



©2000 BY XESS CORP.

---

# Introduction to WebPACK 3.1

---

Using XILINX WebPACK Software to Create  
CPLD Designs



All XS-prefix product designations are trademarks of XESS Corp.

All XC-prefix product designations are trademarks of Xilinx.

## Table of Contents

What This Is and <i>Is Not</i> .....	1
CPLD Programming .....	3
Installing WebPACK.....	5
Getting WebPACK.....	5
Installing WebPACK .....	6
Getting XSTOOLS.....	7
Installing XSTOOLS .....	7
Our First Design.....	8
An LED Decoder.....	8
Starting WebPACK Project Navigator.....	10
Naming Your Project.....	16
Describing Your Design With VHDL .....	18
Checking the VHDL Syntax .....	26
Fixing VHDL Errors .....	27
Synthesizing the Logic circuitry for Your Design.....	28
Fitting the Logic Circuitry Into the CPLD .....	29
Checking the Fit.....	31
Constraining the Fit.....	33
Viewing the Chip.....	39
Generating the Bitstream .....	45
Downloading the Bitstream.....	51
Testing the Circuit.....	53
Hierarchical Design.....	54
A Displayable Counter .....	54

Starting a New Design .....	55
Adding the LED Decoder .....	57
Adding a Counter .....	60
Tying Them Together .....	66
Checking the VHDL Syntax .....	70
Constraining the Design.....	72
Synthesizing the Logic Circuitry for the Design.....	73
Fitting the Logic Circuitry Into the CPLD .....	74
Checking the Fit.....	76
Checking the Timing .....	78
Generating the Bitstream .....	80
Downloading the Bitstream .....	84
Testing the Circuit.....	85
Going Further.....	86

# 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-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 enter a VHDL design.
- How to detect and fix VHDL syntactical errors.
- How to synthesize a netlist from a VHDL 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 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

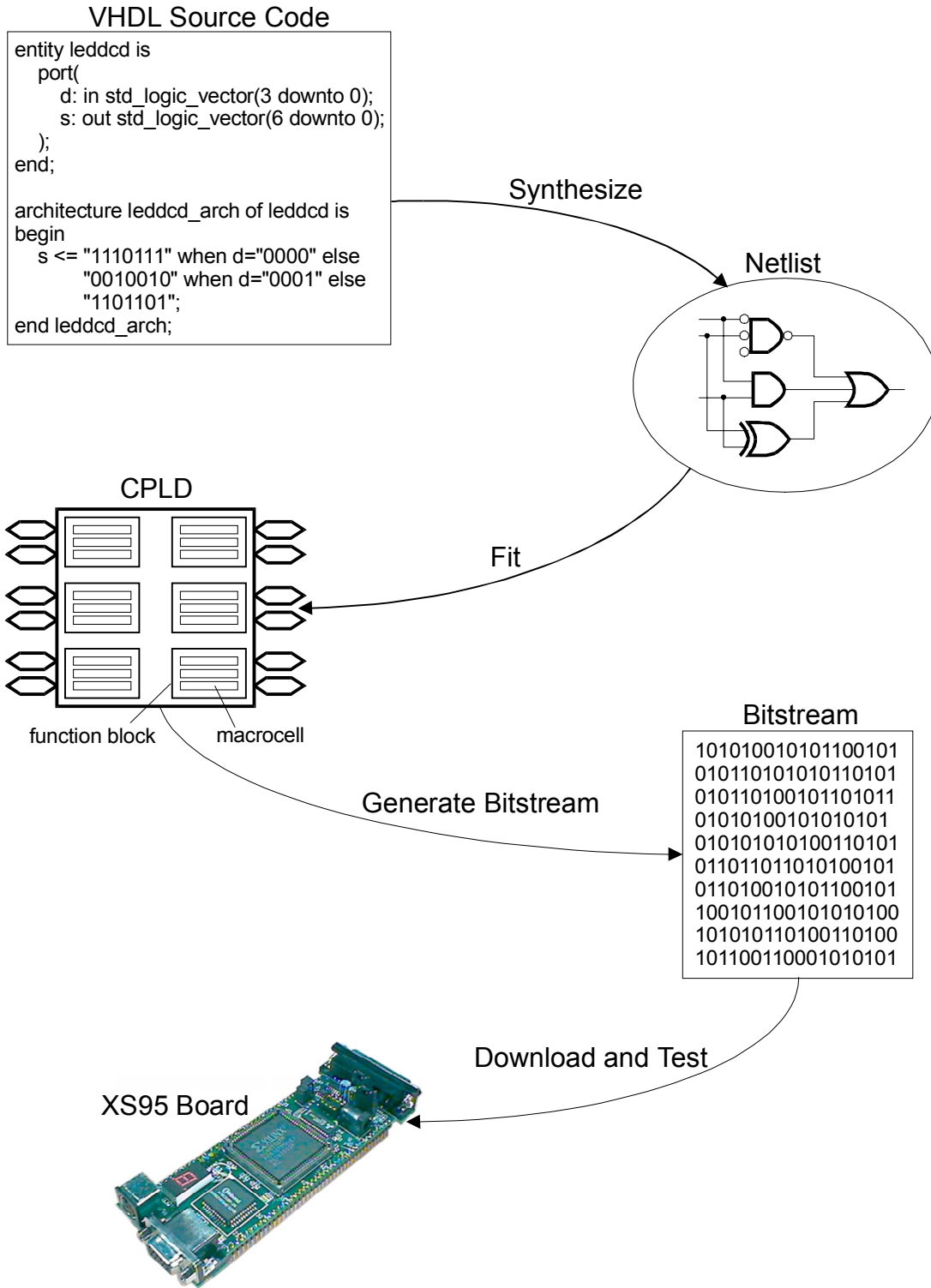
---

## 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, Verilog, or ABEL.
2. You use a *logic synthesizer* program to transform the HDL 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.

That's really all there is to it. The Xilinx WebPACK provides the HDL editor, 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.



# 2

---

# Installing WebPACK

## Getting WebPACK

You can get the WebPACK from <http://www.xilinx.com/sxpresso/webpack.htm>. You will have to register before you can download the WebPACK modules. There are three downloadable modules which form the core of the WebPACK tools:

**Design Entry & Synthesis Tools Module:** This module provides the HDL editor, schematic editor, and logic synthesizer for VHDL, Verilog, and ABEL. The software from module version 3.1WP1.0, released July 24, 2000 was used for this tutorial. The file for the module is [webpack\\_hdl\\_abel.exe](#). The file size is 14.4 MBytes.

**XC9500 Series Fitter Module:** This module provides the fitter for the XC9500 CPLDs. The software from module version 3.1WP1.0, released July 24, 2000 was used for this tutorial. The file for the module is [webpack\\_cpld\\_fitter.exe](#). The file size if 13.2 MBytes.

**JTAG Programmer Module:** This module provides the bitstream generator for the XC9500 CPLDs. The software from module version 3.1WP1.0, released July 24, 2000 was used for this tutorial. The file for the module is [webpack\\_programmer.exe](#). The file size if 10.7 MBytes.

Five more *backPACK* tools provide additional features you may find useful:

**ChipViewer:** This module provides a graphical view of how your logic circuitry fits within the CPLD. The software from module version 3.1WP1.0, released July 24, 2000, was used for this tutorial. The file for the module is [webpack\\_chipviewer.exe](#). The file size is 9.9 MBytes.

**CPLD Capture Libraries:** This module provides libraries of commonly-used logic functions that are used when doing schematic-based logic designs. The libraries from module version 3.1WP1.0, released July

24, 2000, were used for this tutorial. The file for the module is [webpack\\_ecs\\_cpdlib.exe](#). The file size is 17.8 MBytes.

