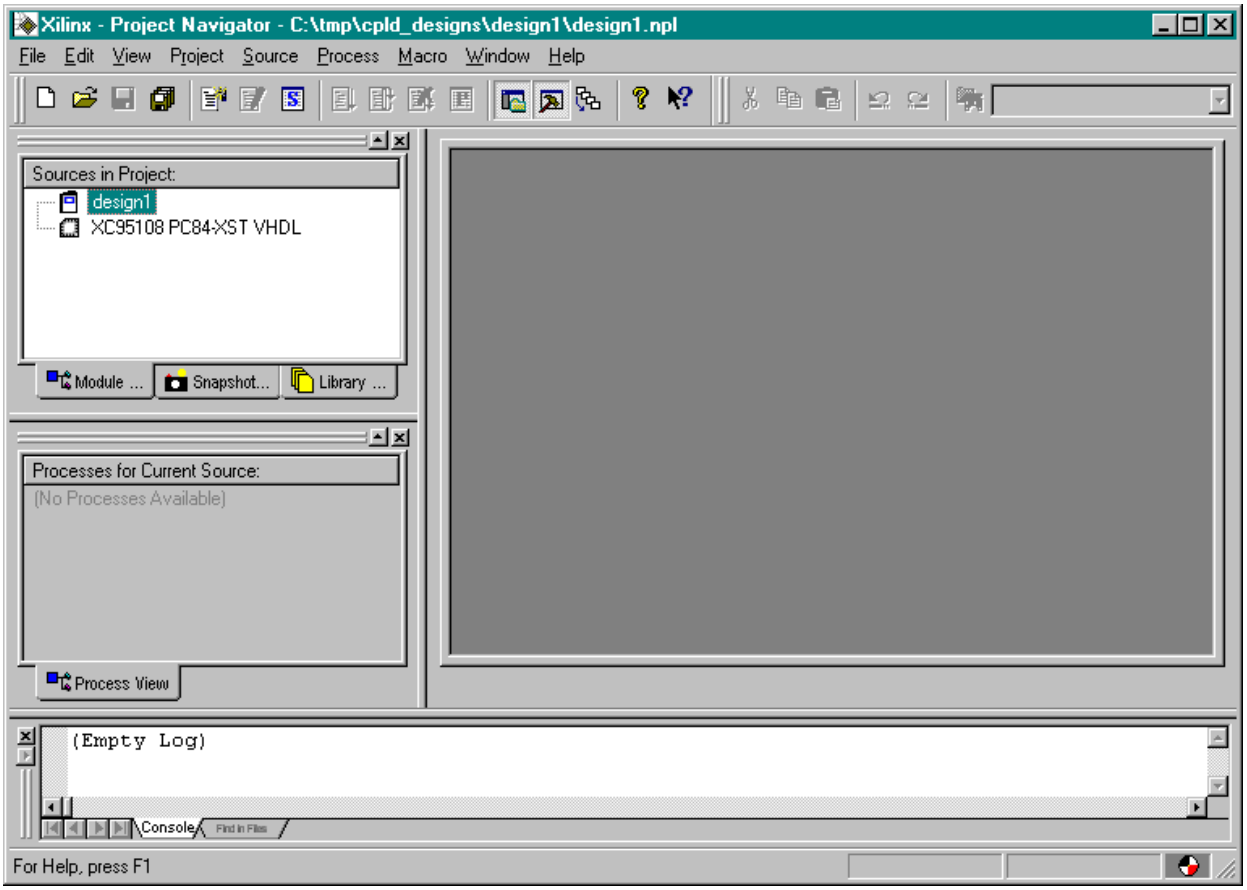


Finally, our design will be done using VHDL so click in the Value field of the Design Flow property and select XST VHDL from the pop-up menu. This enables the Xilinx VHDL synthesizer.



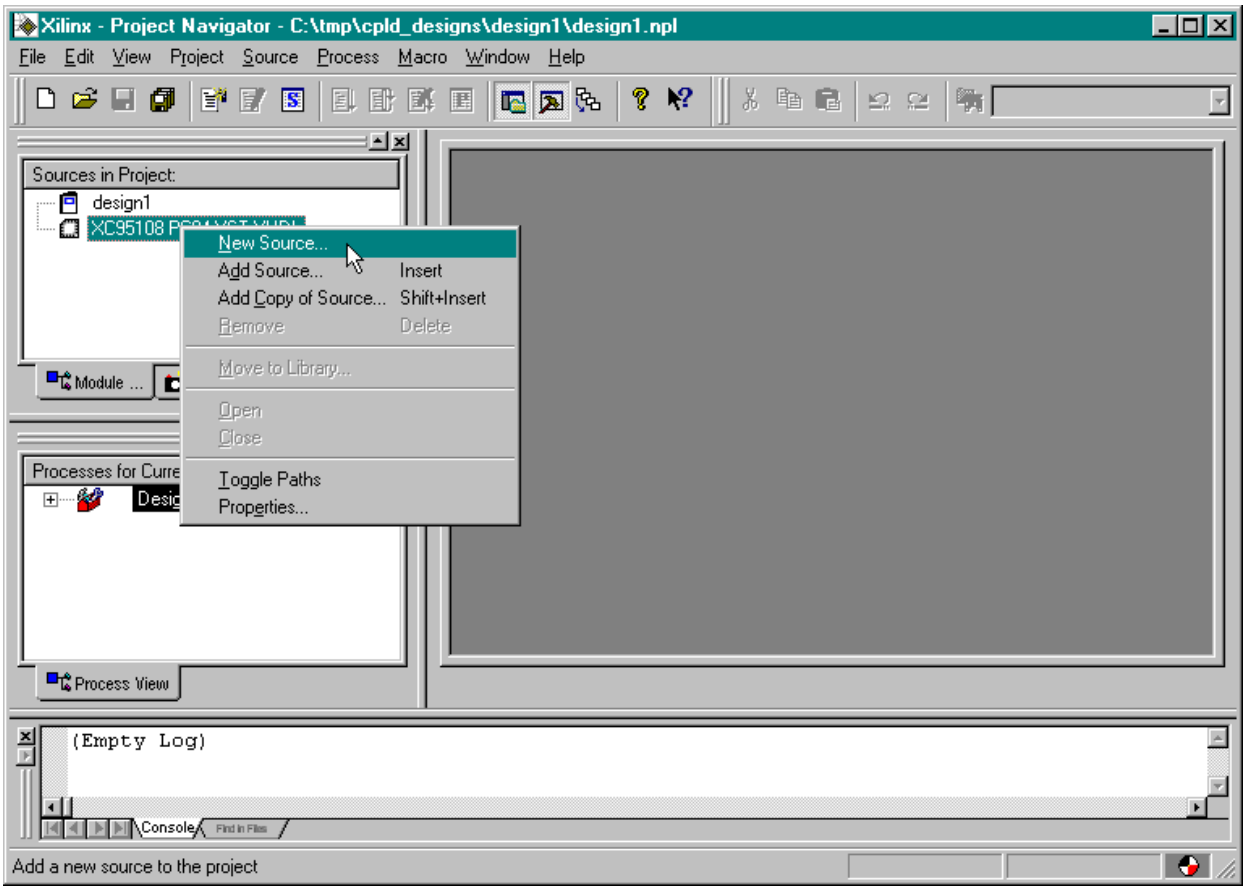
Once all the fields are set, click on OK in the **New Project** window. Now the Sources pane in the **Project Navigator** window contains two items:

1. A project object called design1.
2. A chip object called XC95108 PC84 - XST VHDL.

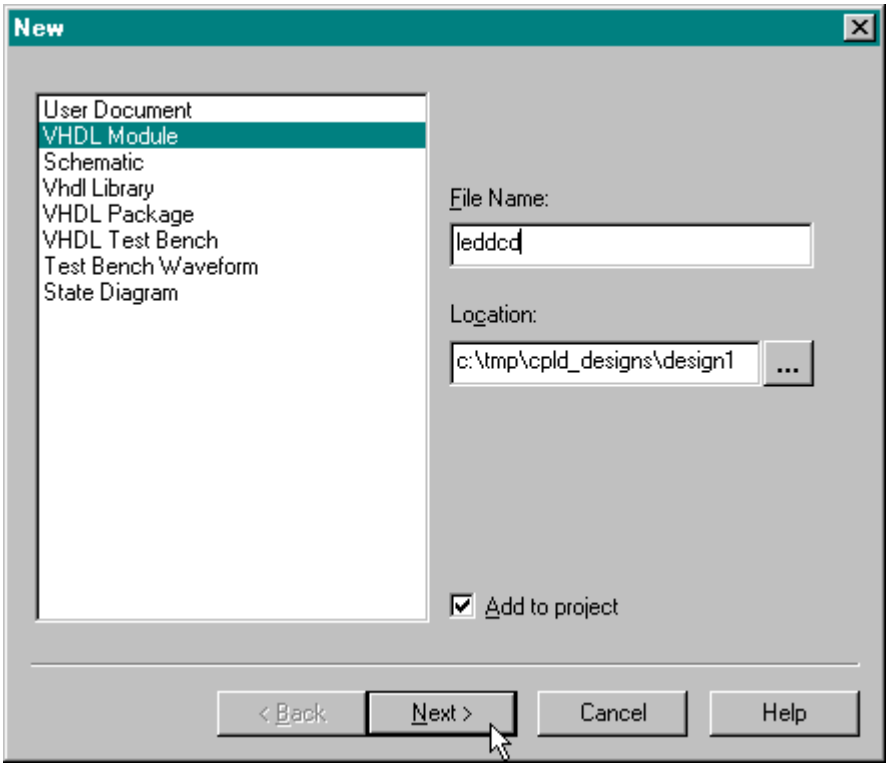


Describing Your Design With VHDL

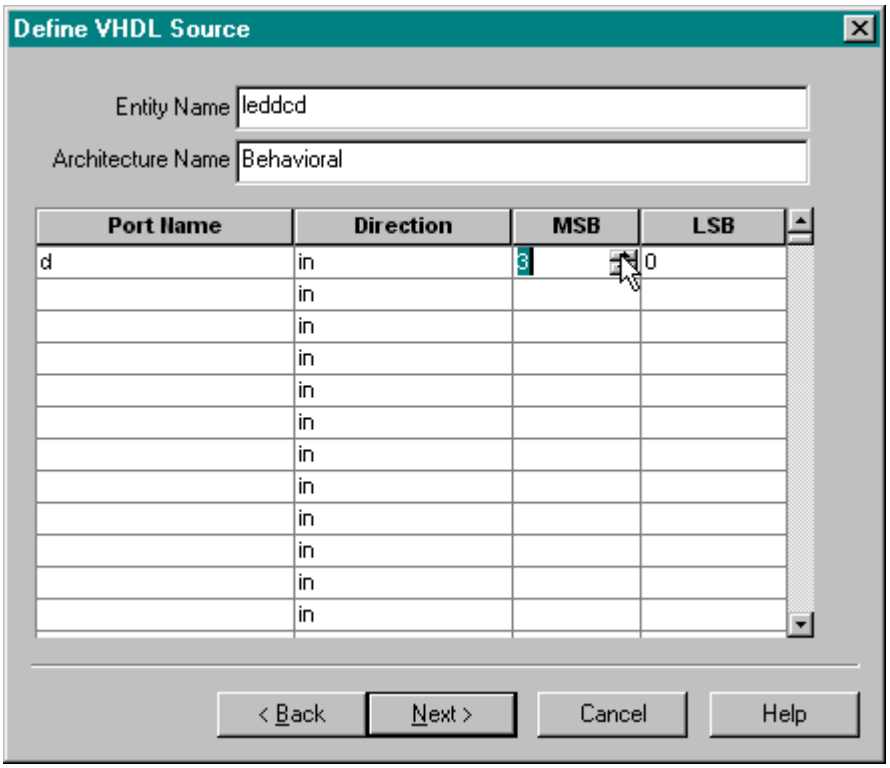
Once all the project set-up is complete, we can begin to actually design our LED decoder circuit. We start by adding a VHDL file to the ***design1*** project. Right-click on the XC95108 PC84 object in the Sources pane and select New Source ... from the pop-up menu as shown below.



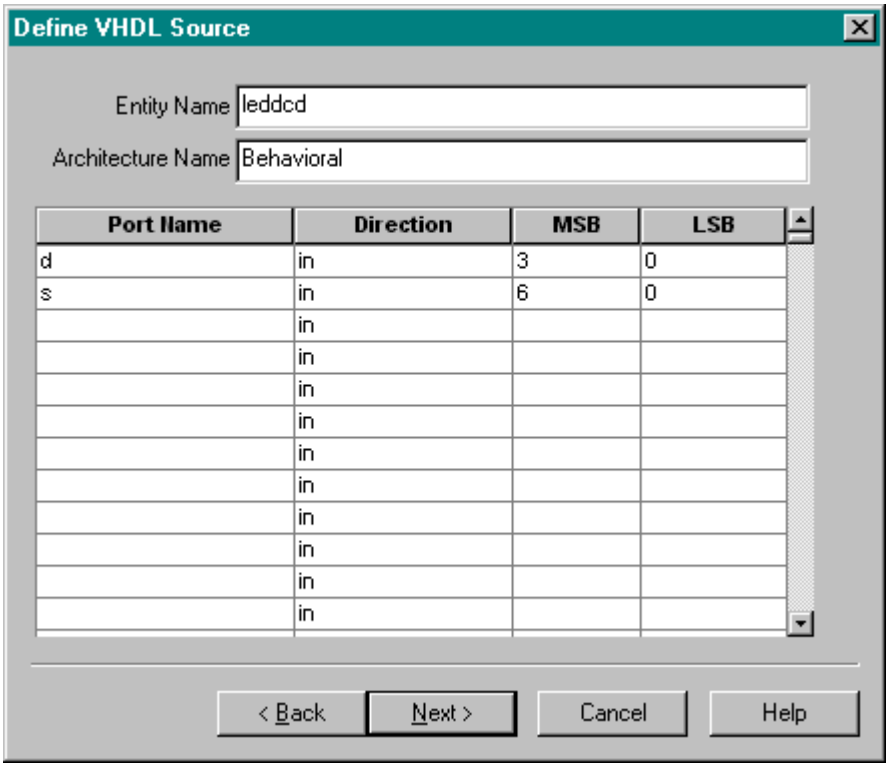
This causes a window to appear where we must select the type of source file we want to add. Since we are describing the LED decoder with VHDL, just highlight the VHDL Module item. Then we type the name of the module, `leddcd`, into the File Name field and click on Next.



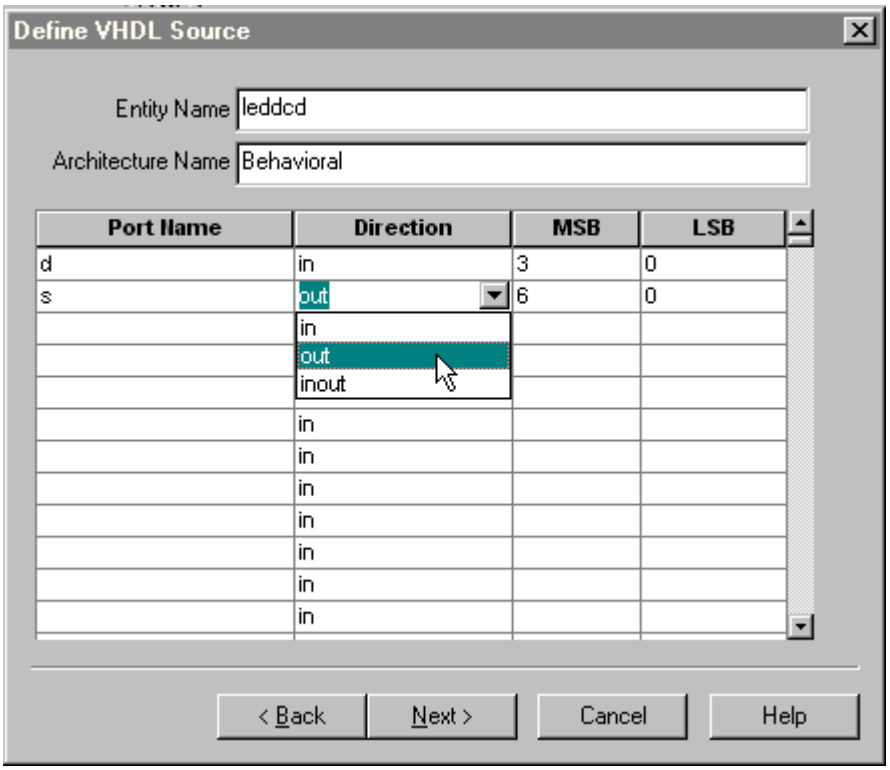
Then the **Define VHDL Source** window appears where we declare the inputs and outputs to the LED decoder circuit. In the first row, click in the Port Name field and type in d (the name of the inputs to the LED decoder). The **d** input bus has a width of four, so click in the MSB field and increment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



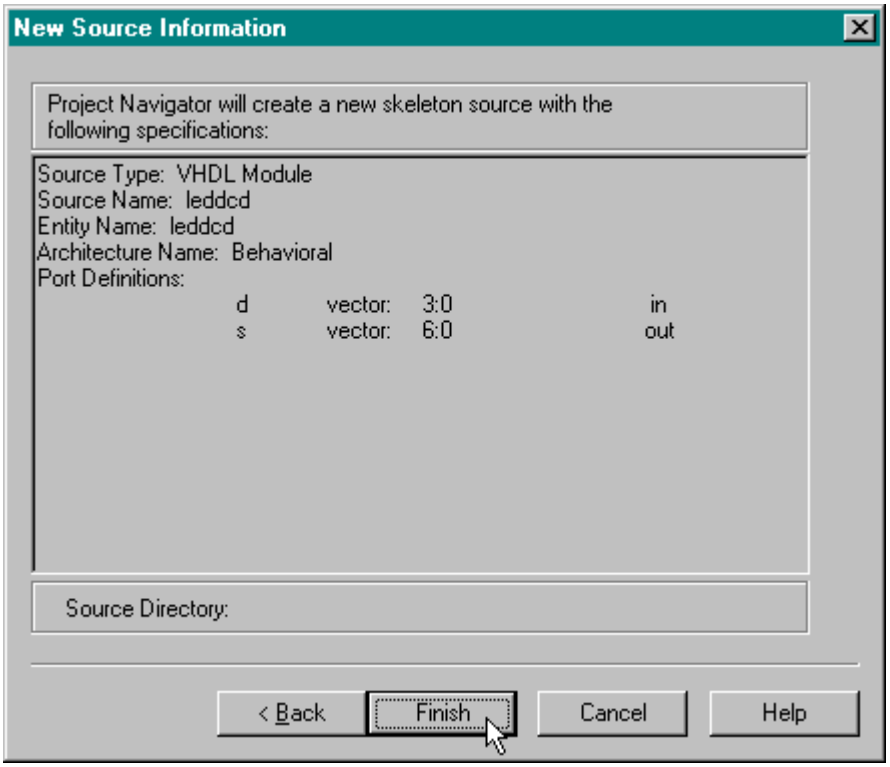
Perform the same operations to create the seven-bit wide **s** bus that drives the LEDs.



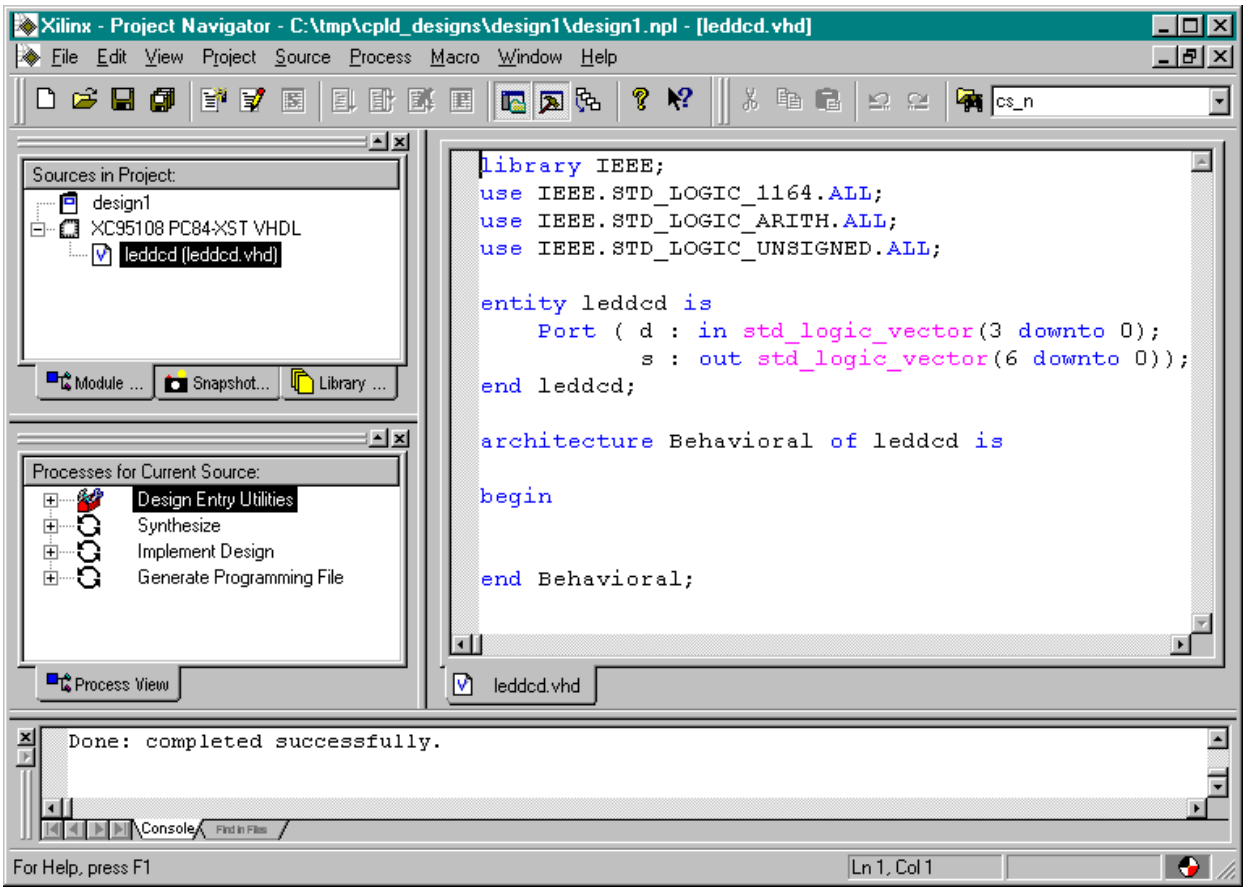
We must also click in the Direction field for the **s** bus and select out from the pop-up menu in order to make the **s** bus signals into outputs.



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 6-9. We will describe the logic operations of the decoder in the architecture section between lines 13 and 16.



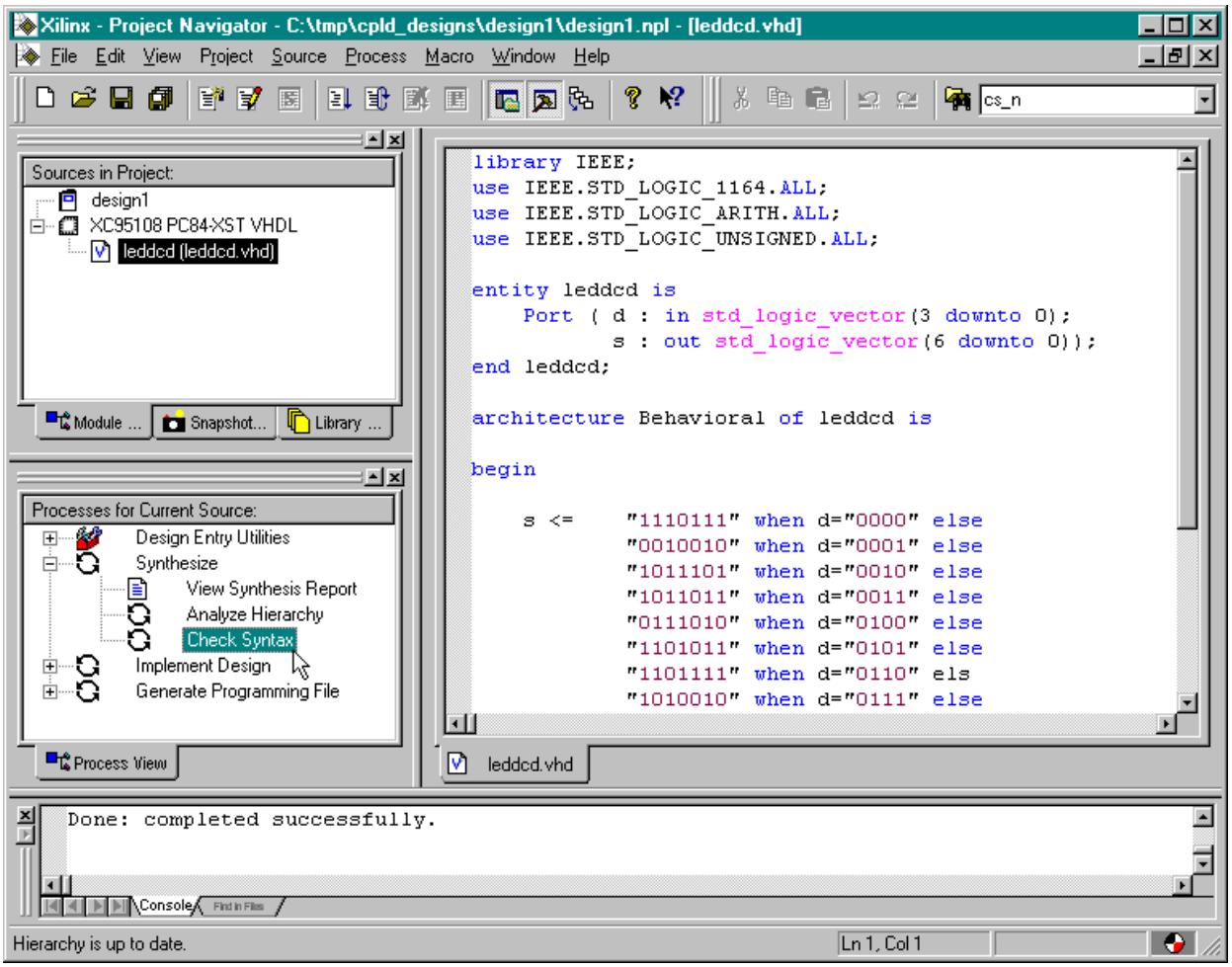
The completed VHDL file for the LED decoder is shown below. The architecture section contains a single statement which assigns a particular seven-bit pattern to the **s** output bus for any given four-bit input on the **d** bus (lines 15-30).

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity leddcd is
    Port ( d : in std_logic_vector(3 downto 0);
           s : out std_logic_vector(6 downto 0));
end leddcd;

architecture Behavioral of leddcd is

begin

    s <=    "1110111" when d="0000" else
            "0010010" when d="0001" else
            "1011101" when d="0010" else
            "1011011" when d="0011" else
            "0111010" when d="0100" else
            "1101011" when d="0101" else
            "1101111" when d="0110" els
            "1010010" when d="0111" else
            "1111111" when d="1000" else
            "1111011" when d="1001" else
            "1111110" when d="1010" else
            "0101111" when d="1011" else
            "0001101" when d="1100" else
            "0011111" when d="1101" else
            "1101101" when d="1110" else
            "1101100"

end Behavioral;
```

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

Checking the VHDL Syntax

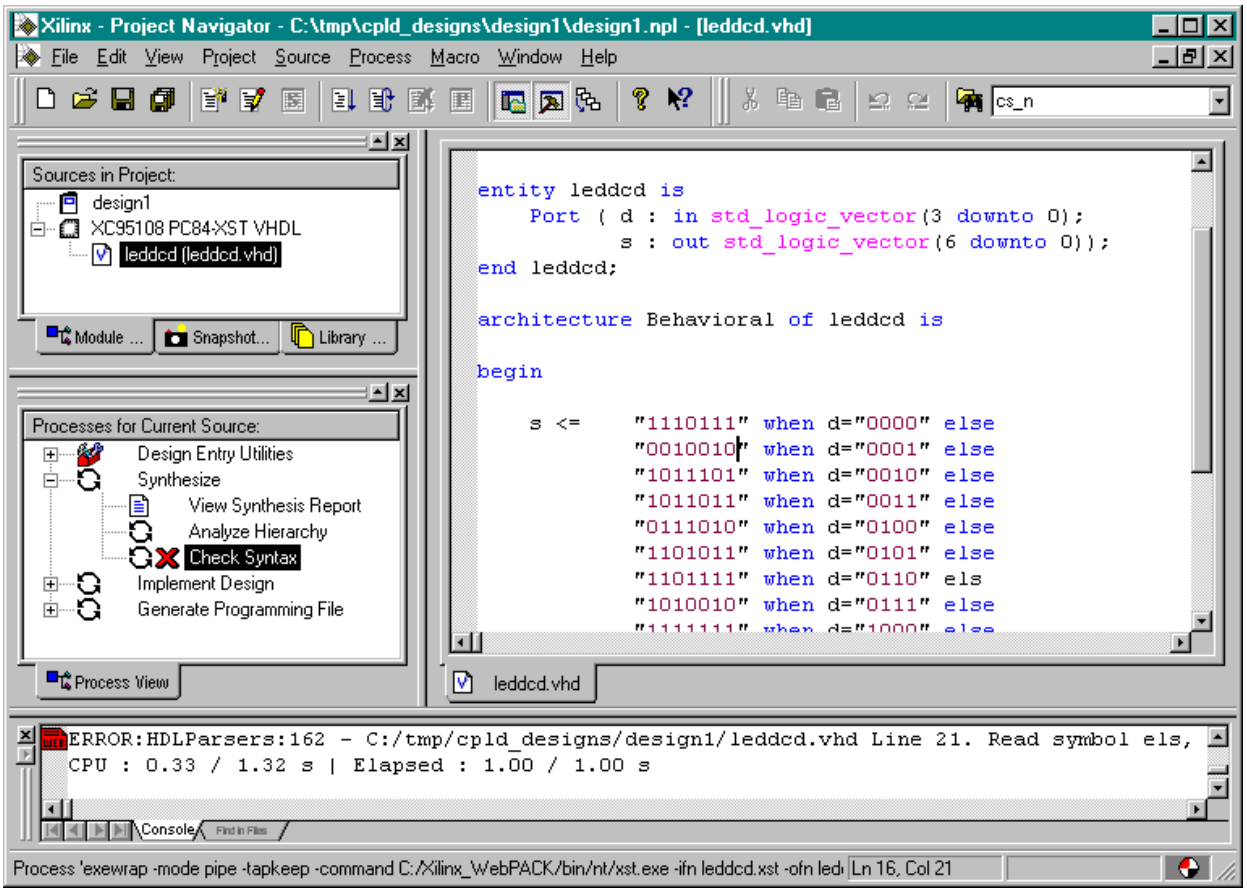
We can check for errors in our VHDL by highlighting the leddcd object in the Sources pane and then double-clicking on Check Syntax in the Process pane as shown below.



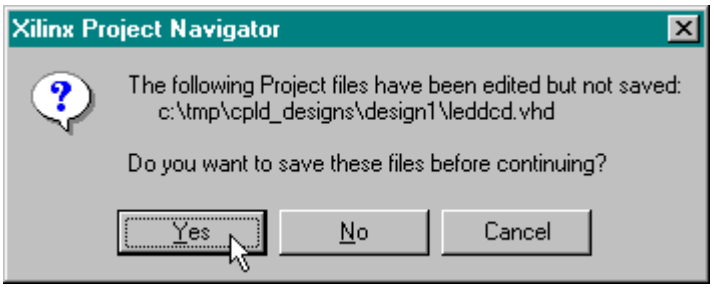
The syntax checking tool grinds away and then displays the result in the process window. In our case, an error was found as indicated by the ✗ next to the Check Syntax process. But what is the error and where is it?

Fixing VHDL Errors

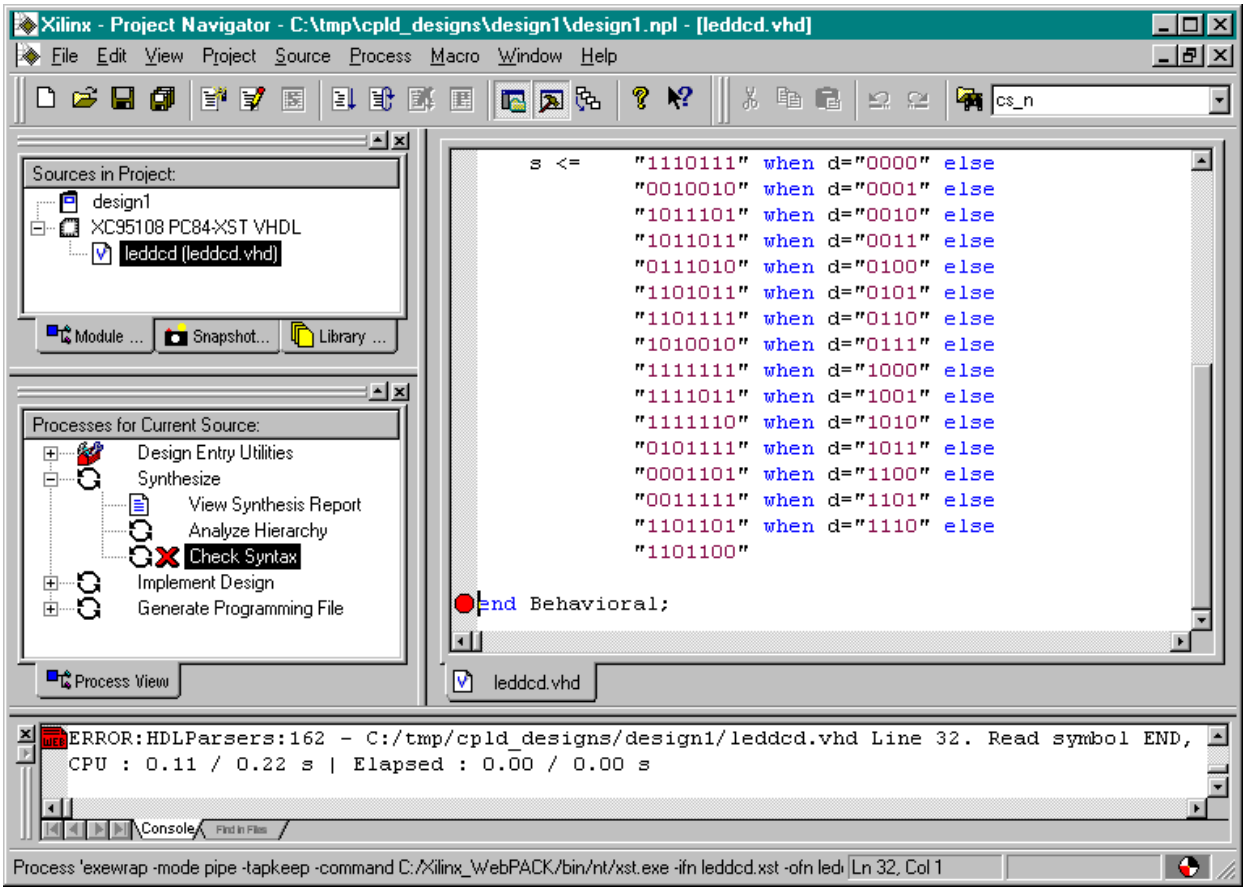
We can find the location of the error by scrolling the log pane at the bottom of the **Project Navigator** window until we find an error message. In this case, the error is located on line 20. You can manually scroll to line 20 in the editor pane, or you can double-click on the error message in the log pane to go directly to the erroroneous line.



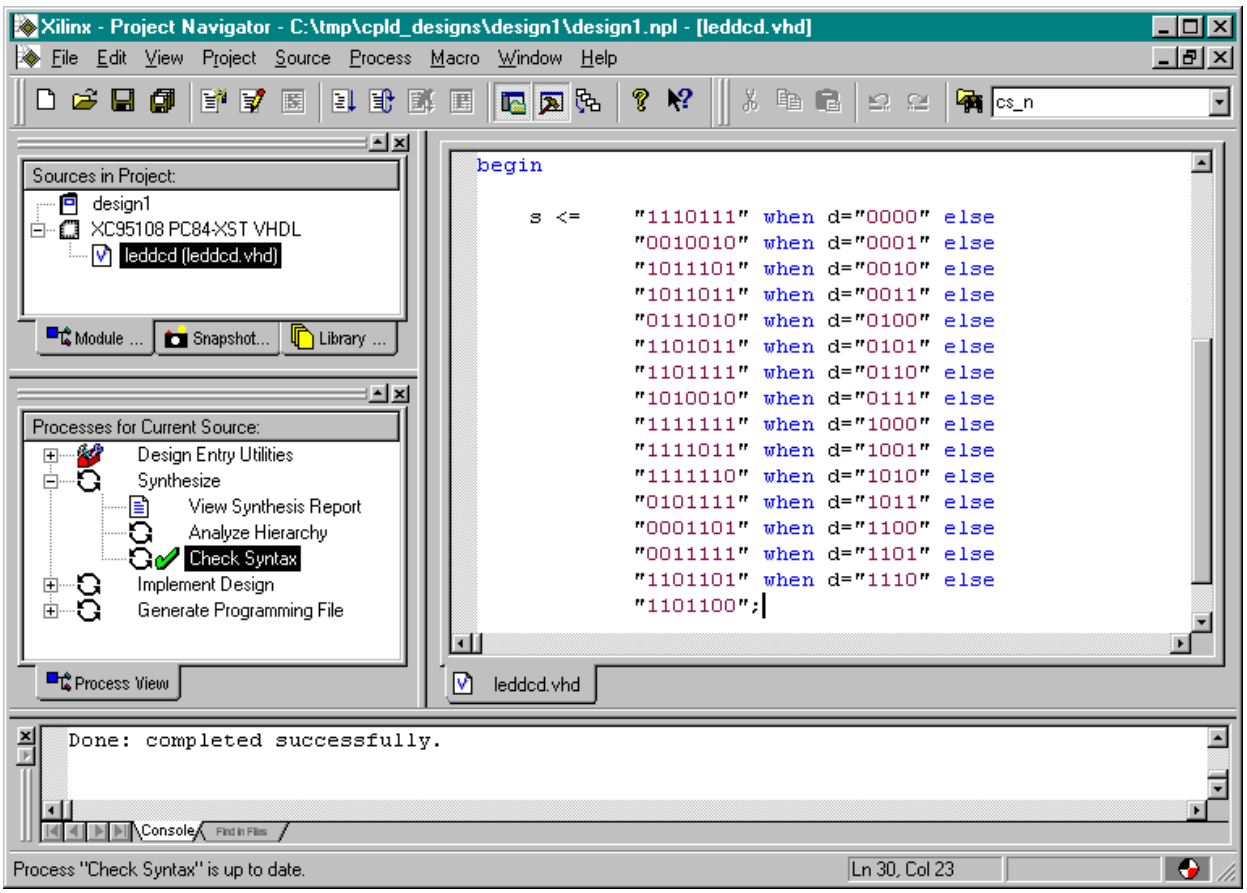
On line 20, we see that we have left the 'e' off the end of the `else` keyword. After correcting this error, we can double-click the on Check Syntax in the Process pane to re-check the VHDL code. We will be asked to save the file before the syntax check proceeds. Click on Yes.



The syntax checker now finds another error on line 31 of the VHDL code.

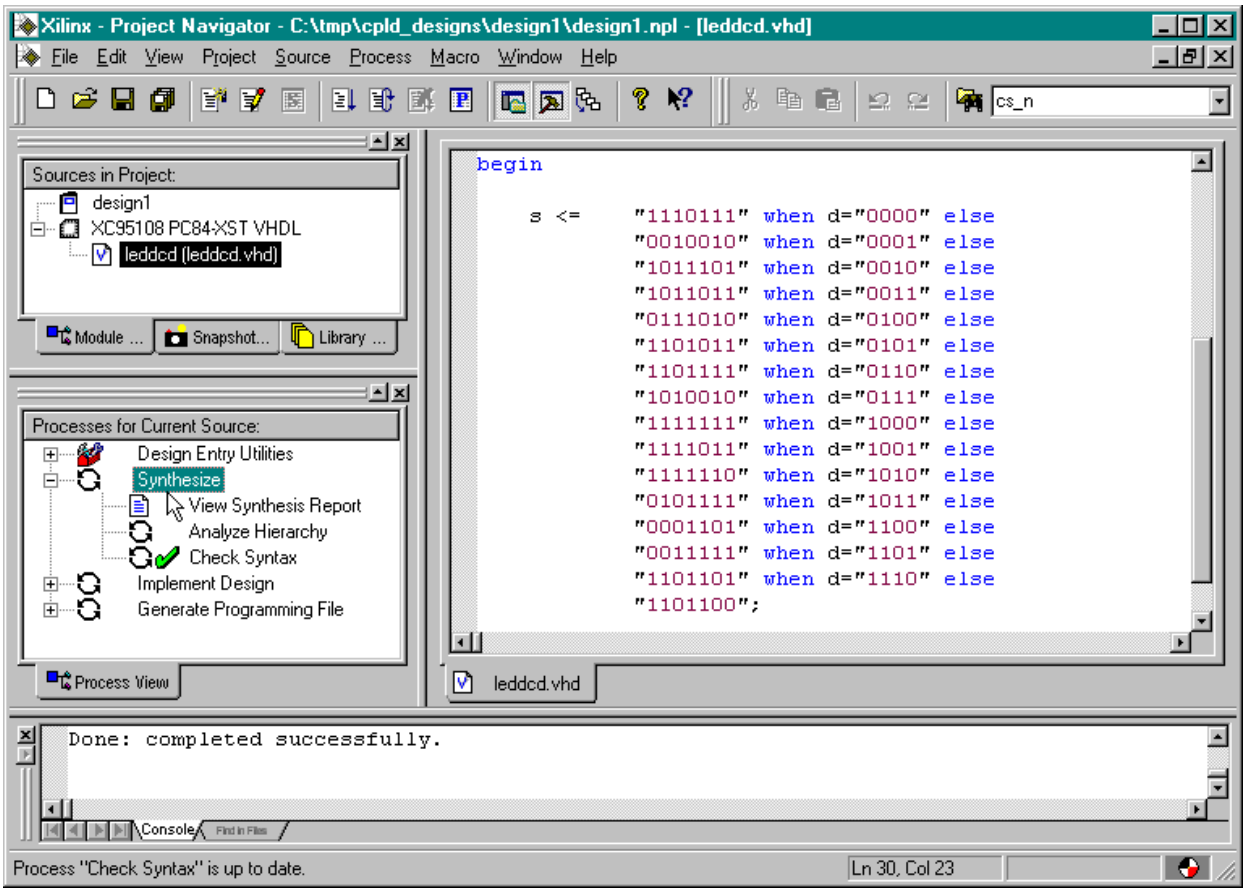


When we look at line 31 we see it is just the end statement for the architecture section. The VHDL syntax checker was expecting to find a ';' and we can see it is missing from the end of line 29. Adding the semicolon to the end of line 29 and save the file. Now when we double-click the Check Syntax process, it runs and then displays a ✔ to indicate there are no more errors.



Synthesizing the Logic circuitry for Your Design

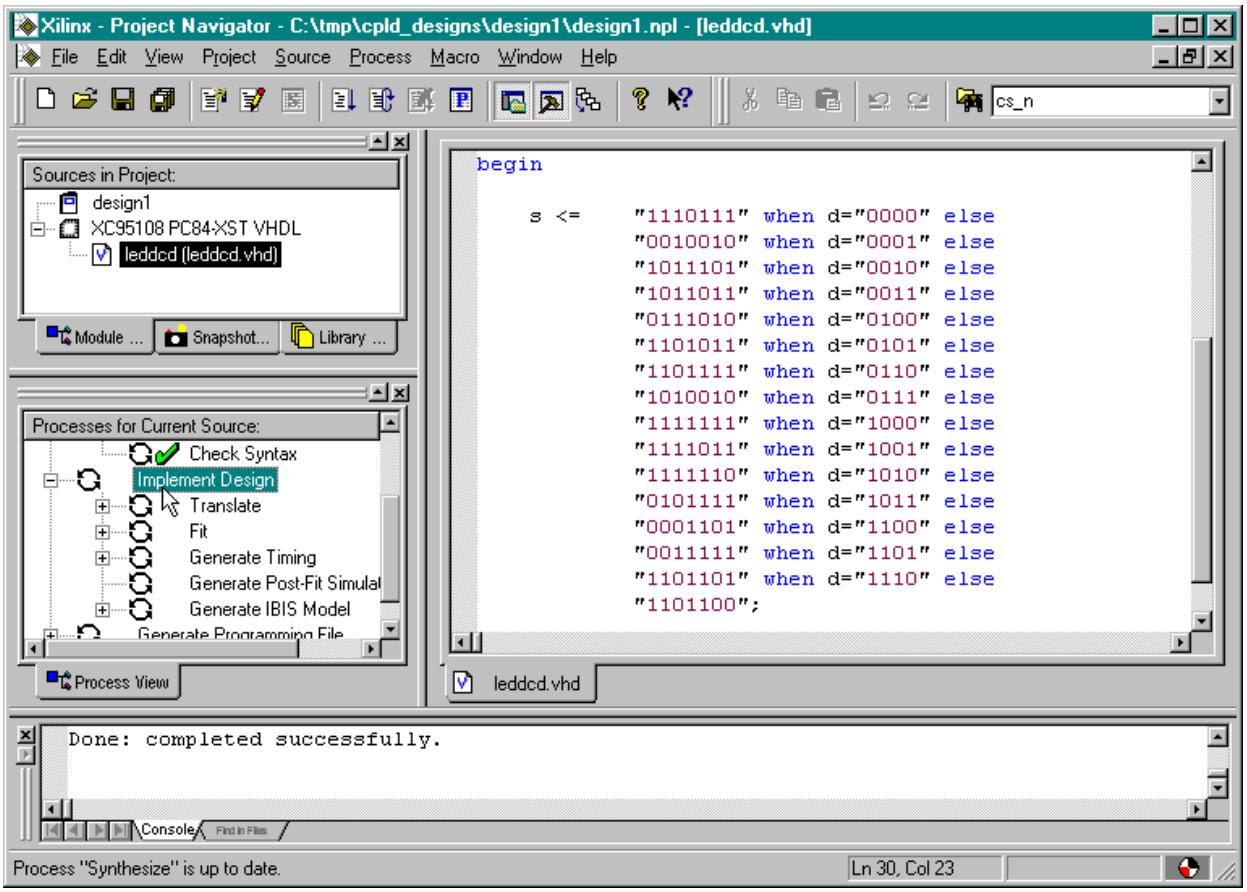
Now that we have valid VHDL for our design, we need to convert it into a logic circuit. This is done by highlighting the leddcd object in the Sources pane and then double-clicking on the Synthesize process as shown below.



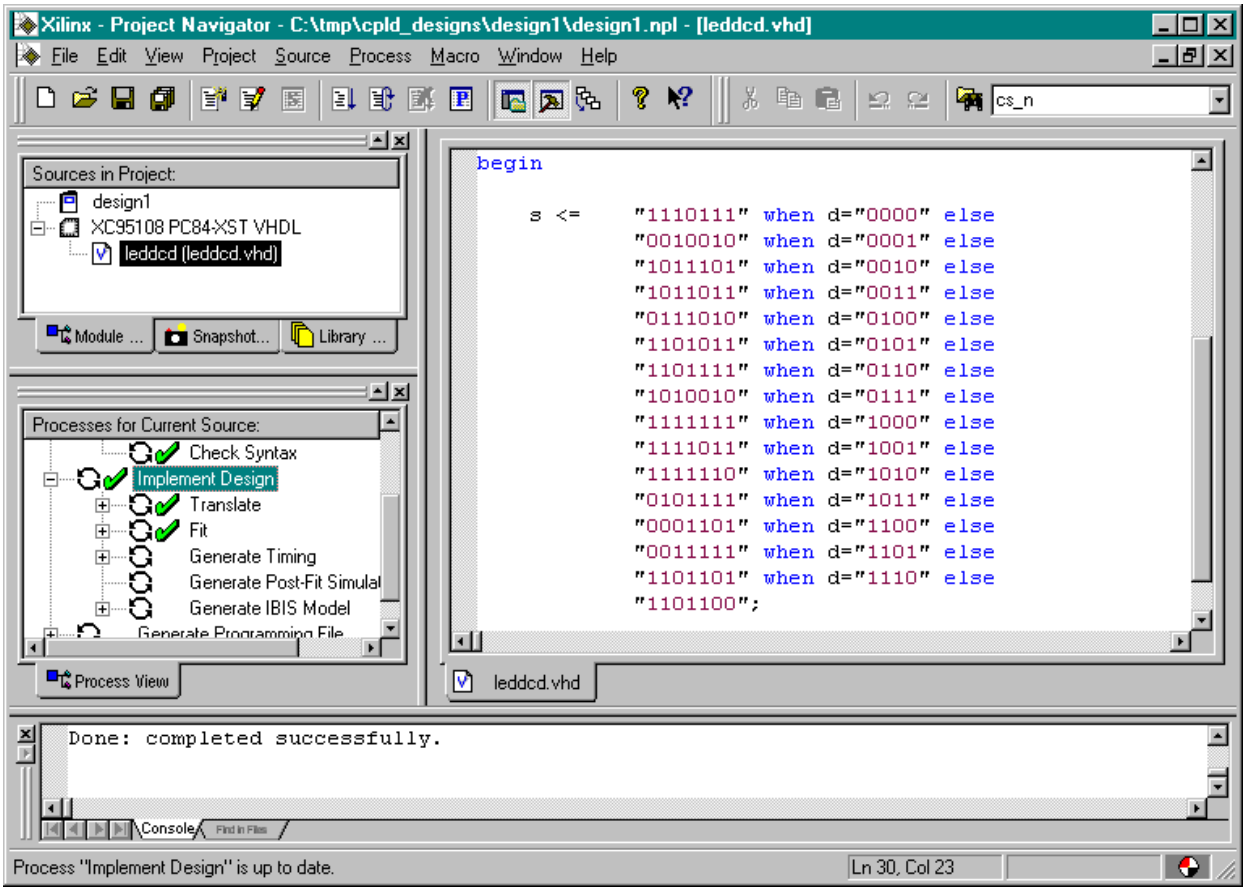
The synthesizer will read the VHDL code and transform it into a netlist of gates. This will take only a minute. If no problems are detected, a ✔ will appear next to the Synthesize process. You can double-click on the View Synthesis Report to see the various synthesizer options that were enabled and some simple statistics for the synthesized design.

Fitting the Logic Circuitry Into the CPLD

We now have a synthesized logic circuit for the LED decoder, but we need to fit it into the logic resources of the CPLD in order to actually use it. We start this process by highlighting the XC95108 PC84 object in the Sources pane and then double-clicking on the Implement Design process.

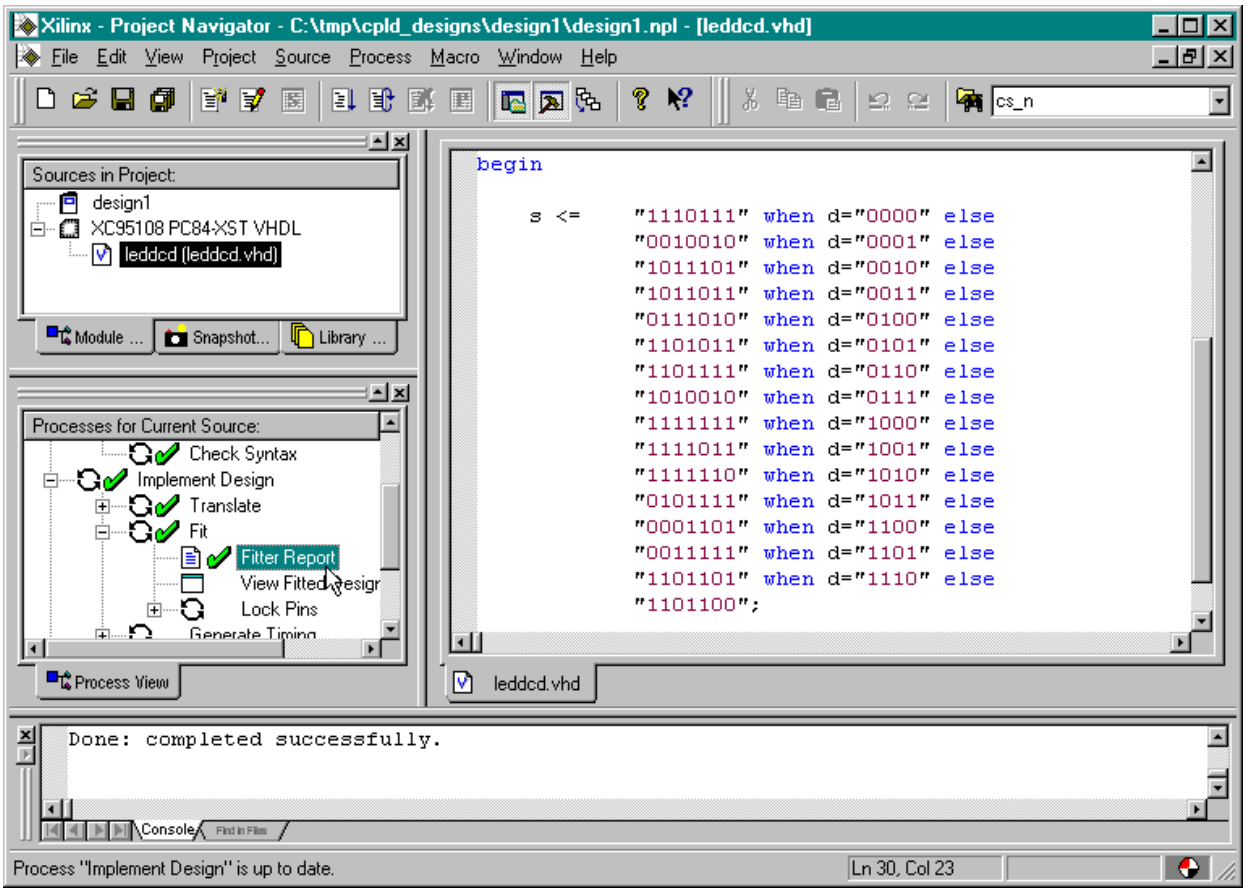


You can watch the progress of the implementation process in the status bar at the bottom of the **Project Navigator** window. For a simple design like the LED decoder, the fitting is completed in seconds (on a 850 MHz Athlon PC with 768 MBytes). A successful implementation is indicated by the ✔ next 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 Fit process maps the netlist into the circuitry elements and routing matrices contained in the device we selected. 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 Fit

We have our design fitted into the XC95108 CPLD, 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 Fitter Report in the Process pane.



This brings up a window containing the fitting statistics for the LED decoder. The top few lines of the file show the LED decoder only uses 7 of the 108 available macrocells in the XC95108 CPLD. And it only uses 11 I/O pins (4 for input, 7 for output).