**StateCAD:** This module provides a graphical state machine editor. The software from module version 3.1WP1.0, released July 24, 2000, was used for this tutorial. The file for the module is [webpack\\_statecad.exe](#). The file size is 4.3 MBytes.

**HDL Benchner:** This module provides an HDL testbench generator. The software from module version 3.1WP1.0, released July 24, 2000, was used for this tutorial. The file for the module is [webpack\\_hdlbencher.exe](#). The file size is 2.1 MBytes.

**MXE BackPACK:** This module provides the ModelSim source-level simulator for the VHDL and Verilog HDLs. The software from module version 3.1WP1.0, released July 24, 2000, was used for this tutorial. The file for the module is [webpack\\_mxe\\_simulator.exe](#). The file size is 17.8 MBytes.

## Installing WebPACK

Install the WebPACK software modules in the following order:

1. Double-click the `webpack_hdl_abel.exe` file. The installation script will run and install the software. Accept the default settings for everything.
2. Double-click the `webpack_cpdl_fitter.exe` file. The installation script will run and install the software. Accept the default settings for everything.
3. If you downloaded the ChipViewer module, then double-click the `webpack_chipviewer.exe` file. The installation script will run and install the software. Accept the default settings for everything.
4. Double-click the `webpack_programmer.exe` file. The installation script will run and install the software. Accept the default settings for everything.

The WebPACK installation may query you about setting some environment variables if you have Xilinx software like Alliance or Foundation already installed. But if you already have these tools you probably don't need WebPACK.