```
cpldfit:  version E.30                                Xilinx Inc.
                                  Fitter Report
Design Name: leddcd                                Date: 10-20-2001,  2:03PM
Device Used: XC95108-7-PC84
Fitting Status: Successful

****************************  Resource Summary  ****************************

Macrocells      Product Terms    Registers       Pins            Function Block
Used            Used             Used            Used            Inputs Used
7  /108 (  6%) 24  /540  (  4%) 0  /108 (  0%) 11 /69  ( 15%) 24 /216 ( 11%)
```

Further down in the fitting report we can see what pins the inputs and outputs use. The **d** inputs have been assigned to pins 72, 17, 83, and 44. The **s** outputs which drive the LED segments have been routed through pins 32, 45, 1, 6, 71, 14, and 57.

```
***************Resources Used by Successfully Mapped Logic******************

** LOGIC **
Signal              Total   Signals Loc     Pwr  Slew Pin  Pin       Pin
Name                Pt      Used            Mode Rate #    Type      Use
s<0>                4       4       FB5_2   STD  FAST 32   I/O       O
s<1>                3       4       FB6_2   STD  FAST 45   I/O       O
s<2>                3       4       FB1_2   STD  FAST 1    I/O       O
s<3>                2       4       FB1_9   STD  FAST 6    I/O       O
s<4>                4       4       FB2_2   STD  FAST 71   I/O       O
s<5>                4       4       FB3_2   STD  FAST 14   I/O       O
s<6>                4       4       FB4_2   STD  FAST 57   I/O       O

** INPUTS **
Signal                              Loc               Pin  Pin       Pin
Name                                                  #    Type      Use
d<0>                                FB2_3             72   I/O       I
d<1>                                FB3_5             17   I/O       I
d<2>                                FB2_16            83   I/O       I
d<3>                                FB5_17            44   I/O       I

End of Resources Used by Successfully Mapped Logic
```

The fitting report even lists the logic equations for each output:

```
; Implemented Equations.

/"s<1>"  =  /"d<0>" * "d<2>" * "d<3>"
       + "d<1>" * "d<2>" * "d<3>"
       + /"d<0>" * "d<1>" * /"d<2>" * /"d<3>"

/"s<2>"  =  "d<0>" * /"d<3>"
       + "d<0>" * /"d<1>" * /"d<2>"
       + /"d<1>" * "d<2>" * /"d<3>"

/"s<3>"  =  /"d<1>" * /"d<2>" * /"d<3>"
       + "d<0>" * "d<1>" * "d<2>" * /"d<3>"

/"s<4>"  =  "d<0>" * "d<1>" * "d<3>"
       + /"d<0>" * "d<1>" * "d<2>"
       + /"d<0>" * "d<2>" * "d<3>"
       + "d<0>" * /"d<1>" * "d<2>" * /"d<3>"

/"s<5>"  =  "d<0>" * "d<1>" * /"d<3>"
       + "d<0>" * /"d<2>" * /"d<3>"
       + "d<1>" * /"d<2>" * /"d<3>"
       + /"d<1>" * "d<2>" * "d<3>"

/"s<6>"  =  /"d<0>" * /"d<1>" * "d<2>"
       + /"d<1>" * "d<2>" * "d<3>"
       + "d<0>" * "d<1>" * /"d<2>" * "d<3>"
       + "d<0>" * /"d<1>" * /"d<2>" * /"d<3>"

/"s<0>"  =  "d<0>" * "d<1>" * "d<2>"
       + "d<0>" * /"d<1>" * /"d<2>" * /"d<3>"
       + /"d<0>" * "d<1>" * /"d<2>" * "d<3>"
       + /"d<0>" * /"d<1>" * "d<2>" * /"d<3>"
```

Constraining the Fit

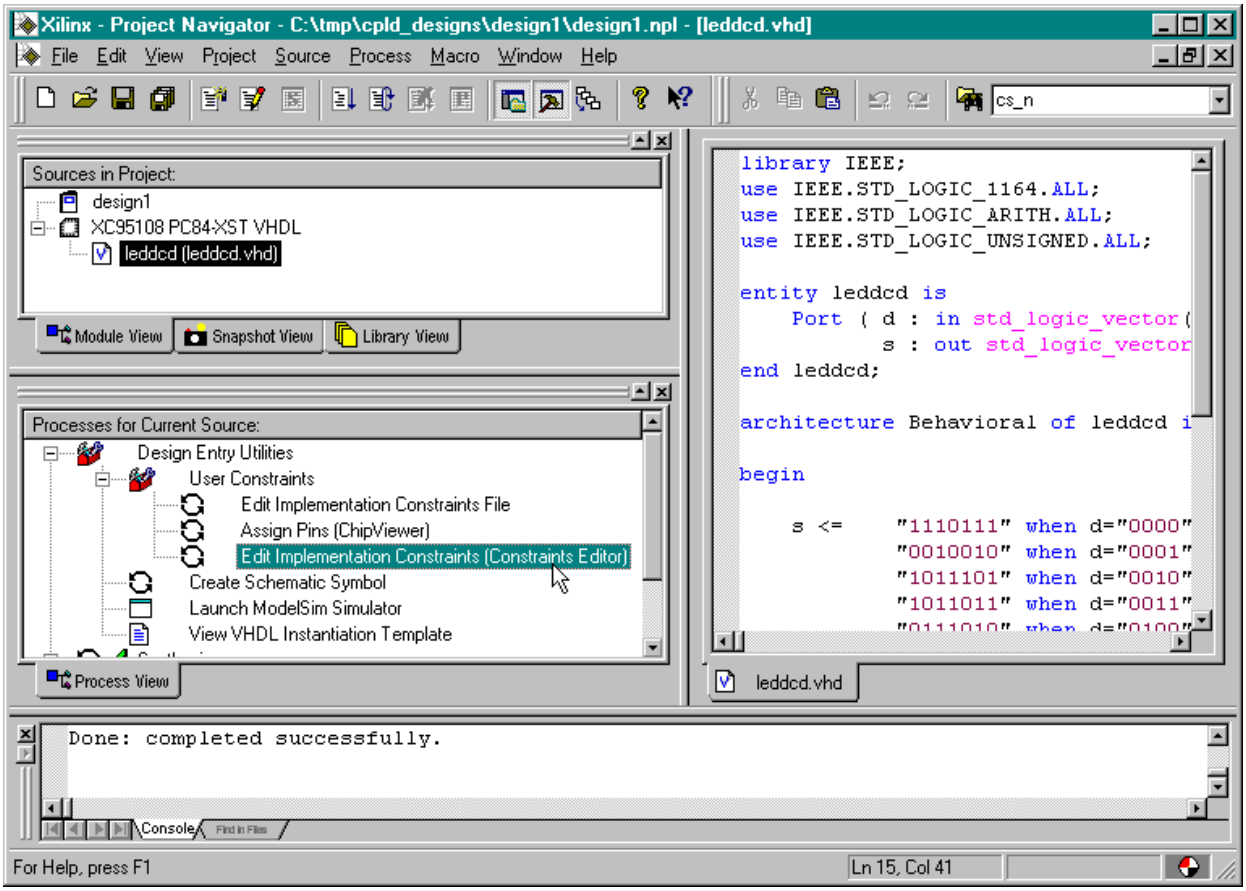
The problem we have now is that the inputs and outputs for the LED decoder were assigned to pins picked by the fitting process, but these are not the pins we actually want to use on the XS95 Board. The CPLD on the XS95 Board has eight inputs which are driven by the PC parallel port and we would like to assign the LED decoder inputs to four of these as follows:

LED Decoder Input	XS95 XC95108 CPLD Pin
d0	P46
d1	P47
d2	P48
d3	P50

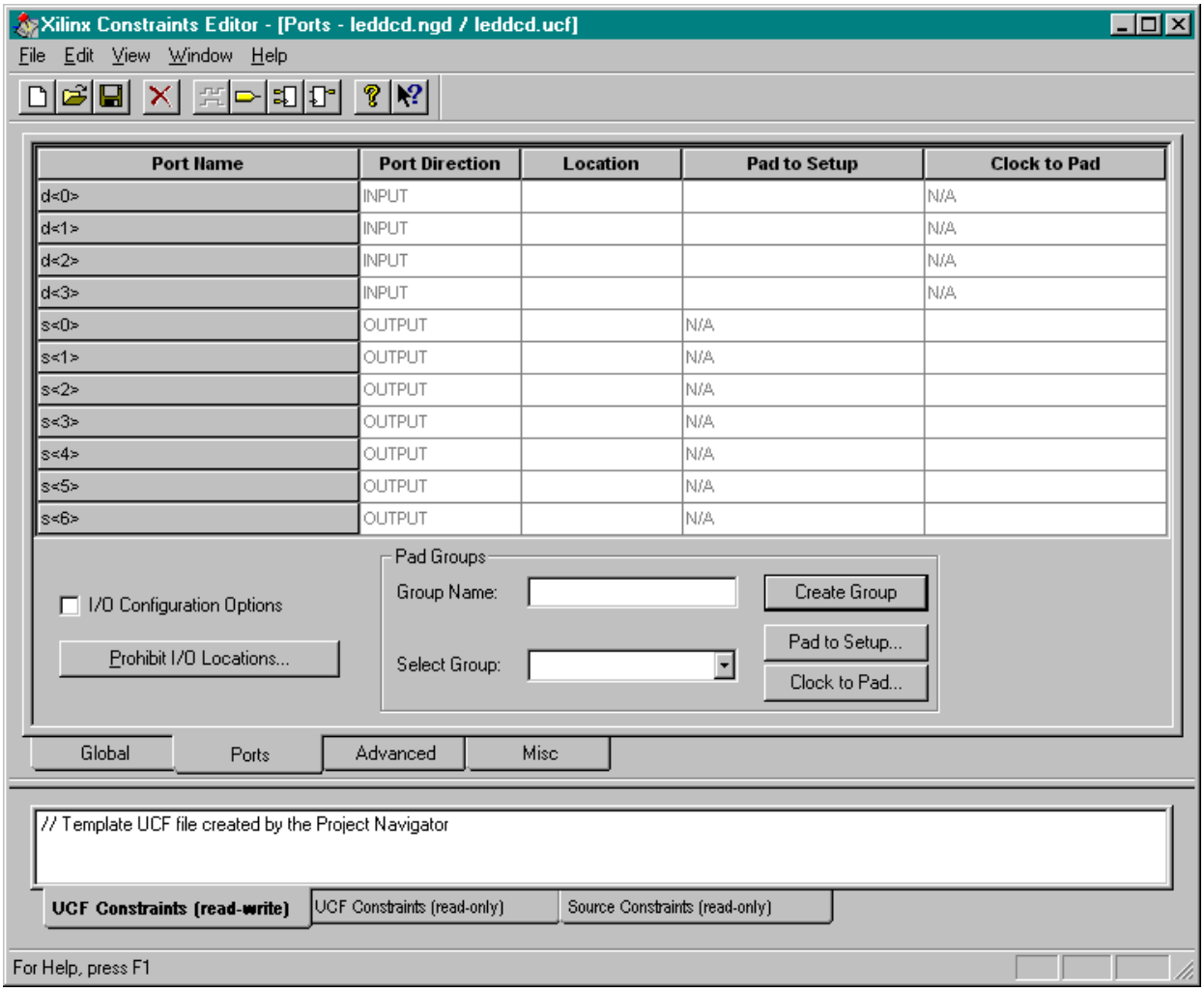
Likewise, the XS95 Board has a seven-segment LED attached to the following pins of the CPLD:

LED Decoder Output	XS95 XC95108 CPLD Pin
s0	P21
s1	P23
s2	P19
s3	P17
s4	P18
s5	P14
s6	P15

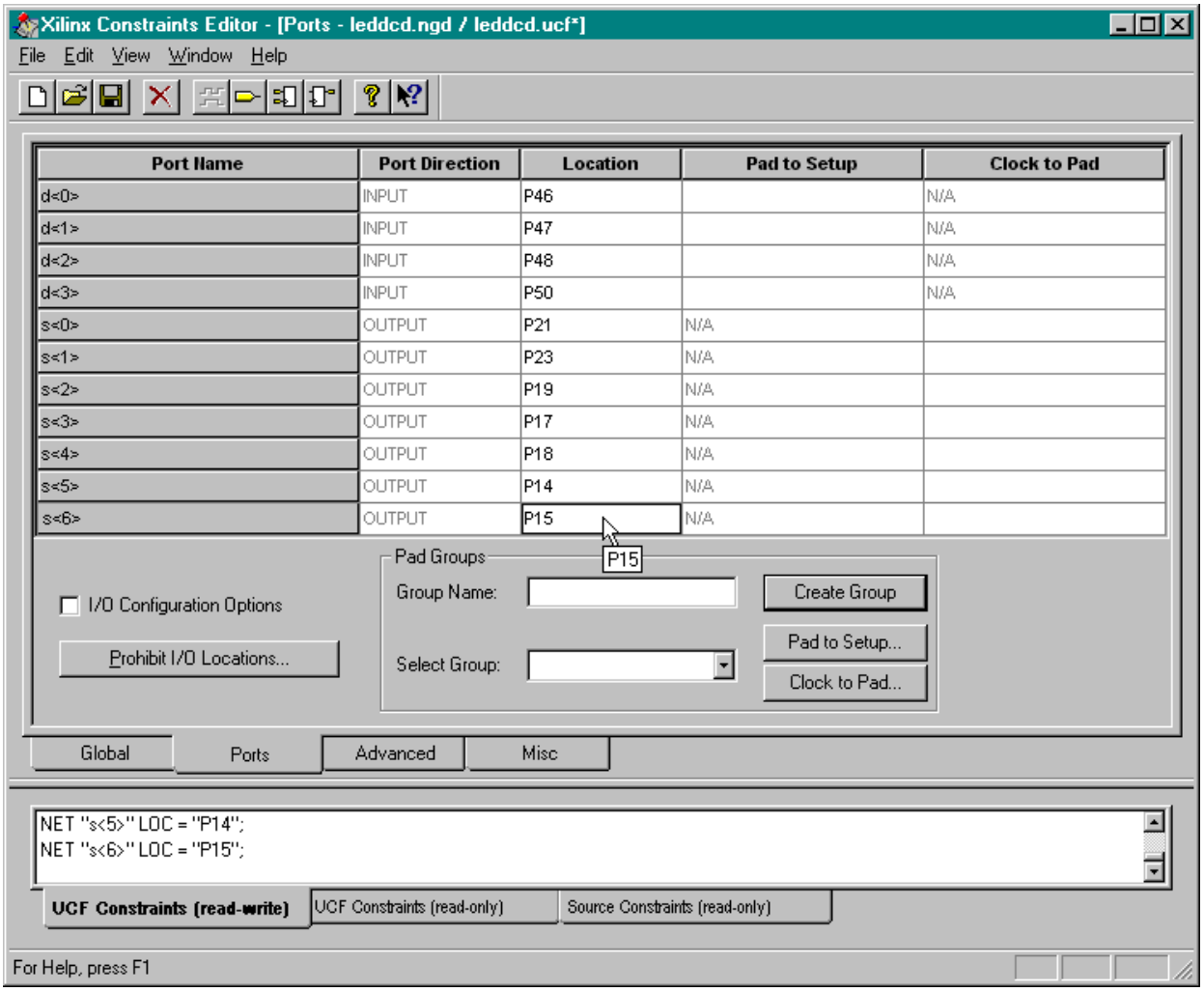
How do we constrain the fitting process so it assigns the inputs and outputs to the pins we want to use? We start by highlighting the leddcd object in the Sources pane and then double-clicking the Edit Implementation Constraints (Constraints Editor) process.



The **Constraints Editor** window appears. Click on the Ports tab in the upper pane. A list of the inputs and outputs for the LED decoder will appear. We can enter our pin assigments here.

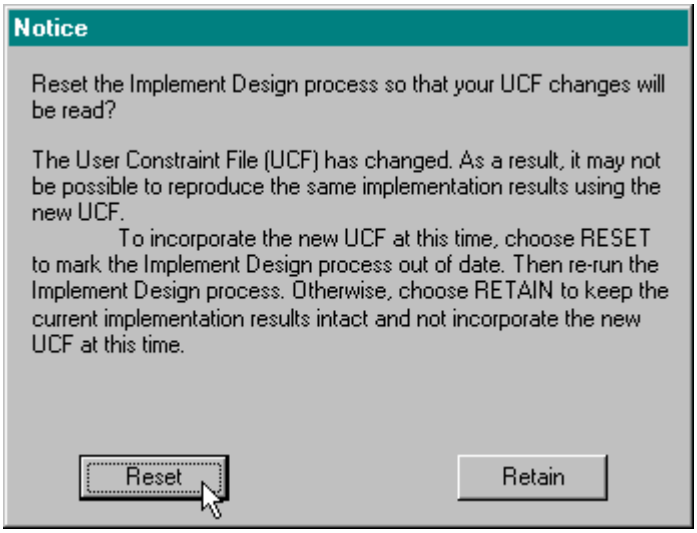


We start by clicking in the Location field for the **d<0>** input. Then just type in the pin assignment for this input: `P46`. Do this for each of the inputs and outputs using the pin assignments from the tables shown above. After doing this, the **Constraints Editor** window appears as follows.



After the pin assignments are entered, click on the button to save the pin assignment constraints. Then select File➔Exit in the **Constraints Editor** window.

Since we are changing the constraints on our design, we are asked to reset the implementation process so it will be re-run with our new constraints. This just means we have to re-run the fitting process again if we want the design to use the pin assignments we just made. Click on Reset and then double-click the Implement Design process to re-fit the design with the new pin assignments.



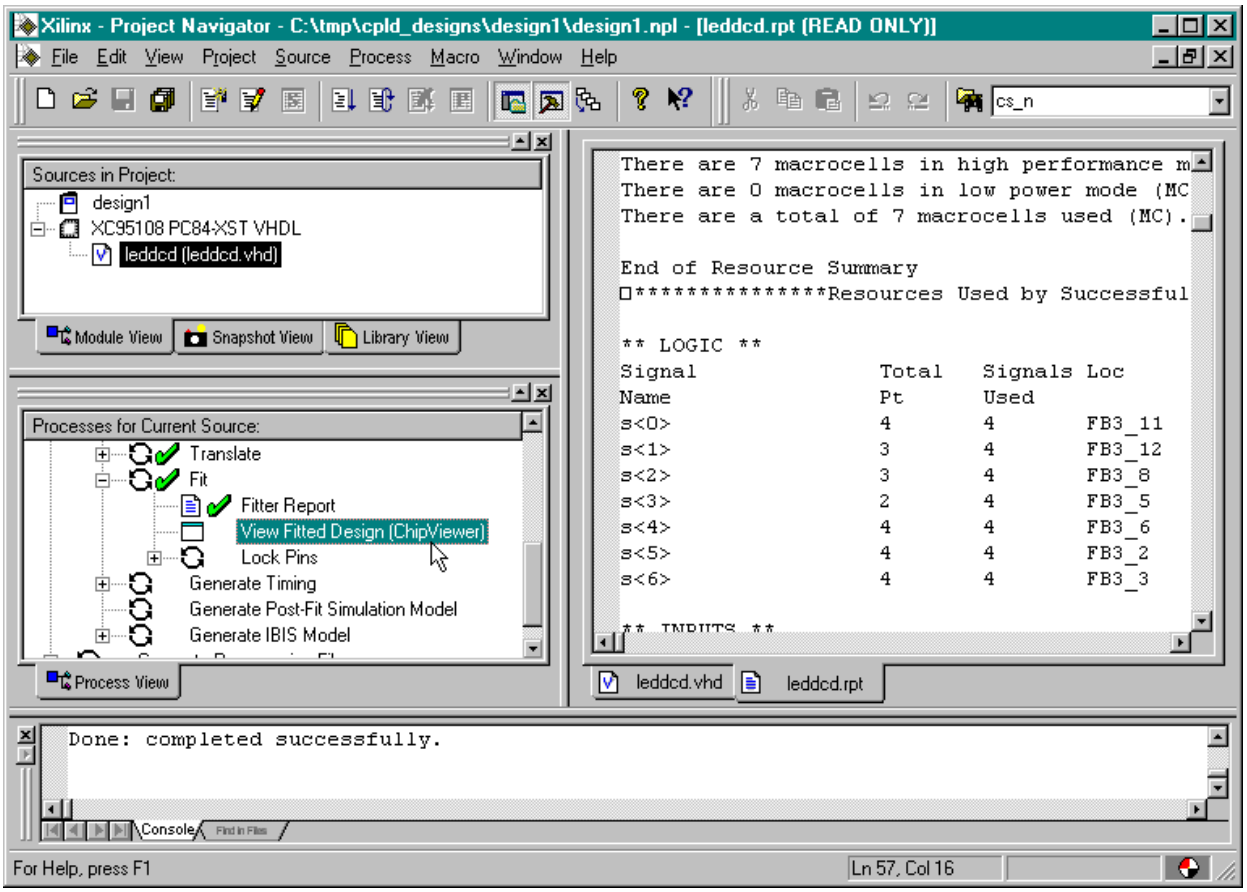
Next double-click on the Fitter Report process to view the pin assignments made by the fitter process. Looking through the fitter report, we see the following:

```
***************Resources Used by Successfully Mapped Logic******************

** LOGIC **
Signal                Total   Signals Loc     Pwr  Slew Pin  Pin       Pin
Name                  Pt      Used            Mode Rate #    Type      Use
s<0>                  4       4       FB3_11  STD  FAST 21   I/O       O
s<1>                  3       4       FB3_12  STD  FAST 23   I/O       O
s<2>                  3       4       FB3_8   STD  FAST 19   I/O       O
s<3>                  2       4       FB3_5   STD  FAST 17   I/O       O
s<4>                  4       4       FB3_6   STD  FAST 18   I/O       O
s<5>                  4       4       FB3_2   STD  FAST 14   I/O       O
s<6>                  4       4       FB3_3   STD  FAST 15   I/O       O

** INPUTS **
Signal                                Loc               Pin  Pin       Pin
Name                                                    #    Type      Use
d<0>                                  FB6_3             46   I/O       I
d<1>                                  FB6_5             47   I/O       I
d<2>                                  FB6_6             48   I/O       I
d<3>                                  FB6_8             50   I/O       I