## 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 [xstooset.exe](#) file V3.0. The file size is 1.1 MBytes.

## Installing XSTOOLS

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

# 3

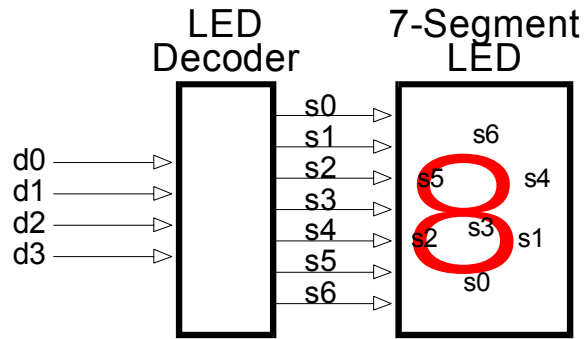
## Our First Design

### 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:

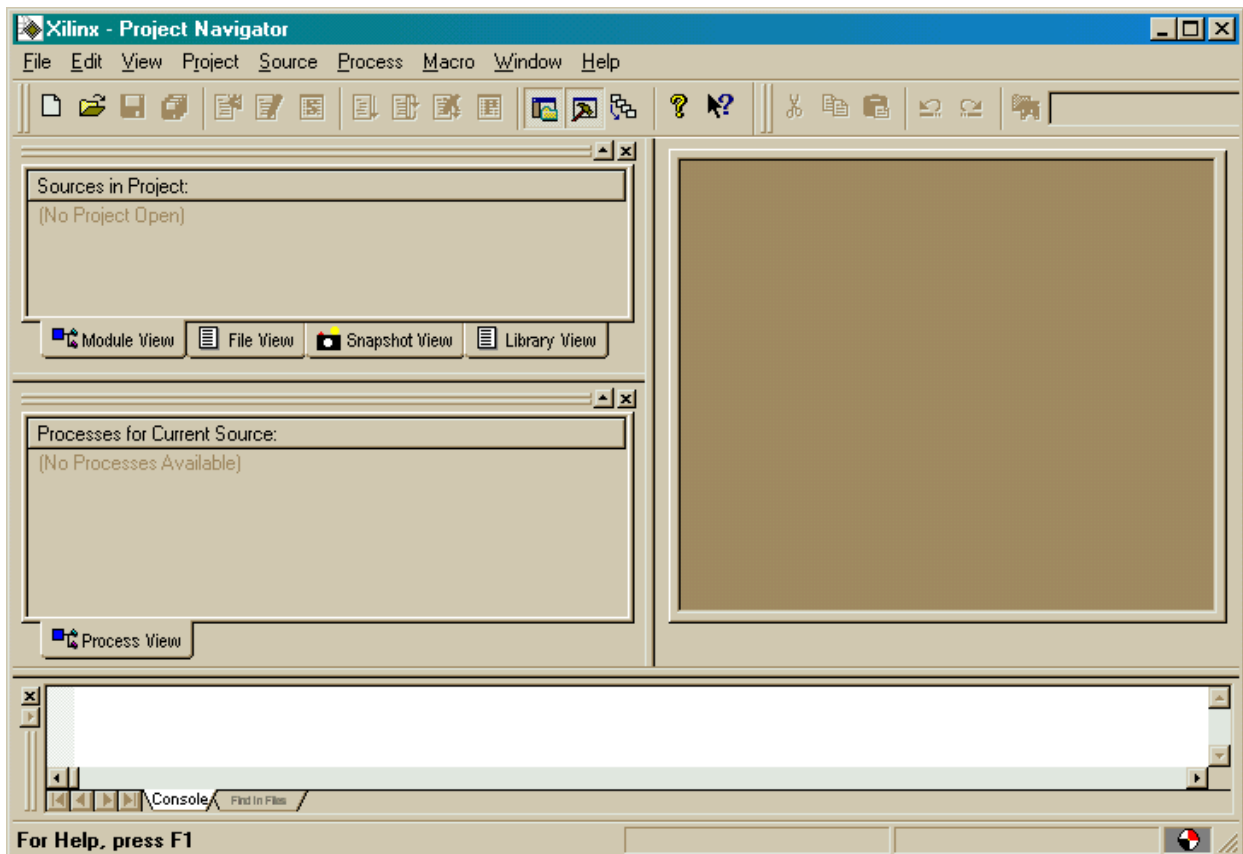


## Starting WebPACK Project Navigator

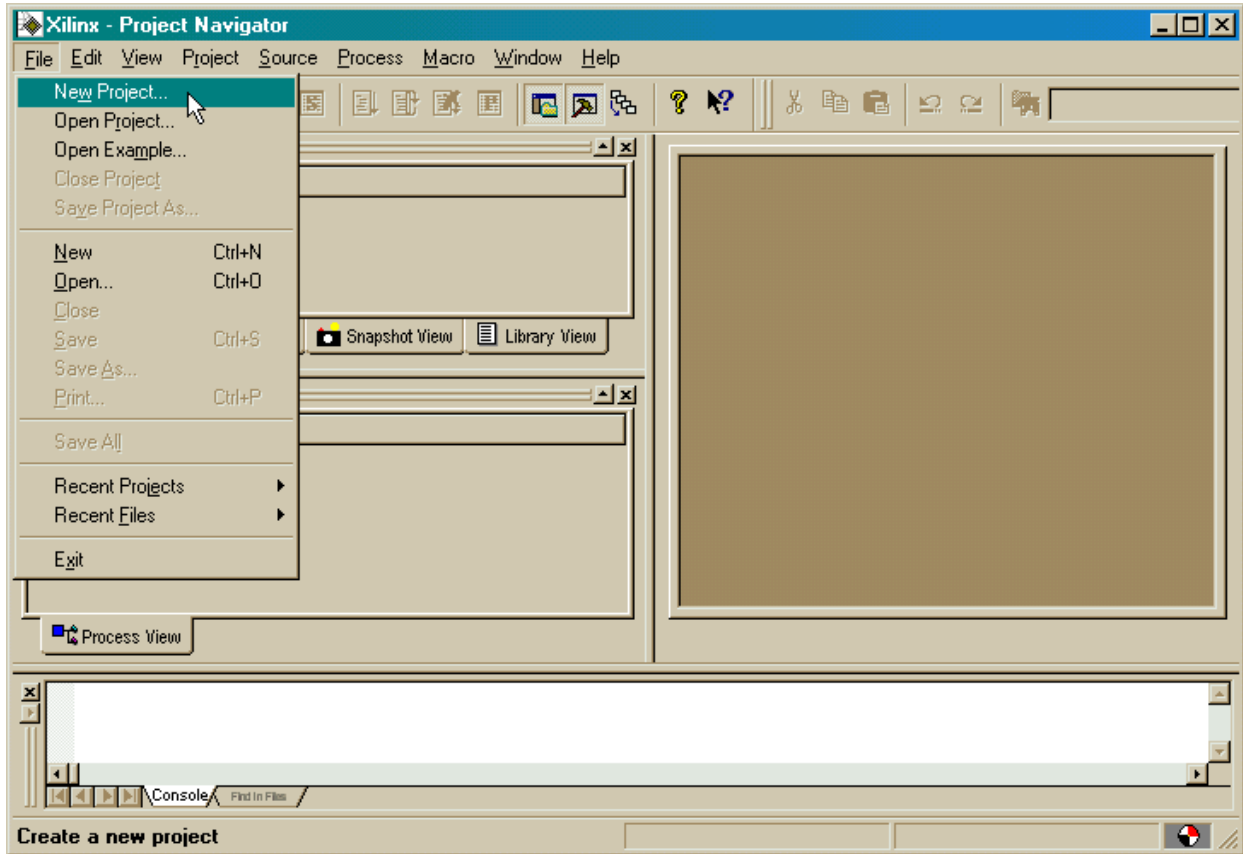


We start WebPACK by double-clicking the icon, on the desktop. This will bring up an empty project window as shown below. The window has four panes:

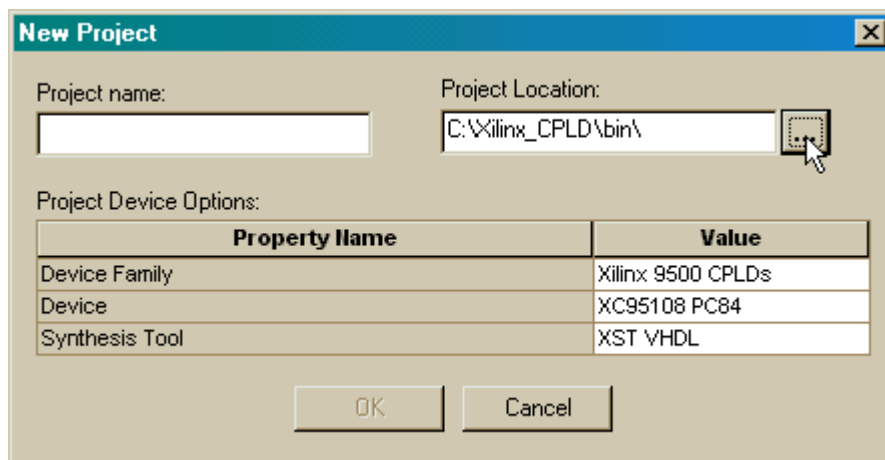
1. A **source pane** that shows the organization of the source files that make up our design. There are four tabs so we can view the source files, functional modules, or HDL libraries for our project or look at various snapshots of the project.
2. A **process pane** that lists the various operations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



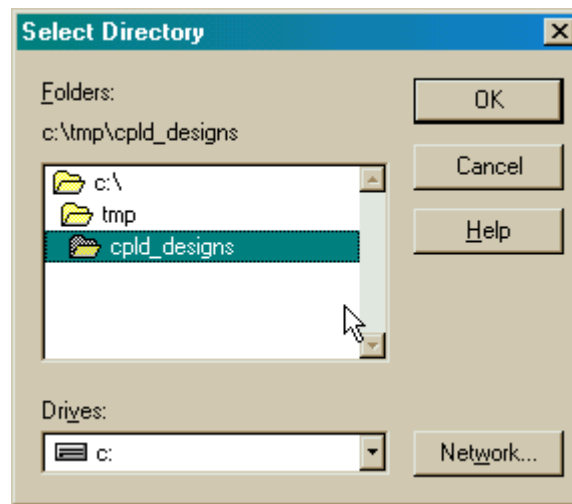
To start our design, we must create a new project by selecting the File⇒New Project item from the menu bar.



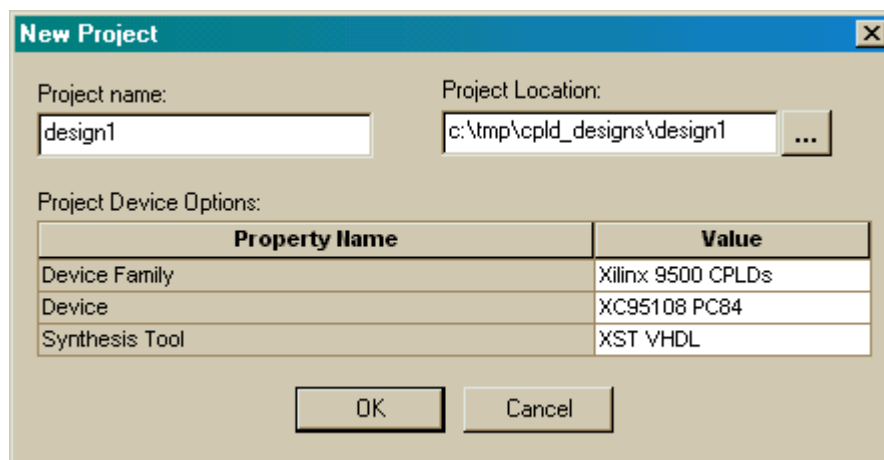
This brings up the **New Project** window where we can enter the location of our project files, project name, the target CPLD device for this design, and the tools used to synthesize logic from our source files.



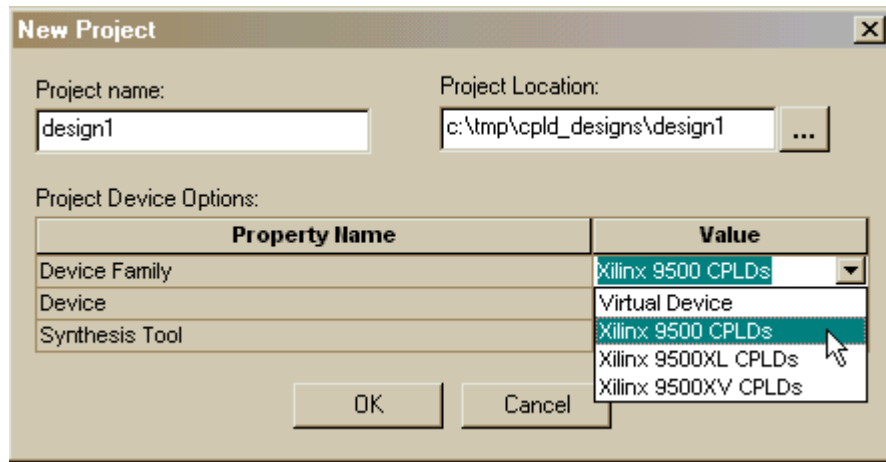
Click on the ... button next to the Project Location field and use the **Select Directory** window to select a folder where our project files will be stored. For our design examples, we will store everything in the C:\tmp\cpld\_designs folder.