End of Resources Used by Successfully Mapped Logic
```

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

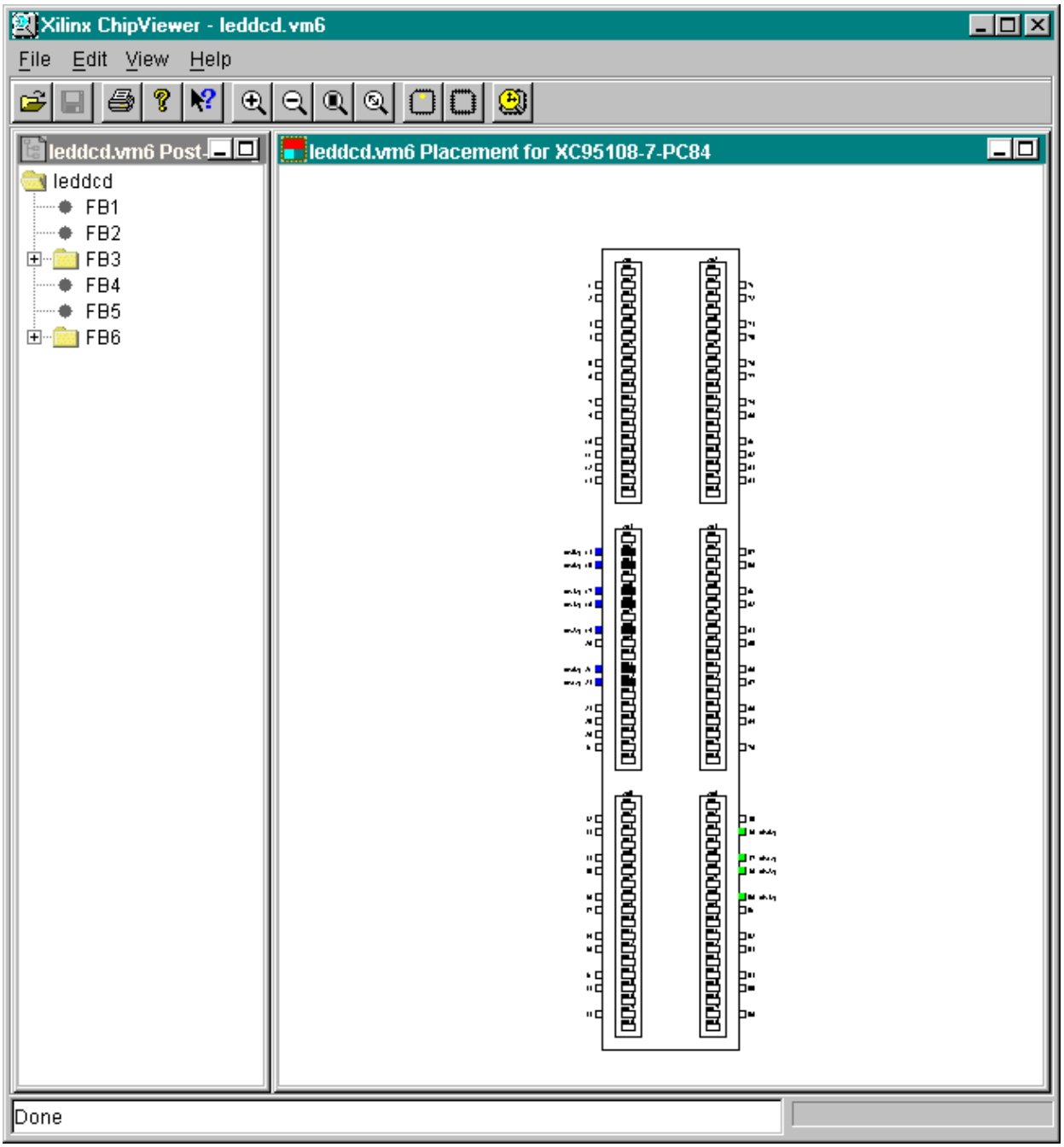
After the implementation process completes, you can get a graphical depiction of how the logic circuitry and I/O are assigned to the CPLD macrocells and pins. Just highlight the leddcd object in the Sources pane and then double-click the View Fitted Design (ChipViewer) process.



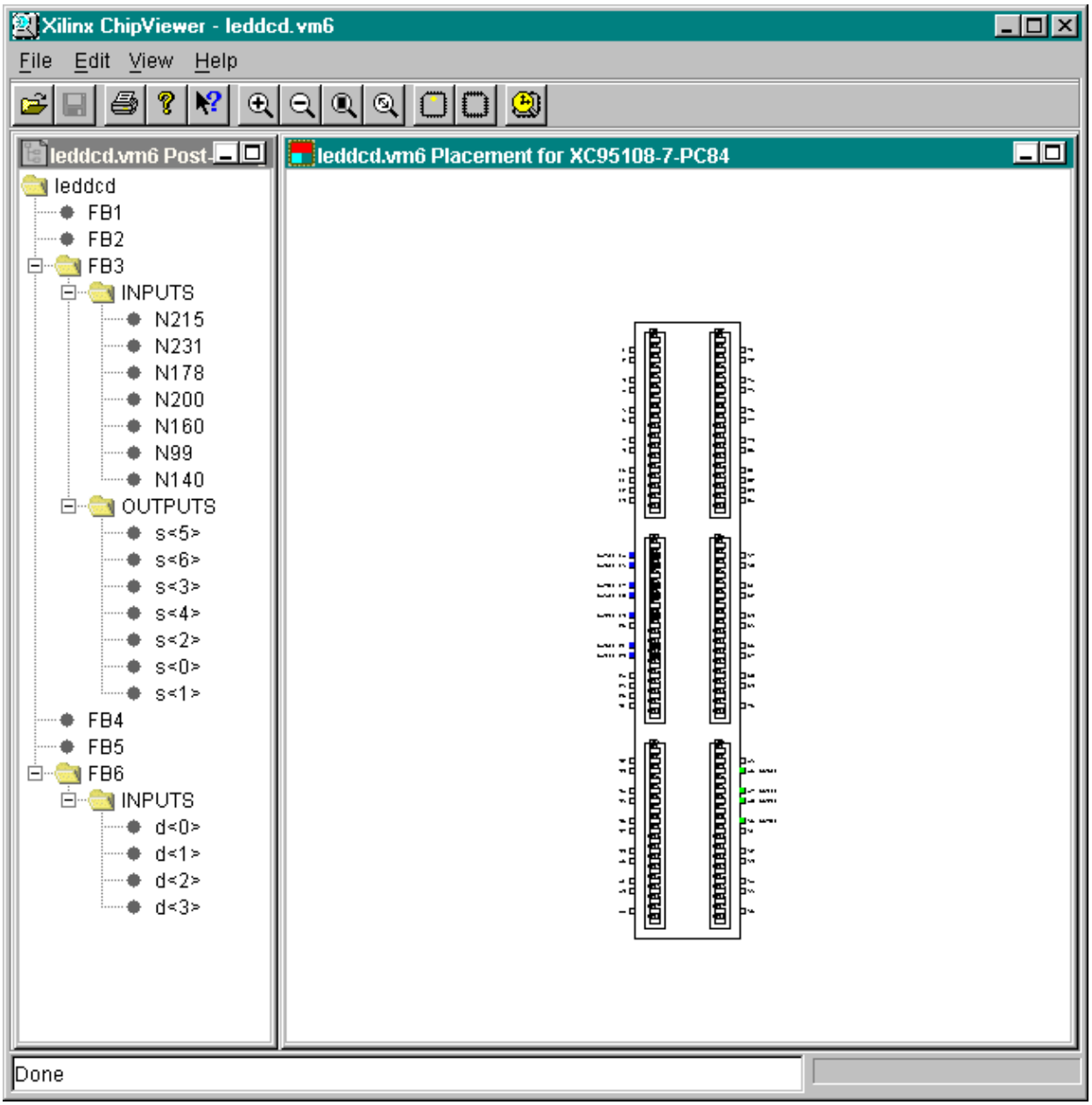
The **ChipViewer** window will appear containing two panes:

1. The left-hand pane lists the LED decoder inputs and outputs assigned to the various function blocks in the XC95108 PC84 CPLD.

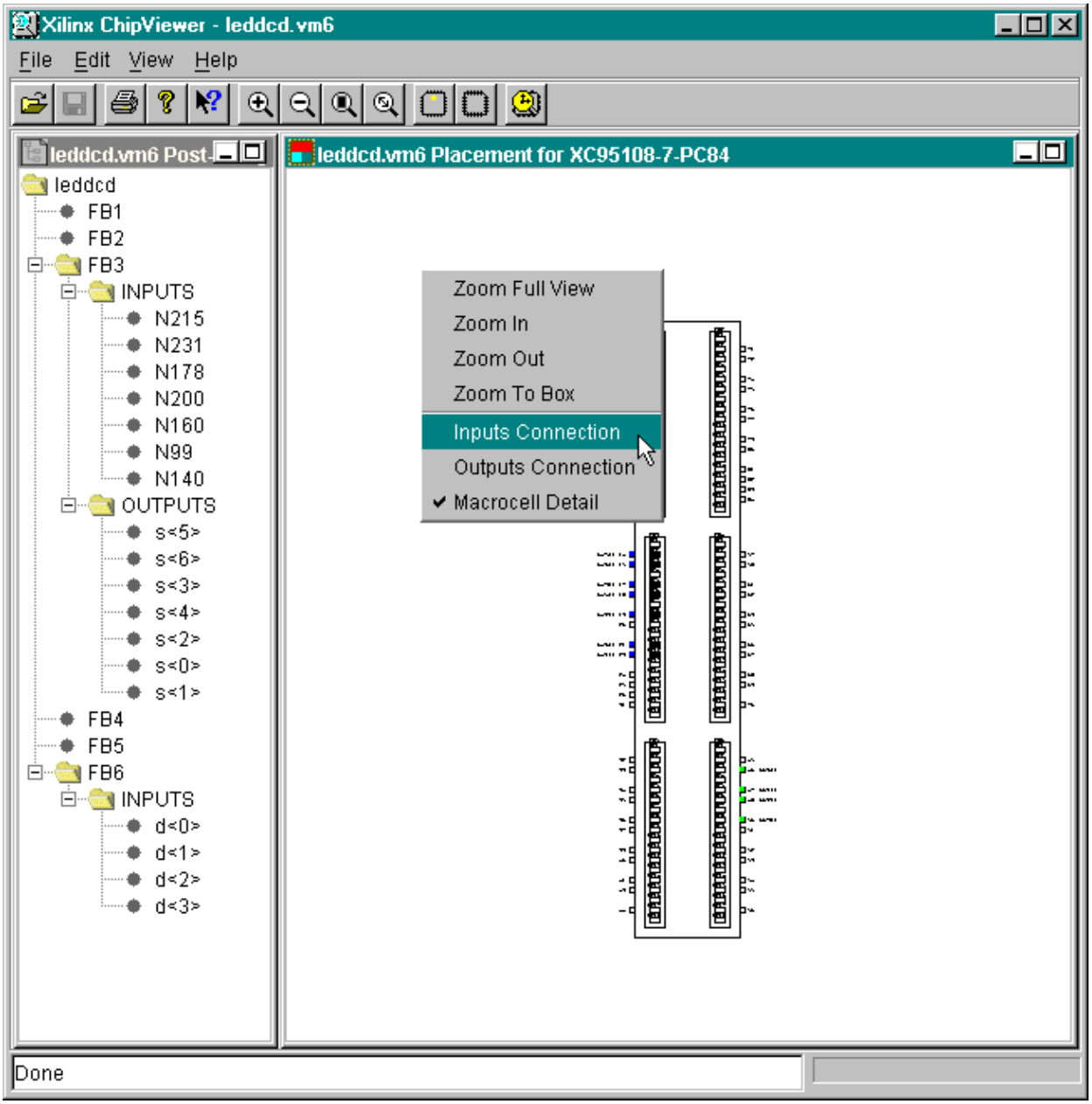
2. The right-hand pane shows the 108 macrocells of the CPLD arranged into six groups of 18 cells each. The 69 I/O pins are also shown. (The 15 pins used for Vcc, GND, and programming are not shown.)



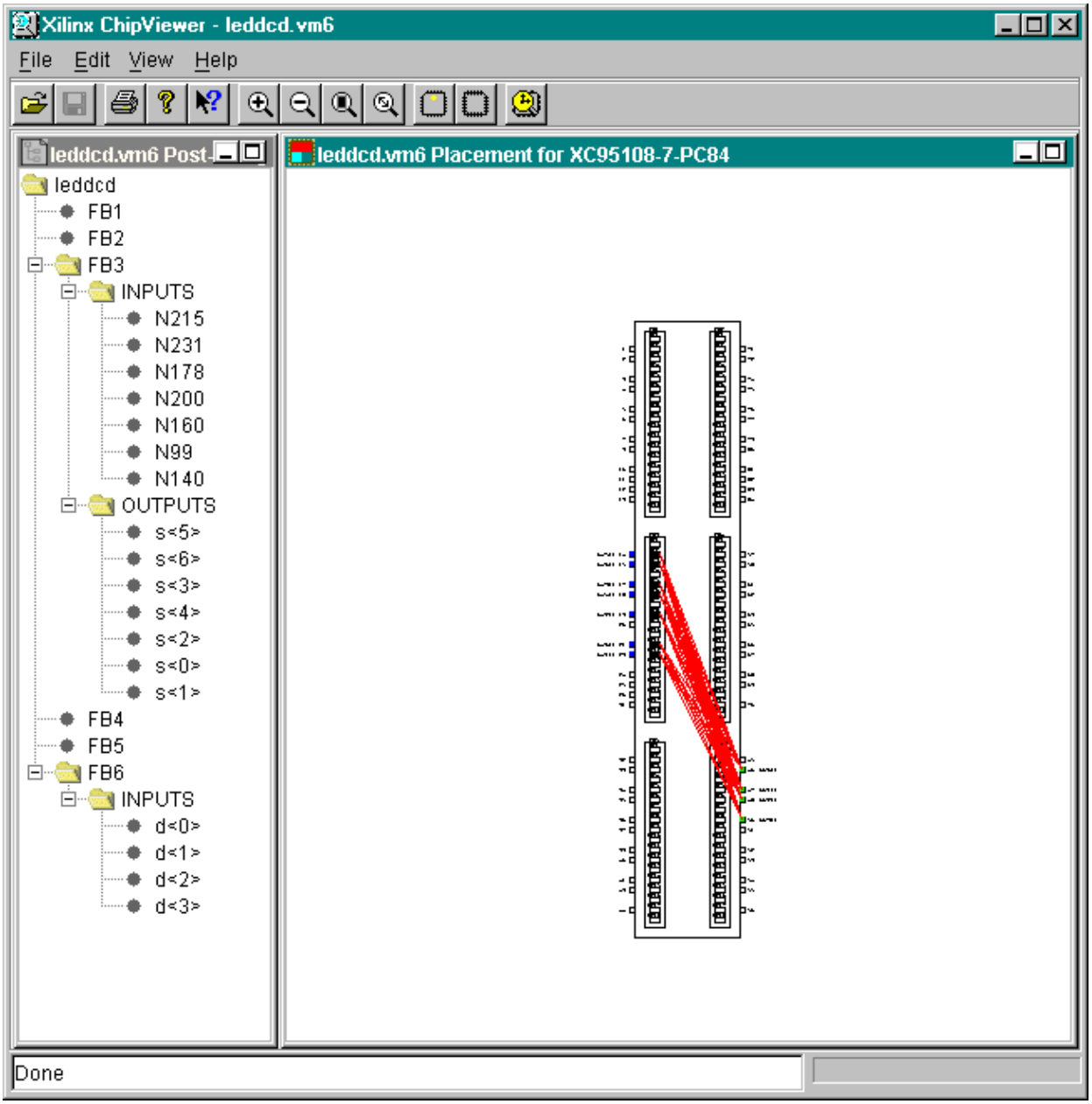
The function blocks with ⊞ icons next to them contain macrocells that are used in our design. (The remaining function blocks are not used by our design.) Clicking on the ⊞ next to FB6, we can see that the inputs enter through pins attached to function block 6. Meanwhile, all the outputs are generated by macrocells in function block 3.



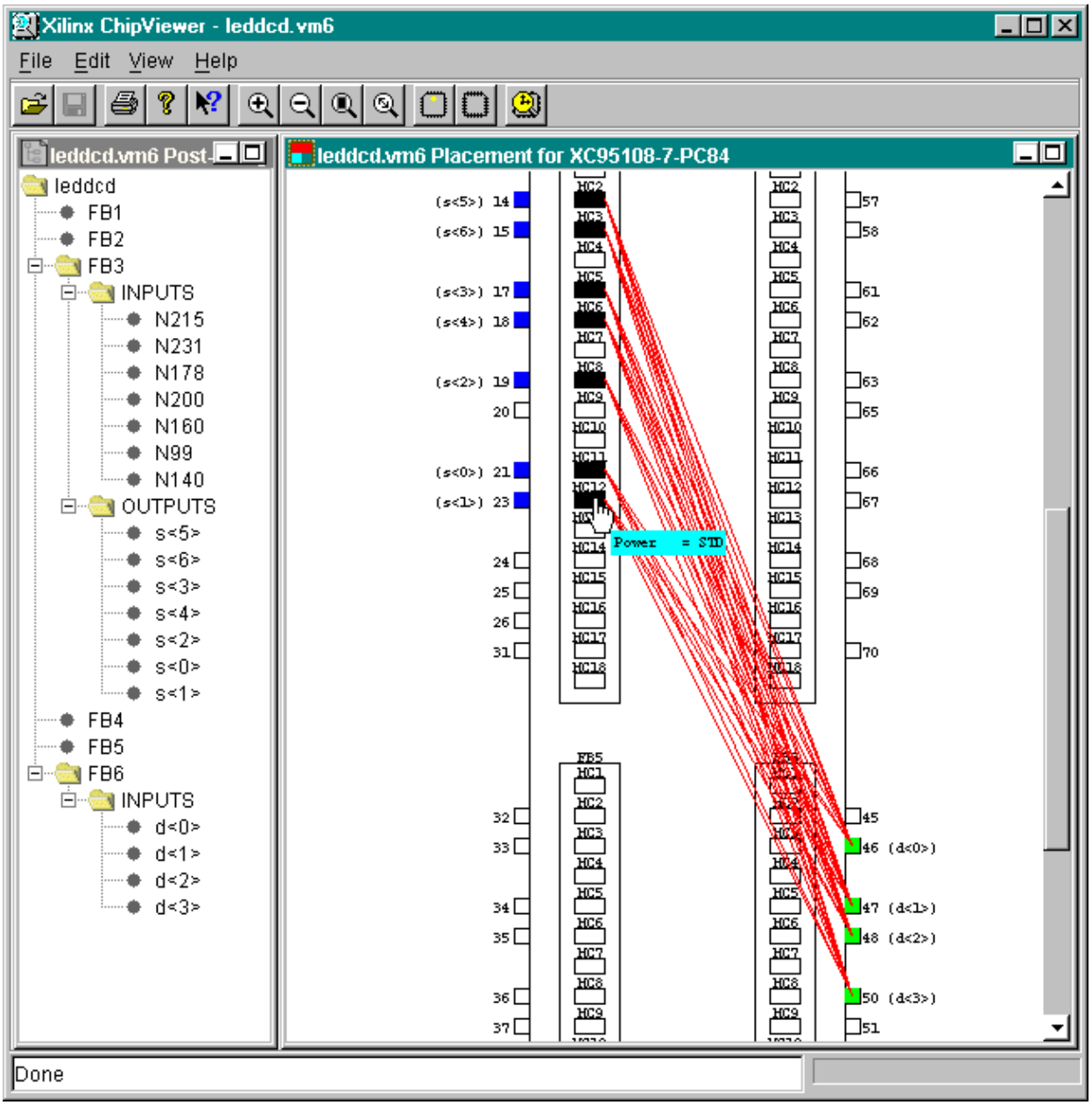
We can see which inputs affect each output by right-clicking in the right-hand pane and selecting the Inputs Connection item from the pop-up menu.



This causes red connecting lines to be drawn from each green-colored input to the blue-colored outputs it affects as shown below. For the LED decoder, every input affects every output so there are seven lines connecting each input to every output.

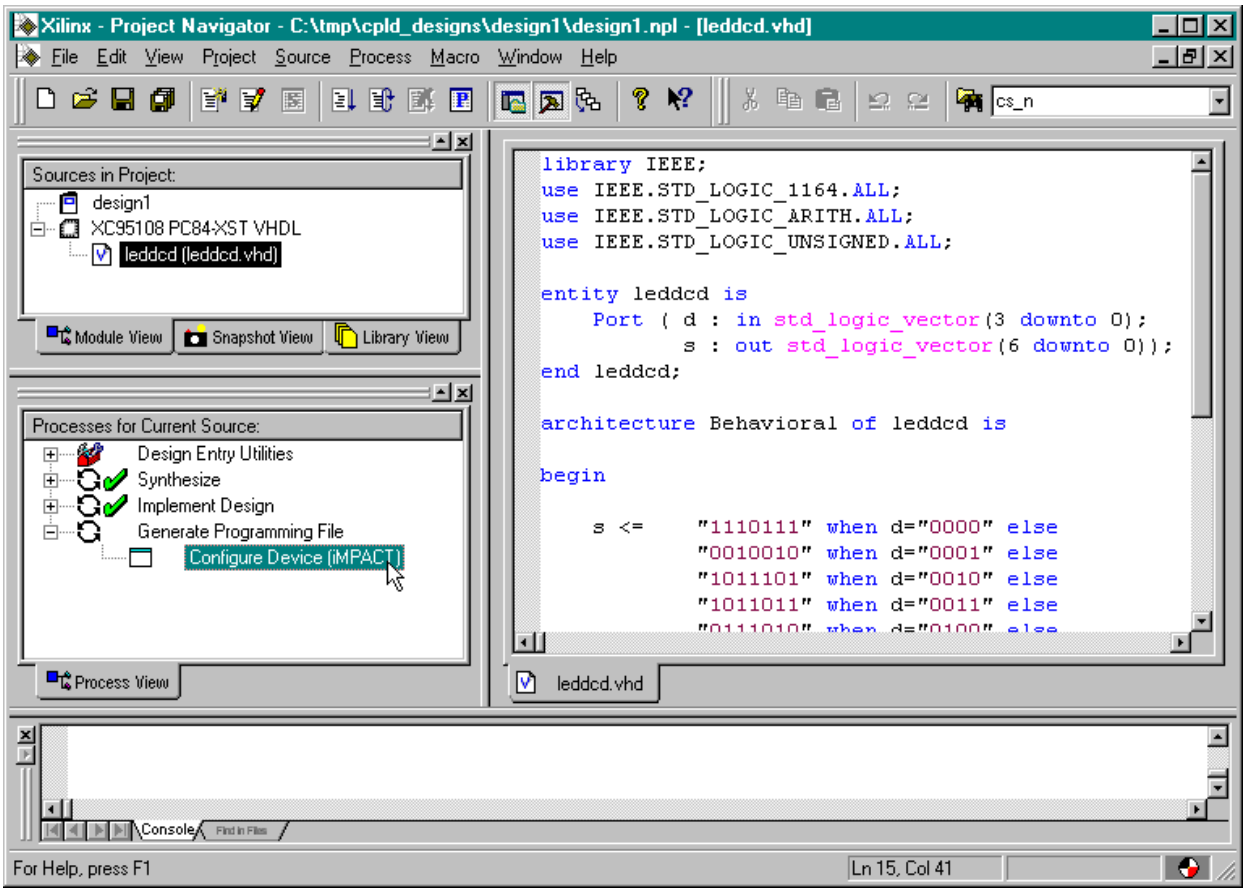


We can see more detail by clicking on the button several times to expand the right-hand pane. We can see the name of each output and the pin it is assigned to. By placing the mouse pointer over a particular pin, we get some information on the settings for the configuration options for the pin and the attached macrocell. For example, macrocell 12 of function block 3 is configured in the standard power consumption mode, and pin 23 is set for the maximum slew rate. We can also double-click on a macrocell to expand it into a separate window where we can see a list of its inputs, the Boolean equation for its output, and a rather uninteresting block-level schematic.

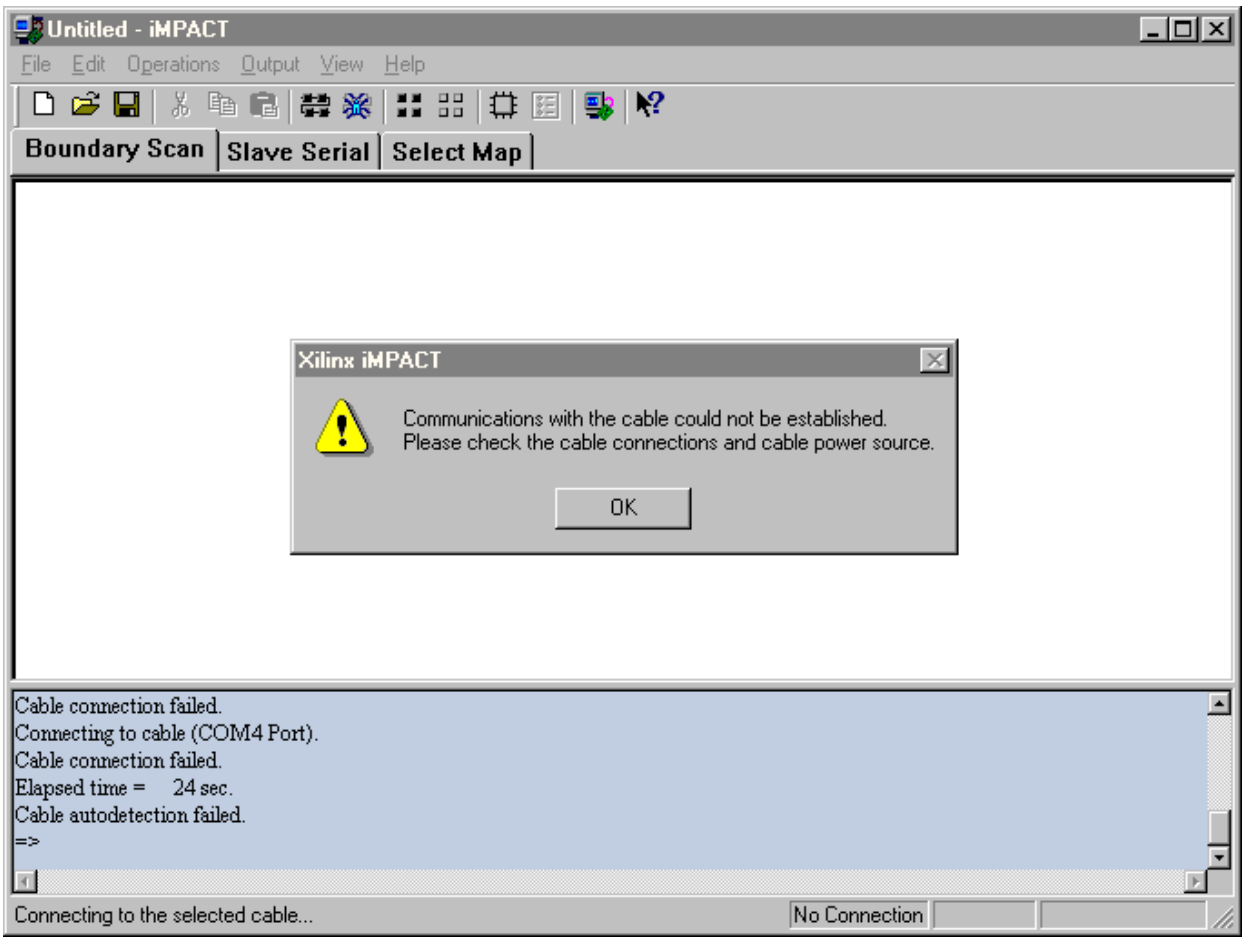


Generating the Bitstream

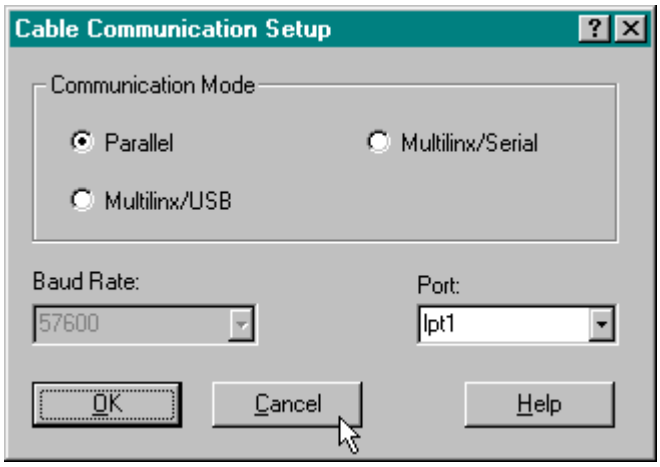
Now that we have synthesized our design and fitted it to the CPLD with the correct pin assignments, we are ready to generate the bitstream that is used to program the actual chip. To initiate the programmer, we highlight the leddcd object in the Sources pane and double-click on the Configure Device (iMPACT) process.



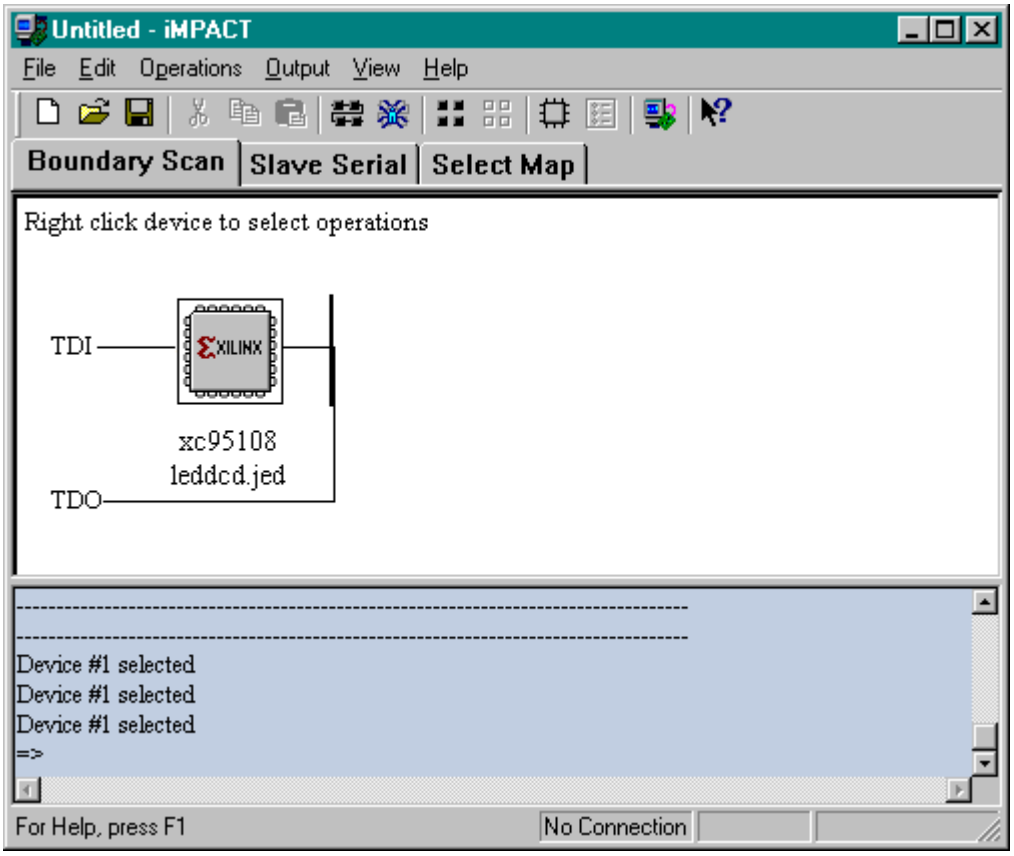
The **iMPACT** window will appear and it will try and fail to establish a connection with the CPLD through the various ports of the PC. This is normal since the XS95 Board is not made to interface directly with the iMPACT software. Just click on the OK button and proceed.



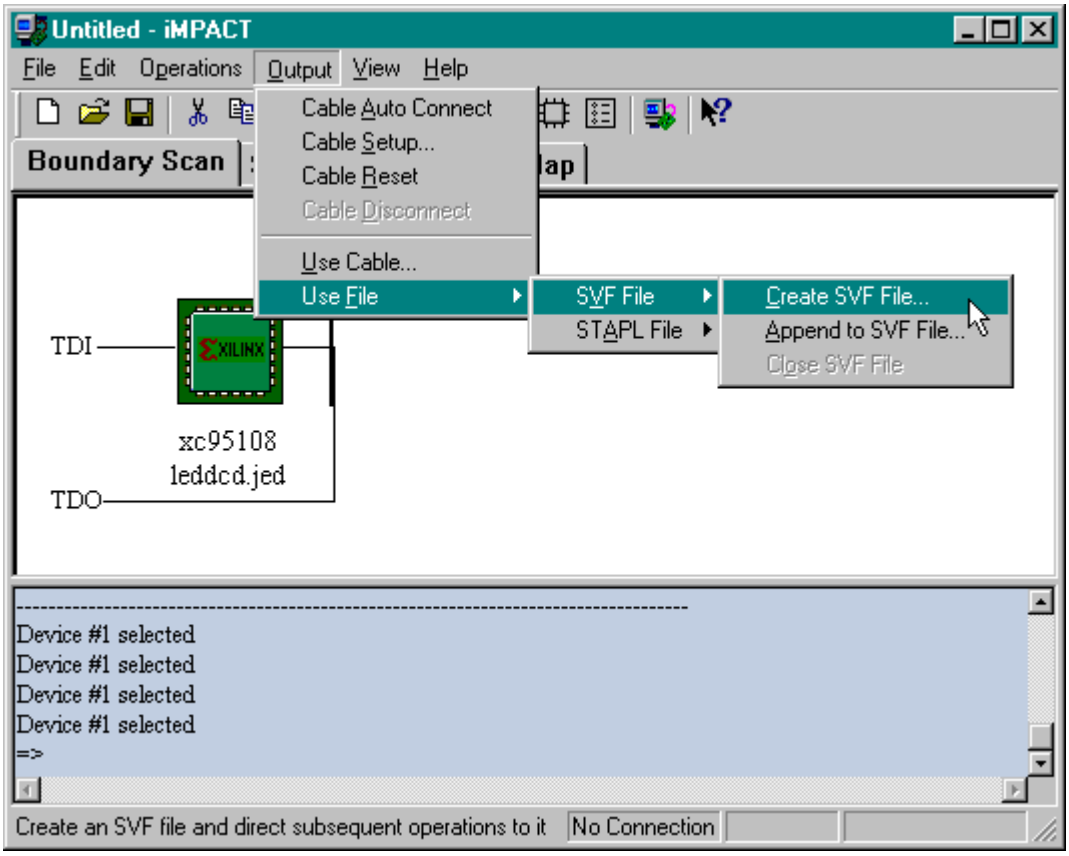
Since iMPACT cannot determine how the CPLD is connected to the PC, it will ask you what type of communication link the connection uses. Since there is no appropriate link for the XS95 Board, click the Cancel button.



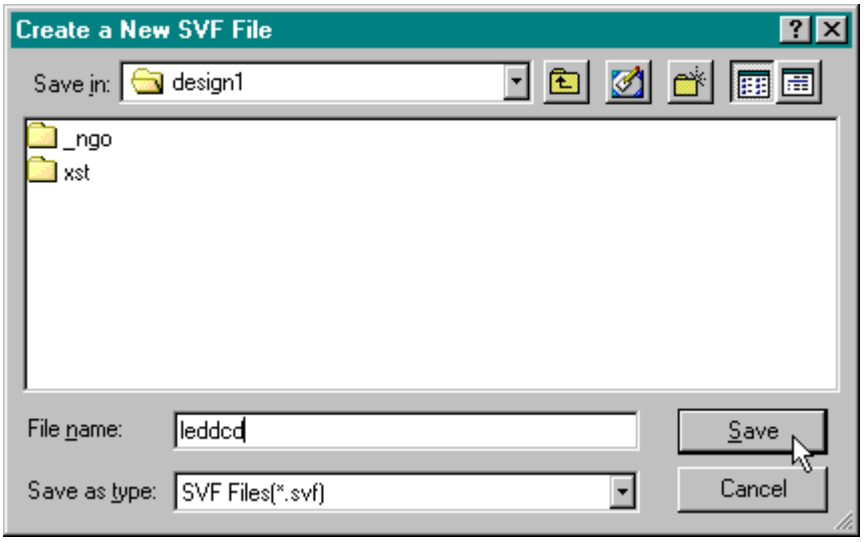
The **iMPACT** window now shows the JTAG chain of chips that are to be programmed. We only have one chip in our LED decoder design, so only one XC95108 CPLD is shown.



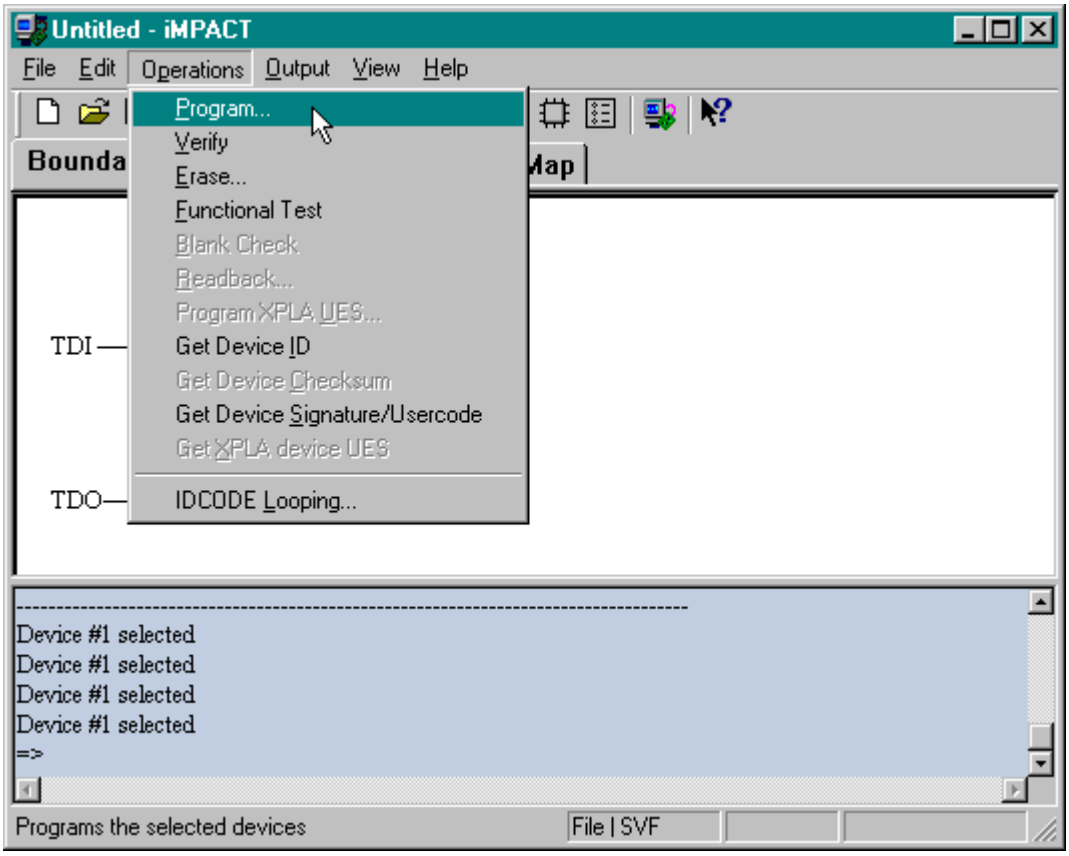
We proceed by selecting the destination for the bitstream. The XS95 Board has a separate utility called gxsload for programming the CPLD, so we need to store the bitstream into a file that gxsload can read. To do this, select Output➔Use File➔SVF File➔Create SVF File... as follows.



Now a window appears where we can enter the name for the file that will hold the bitstream. We can click on the Save button to accept the default name of leddcd.svf.

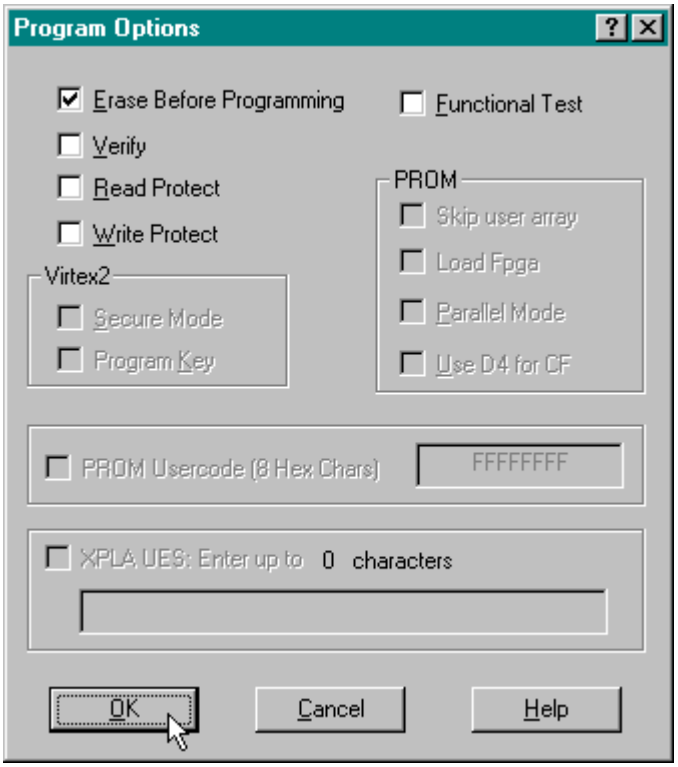


At this point we can generate the bitstream that will be stored in the leddcd.svf file. Click on the Operations➔Program... menu item to initiate the actual bitstream generation process.

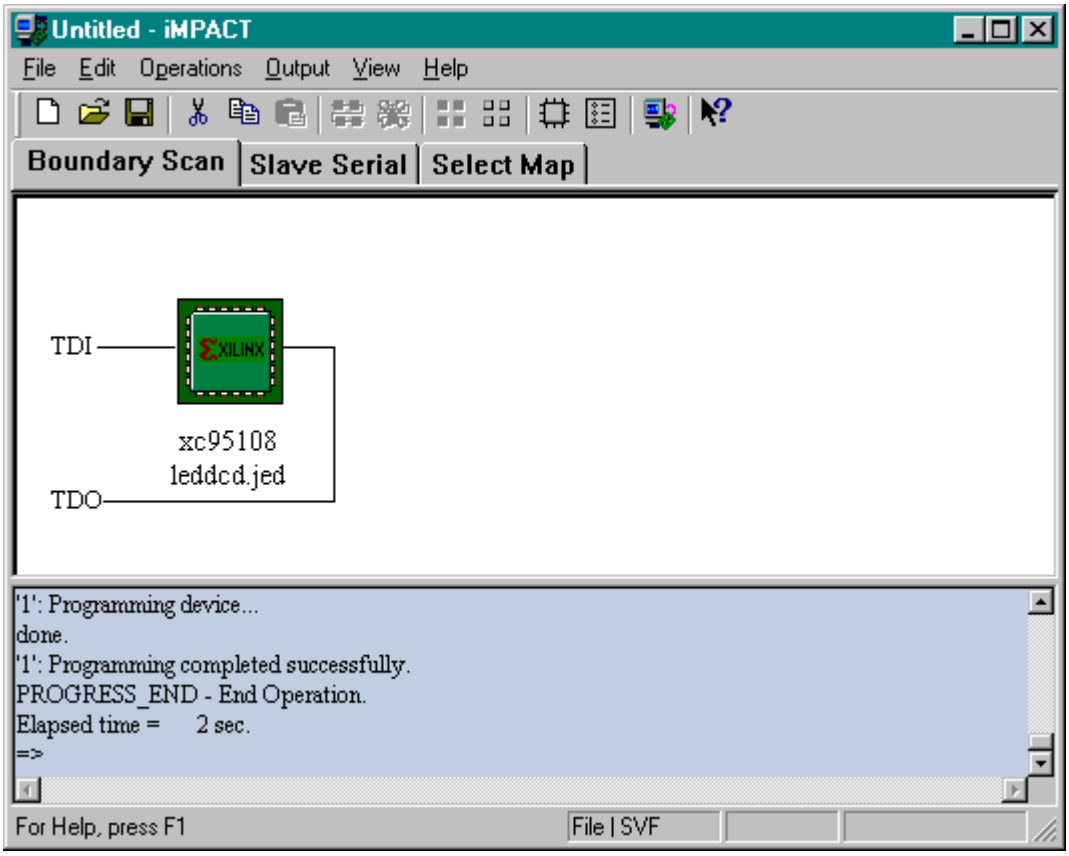


A **Program Options** window appears where we can set some options for the generated bitstream. We usually check the Erase Before Programming box so that the Flash storage in the CPLD will be erased before we start loading a new design. (The only time we can leave this box unchecked is when we are programming a CPLD which we know is already erased.)

The other two options whch are of interest are Write Protect and Read Protect. Checking the Write Protect box generates a bitstream which programs the CPLD so that it cannot be reprogrammed. (Don't worry, the device can still be erased if you want to re-use it.) The Read Protect option prevents anyone from getting the bitstream out of the CPLD so they can't steal the design. We won't enable either of these options.



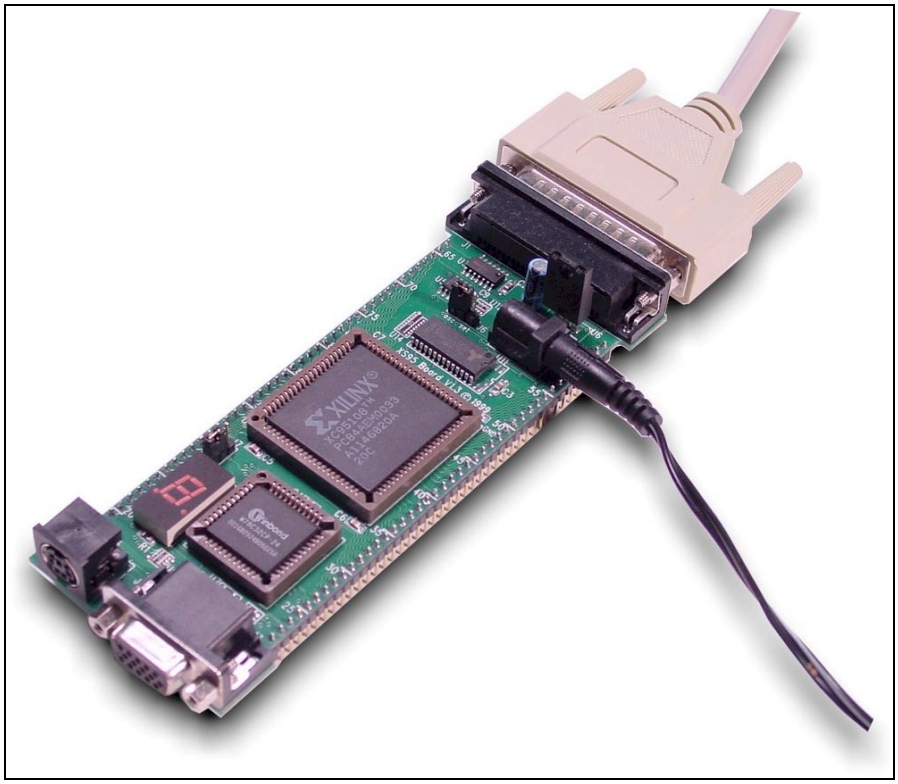
After we click OK in the **Program Options** window, the bitstream generation process begins. The progress is reported in the lower pane of the **iMPACT** window:



Once the bitstream file is generated, click on File➔OK to close the window. (You will be asked if you want to save the configuration. Don't bother.)

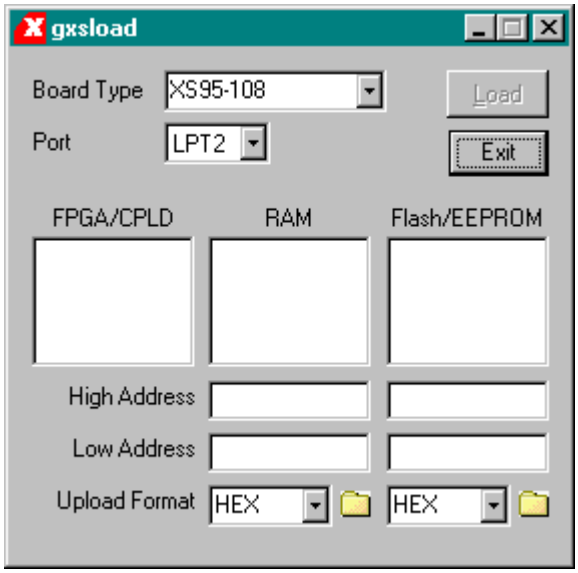
Downloading the Bitstream

Now we have to get the bitstream file programmed into the CPLD of the XS95 Board. The XS95 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.

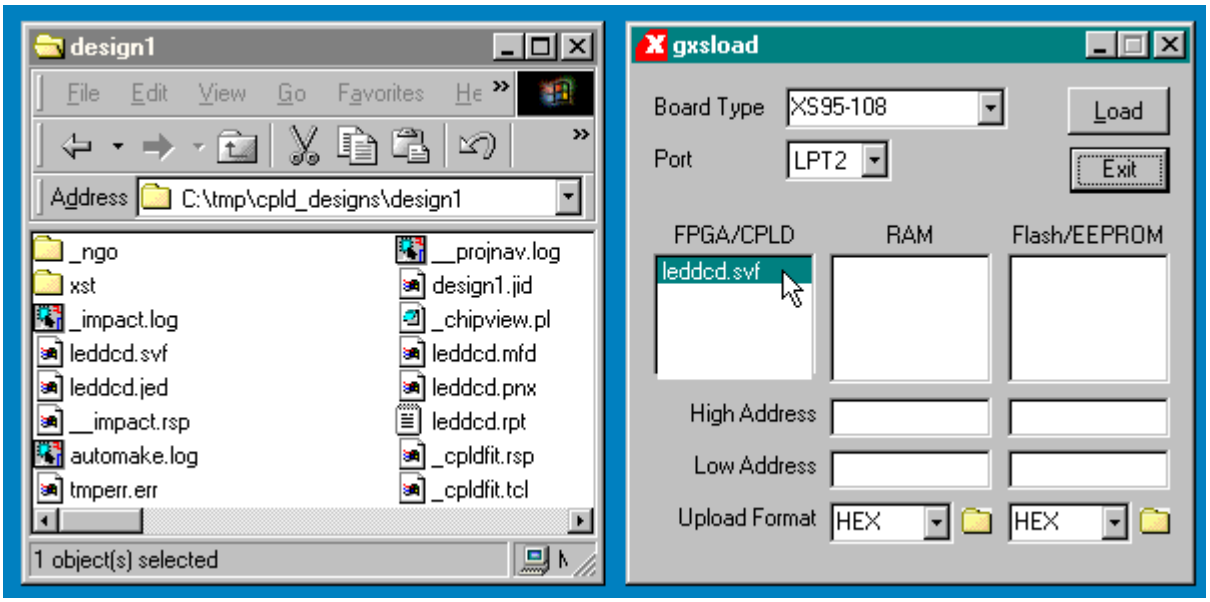


The XS95 Board is programmed using the gxsload utility. We double click the 

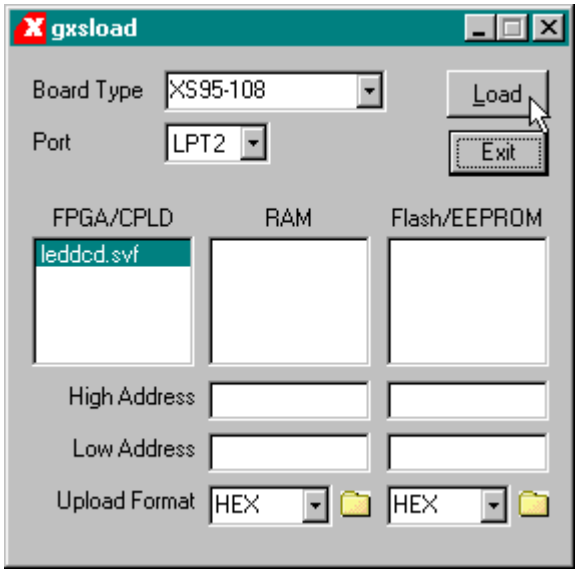
icon to bring up the following window:

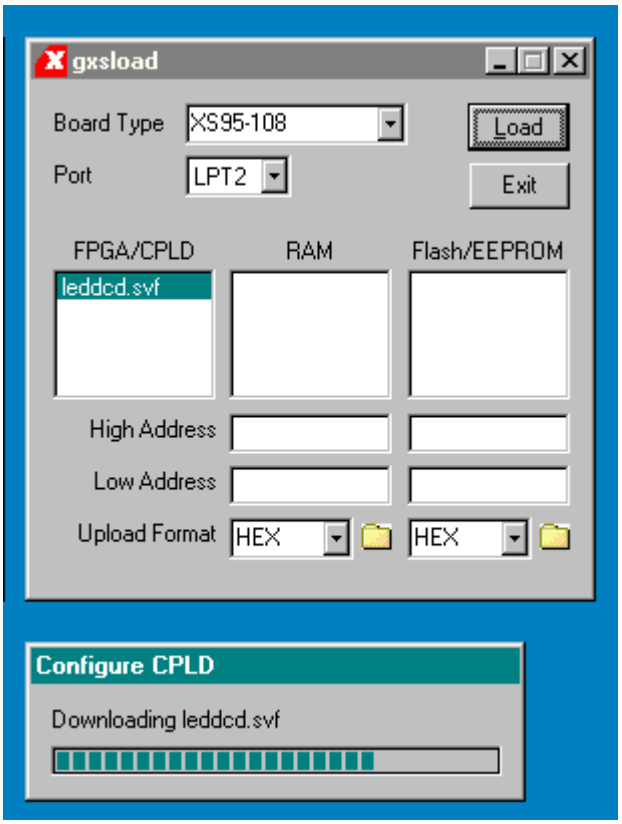


Then we open a window that shows the contents of the folder where we have stored our LED decoder design (C:\tmp\cpld_designs in this case). We just drag the leddcd.svf file from the **design1** window into the **gxsload** window.



Then we click on the Load button to initiate the programming of the XS95 Board. Downloading the leddcd.svf file will take under a minute.



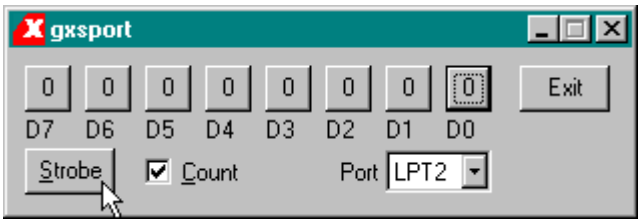


Testing the Circuit

Once the CPLD on the XS95 Board is programmed, we can begin testing the LED decoder. The eight data pins of the PC parallel port connect to the CPLD 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 XS95 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 CPLD. For example, setting (D3,D2,D1,D0) = (1,1,1,0) will cause E to appear on the seven-segment LED of the XS95 Board.



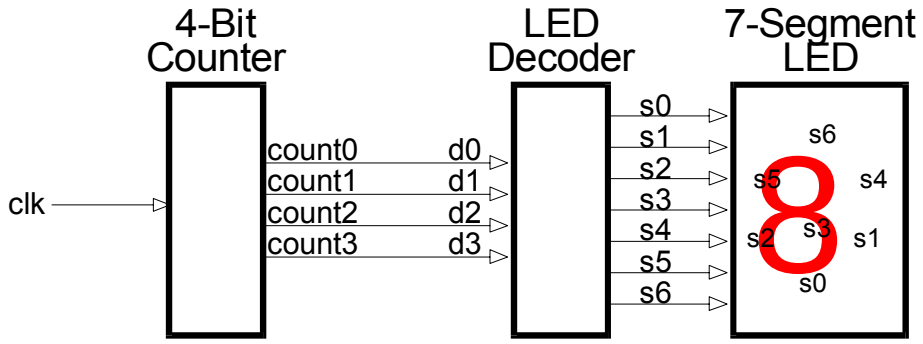
If you check the Count box in the **gxsport** window, then each click on the Strobe button increments the eight-bit value represented by D7-D0. This makes it easy to check all sixteen input combinations.

4

Hierarchical Design

A Displayable Counter

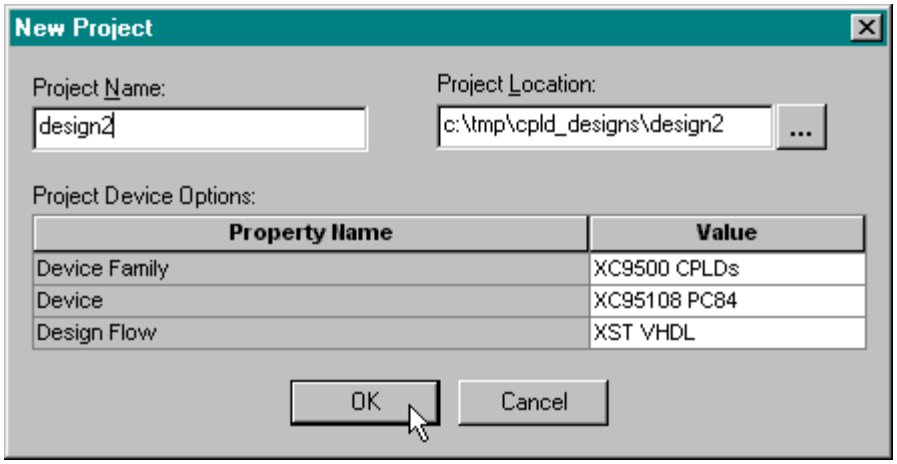
We went through a lot of work for our first CPLD design, so we will reuse it in this design: a four-bit counter whose value is displayed on a seven-segment display. The counter will increment on a rising edge of the clock. The four-bit output from the counter enters the LED decoder whereupon the counter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



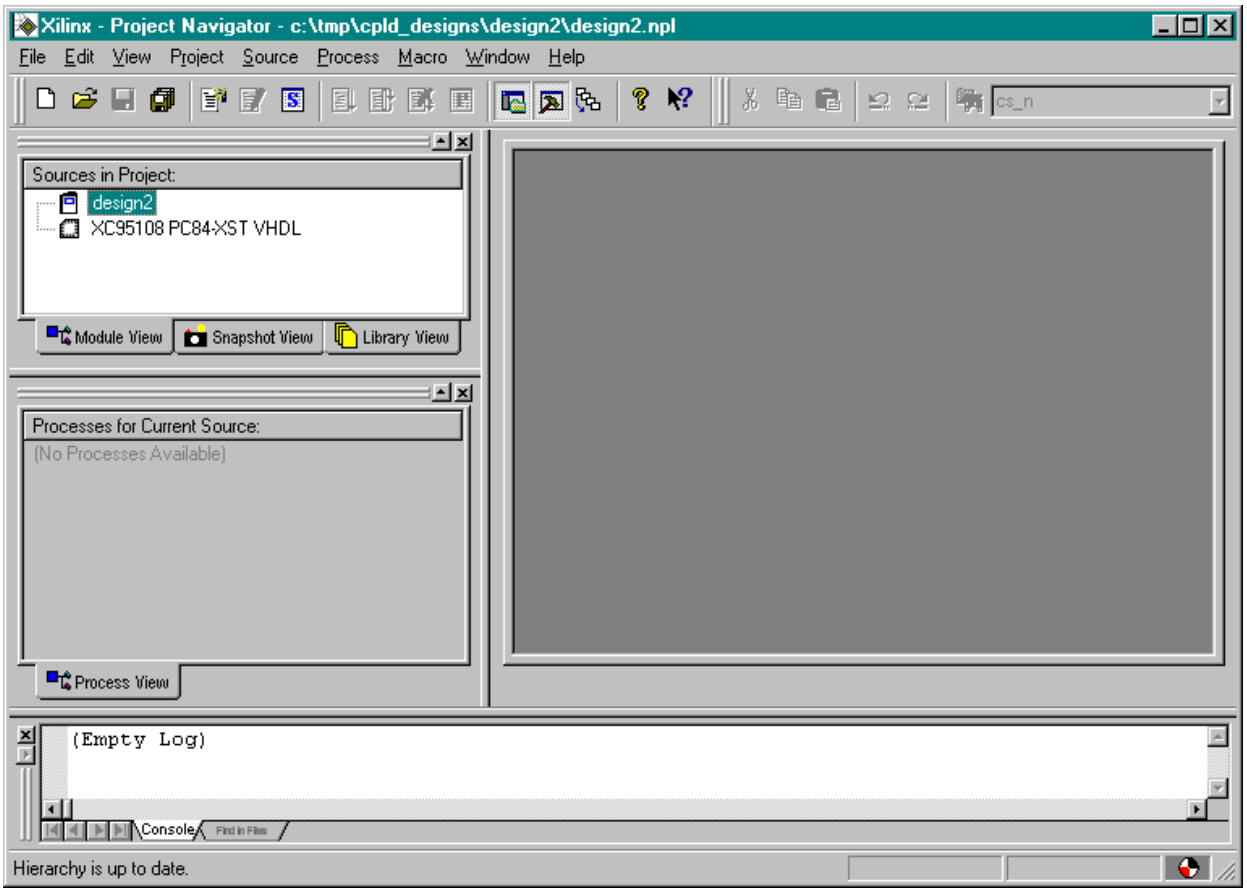
This design is hierarchical in nature. The LED decoder and counter are modules which are interconnected within a top-level module.

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\cpld_designs. The other properties in the **New Project** window retain the same values we set in the previous project.

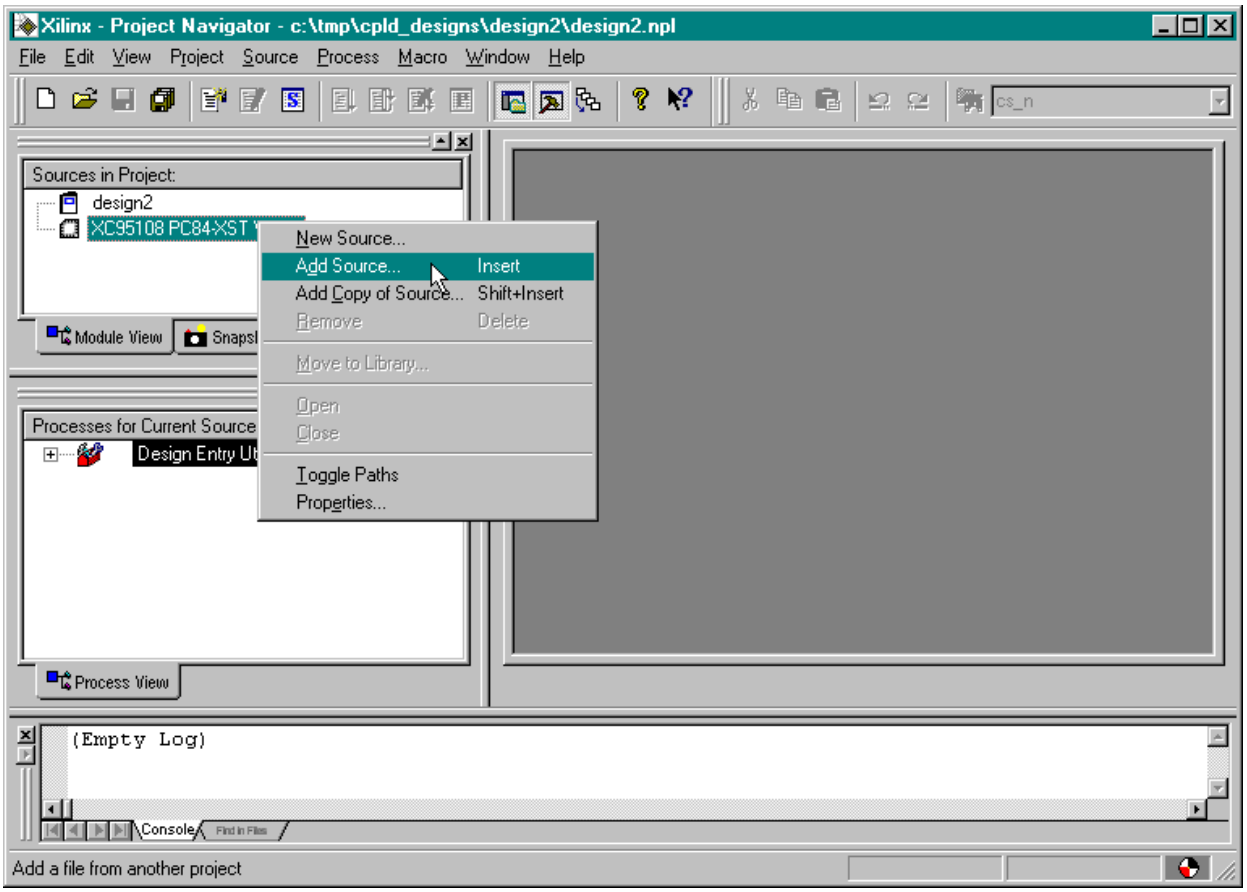


Once we click on OK in the **New Project** window, the **Project Navigator** window appears as shown below.

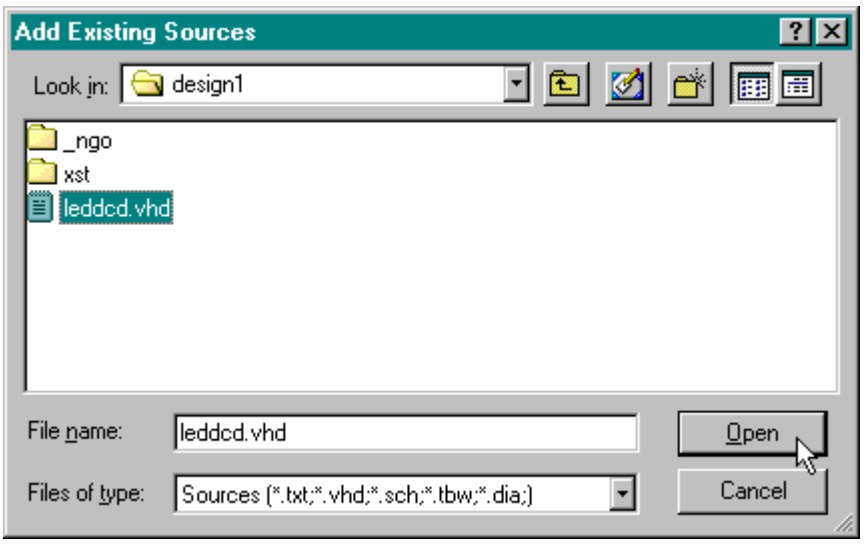


Adding the LED Decoder

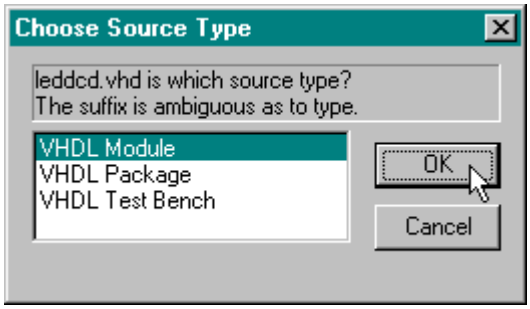
The first thing we do after getting the ***design2*** project started is to add the LED decoder module. We do this by right-clicking on the XC95108 PC84 object in the Sources pane and selecting Add Source ... from the pop-up menu.



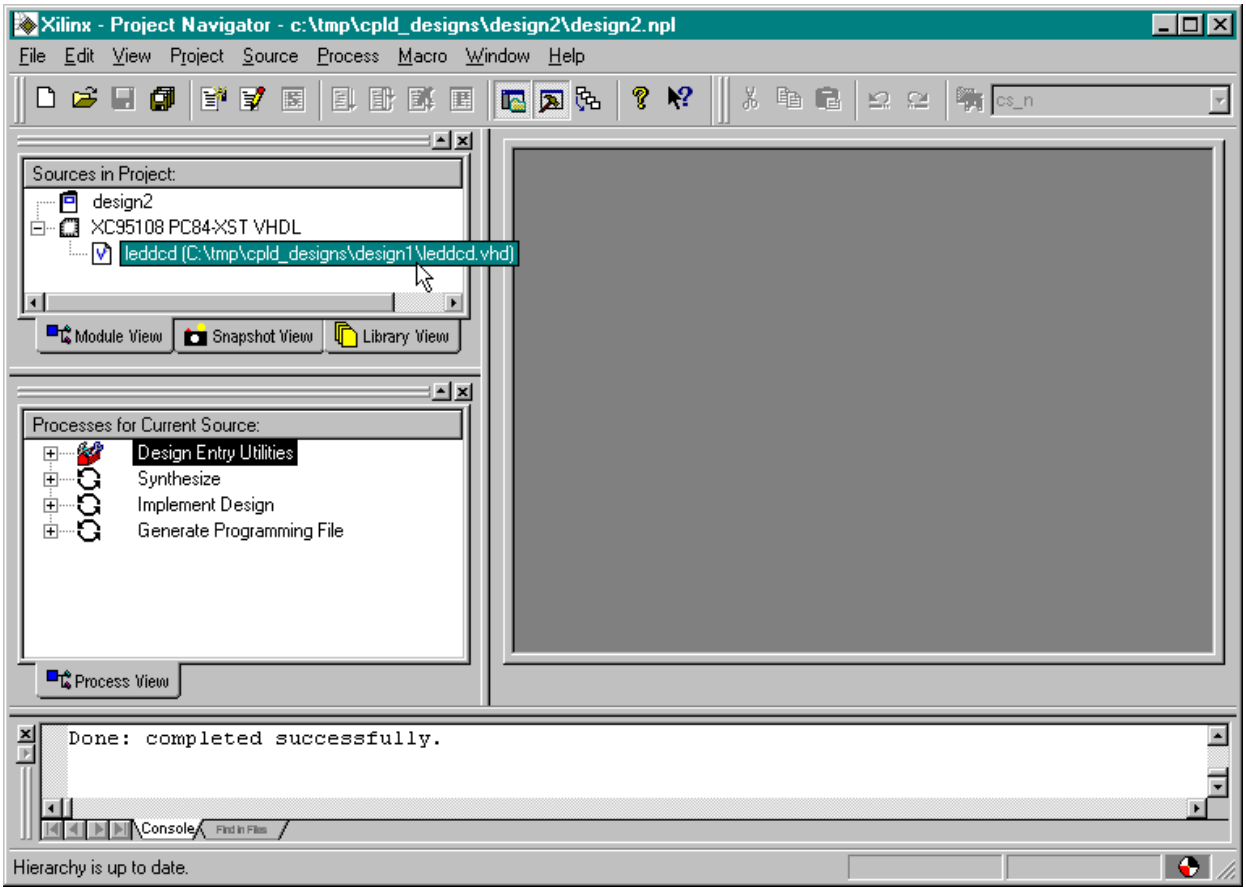
The **Add Existing Sources** window appears and we move to the C:\tmp\cpld_designs\design1 folder. Then we highlight the leddcd.vhd file that contains the VHDL source code for the LED decoder.



After clicking on Open, a window appears that asks us the type of file we are adding to the project.

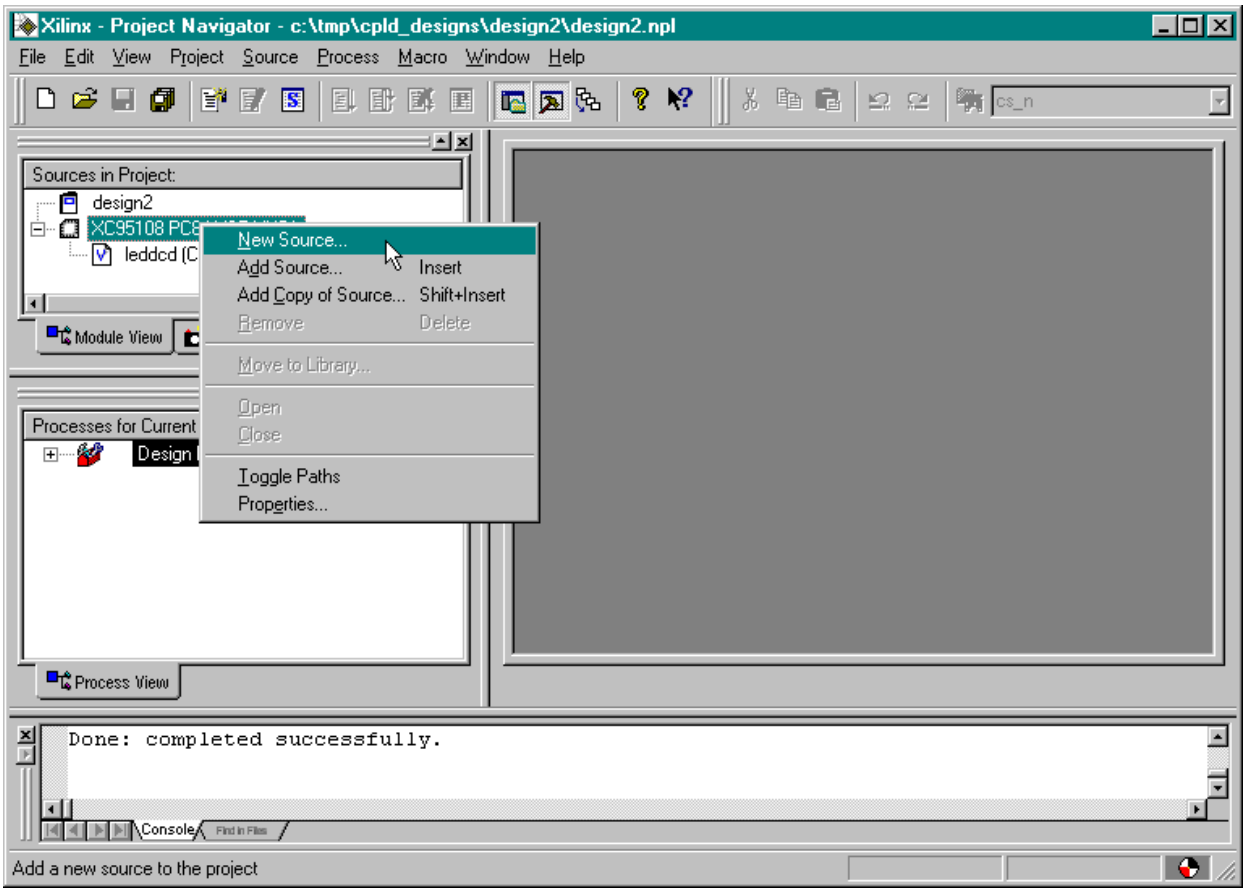


We select VHDL Module since the leddcd.vhd file contains a standard VHDL description of a circuit. (Packages contain extra syntactical elements for modules meant to be used as a library. Test benches contain VHDL code that exercises other VHDL modules through a sequence of tests.) After clicking OK, we see that the LED decoder module has been added to the Source pane of the **Project Navigator** window on the next page.

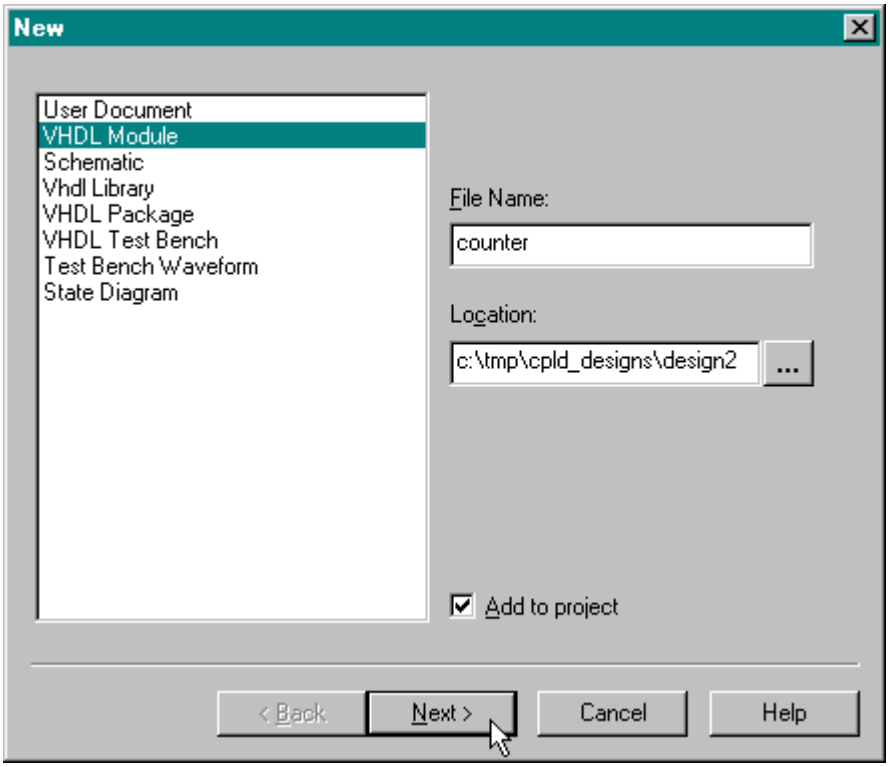

Xilinx - Project Navigator - c:\tmp\cpld_designs\design2\design2.npl
File Edit View Project Source Process Macro Window Help
Sources in Project:
design2
XC95108 PC84-XST VHDL
leddcd (C:\tmp\cpld_designs\design1\leddcd.vhd)
Module View
Snapshot View
Library View
Processes for Current Source:
Design Entry Utilities
Synthesize
Implement Design
Generate Programming File
Process View
Done: completed successfully.
Console
Find in Files
Hierarchy is up to date.

Adding a Counter

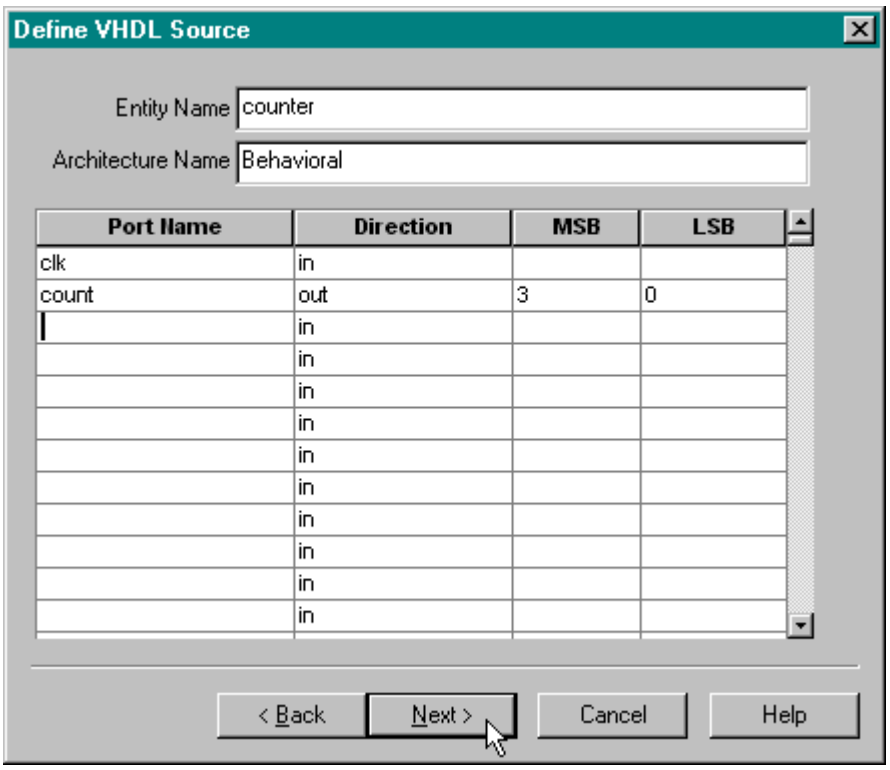
Now we have to add the counter to our design. We don't have a counter module yet, so we have to build one with VHDL. Right-click on the XC95108 PC84 object and select New Source... from the pop-up menu.



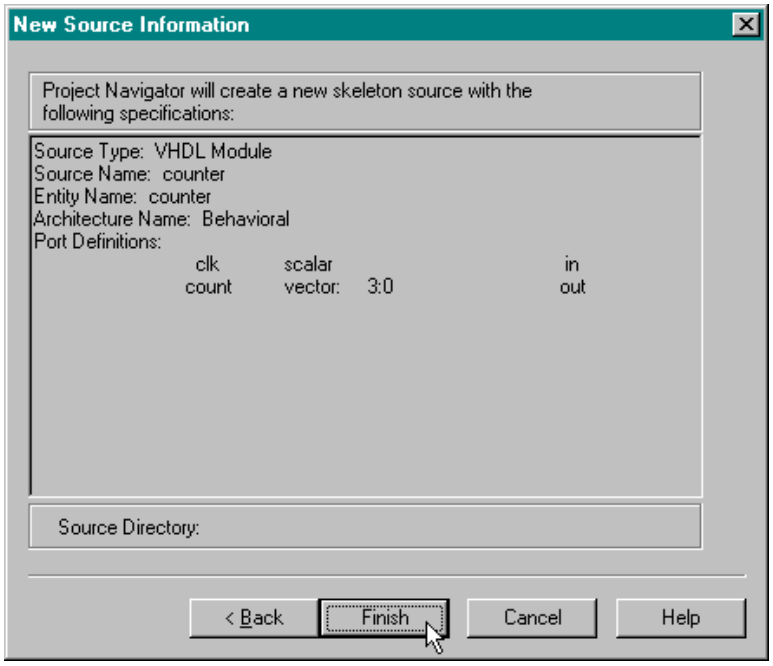
As in the previous example, we are prompted for the type of file we want to add to the project. Once again, we select the VHDL Module menu item. Then we type `counter` into the File Name field and click on the Next button.



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:

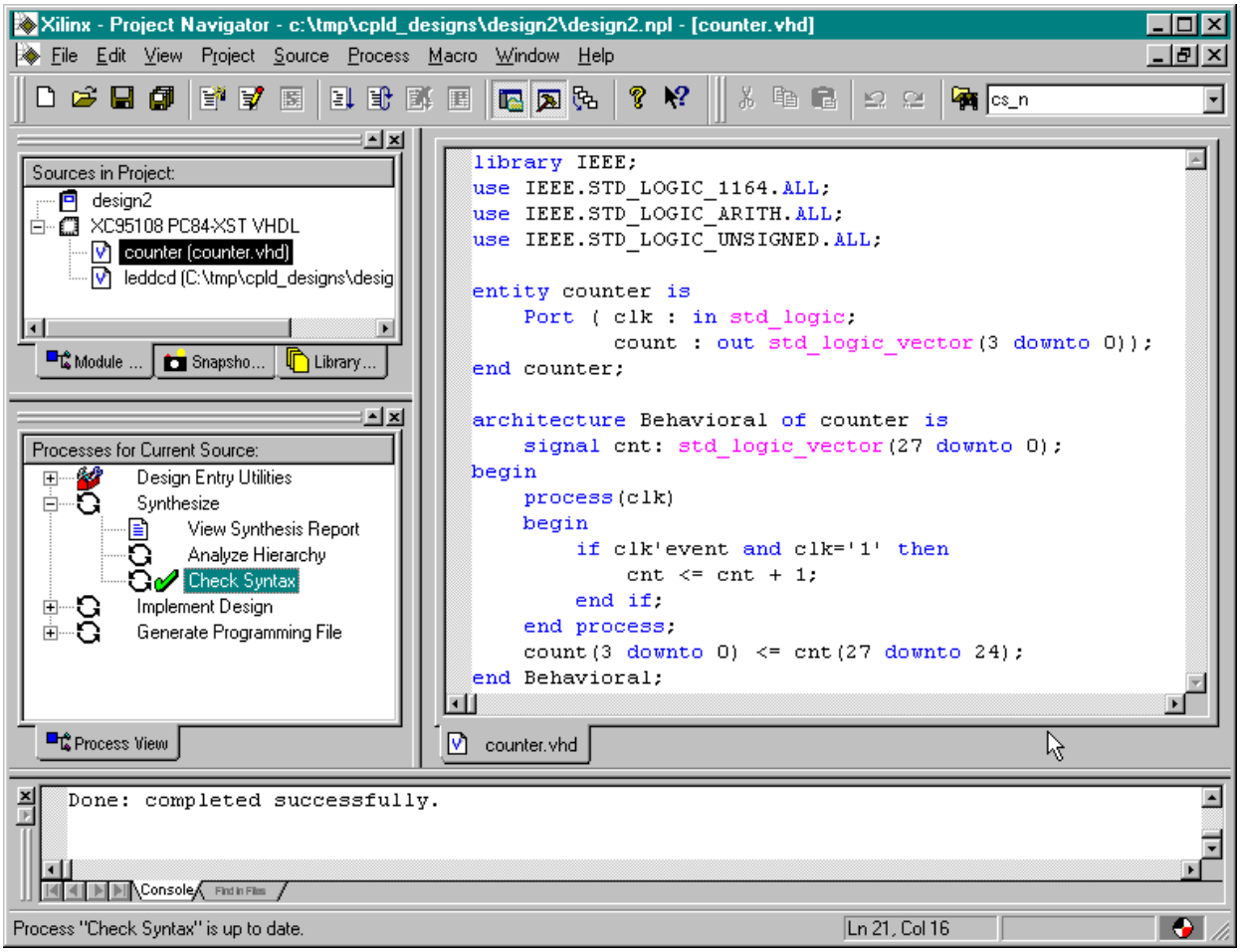
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity counter is
    Port ( clk : in std_logic;
           count : out std_logic_vector(3 downto 0));
end counter;

architecture Behavioral of counter is
    signal cnt: std_logic_vector(27 downto 0);
begin
    process(clk)
    begin
        if clk'event and clk='1' then
            cnt <= cnt + 1;
        end if;
    end process;
    count(3 downto 0) <= cnt(27 downto 24);
end Behavioral;
```