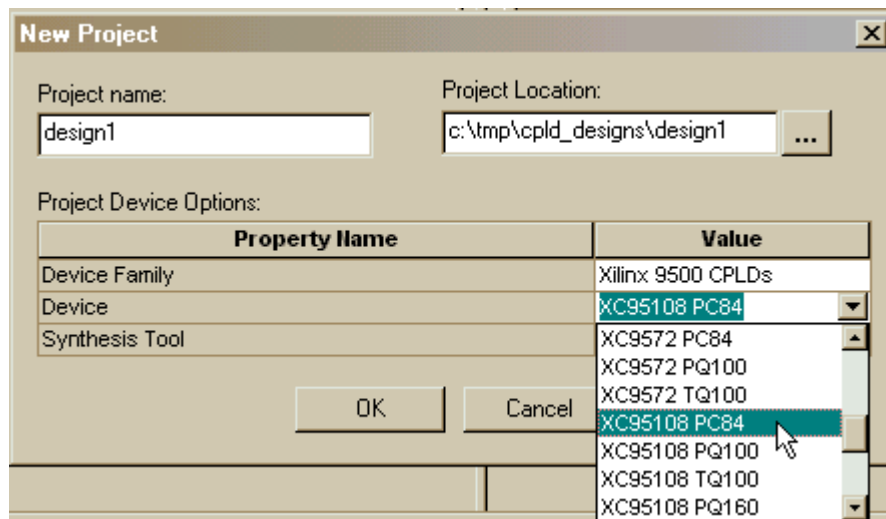
Next we will give our LED decoder design the descriptive title of `design1` by typing it into the Project name field.



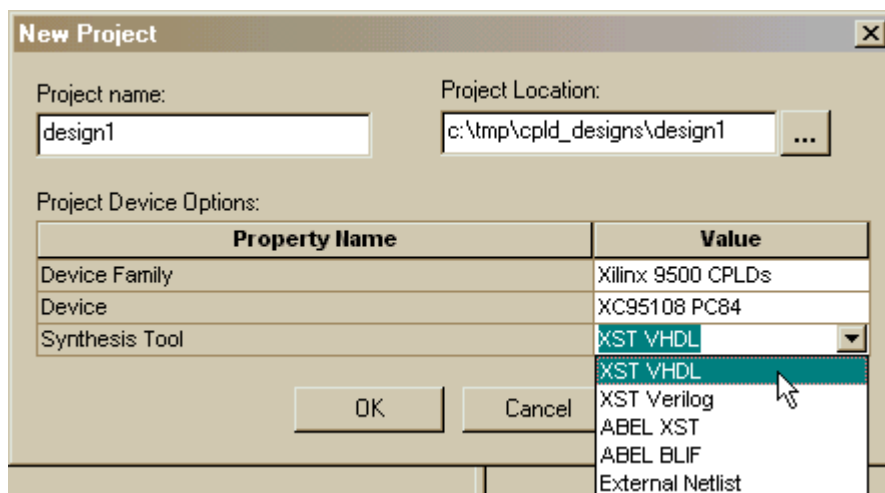
To set the family of CPLD devices we will target with this design, click in the Value field of the Device Family property. Select the Xilinx 9500 CPLDs entry in the pop-up menu that appears.



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.

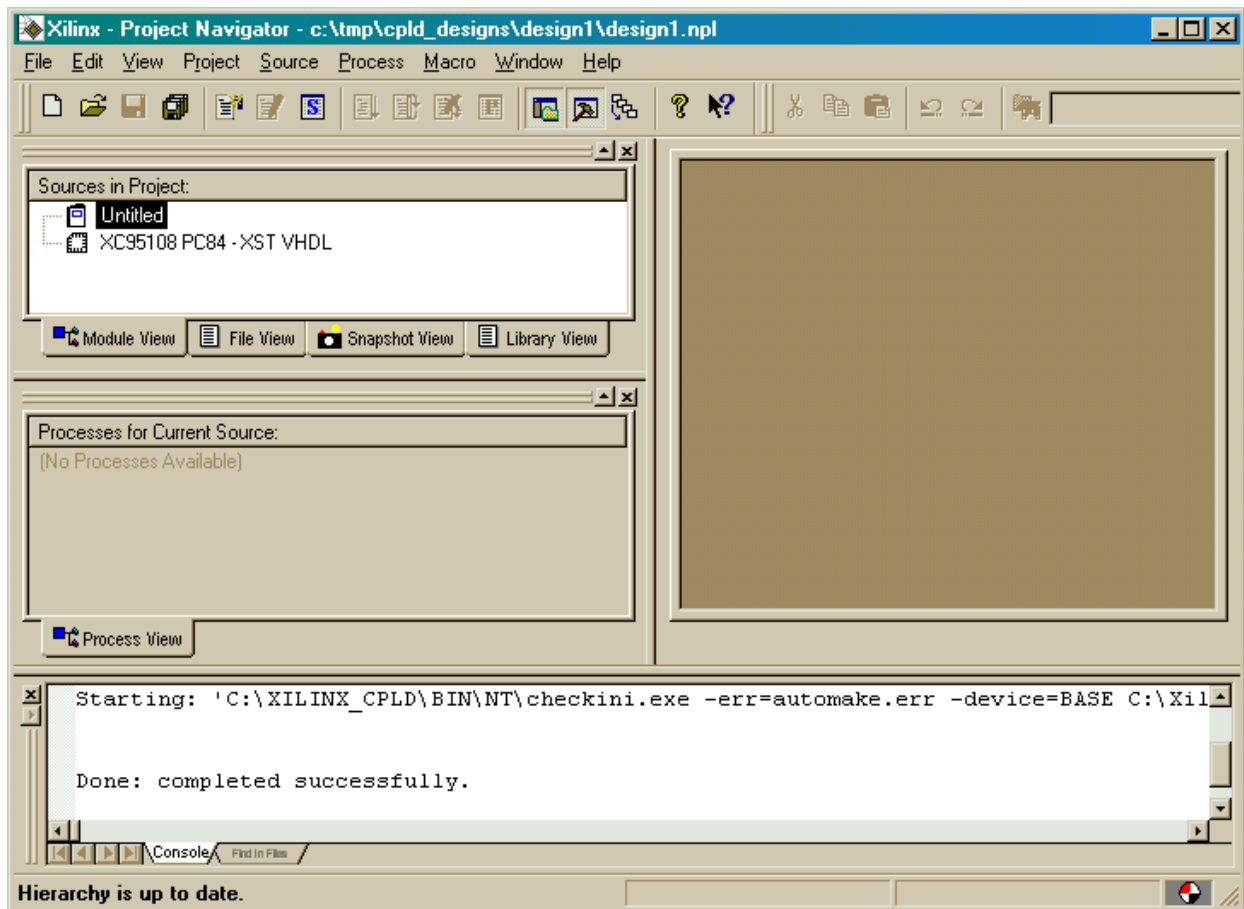


Finally, our designs will be done using VHDL so click in the Value field of the Synthesis tool property and select XST VHDL from the pop-up menu. This activates the Xilinx VHDL synthesizer.



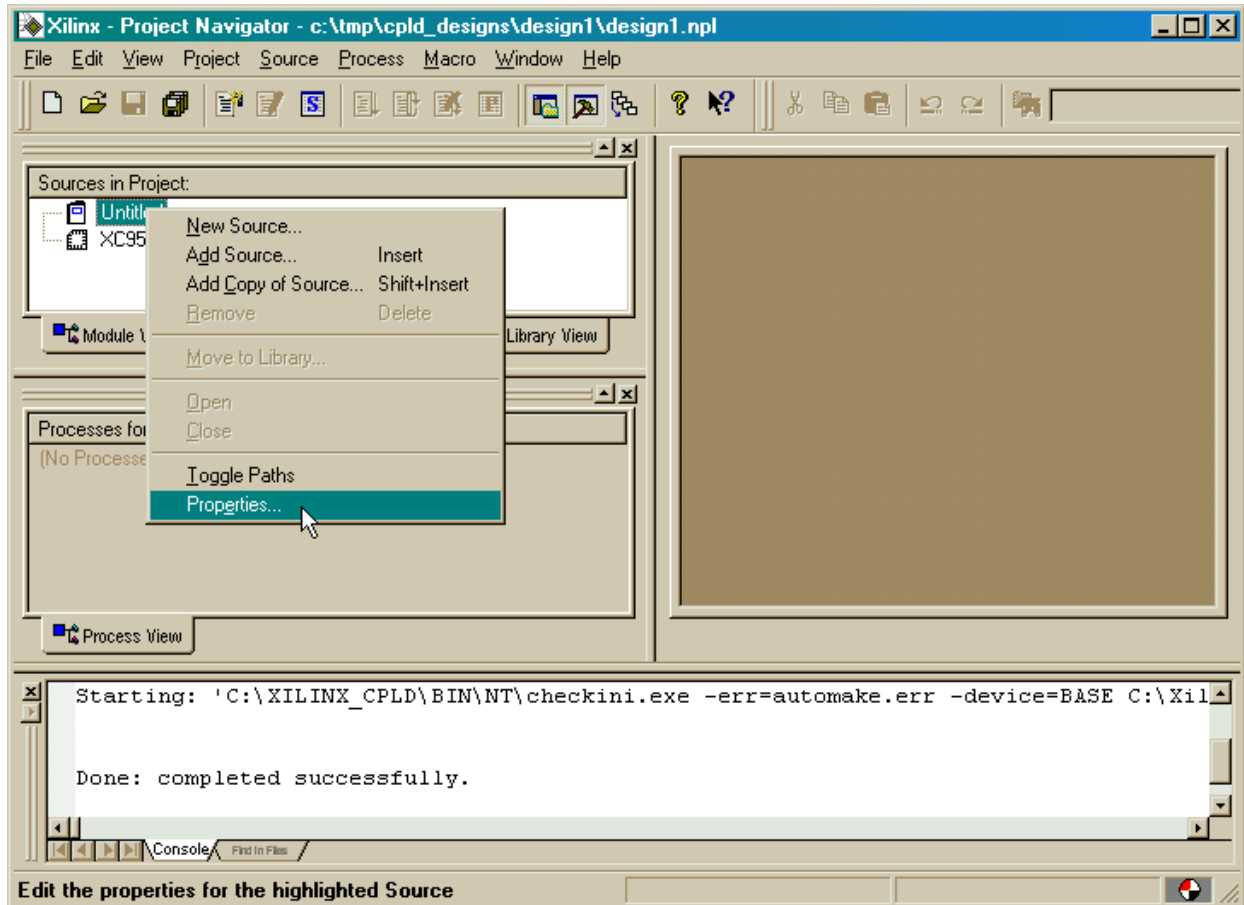
Once all the fields are set, click on OK in the **New Project** window. Now the Sources pane in the **Project Navigator** window contains two items:

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

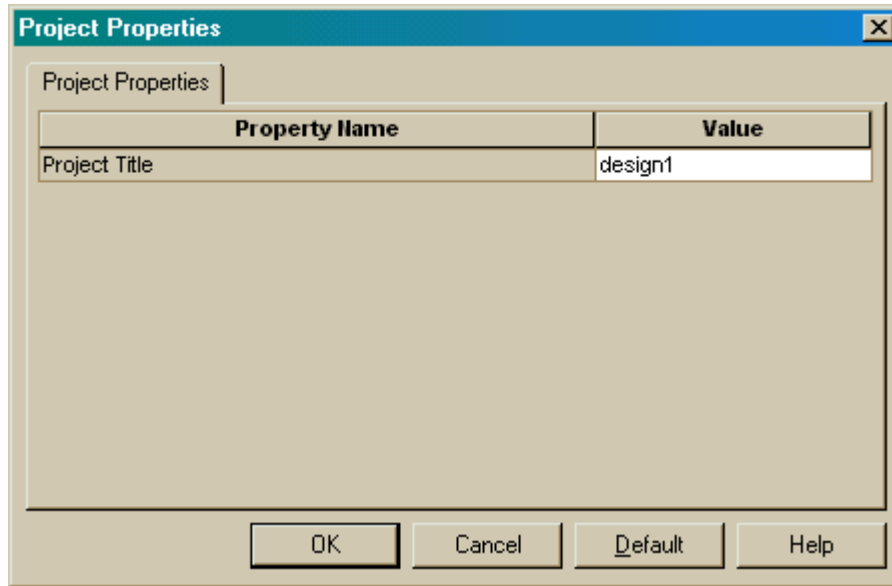


## Naming Your Project

Now we will name the project by right-clicking the mouse on the Untitled project object and selecting the Properties item from the pop-up menu.



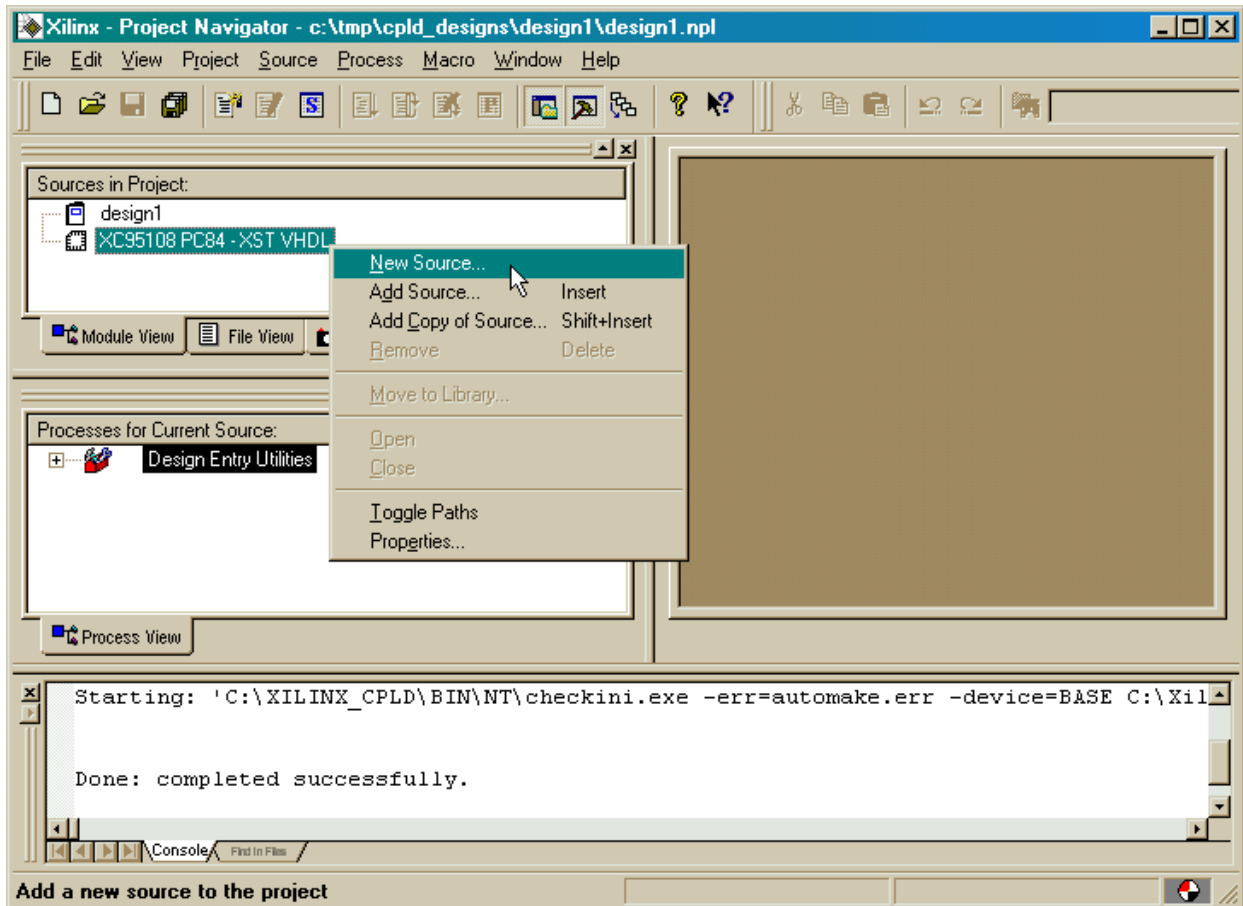
This displays the **Project Properties** window that contains a single Project Properties tab. The tab has a single slot where we can type the Project Title. I chose design1 to keep it consistent with the project name we chose previously, but you can choose any name.