counter.vhd *

Line 12 declares a 28-bit signal, **cnt**, that is the current value of the counter. The process on lines 15-20 controls when counter increments. The condition clause of line 16 is only true when the value on the **clk** input goes from 0 to 1. Then the statement on line 17 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 22 places the upper four bits of the current counter value onto the outputs of the module.

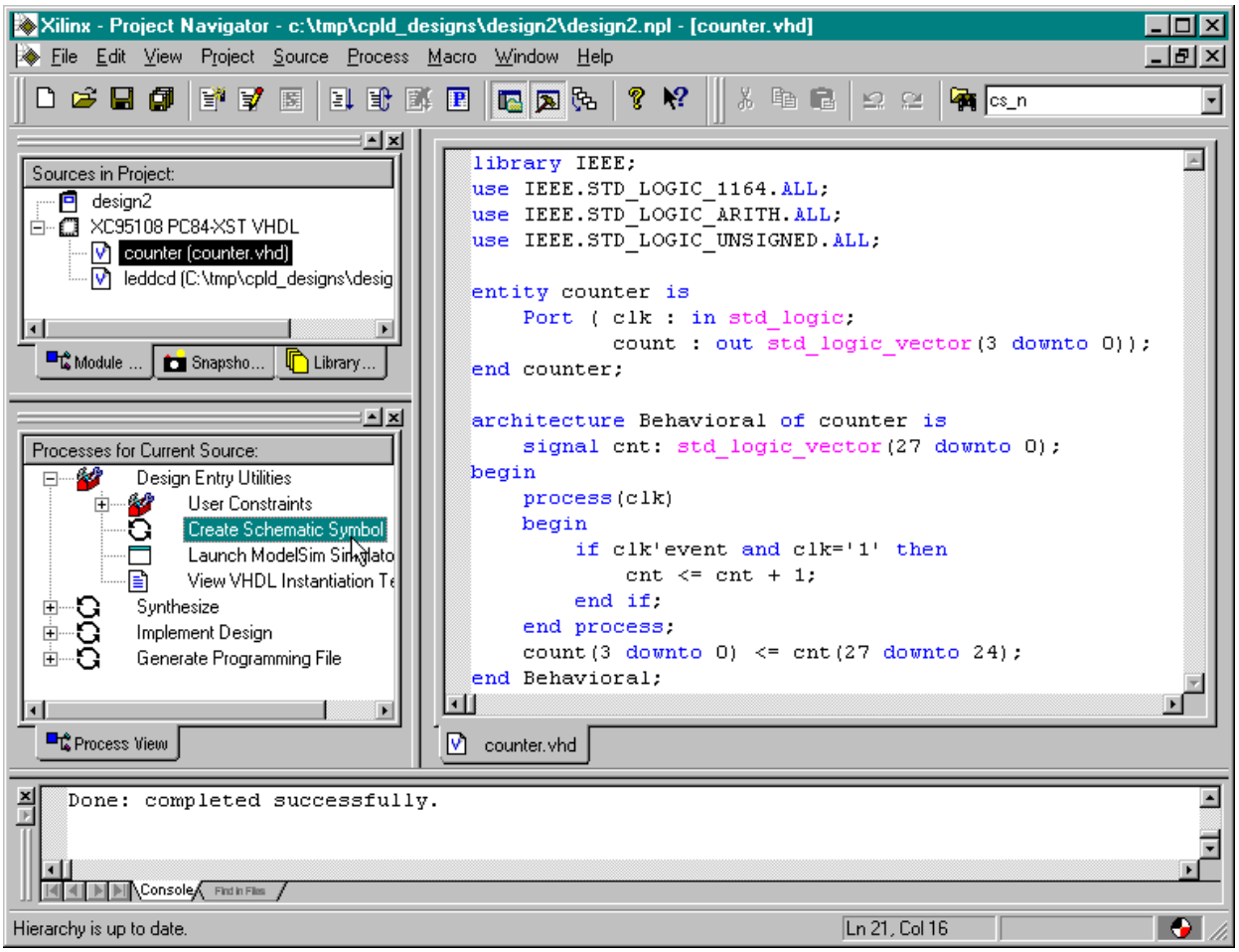
Why are we building a 28-bit counter and using only the upper four bits? The counter will be driven by the oscillator on the XS95 Board which has a default frequency of 50 MHz. The LED display would be changing much too quickly at this frequency. By connecting the LED decoder to the upper four bits of 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 seen.

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.

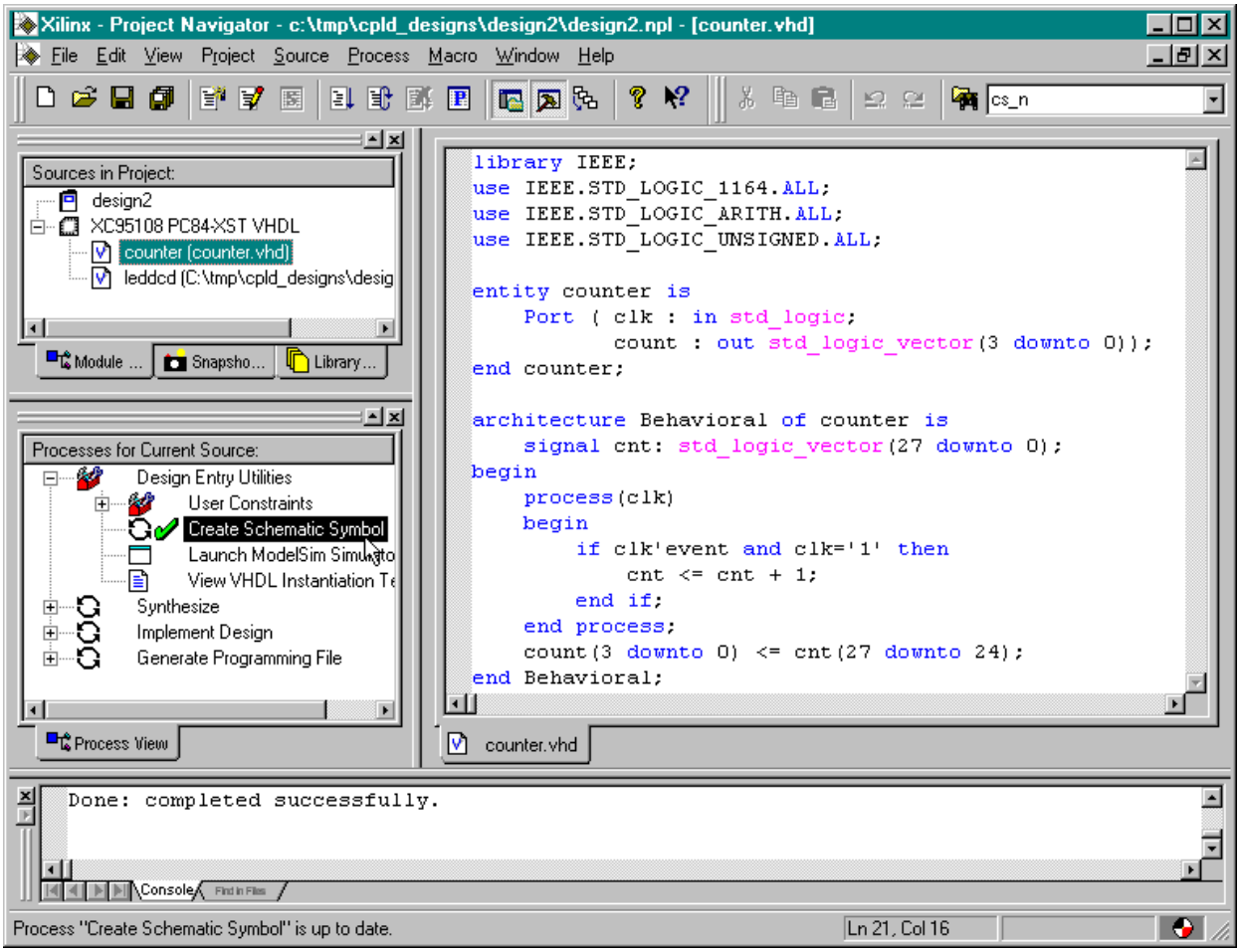


Tying Them Together

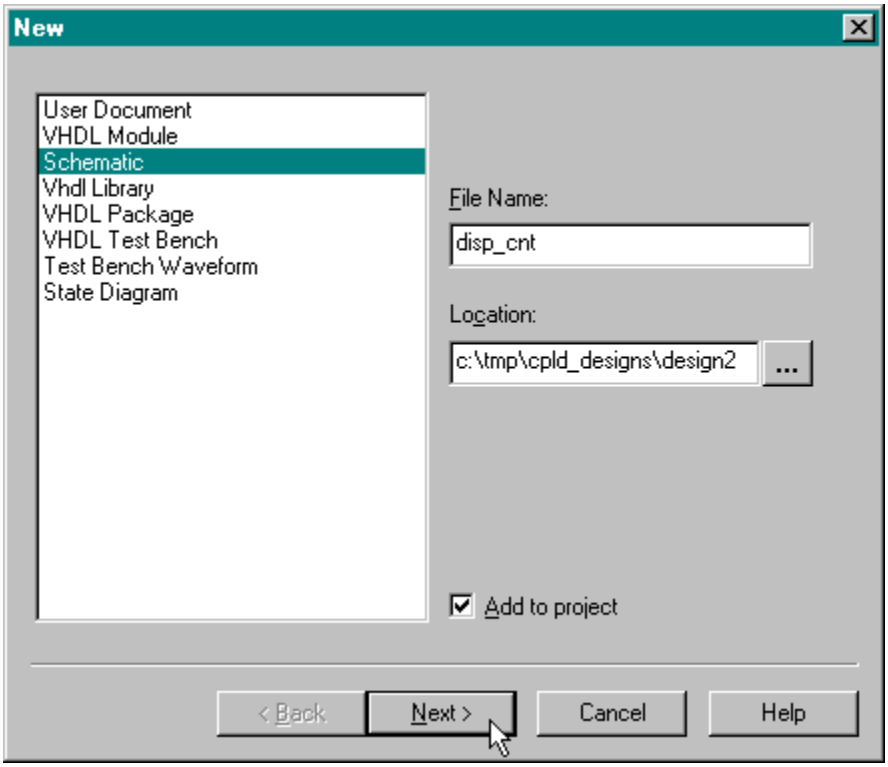
We have the LED decoder and the counter, but now we need to tie them together to build the displayable counter. We will do this by connecting the counter to the LED decoder in a top-level schematic. Before we can do this, we have to create schematic symbols for both the counter and LED decoder modules.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



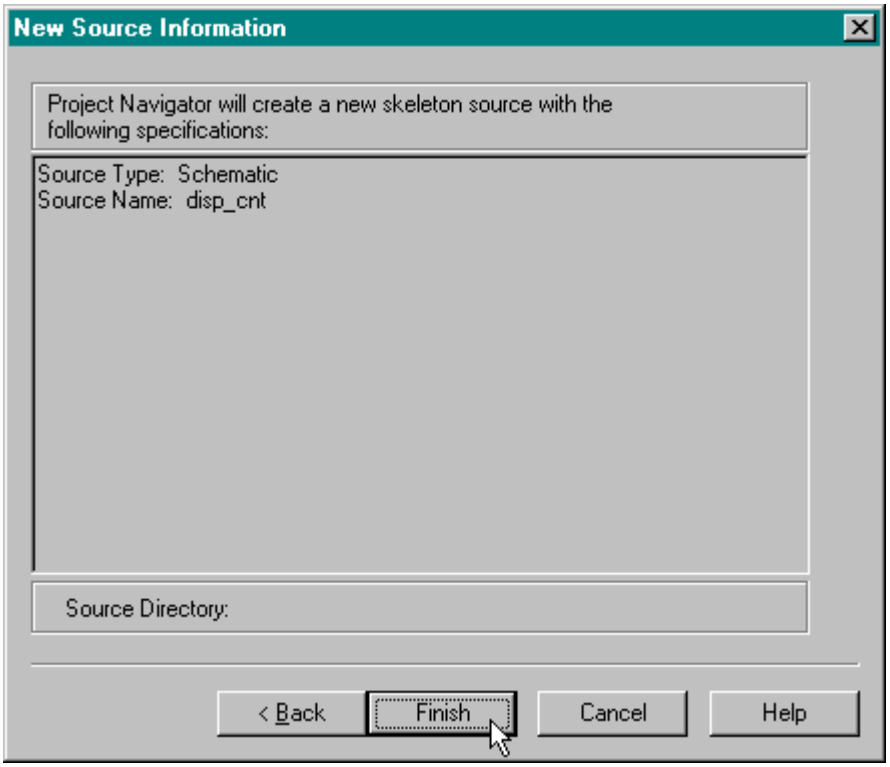
A ✔ will appear next to the Create Schematic Symbol process after the symbol is created. Repeat this procedure to create the schematic symbol for the LED decoder.



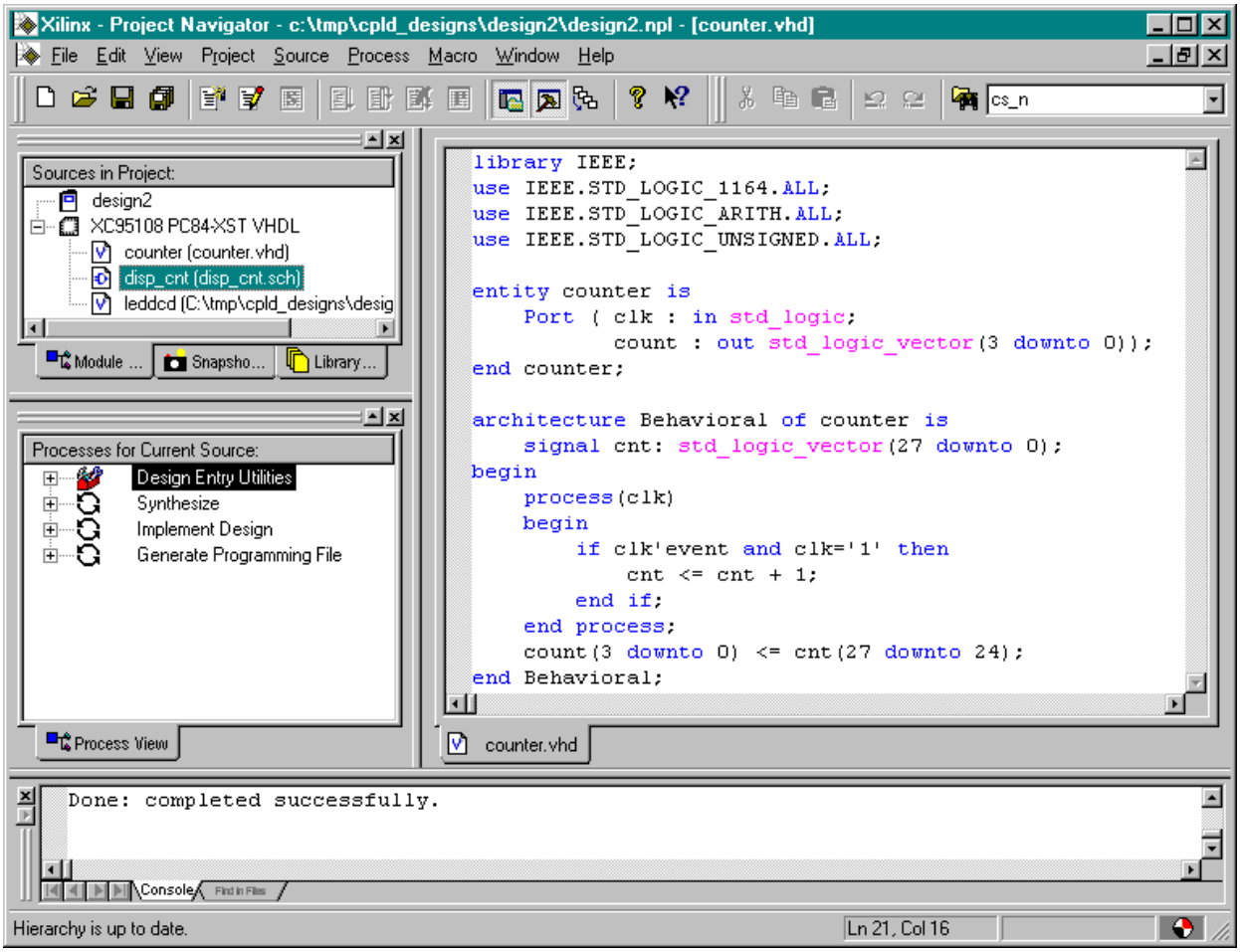
Once the schematic symbols for the lower-level modules are built, we can add the top-level schematic to the project. Right-click on the XC95108 PC84 object and select New Source... from the pop-up menu. Then highlight the Schematic entry in the **New** window and name the schematic **disp_cnt**. Then click on Next.



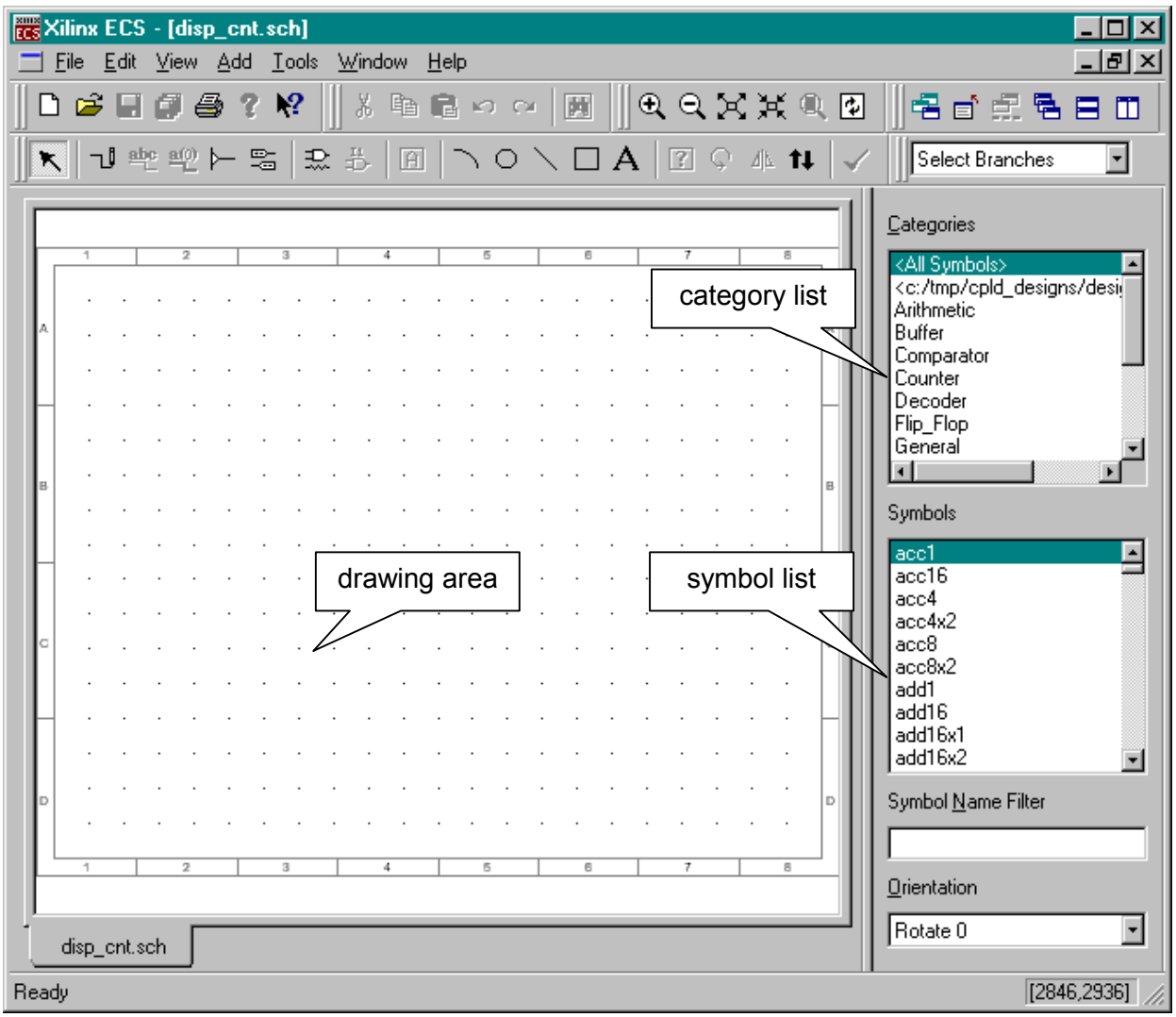
There is very little to do when setting-up a schematic, so just click on the Finish button in the **New Source Information** window that appears.



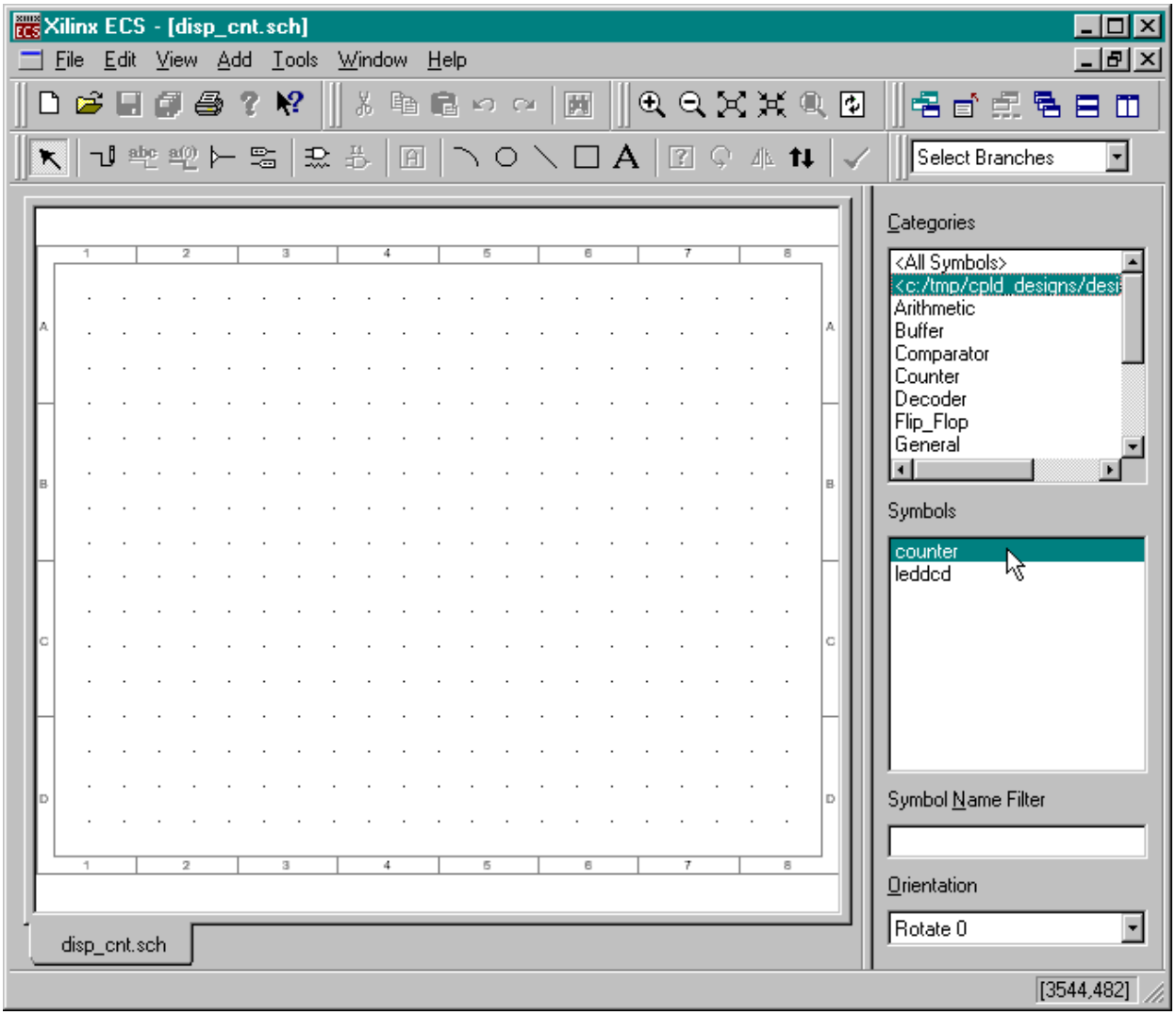
Now the disp_cnt schematic object has been added to the Sources pane. Double-click it to open a schematic window.



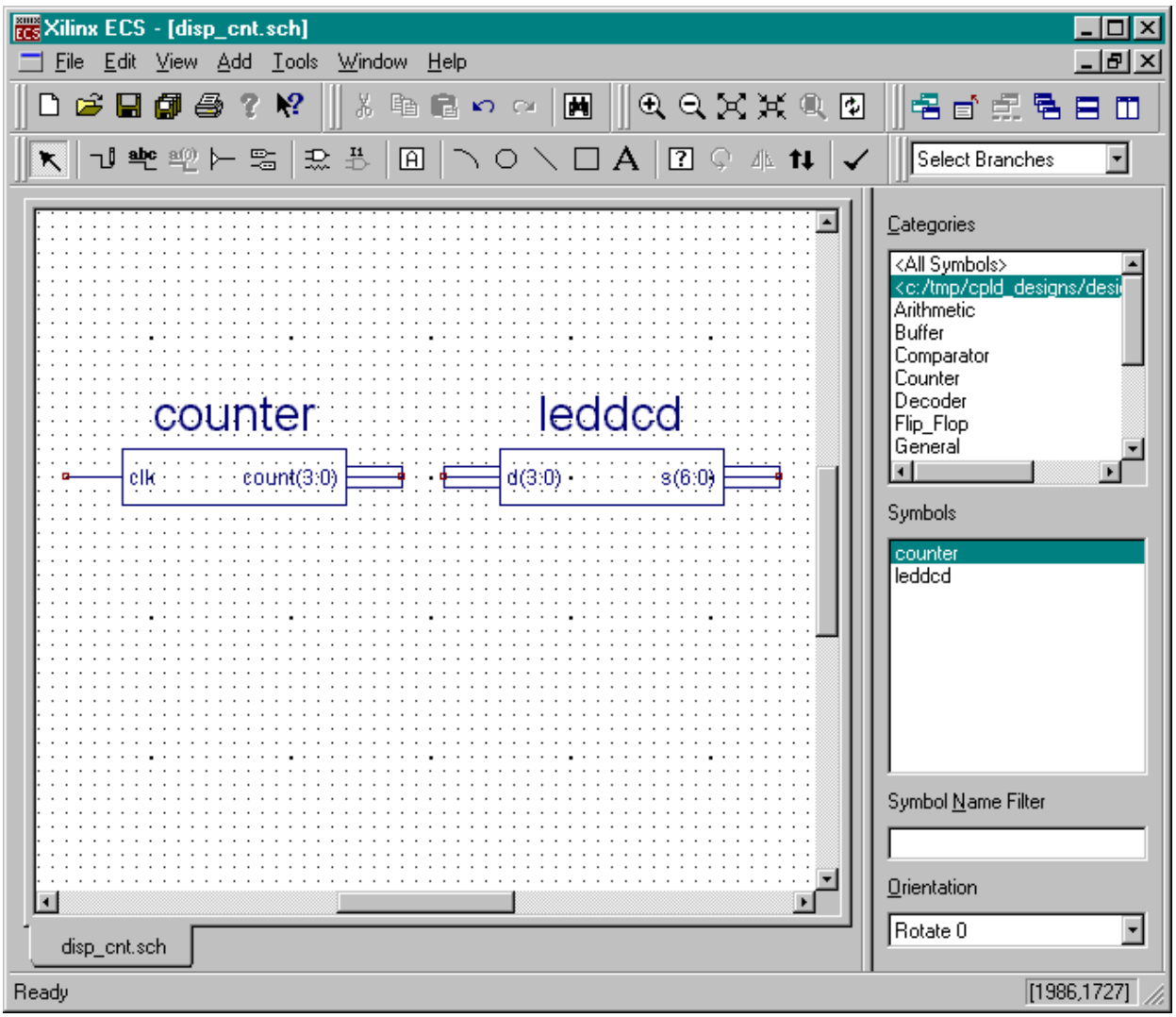
The schematic editor window has a drawing area and a list of categories for various logic circuit elements that can be used in a schematic. Below that is the list of symbols for circuit elements in a highlighted category.



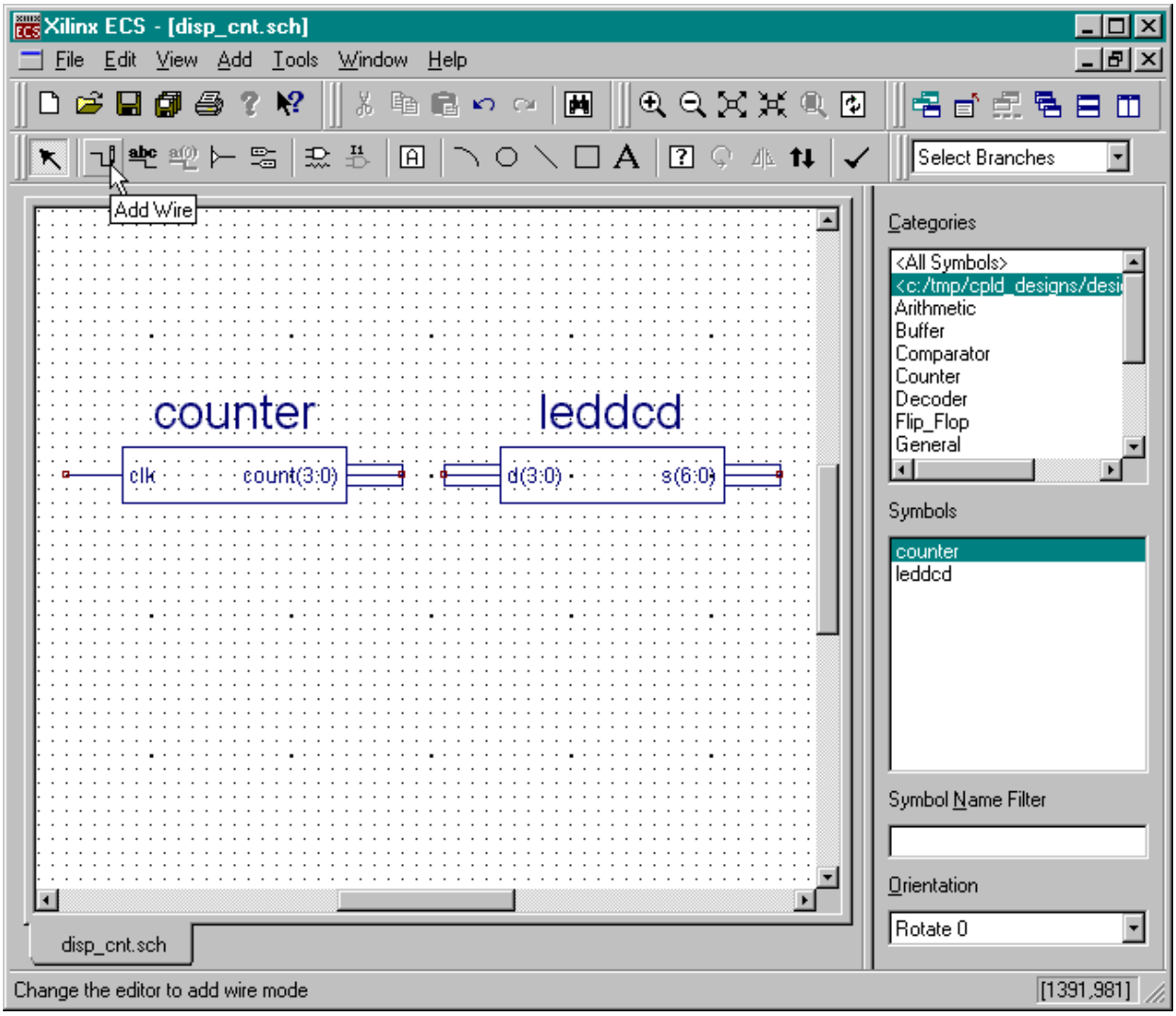
To start creating the top-level schematic, highlight the second entry in the category list. The c:/tmp/cpld_designs/design2 category contains the schematic symbols for the ***design2*** project's counter and LED decoder modules. We can see the names of these modules in the symbol list.



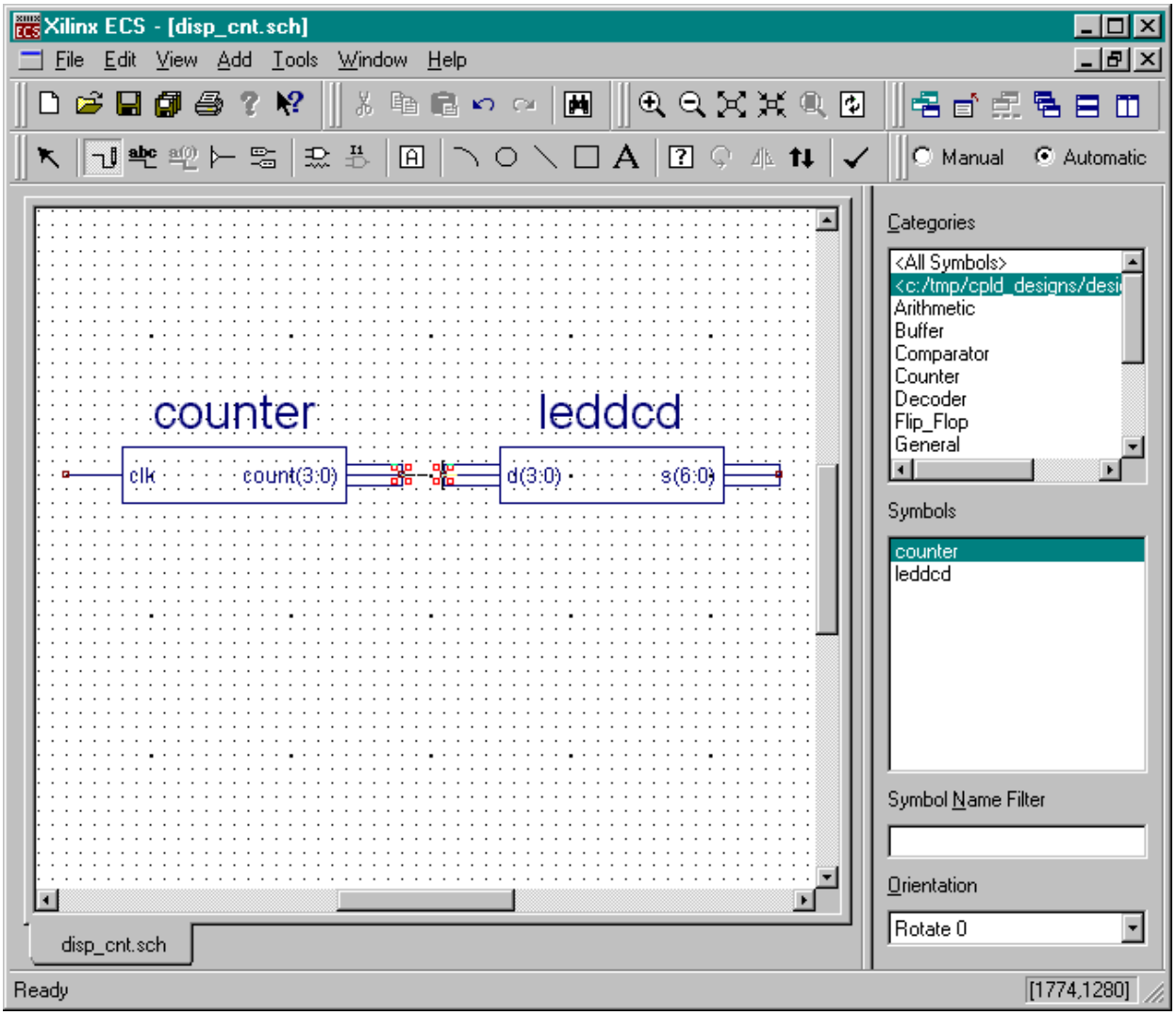
Click on the counter entry in the Symbols list. Then move the mouse cursor into the drawing area and left-click to place an instance of the counter into the schematic. Repeat this process with the leddcd module to arrive at the result shown below.



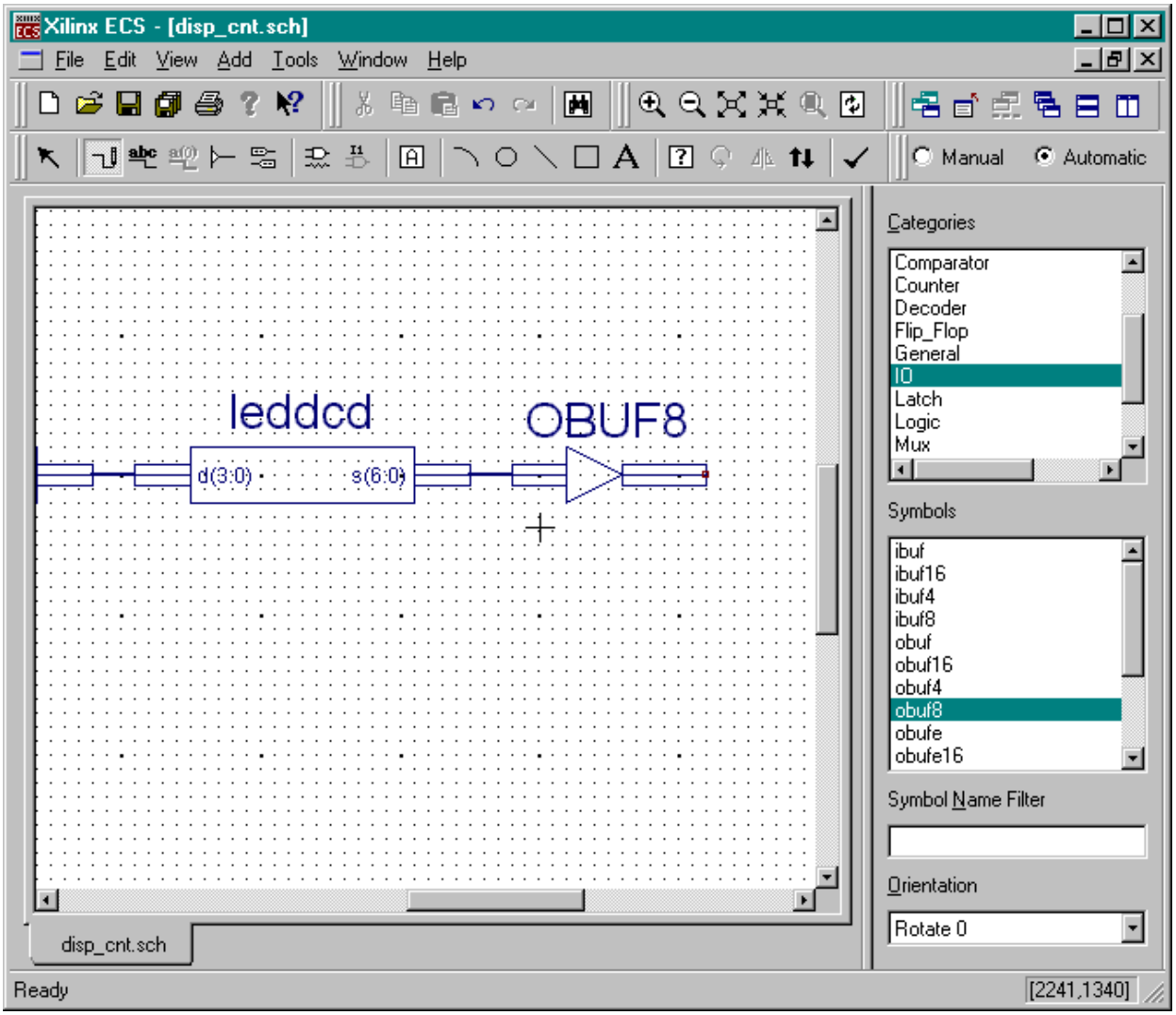
Next, click on the button to begin adding wires to the schematic.



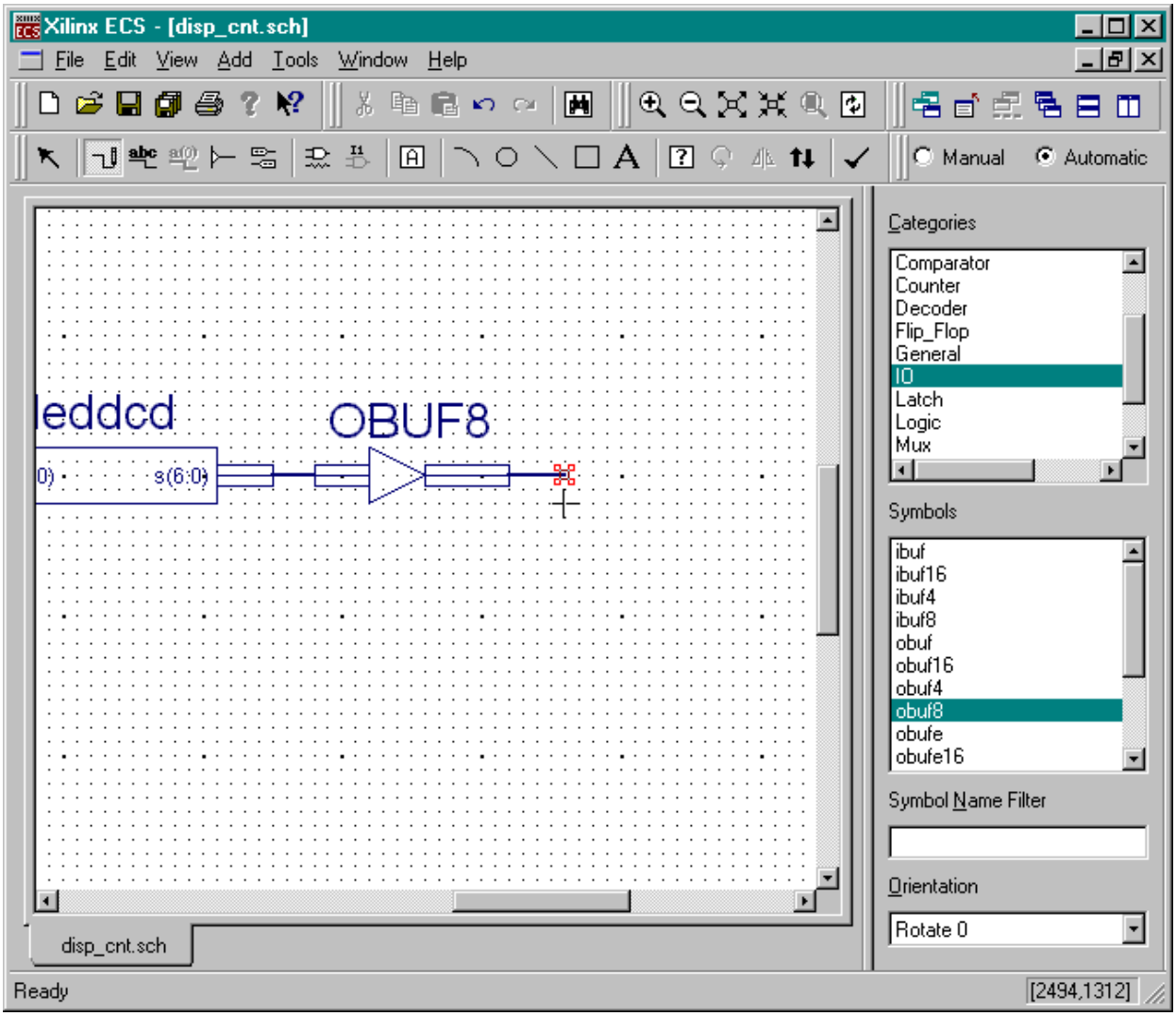
Left-click the mouse on the **count(3:0)** bus on the right-hand edge of the **counter** module. Then left-click on the **d(3:0)** bus on the left-hand edge of the **leddcd** module. As a result of this procedure, a four-bit bus is created between the output of the counter module and the input of the LED decoder module. Either click in the same endpoint or hit the ESC key to stop adding segments to the bus.



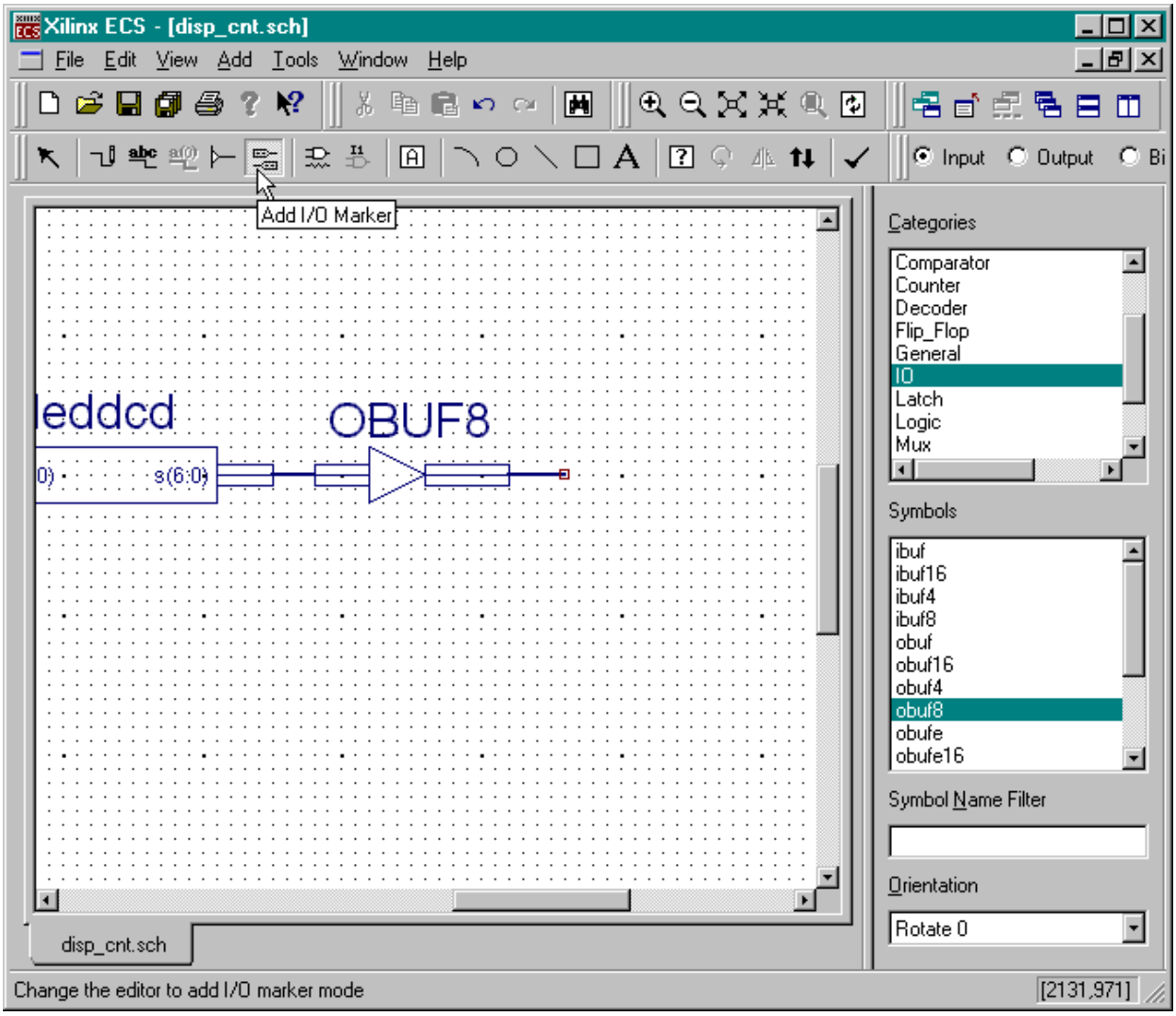
Now highlight the IO category and select a byte-wide output buffer (OBUF8) from the list of symbols. Attach the output buffer to the output of the LED decoder as shown below.