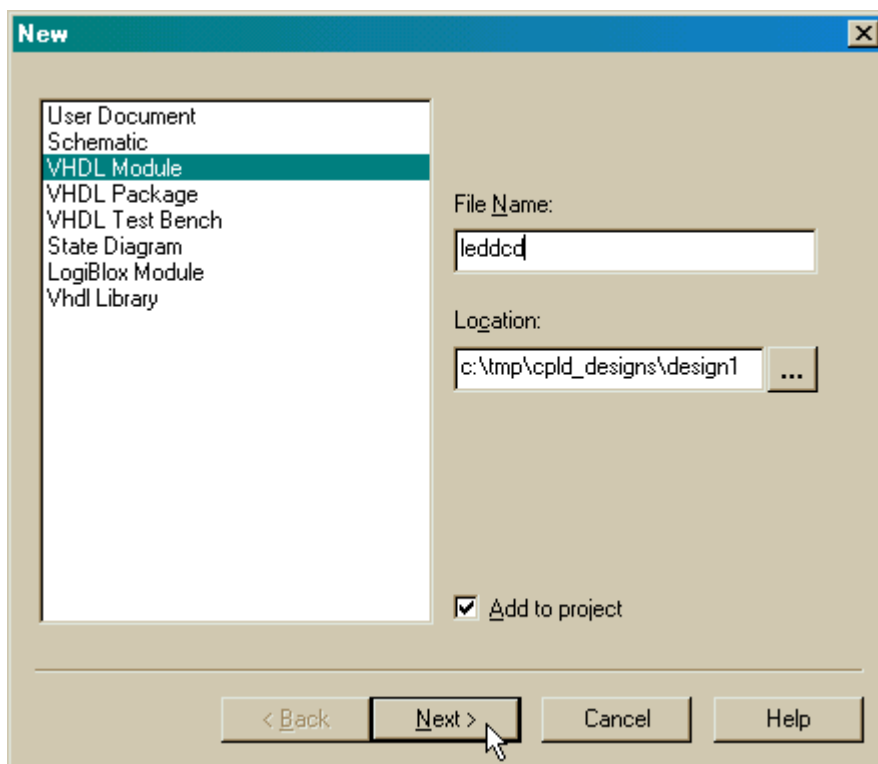
Then we click on OK to close the window. The name of the project object in the Source pane will be changed to design1.

## 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 and click on Next.



Then the **Define VHDL Source** window appears where we name the module entity and the name of the architecture section for the entity. In the window shown below, we have chosen the entity name to be leddcd, and the architecture section will be called leddcd\_arch.

Define VHDL Source

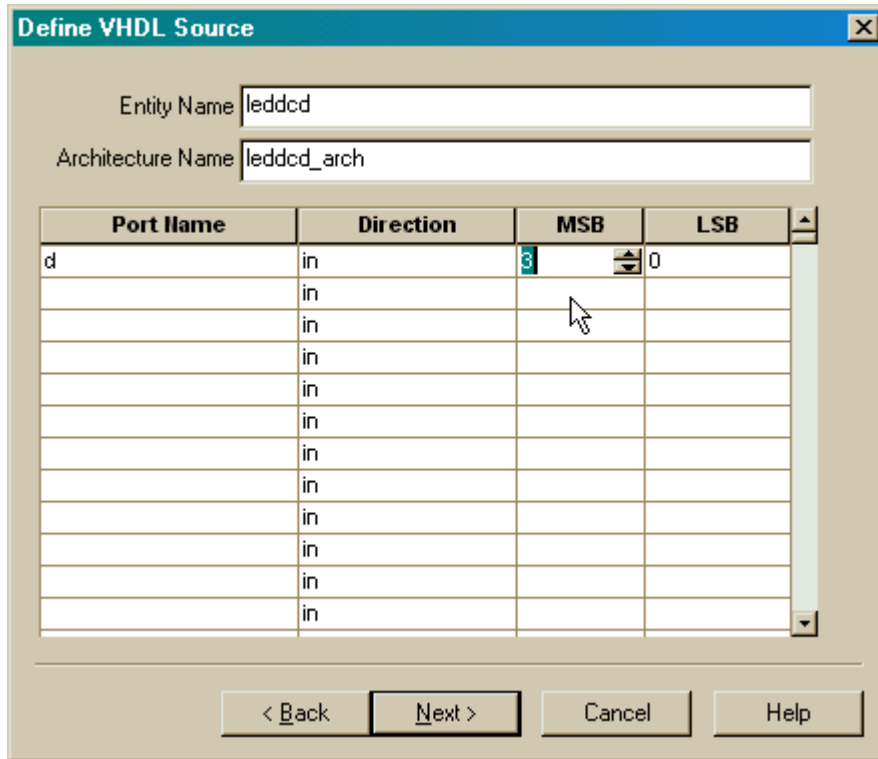
Entity Name: leddcd

Architecture Name: leddcd\_arch

Port Name	Direction	MSB	LSB
	in		
	in		
	in		
	in		
	in		
	in		
	in		
	in		
	in		
	in		
	in		
	in		
	in		

< Back   Next >   Cancel   Help

We declare the inputs and outputs to the LED decoder circuit in the same window. 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.