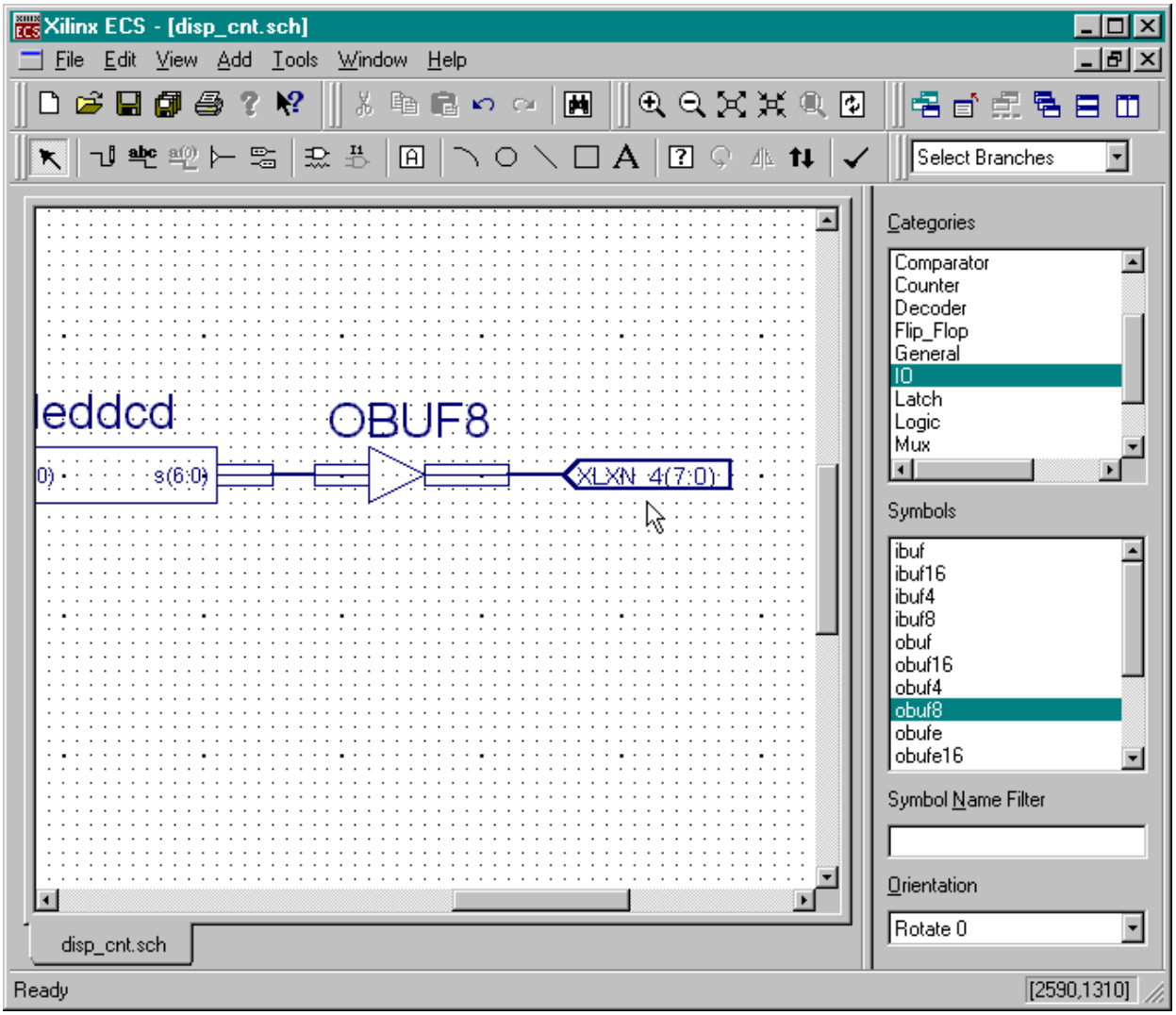
Next attach a short bus segment to the output of the byte-wide buffer.



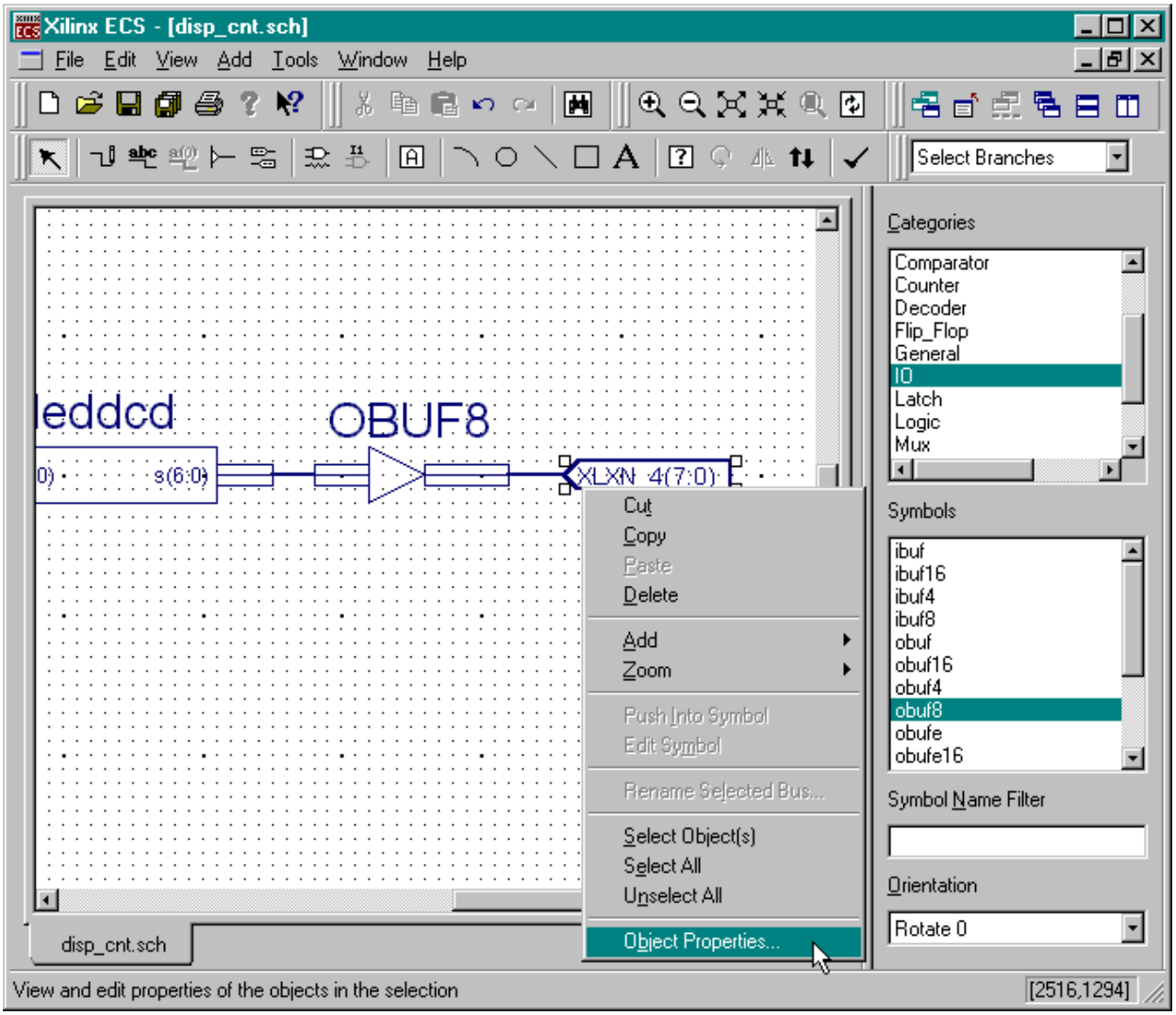
Now click on the button for adding I/O markers.



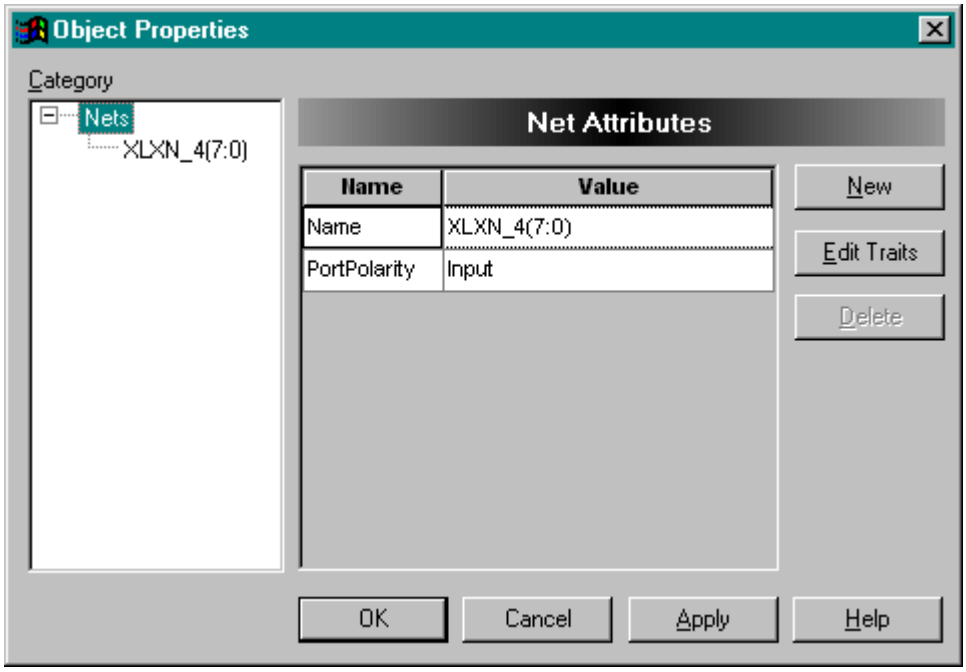
Then click on the other end of the newly-added bus segment to create a byte-wide set of output pins.



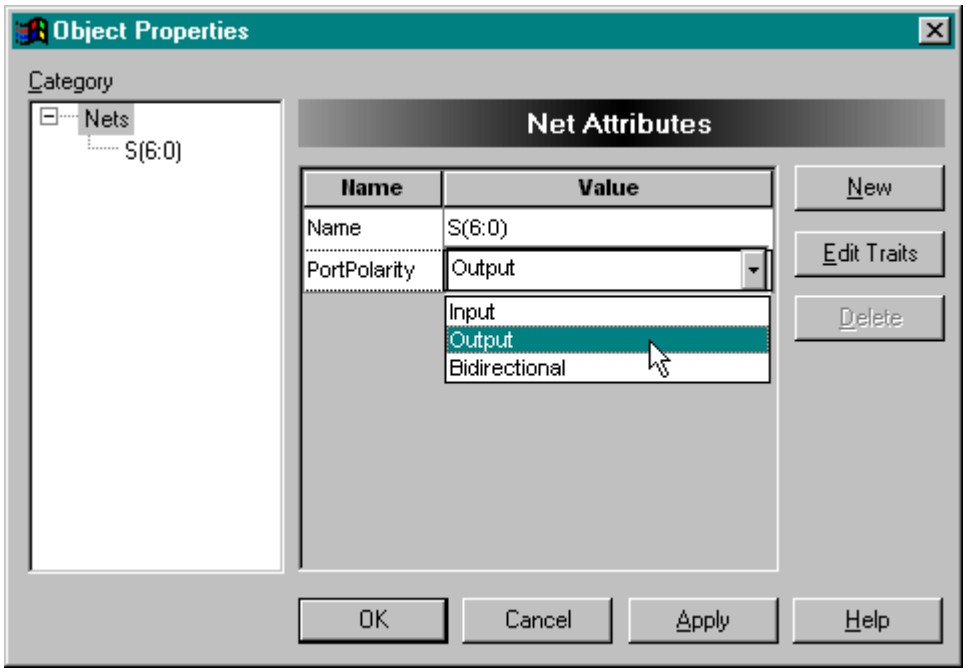
The output pins automatically assume the same name as the bus to which they are attached but this name was automatically generated and doesn't carry a lot of meaning. To change the name of the outputs (and the associated bus), right-click on the I/O marker and select Object Properties... from the pop-up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



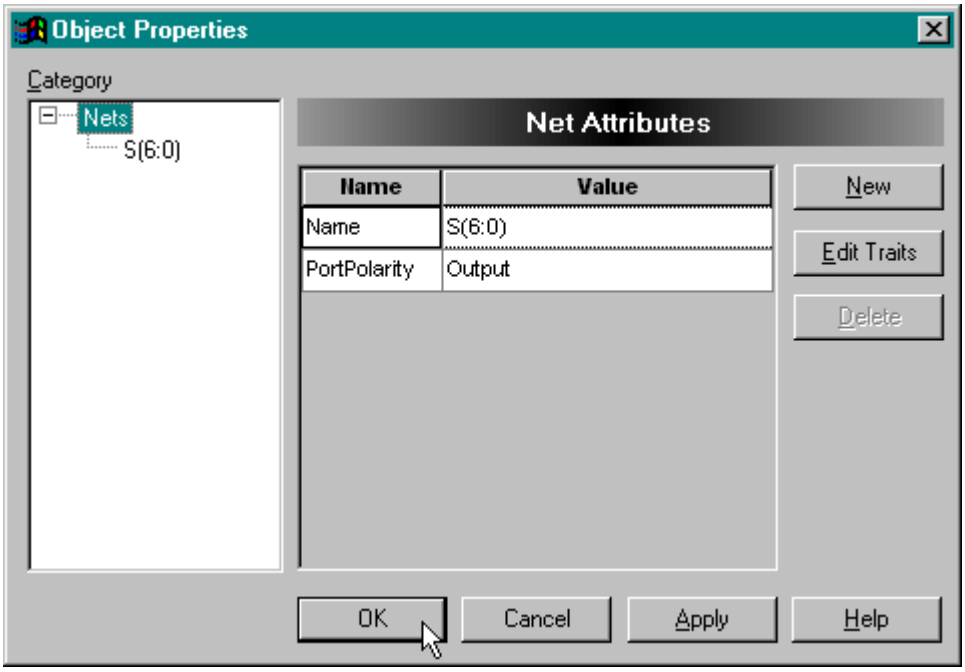
The **Object Properties** window allows us to set the name and direction of the pins.



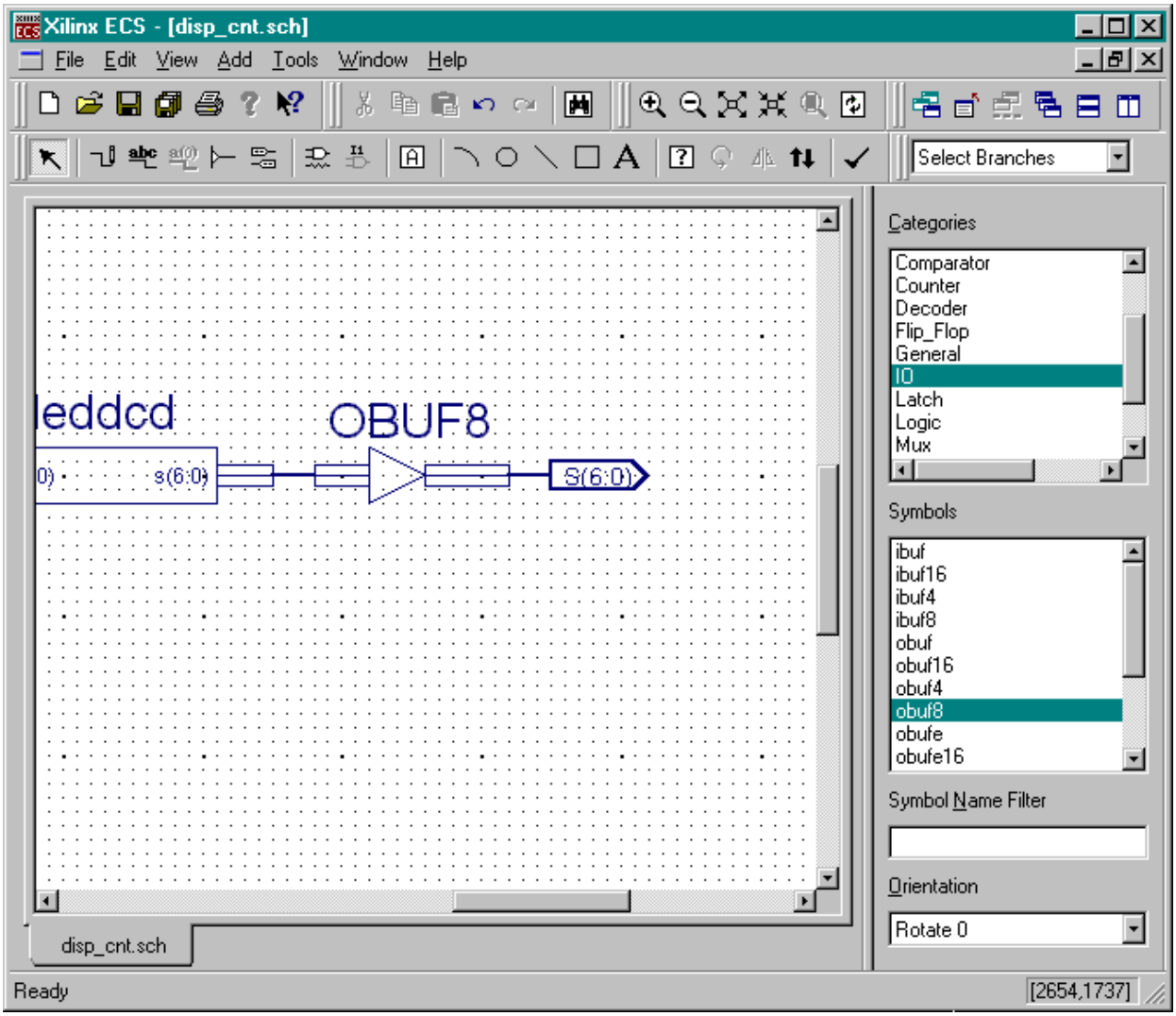
Replace the existing bus name with a seven-bit bus for driving the LED segments: **S(6:0)**. Then set the direction of the bus pins to Output.



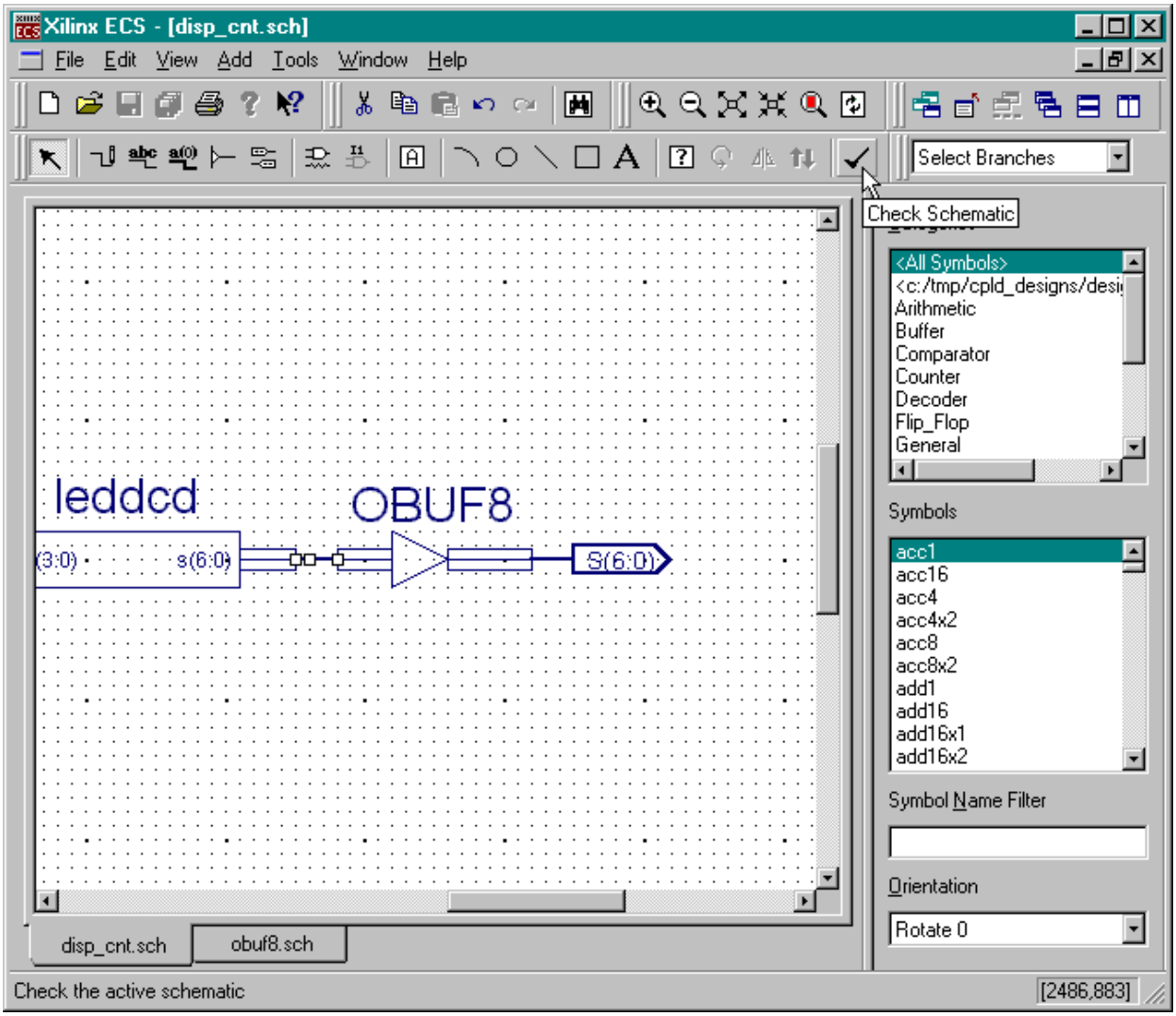
Next click on the OK button to close the **Object Properties** window.



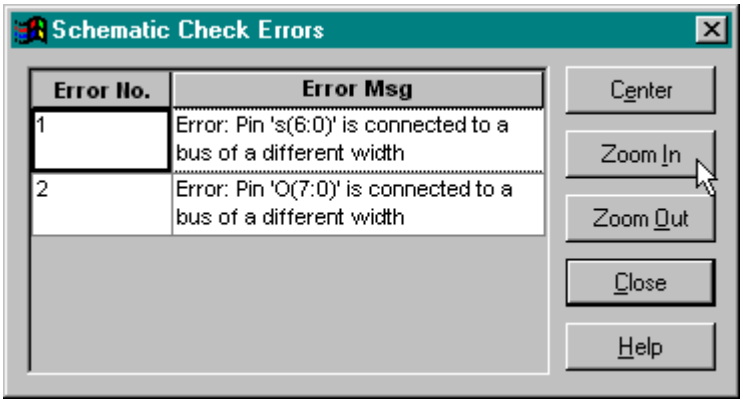
The output pins now appear with their new name, width, and direction.



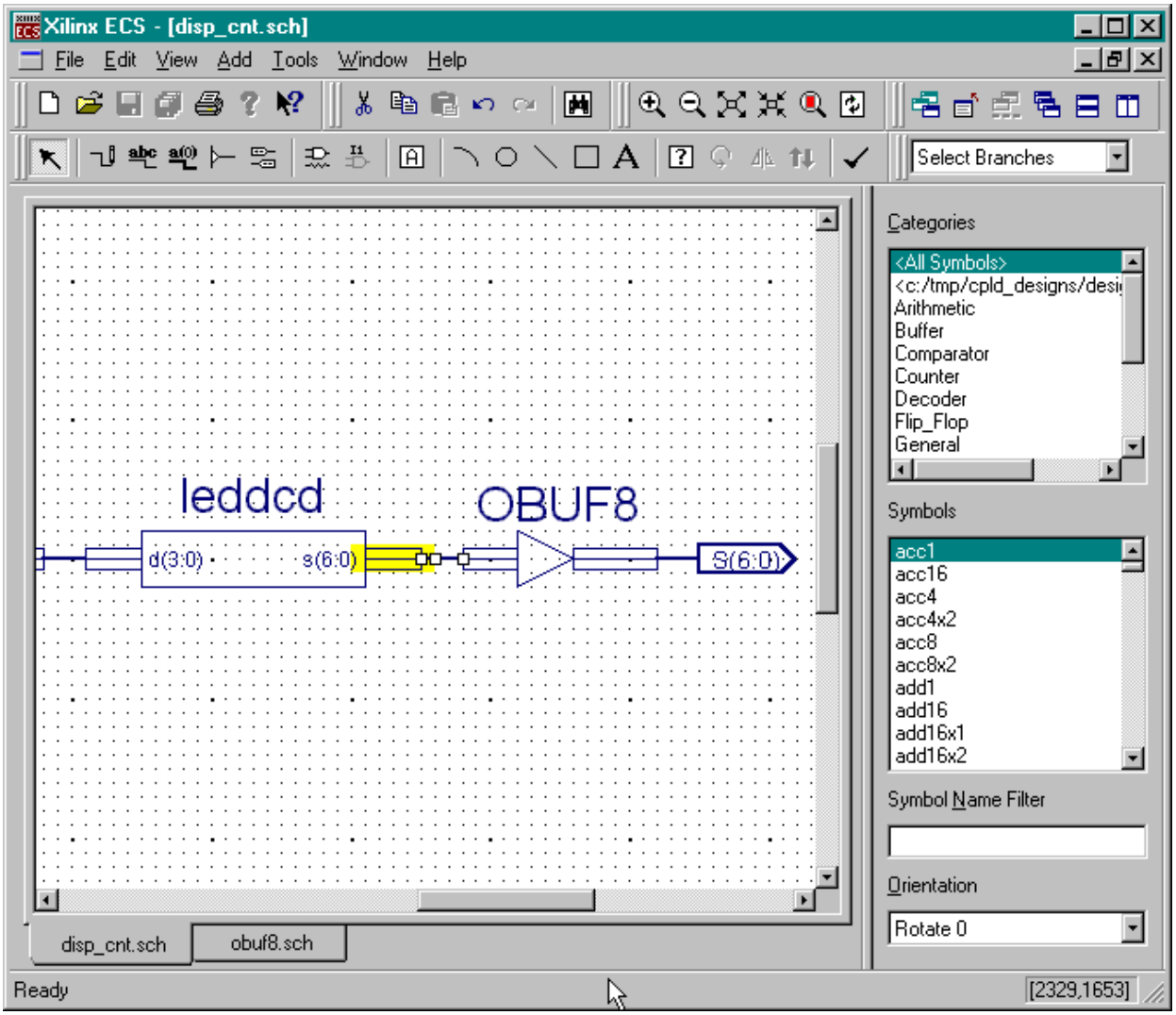
At this point it makes sense to check the schematic to see if there are any errors such as unterminated wire stubs or mismatched bus widths. Click on the button to perform a schematic check.



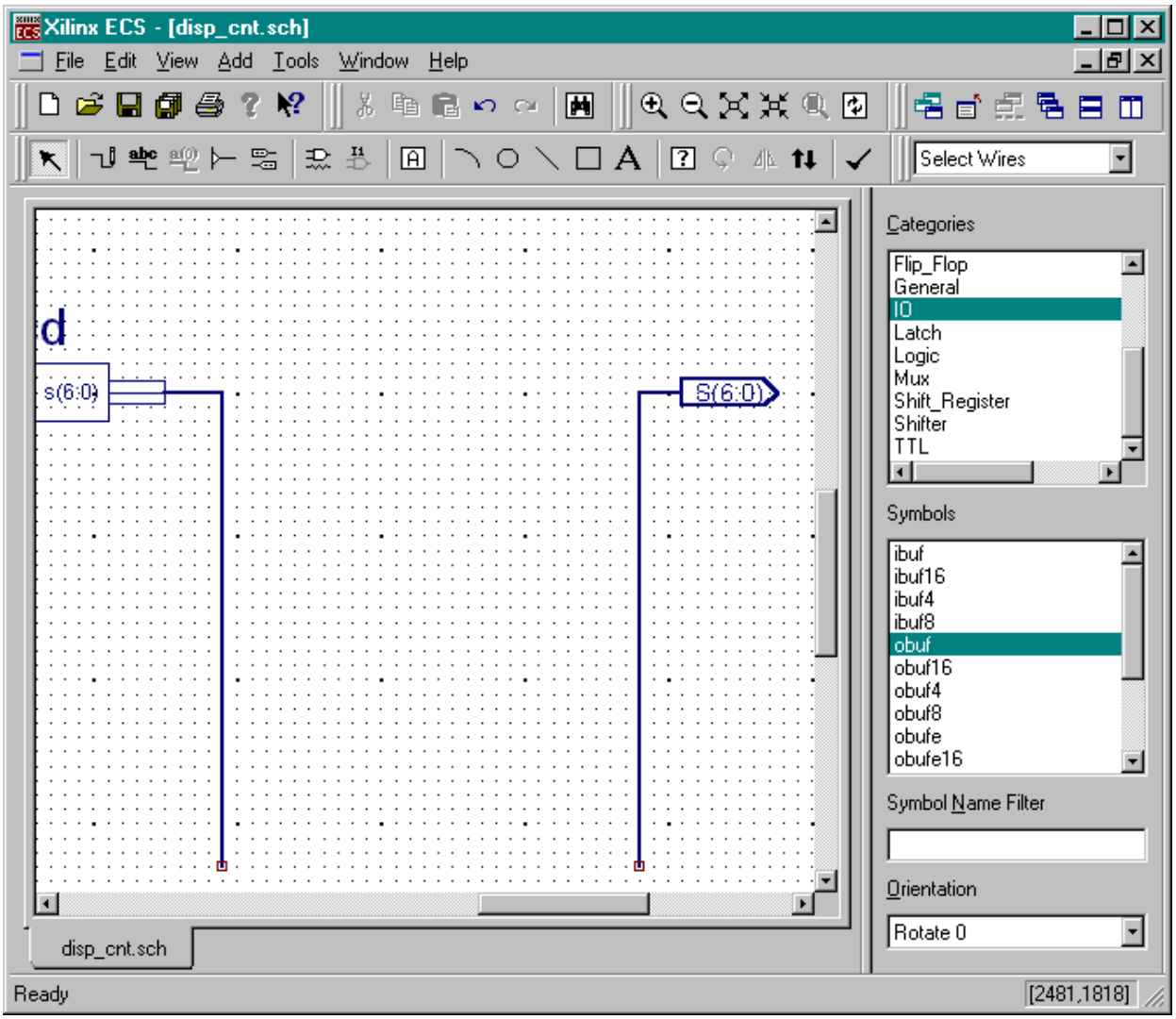
The **Schematic Check Errors** window will appear showing two errors. We can find the place in the schematic where the error occurs by clicking on the associated error message. Then clicking on the Zoom In button to see an enlarged view of the area where the error lies.



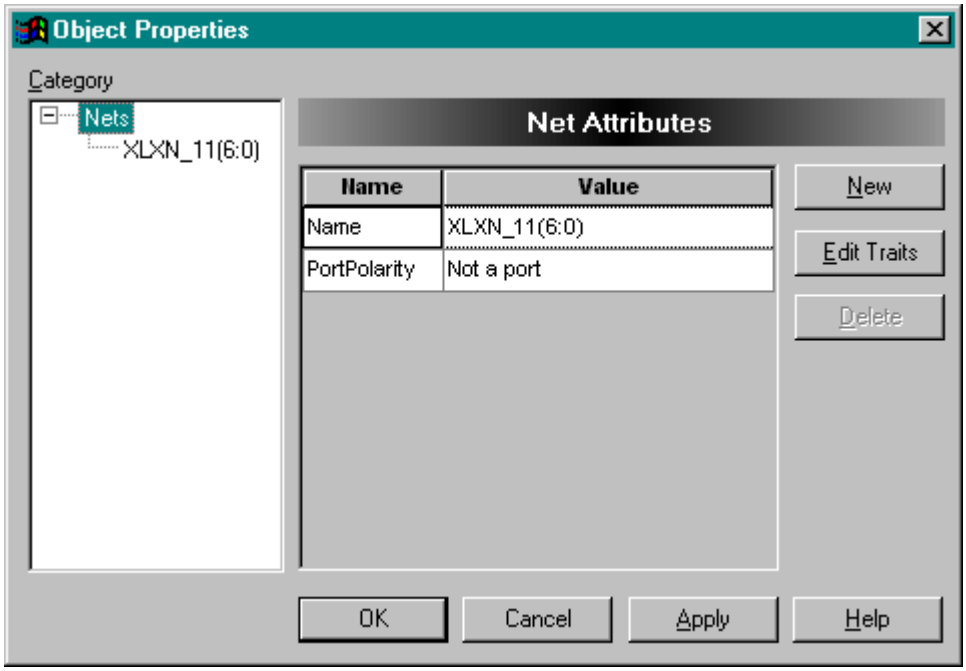
The first error indicates that the seven-bit output of the LED decoder does not match with the byte-wide input of the output buffer symbol. Note how the output of the leddcd symbol is highlighted to indicate the error. The second error is similar to the first in that the byte-wide output of the OBUF8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



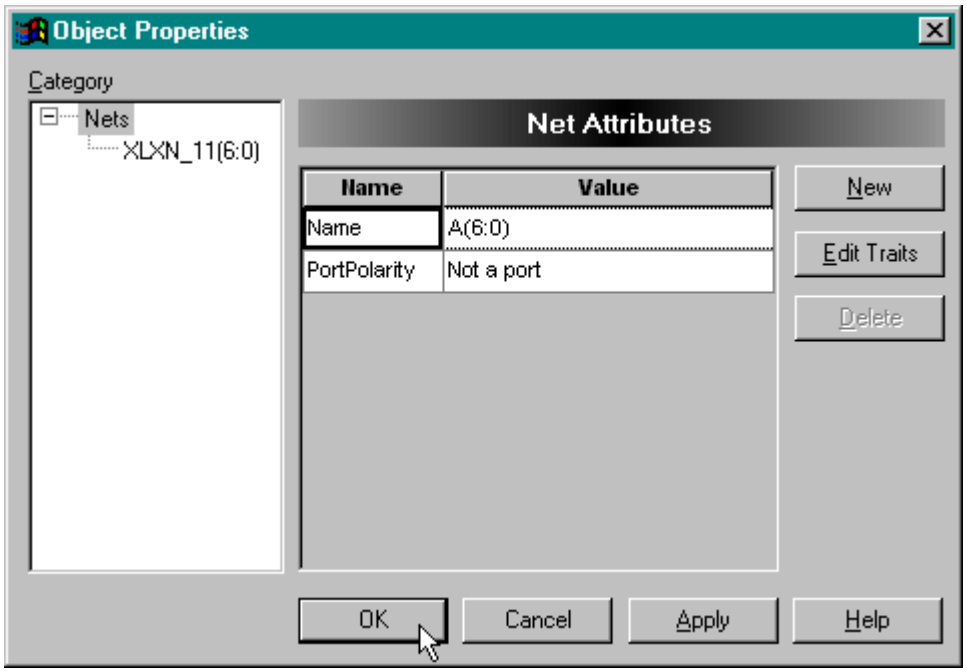
We could try solve these problems by using subsets of the buses or by adjusting the width of the output buffers. But the simplest solution is to remove the byte-wide output buffer and replace it with a group of seven output buffers. To start this process, click on the button and then click on the OBUF8 symbol. Then press the delete key. Then click on the button and add bus segments to the schematic as shown below.



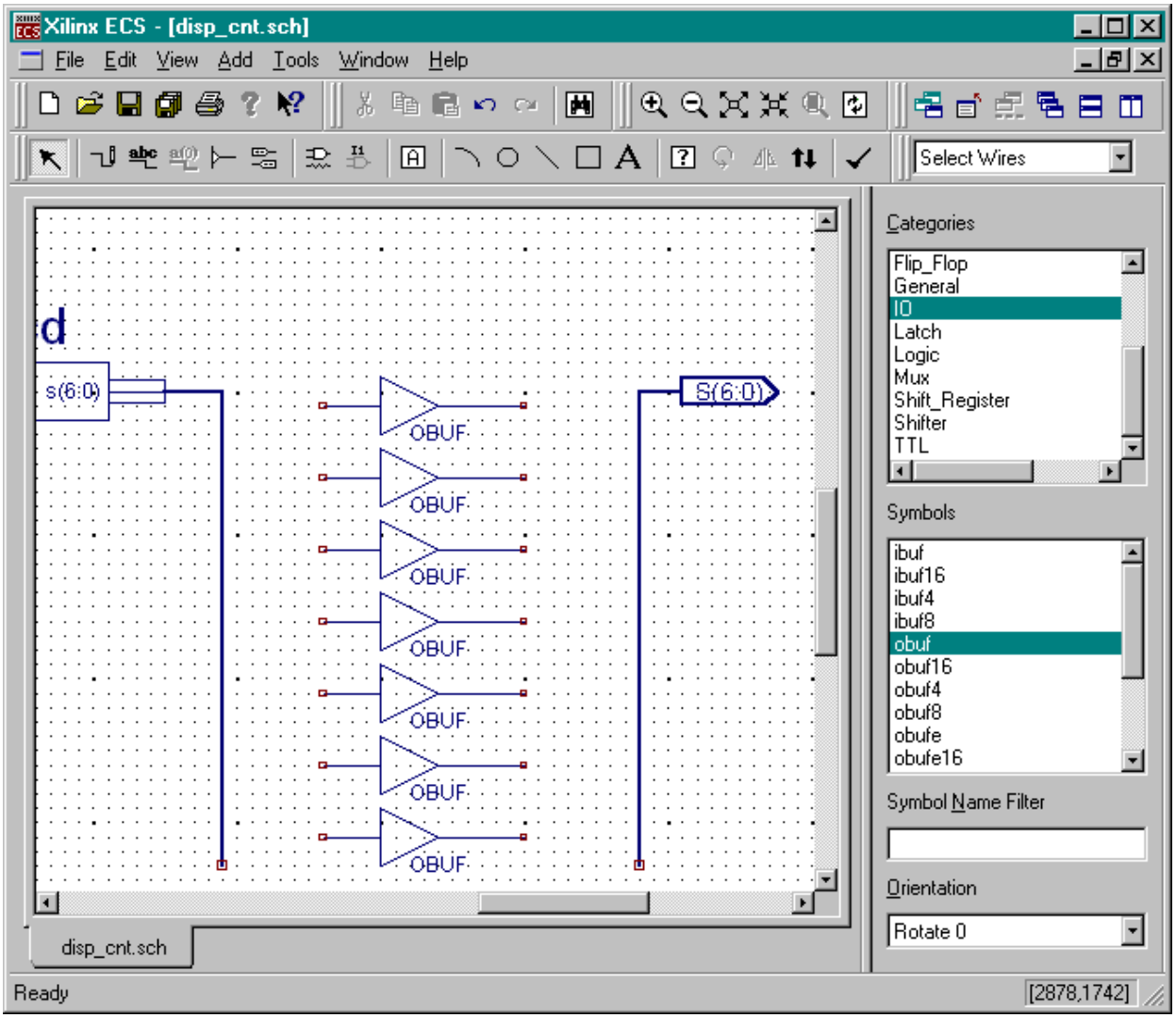
At this point it is a good idea to rename the bus connected to the LED decoder output so it has a less cumbersome name. Double-click on the bus and the **Object Properties** window will appear.



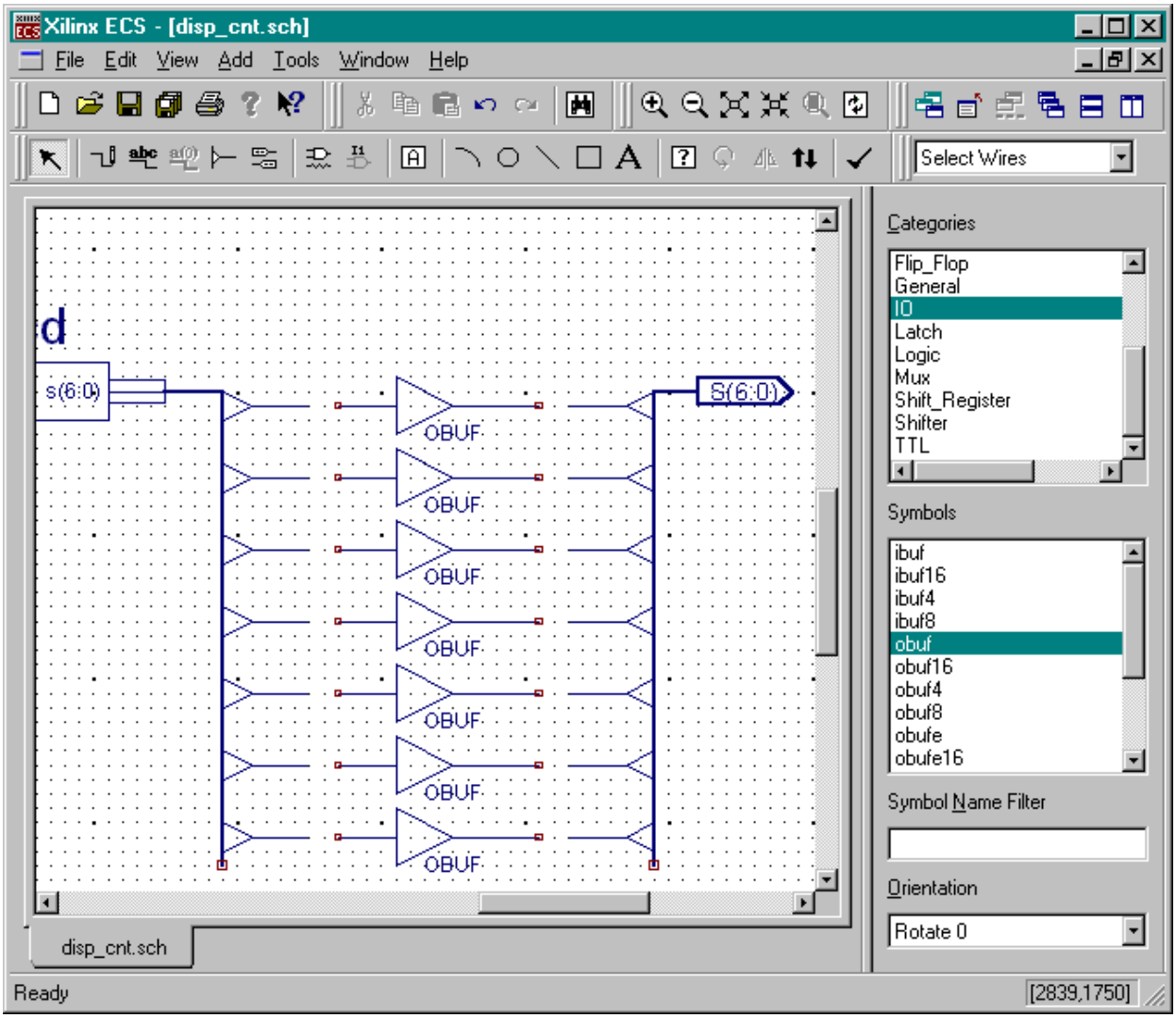
Replace the automatically-assigned name for the seven-bit bus, **XLXN_11(6:0)**, with the bus name **A(6:0)**. Then click on the OK button.



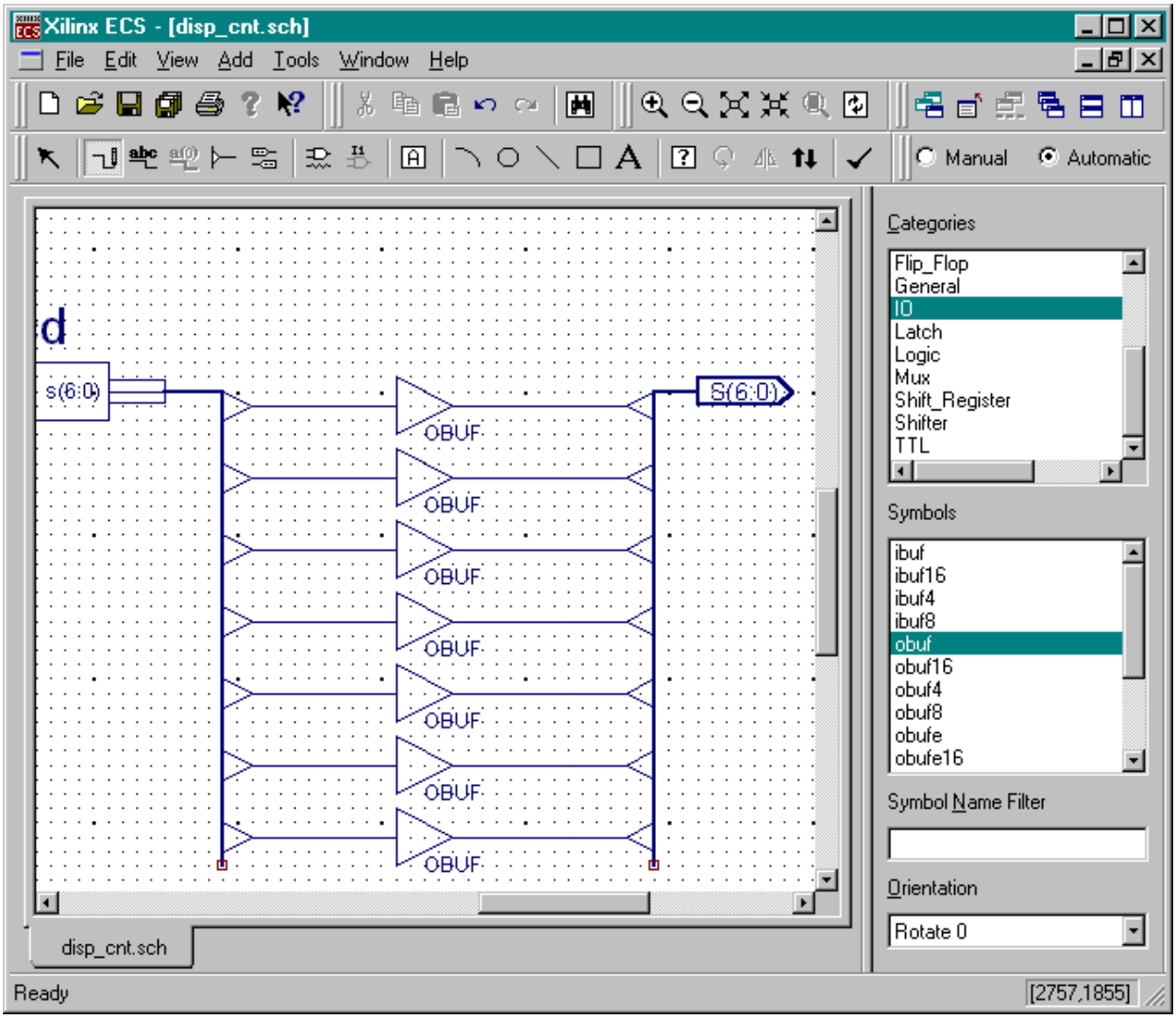
Next select a single-bit output buffer, obuf, from the list of symbols and drop seven of them into the schematic drawing area.



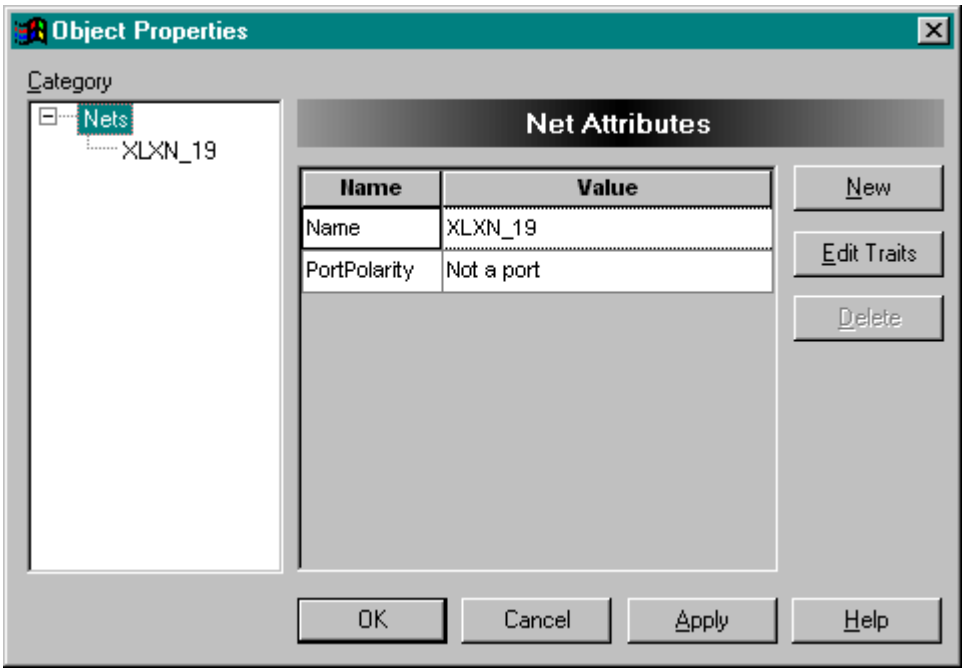
Now the output buffers have to be attached to the buses. Click on the button and attach seven bus taps to each bus as shown below. (Use the rotation button to rotate the bus tap symbol.)



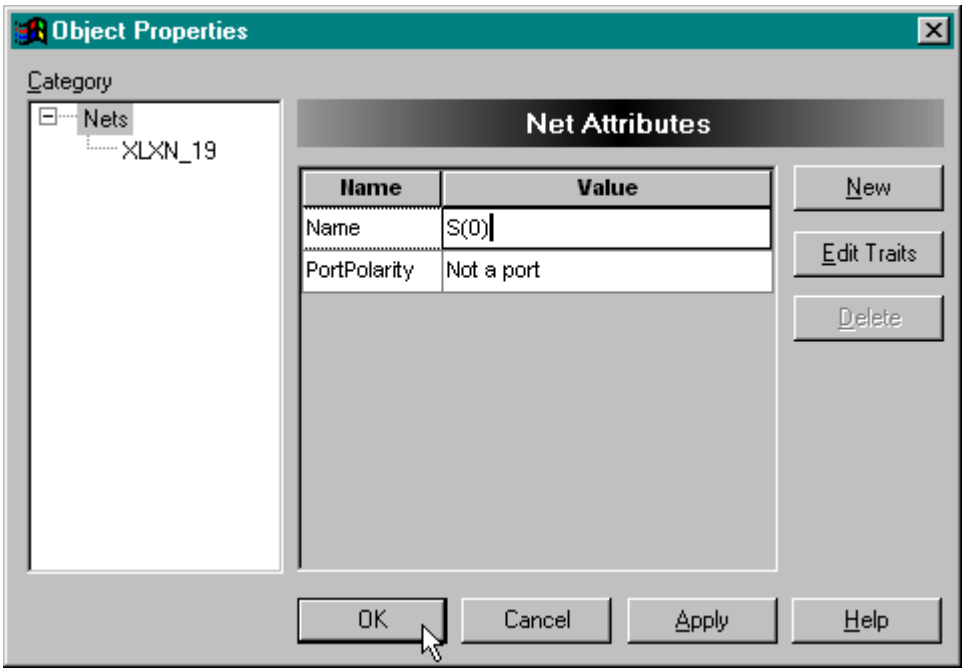
Once all the bus taps are in place, attach the output buffers to the taps using wire segments.



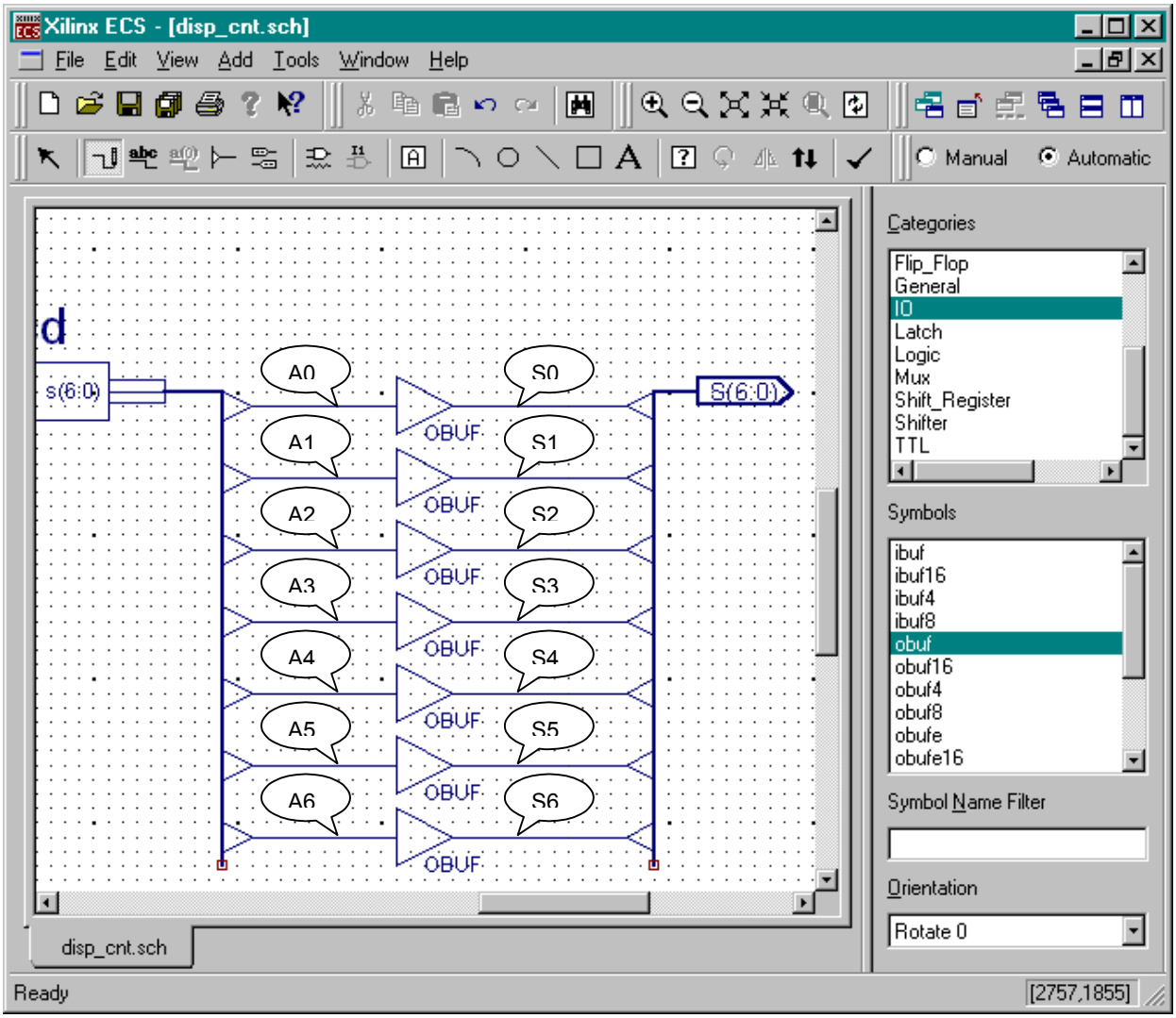
Now the question becomes: “How do we know each output buffer is attached to the right LED decoder output and output pin?” The answer is: “We don’t!” We have to manually set the connections to the buses to make sure they are correct. Double-click on the wire segment connecting the output buffer to the S(6:0) output bus. (Make sure you double-click the wire segment and not the bus tap symbol or the OBUF symbol.) The **Object Properties** window will appear with the name for the wire segment that was automatically assigned by the schematic editor.



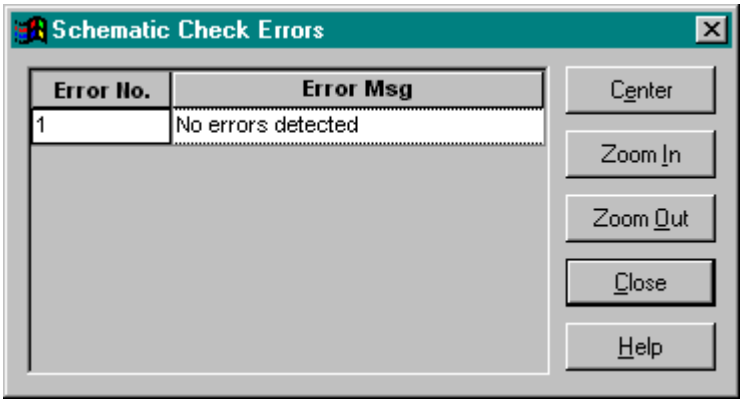
Change the name of the wire segment to S(0) which is the least-significant bit of the output bus. Then click on OK.



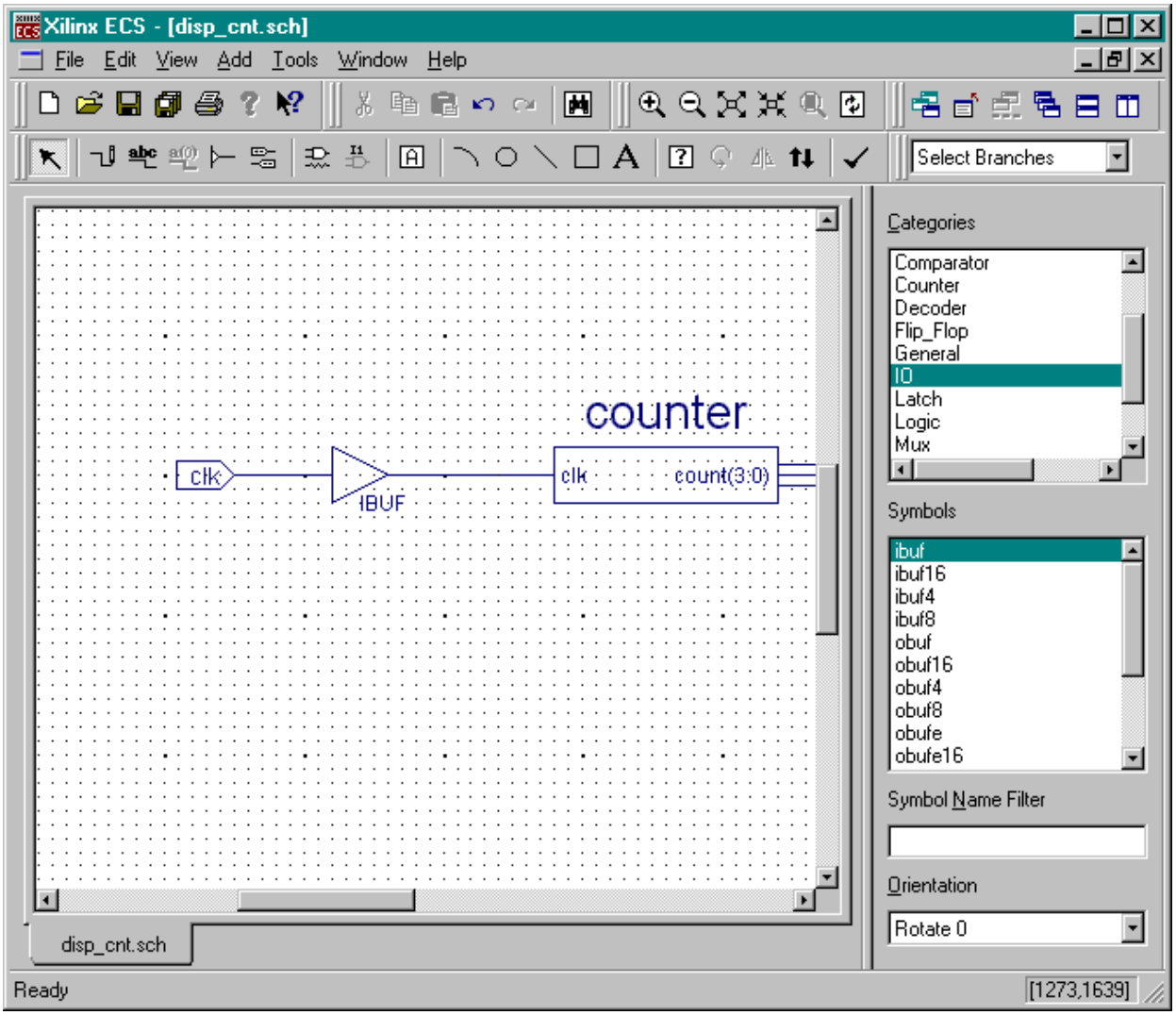
Repeat the process to rename each wire segment as shown below. (The visible labels for each wire segment were added afterward. The wire segment labels will not be shown by the schematic editor.)



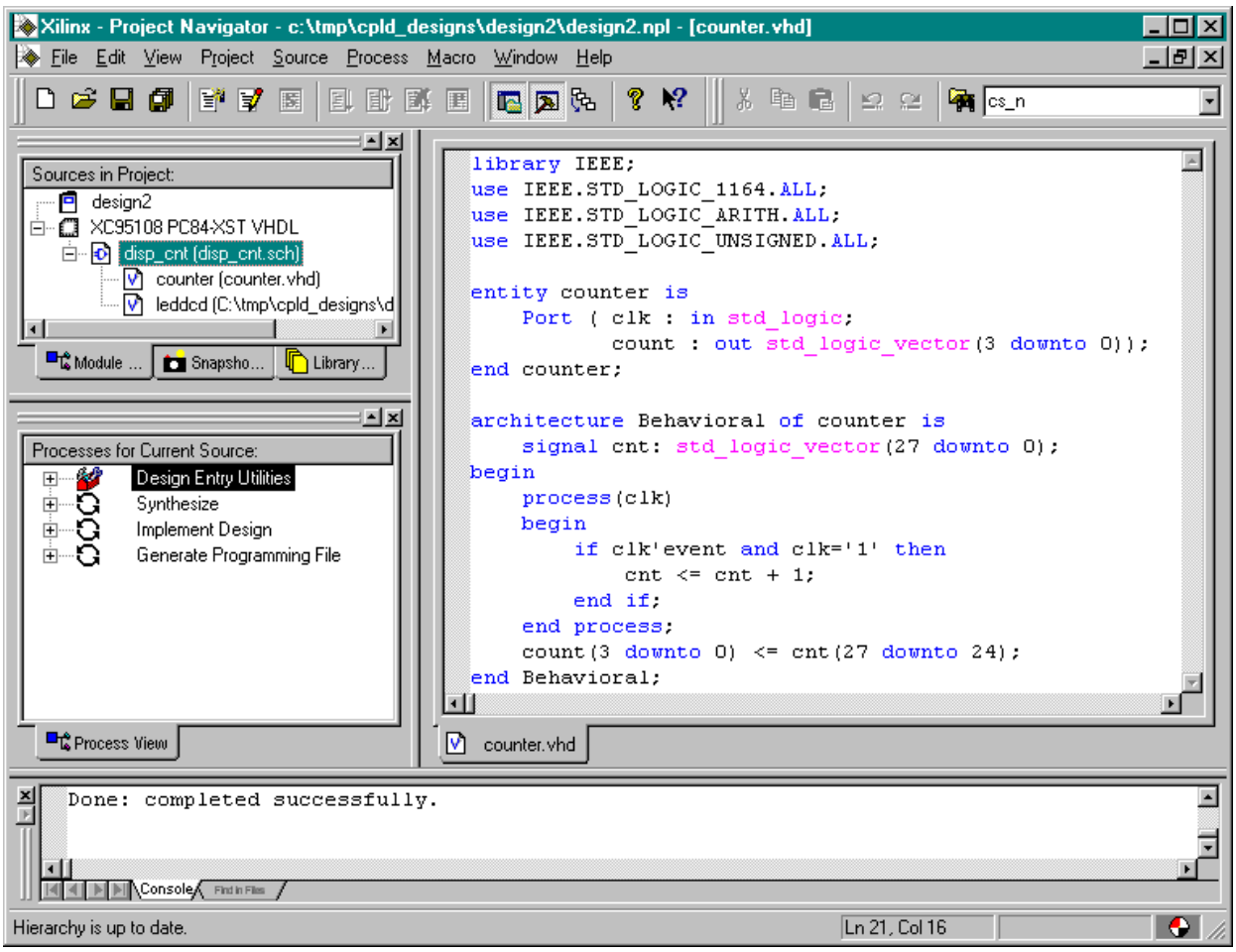
Now when we click on the schematic check button,  , we see the errors have been corrected.



Once the outputs from the circuit are in place, we can create the analogous circuitry for the input. We connect a single input buffer module to the clock input of the counter and then we connect a single input I/O marker to the IBUF symbol. After this, perform another schematic check to detect any errors, save the schematic using the File➔Save command and then close the schematic editor.

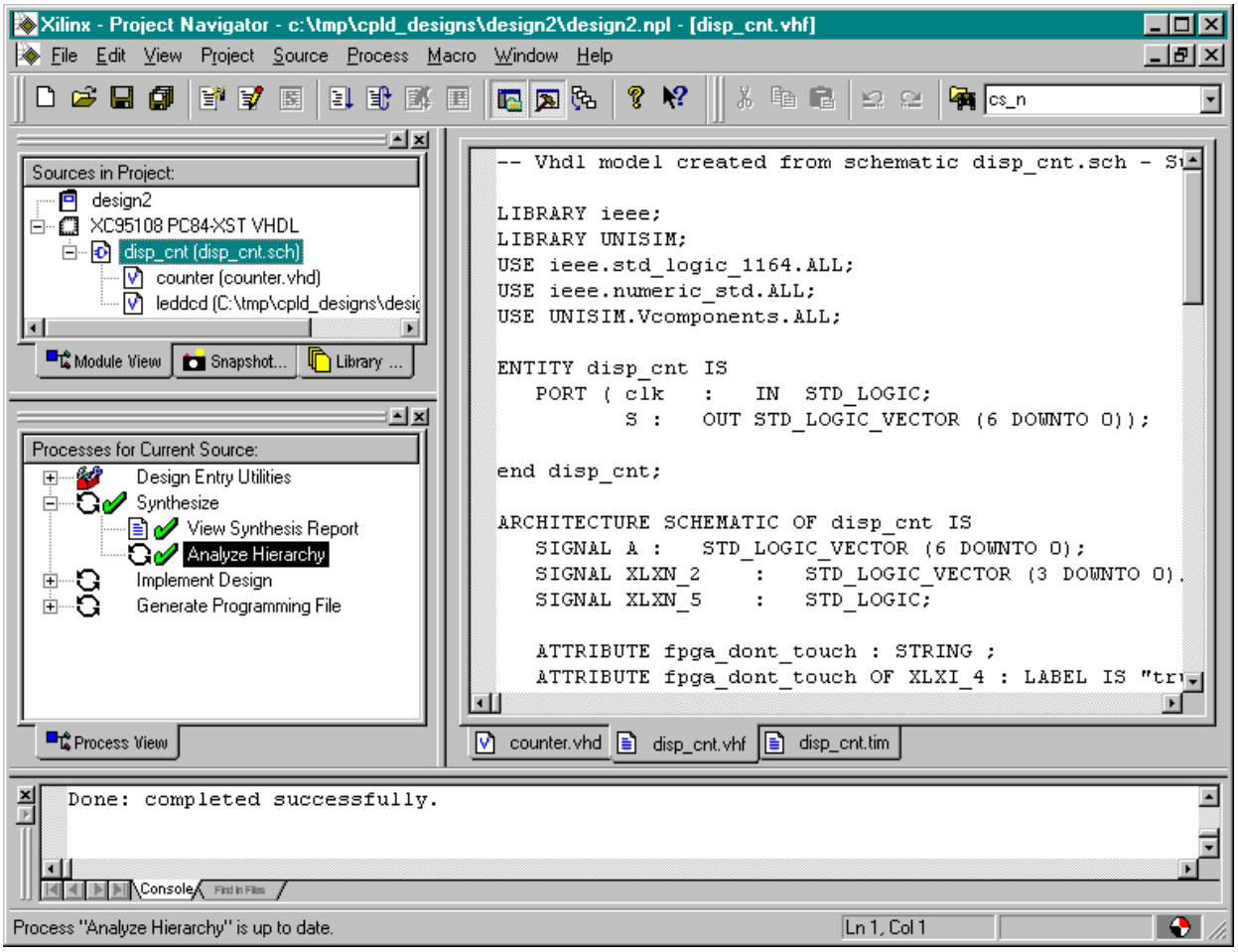


Once we save the schematic for the top-level module, we see the updated hierarchy in the Sources pane of the **Project Navigator** window. Now the **counter** and **leddcd** modules are shown as lower-level modules that are included within the top-level **disp_cnt** module.



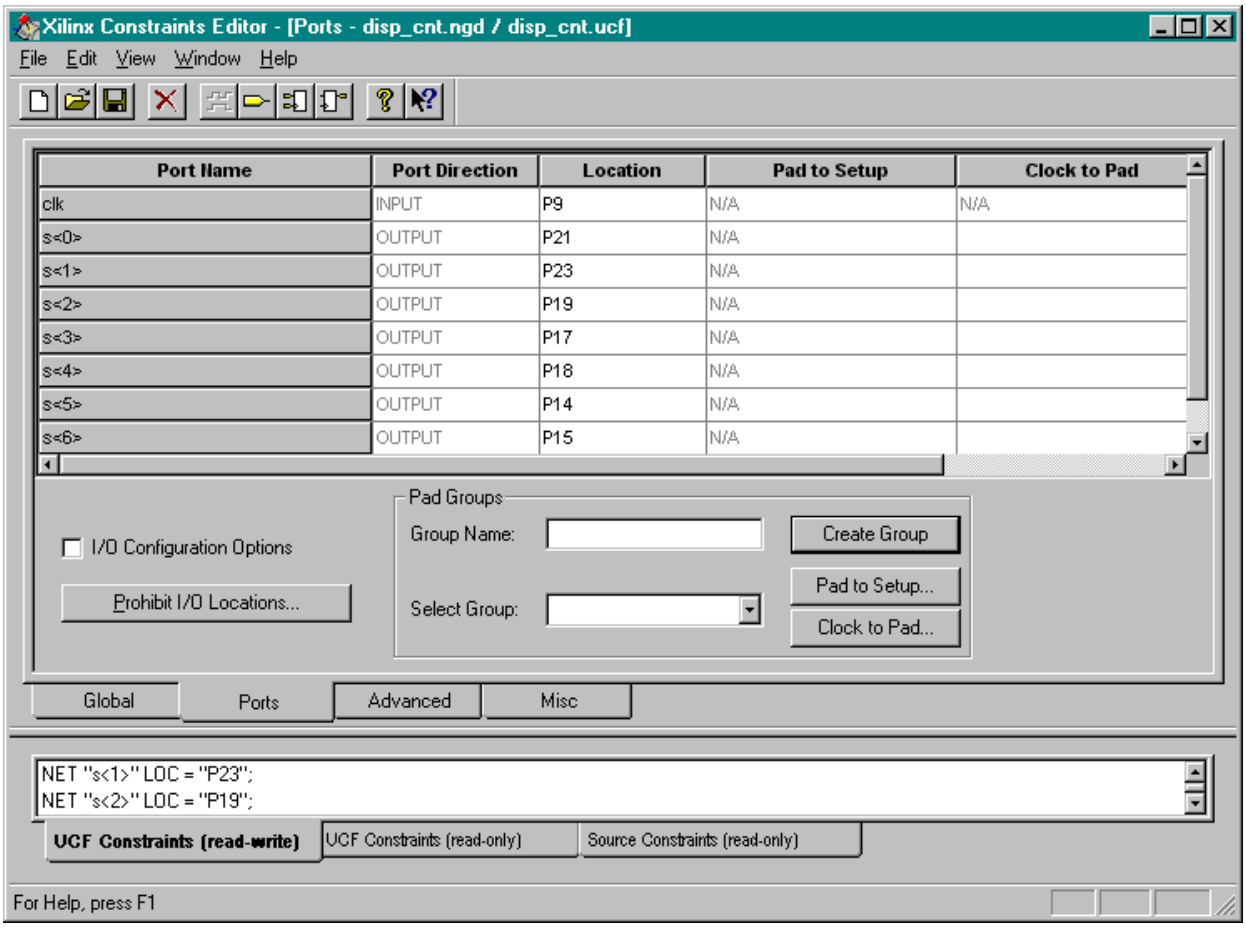
Checking the VHDL Syntax

We can check the entire design by highlighting the disp_cnt object in the Sources pane and then double-clicking the Analyze Hierarchy process. This checks the VHDL for each module and their interconnections with each other. The ✔ that appears after the Analyze Hierarchy process completes shows we have no syntax problems in our modules.



Constraining the Design

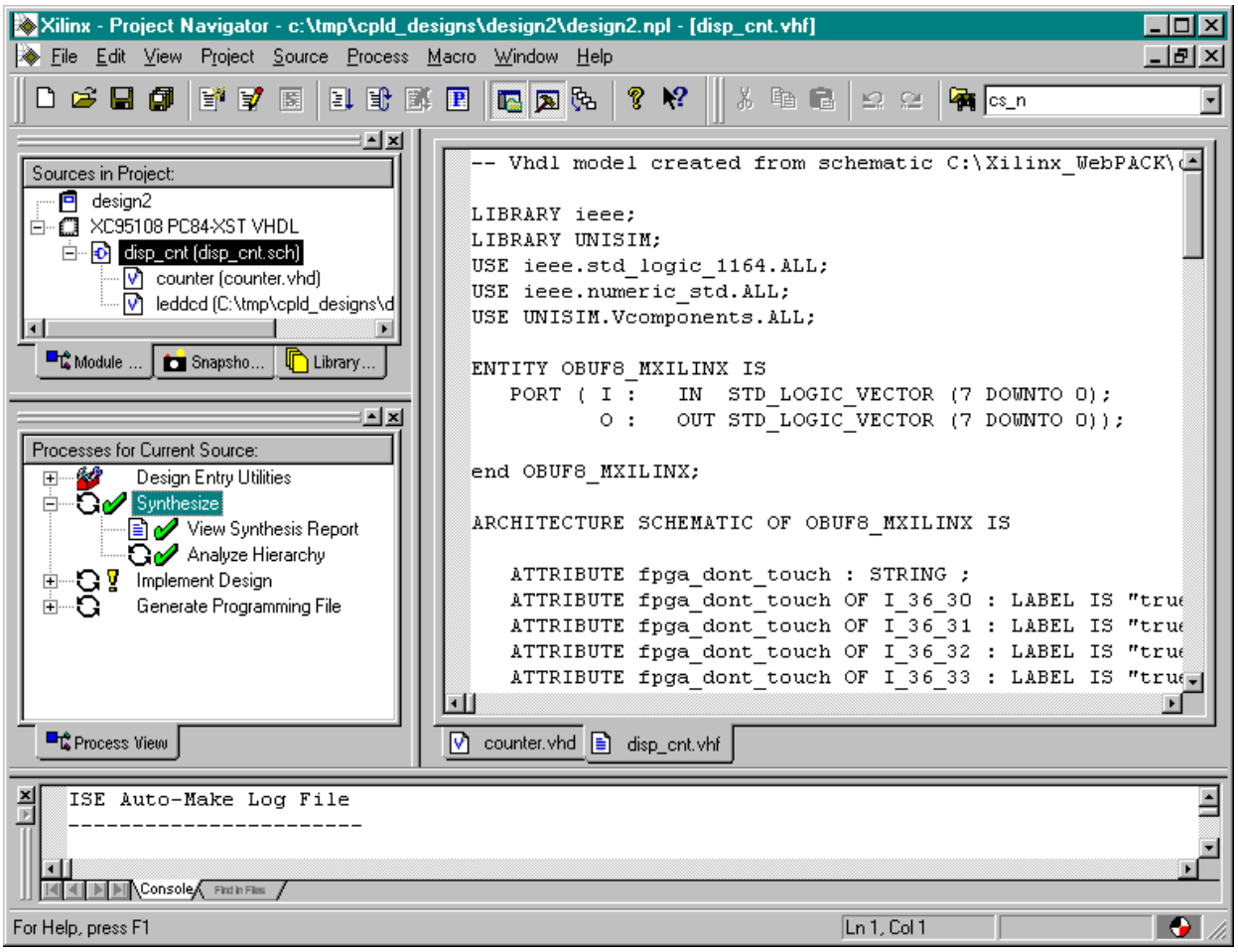
Before synthesizing the displayable counter, we need to assign the pins which the inputs and outputs will use. Highlight the disp_cnt object in the Sources pane and then double-click the Edit Implementation Constraints (Constraints Editor) process. In the Ports tab of the **Constraint Editor** window that appears, set the pin assignments for the clock input and LED segment drivers as follows:



Assigning the **clk** input to pin P9 lets us use the onboard oscillator of the XS95 Board to drive the counter. The output assignments connect the displayable counter to the seven-segment LED on the XS95 Board as in the previous design example.

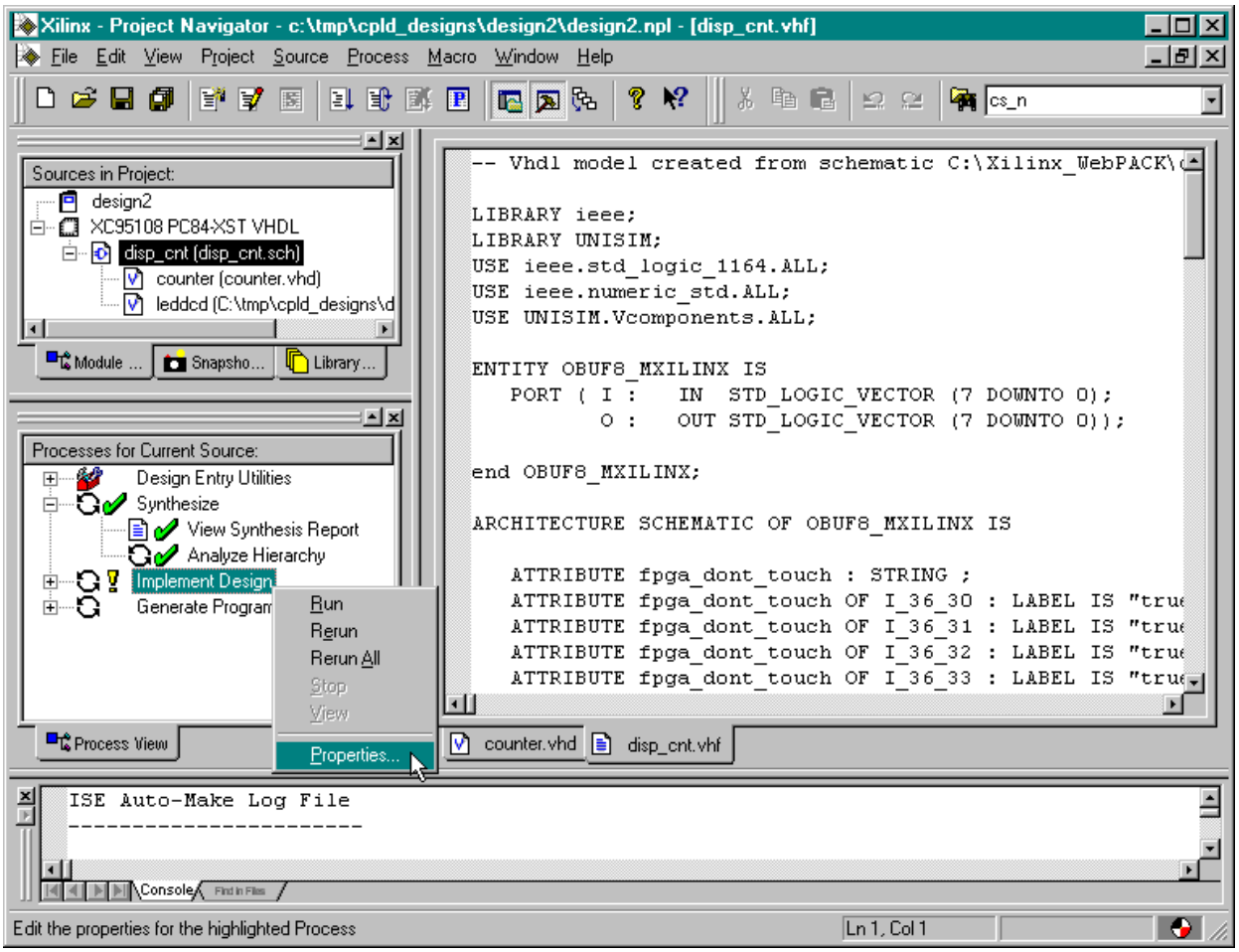
Synthesizing the Logic Circuitry for the Design

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.

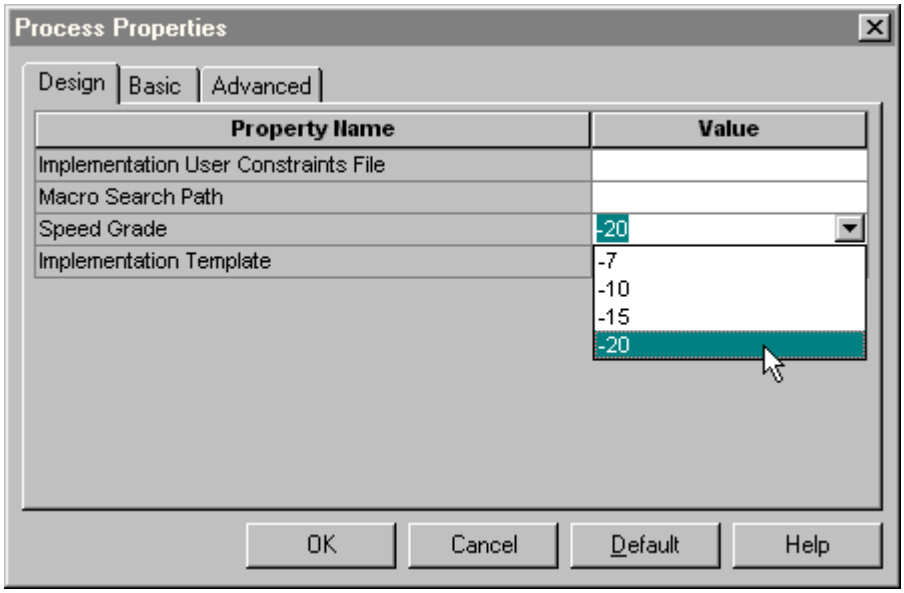


Fitting the Logic Circuitry Into the CPLD

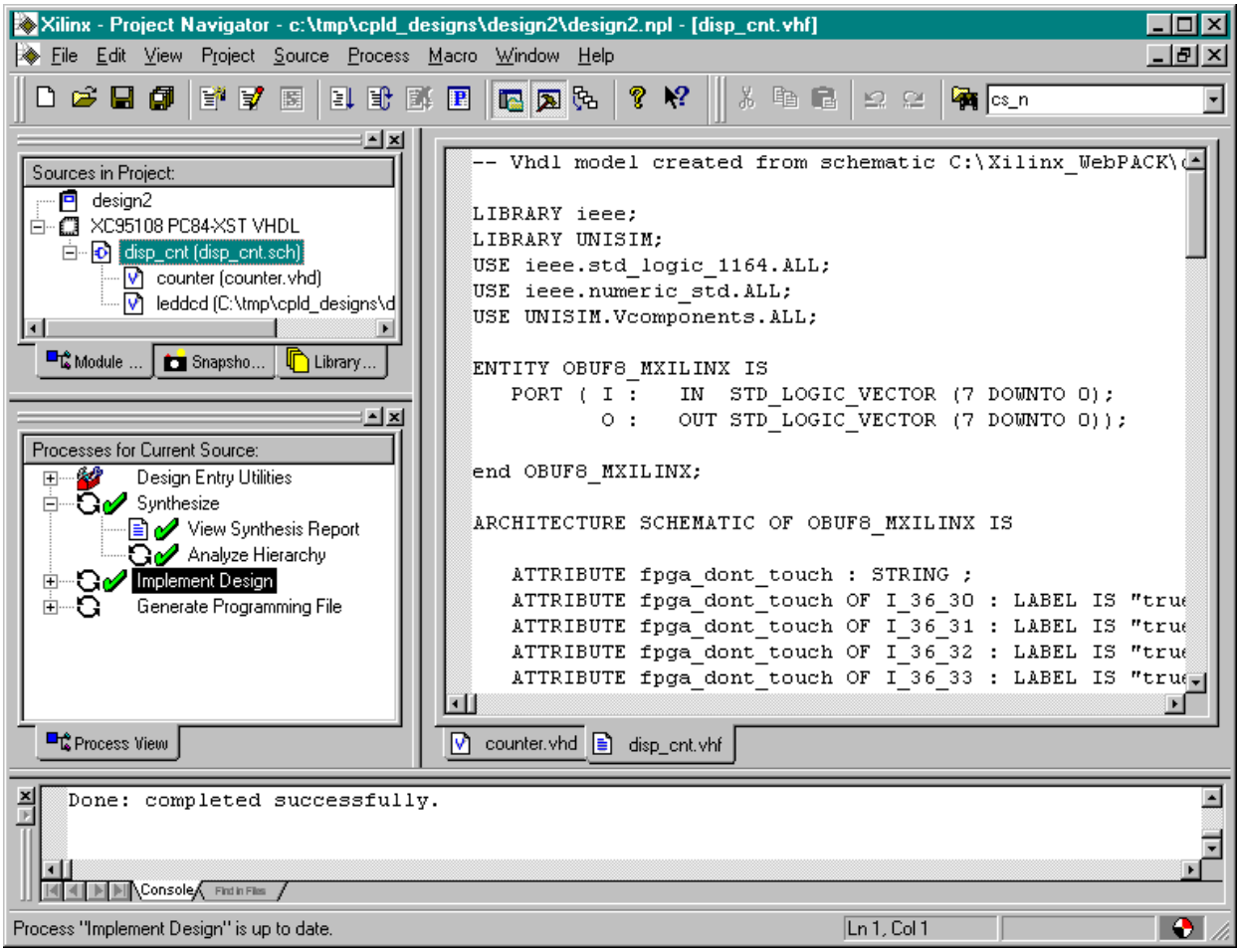
Once the netlist is synthesized, we can begin the process of fitting it into the CPLD. Before activating the fitting process, however, we will give the fitter some information on the speed of the CPLD we are targeting. The XC95108 on the XS95 Board has a -20 speed grade which means that the pad-to-pad delay through a single macrocell is 20 ns. To set the device speed, right-click on the Implement Design process and select the Properties item in the pop-up menu.