Define VHDL Source

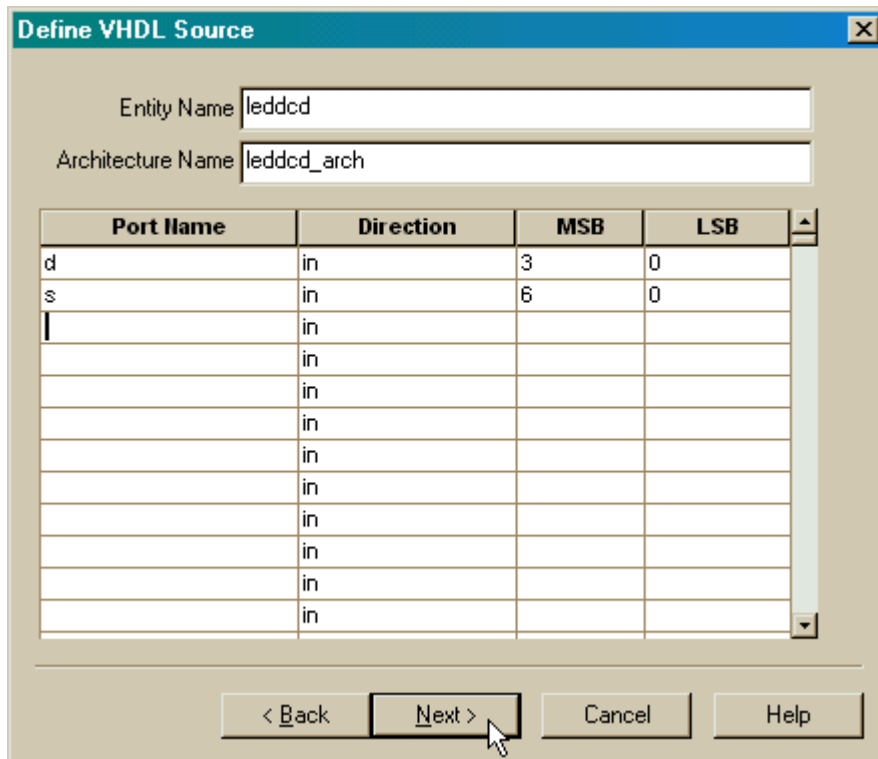
Entity Name: leddcd

Architecture Name: leddcd\_arch

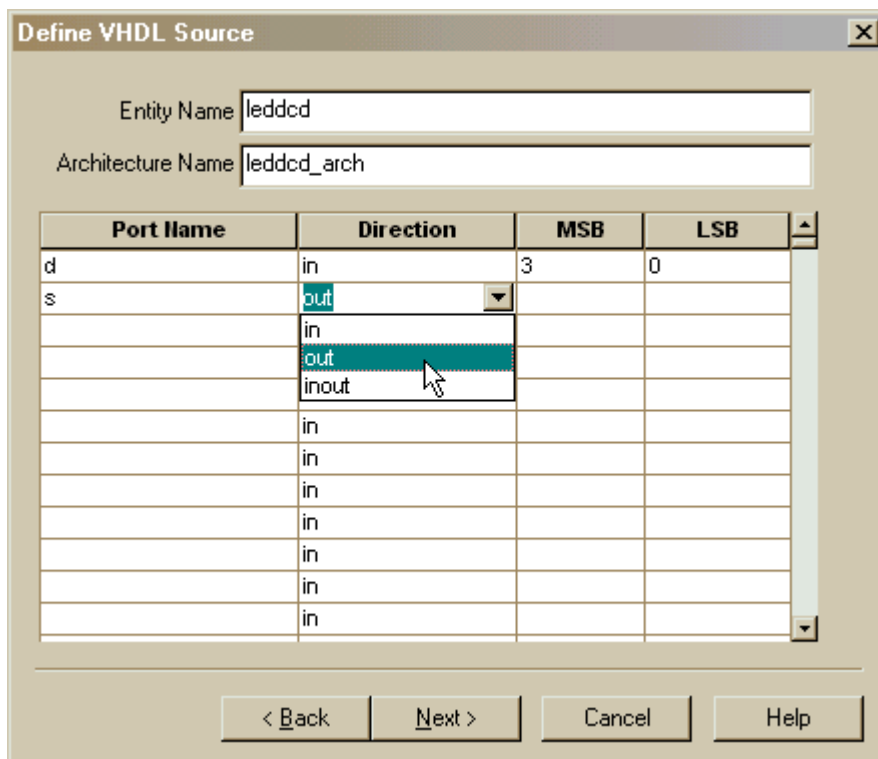
Port Name	Direction	MSB	LSB
d	in	3	0
	in		
	in		
	in		
	in		
	in		
	in		
	in		
	in		
	in		
	in		
	in		
	in		
	in		

< Back   Next >   Cancel   Help

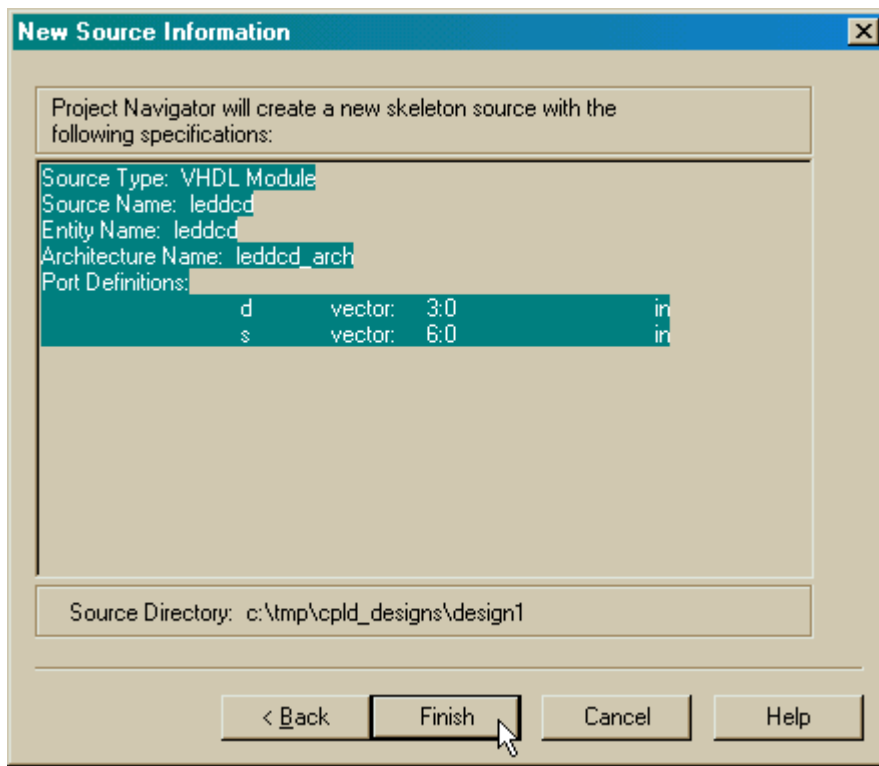
Perform the same operations to create the seven-bit wide **s** bus that drives the LEDs.



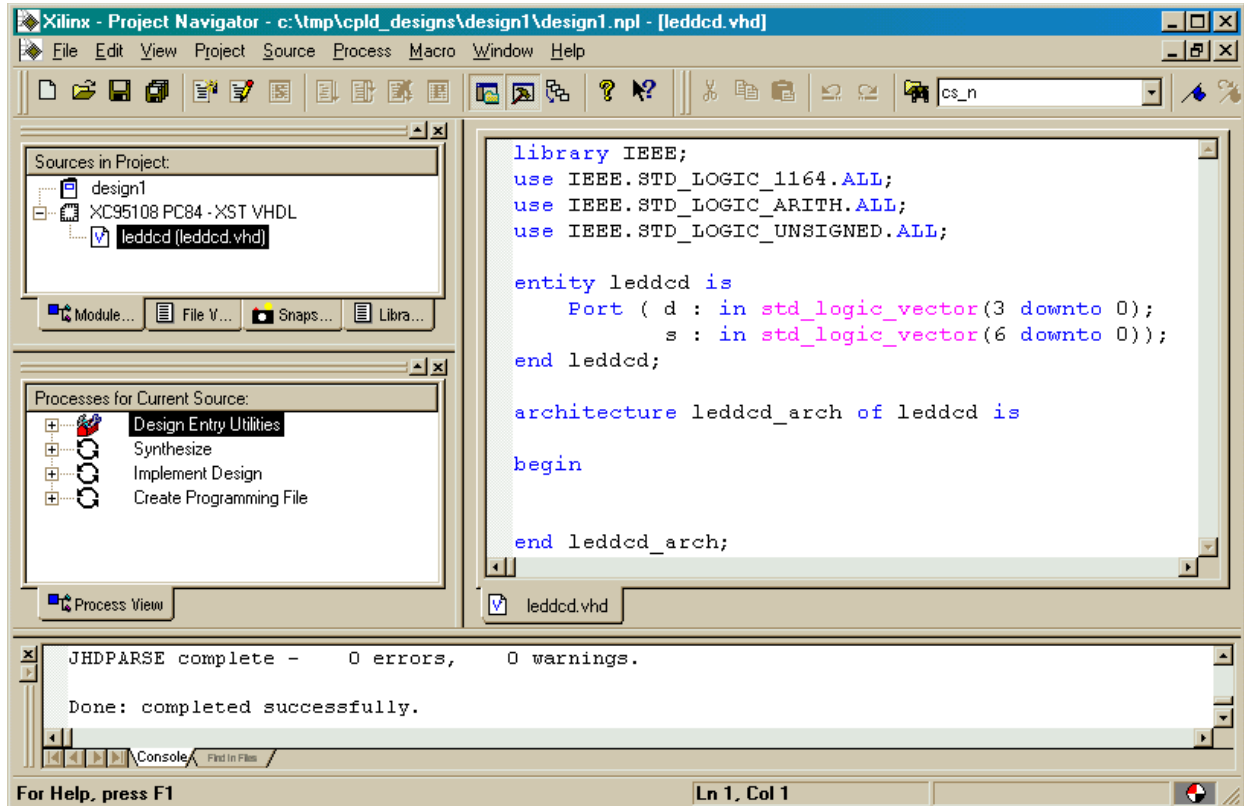
We must also click in the Direction field for the **s** bus and select out from the pop-up menu in order to make the **s** bus signals into outputs.