In the **Process Properties** window that appears, we set the speed to -20 in the Speed Grade field of the Design tab. Then we click on OK. There are many other parameters we can adjust to affect the fitting process, but we don't need to alter any of them from their default values for this design. (We would probably adjust these parameters if we were pushing the CPLD to the limit in terms of logic density or operating speed.)



Once the speed grade of the CPLD is set, we can double-click on the Implement Design process to initiate the fitting process. The ✔ that appears indicates that the fitting process was successful.



Checking the Fit

After the fitting process is done, we can check the logic utilization by double-clicking on the Fitter Report process. At the top of the file we find:

```
cpldfit:  version E.30                                Xilinx Inc.
                                   Fitter Report
Design Name: disp_cnt                              Date: 10-21-2001, 11:49AM
Device Used: XC95108-20-PC84
Fitting Status: Successful

****************************  Resource Summary  ****************************

Macrocells      Product Terms    Registers       Pins            Function Block
Used            Used             Used            Used            Inputs Used
35 /108 ( 32%) 52  /540  (  9%) 28 /108 ( 25%) 8  /69  ( 11%) 48 /216 ( 22%)
```

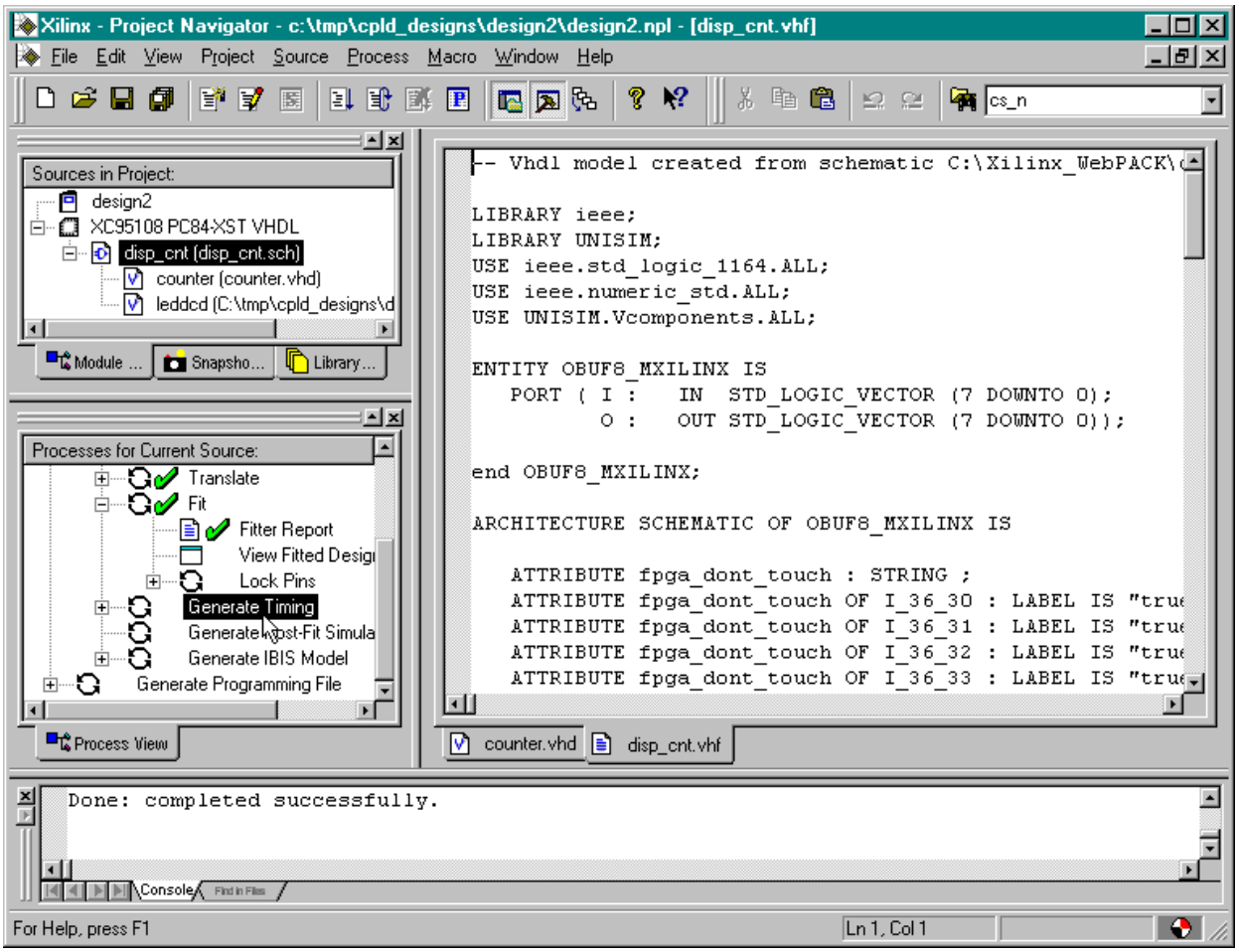
The displayable counter consumes 35 of the 108 macrocells: 28 for the four-bit counter and 7 for the LED decoder. Looking further, we find the pin assignments for the clock input and LED decoder outputs match the assignments we made in the Constraint Editor:

```
***************Resources Used by Successfully Mapped Logic******************

** LOGIC **
Signal                Total   Signals Loc      Pwr  Slew Pin  Pin       Pin
Name                  Pt      Used             Mode Rate #    Type      Use
N52                   1       24      FB3_18   STD            (b)       (b)
N53                   1       25      FB3_17   STD       31   I/O       (b)
N54                   1       26      FB3_16   STD       26   I/O       (b)
N55                   1       27      FB3_15   STD       25   I/O       (b)
s<0>                  4       4       FB3_11   STD  FAST 21   I/O       O
s<1>                  3       4       FB3_12   STD  FAST 23   I/O       O
s<2>                  3       4       FB3_8    STD  FAST 19   I/O       O
s<3>                  2       4       FB3_5    STD  FAST 17   I/O       O
s<4>                  4       4       FB3_6    STD  FAST 18   I/O       O
s<5>                  4       4       FB3_2    STD  FAST 14   I/O       O
s<6>                  4       4       FB3_3    STD  FAST 15   I/O       O
xlxi_1/cnt_0          1       1       FB2_18   STD            (b)       (b)
xlxi_1/cnt_1          1       1       FB2_17   STD       84   I/O       (b)
xlxi_1/cnt_10         1       10      FB1_18   STD            (b)       (b)
xlxi_1/cnt_11         1       11      FB1_17   STD       13   I/O       (b)
xlxi_1/cnt_12         1       12      FB1_16   STD       12   GCK/I/O   (b)
xlxi_1/cnt_13         1       13      FB1_15   STD       11   I/O       (b)
xlxi_1/cnt_14         1       14      FB1_14   STD       10   GCK/I/O   (b)
xlxi_1/cnt_15         1       15      FB1_13   STD            (b)       (b)
xlxi_1/cnt_16         1       16      FB1_12   STD       9    GCK/I/O   GCK
xlxi_1/cnt_17         1       17      FB1_11   STD       7    I/O       (b)
xlxi_1/cnt_18         1       18      FB1_10   STD            (b)       (b)
xlxi_1/cnt_19         1       19      FB3_14   STD       24   I/O       (b)
xlxi_1/cnt_2          1       2       FB2_16   STD       83   I/O       (b)
xlxi_1/cnt_20         1       20      FB3_13   STD            (b)       (b)
xlxi_1/cnt_21         1       21      FB3_10   STD            (b)       (b)
xlxi_1/cnt_22         1       22      FB3_9    STD       20   I/O       (b)
xlxi_1/cnt_23         1       23      FB3_7    STD            (b)       (b)
xlxi_1/cnt_3          1       3       FB1_9    STD       6    I/O       (b)
xlxi_1/cnt_4          1       4       FB1_8    STD       5    I/O       (b)
xlxi_1/cnt_5          1       5       FB1_7    STD            (b)       (b)
xlxi_1/cnt_6          1       6       FB1_6    STD       4    I/O       (b)
xlxi_1/cnt_7          1       7       FB1_5    STD       3    I/O       (b)
xlxi_1/cnt_8          1       8       FB1_4    STD            (b)       (b)
xlxi_1/cnt_9          1       9       FB1_3    STD       2    I/O       (b)

** INPUTS **
Signal                                Loc               Pin  Pin       Pin
Name                                                    #    Type      Use
clk                                   FB1_12            9    GCK/I/O   GCK

End of Resources Used by Successfully Mapped Logic
```

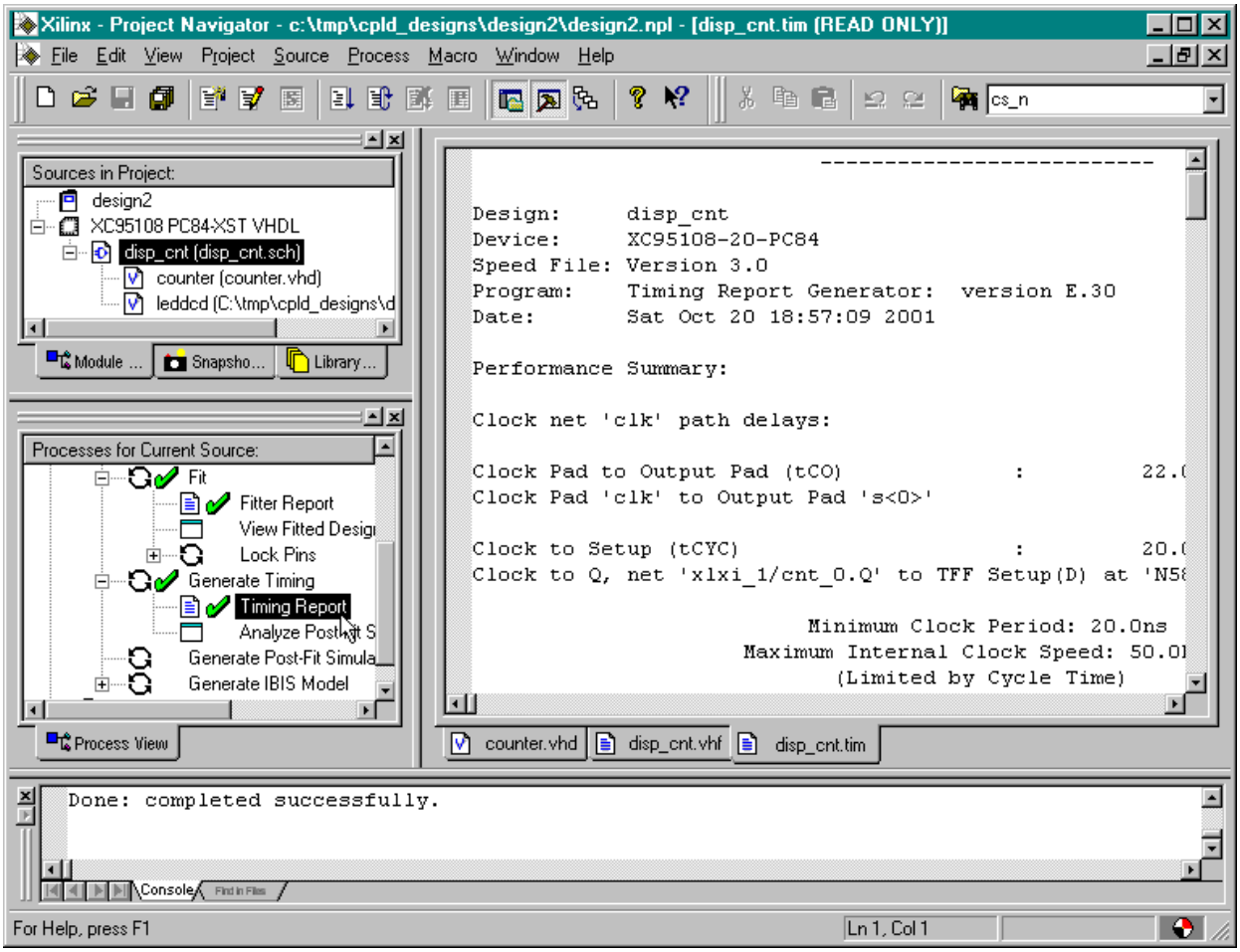
Also note that there are 28 signals in addition to the input and outputs. There are the upper four output bits from the counter (**N26**, **N27**, **N28**, and **N29**) and the twenty-four lower bits of the counter (**xlxi_1/cnt_***). They will not appear on the pins of the CPLD because their macrocells have been buried. In effect, a macrocell is buried when the output buffer from the macrocell to its associated I/O pin is placed in a high-impedance state.

Checking the Timing

We have the displayable counter synthesized and fitted to the XC95108 CPLD with the correct pin assignments. But how fast can we run the counter? To find out, double-click on the Generate Timing process.



After the static timing delays are calculated, double-click the Timing Report to view the results of the analysis.



The timing report contains the following information:

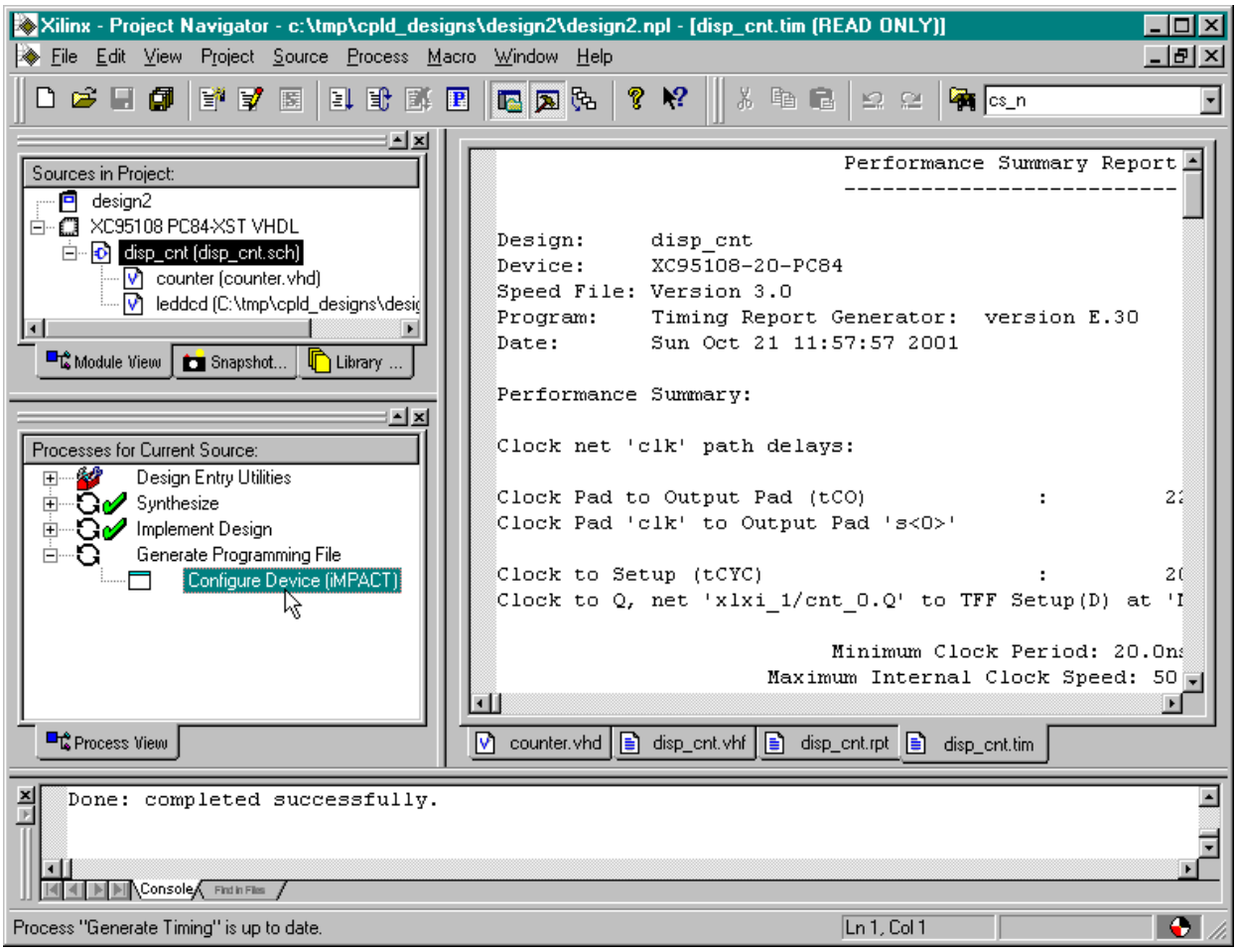
```
                          Performance Summary Report
                          ---------------------------

Design:      disp_cnt
Device:      XC95108-20-PC84
Speed File:  Version 3.0
Program:     Timing Report Generator:  version E.30
Date:        Sun Oct 21 11:51:18 2001

Performance Summary:

Clock net 'clk' path delays:

Clock Pad to Output Pad (tCO)              :         22.0ns (2 macrocell levels)
Clock Pad 'clk' to Output Pad 's<0>'                                        (GCK)

Clock to Setup (tCYC)                      :         20.0ns (1 macrocell levels)
Clock to Q, net 'xlxi_1/cnt_0.Q' to TFF Setup(D) at 'N52.D'                 (GCK)

                          Minimum Clock Period: 20.0ns
```

```
Maximum Internal Clock Speed: 50.0Mhz
(Limited by Cycle Time)
```

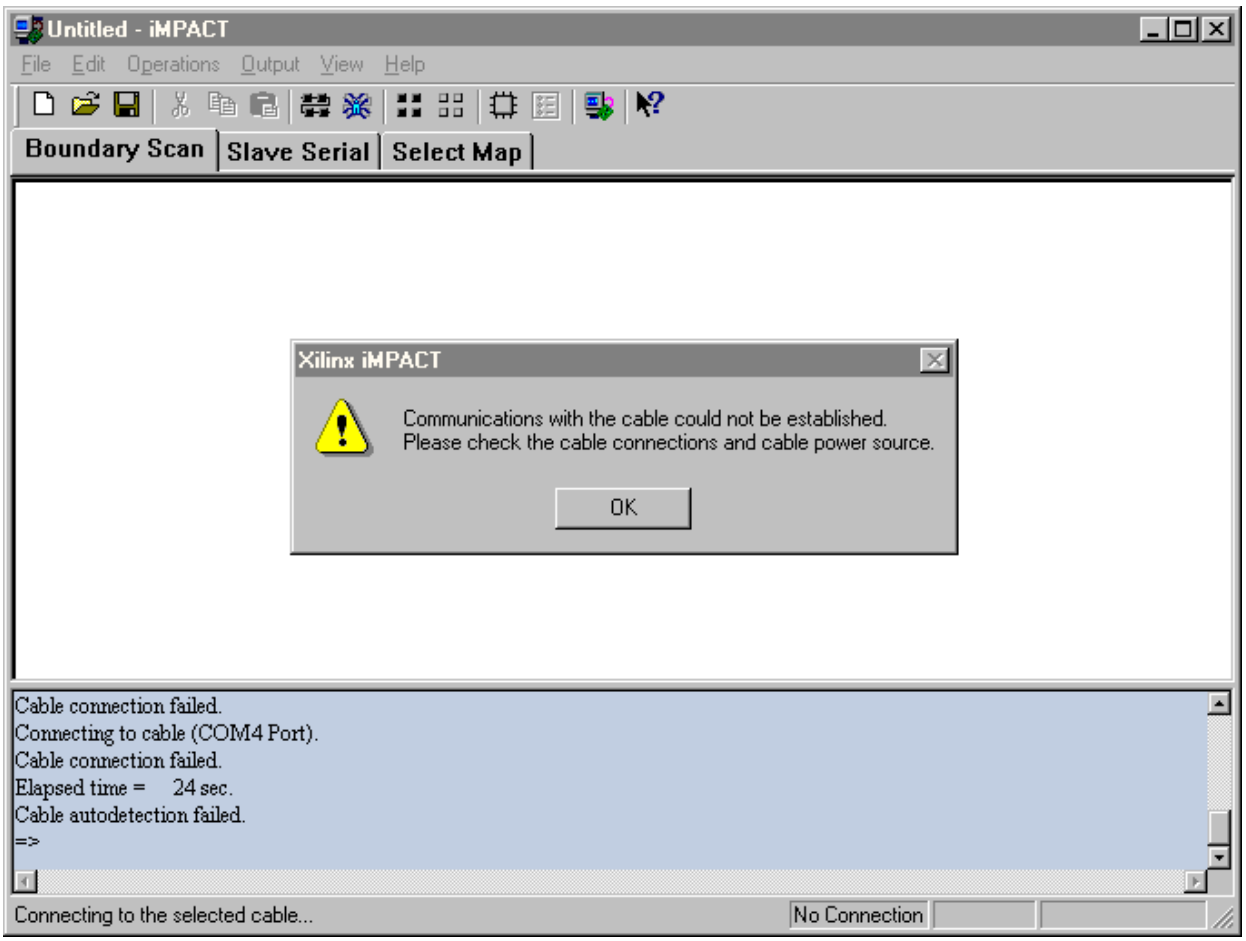
This table says that there is a 22.0 ns time delay between the rising edge on the clk input and a change in any one of the LED driver outputs. (This is the reason we changed the speed grade property in the fitter: so the reported delay would be accurate for our chip.) The minimum clock period is stated to be 20 ns so the design should run at 50 MHz.

Generating the Bitstream

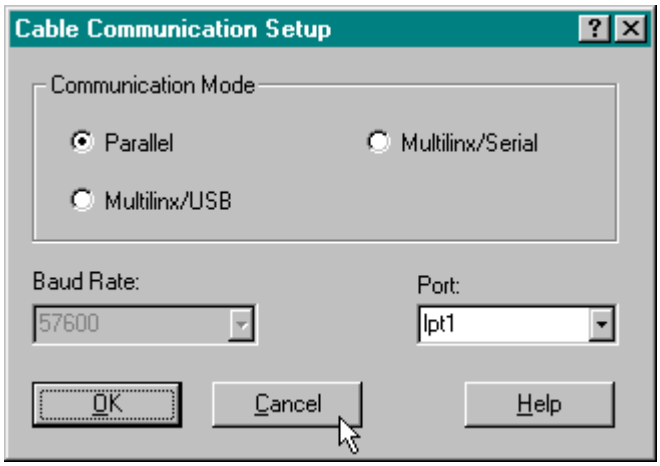
Now we are ready to generate the bitstream for the displayable counter. To initiate the programmer, we highlight the disp_cnt object in the Source pane and double-click on the Configure Device (iMPACT) process.



The **iMPACT** window will appear and once again it will try and fail to establish a connection with the CPLD through the various ports of the PC. As in the previous design example, just click on the OK button and proceed.

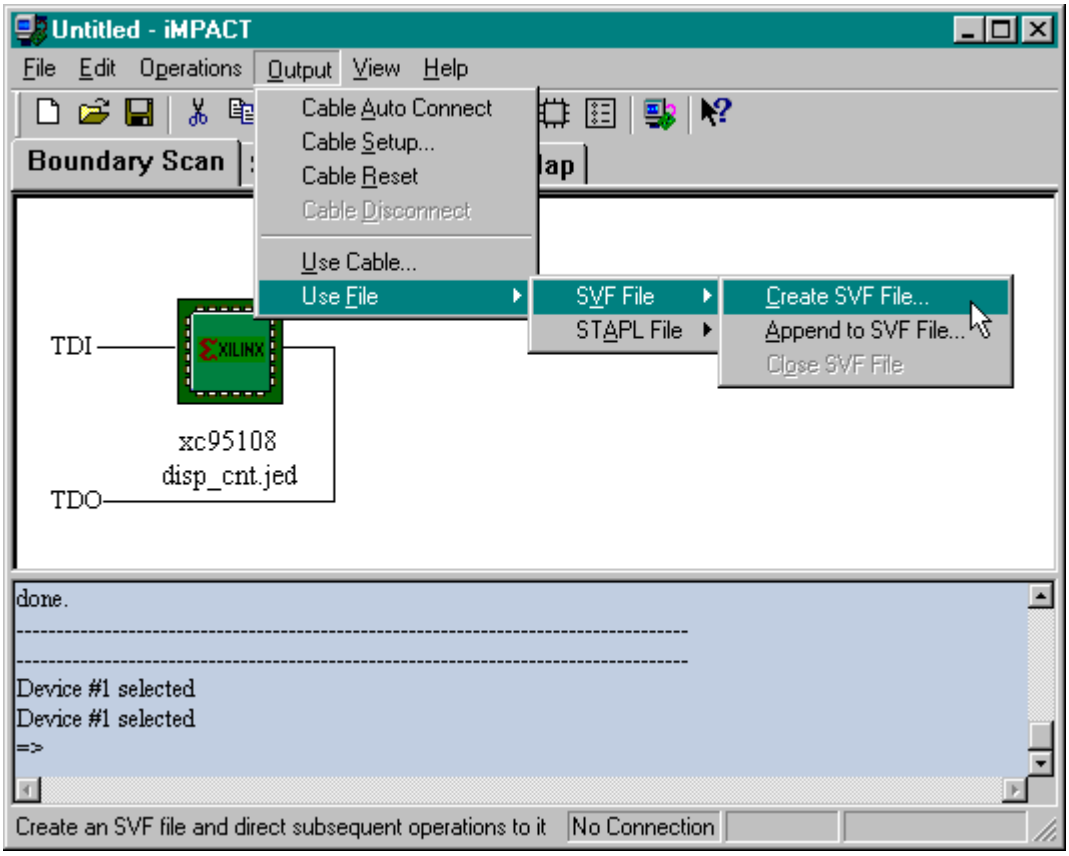


Click the Cancel button in the next window and proceed.

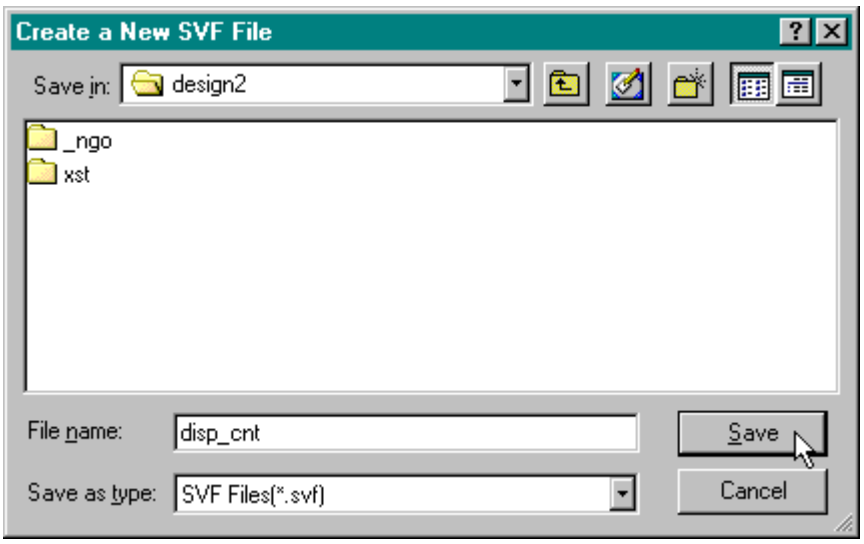


The **iMPACT** window now shows the JTAG chain of chips that are to be programmed. We only have one chip in our LED decoder design, so only one XC95108 CPLD is shown. Click on the

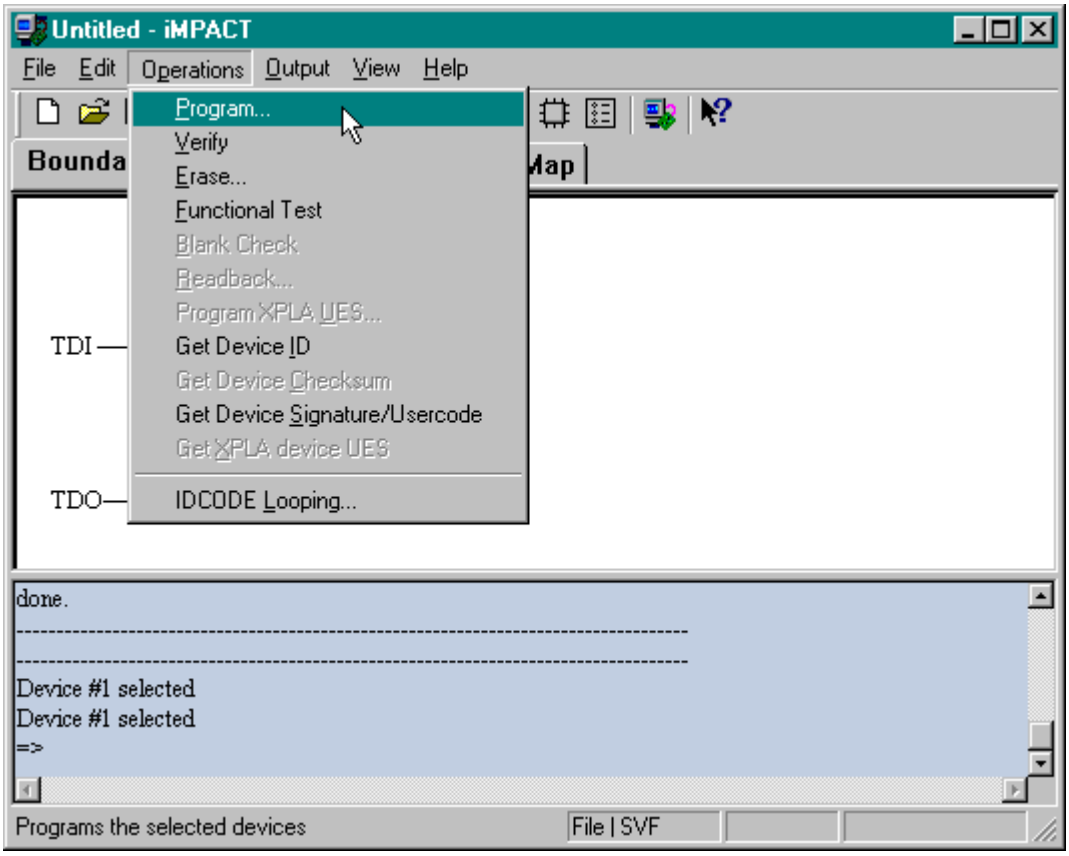
xc95108 icon to highlight it. Then continue by selecting an SVF file as the destination for the bitstream so we can use GXSLOAD for programming the CPLD in the XS95 Board.



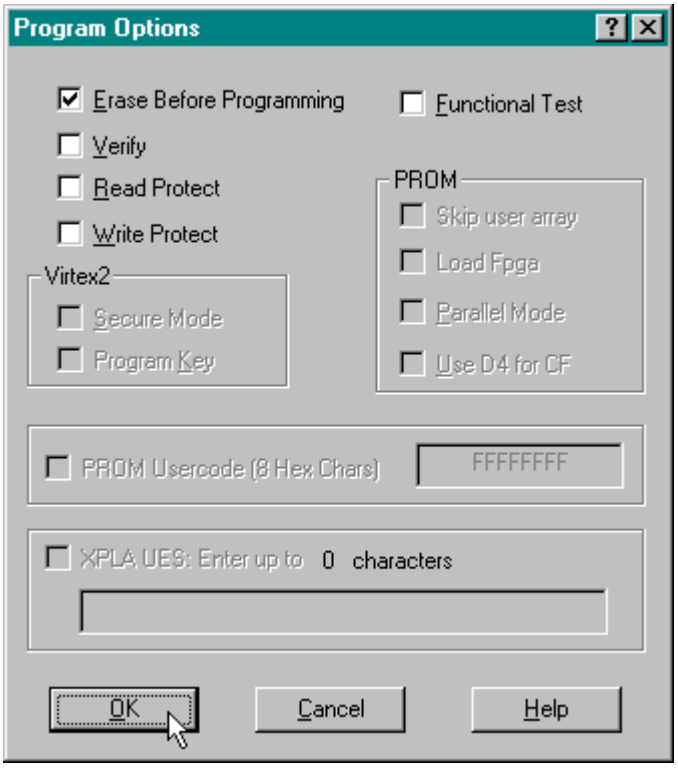
Now a window appears where we can enter the name for the file that will hold the bitstream. We can click on Save to accept the default name of disp_cnt.svf.



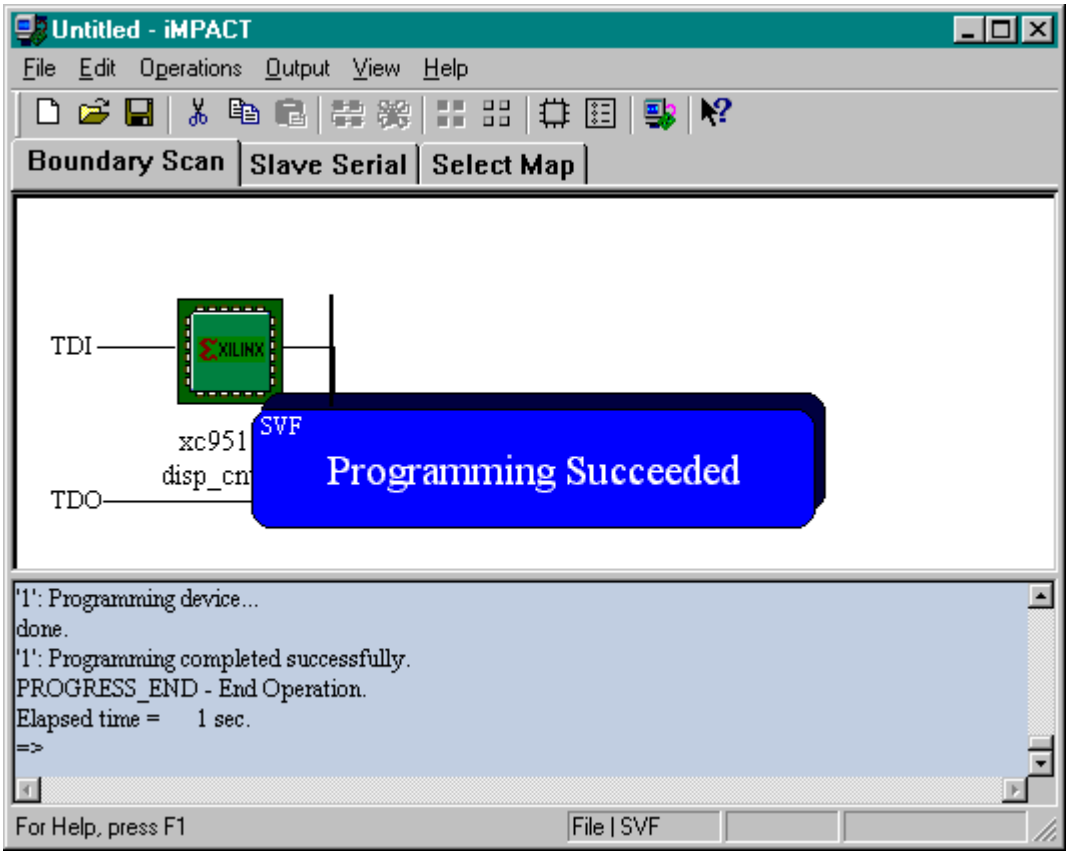
We then proceed to generate the bitstream by selecting the Operations➔Program menu item.



A **Program Options** window appears where we can set the Erase Before Programming option for the generated bitstream. Once we click OK in the **Program Options** window, the bitstream generation process begins.



The progress is reported in the lower pane of the **iMPACT** window:

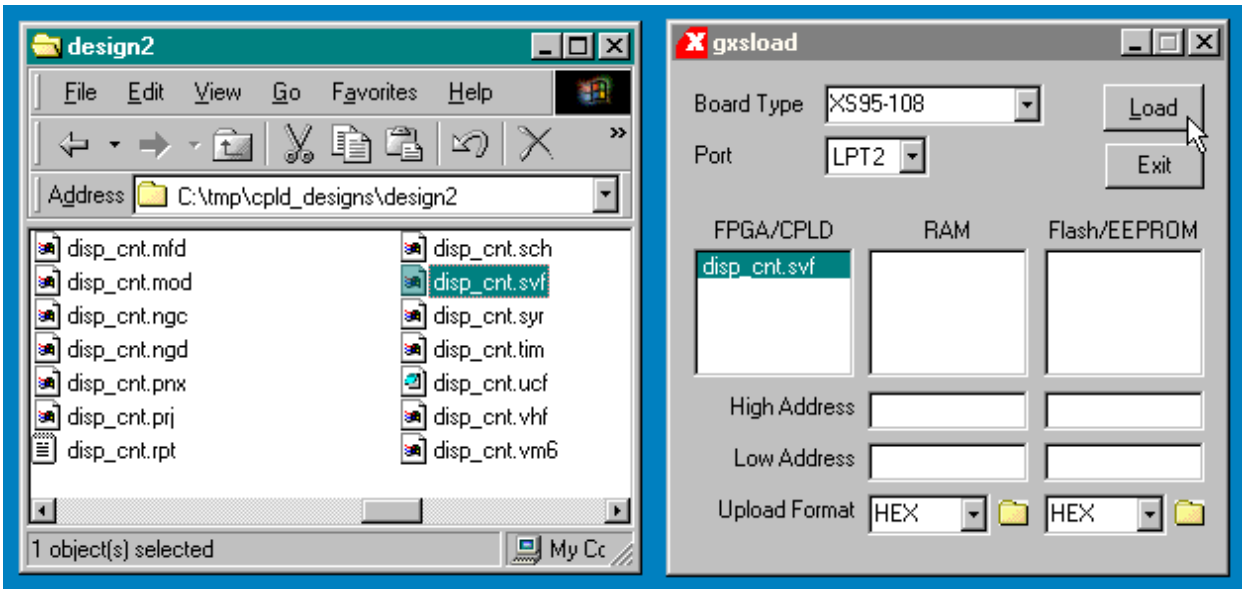


Once the bitstream file is generated, we click on File➔Exit to close the window.

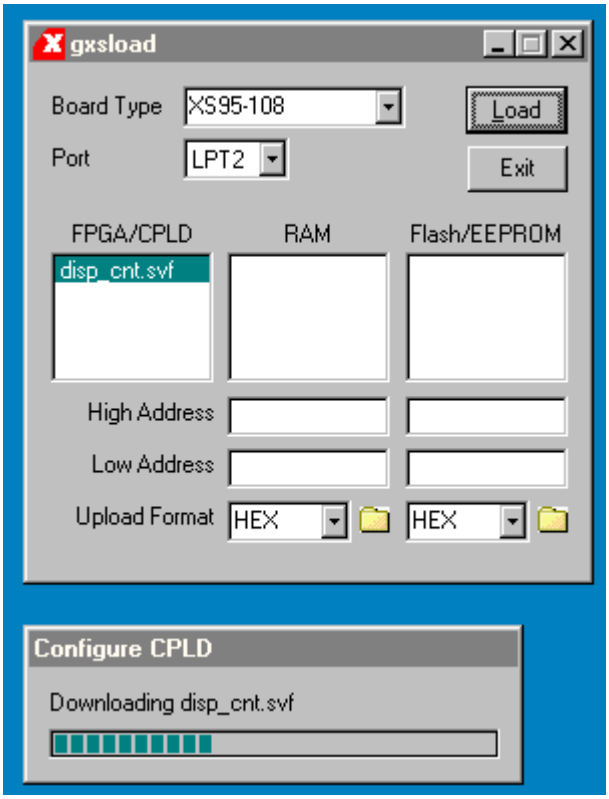
Downloading the Bitstream

Now we download the bitstream file into the CPLD of the XS95 Board. We double click the 

icon to bring up the **gxsload** window. Then we drag the disp_cnt.svf file from the C:\tmp\cpld_designs\design2 folder and drop it into the **gxsload** window.



Click the Load button and the XC95108 CPLD programming starts and completes in about a minute.



Testing the Circuit

Once the XC95108 CPLD on the XS95 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.