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 right-hand pane of the **Project Navigator** window displays a VHDL skeleton for our LED decoder. (We will 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 14-29).

```

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 leddcd_arch 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"

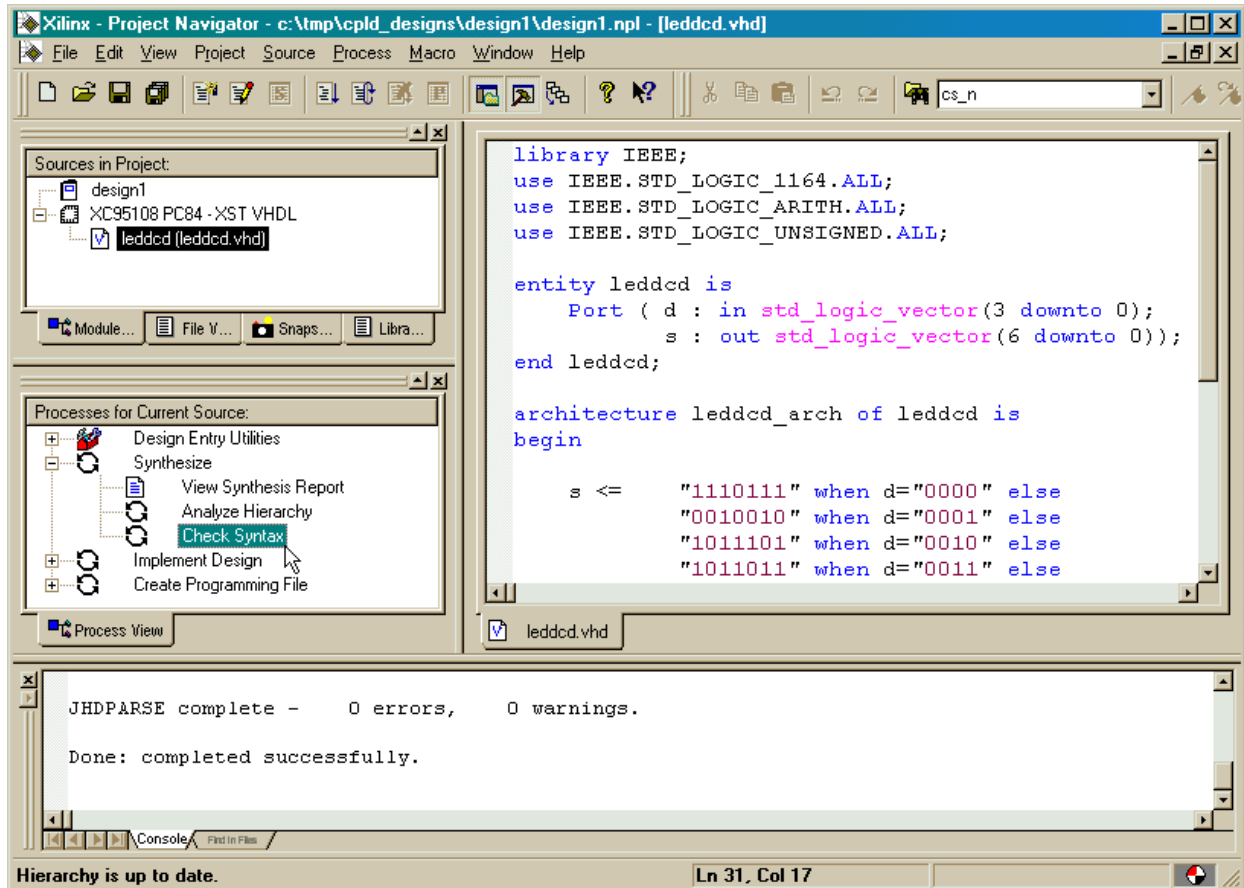
end leddcd_arch;

```

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

## Checking the VHDL Syntax

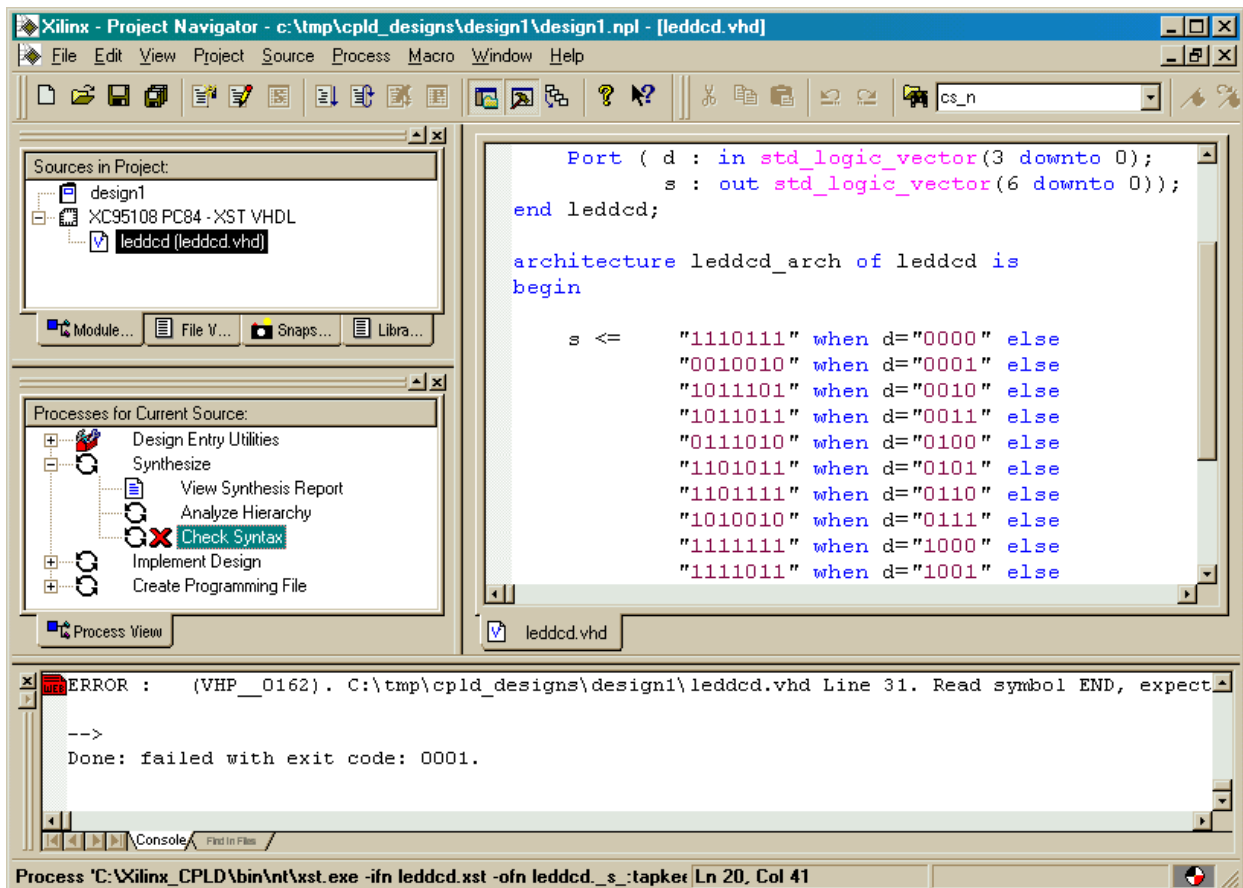
We can check for errors in our VHDL by highlighting the leddcd object in the Sources pane and then double-click on Check Syntax in the Process pane as shown below.




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

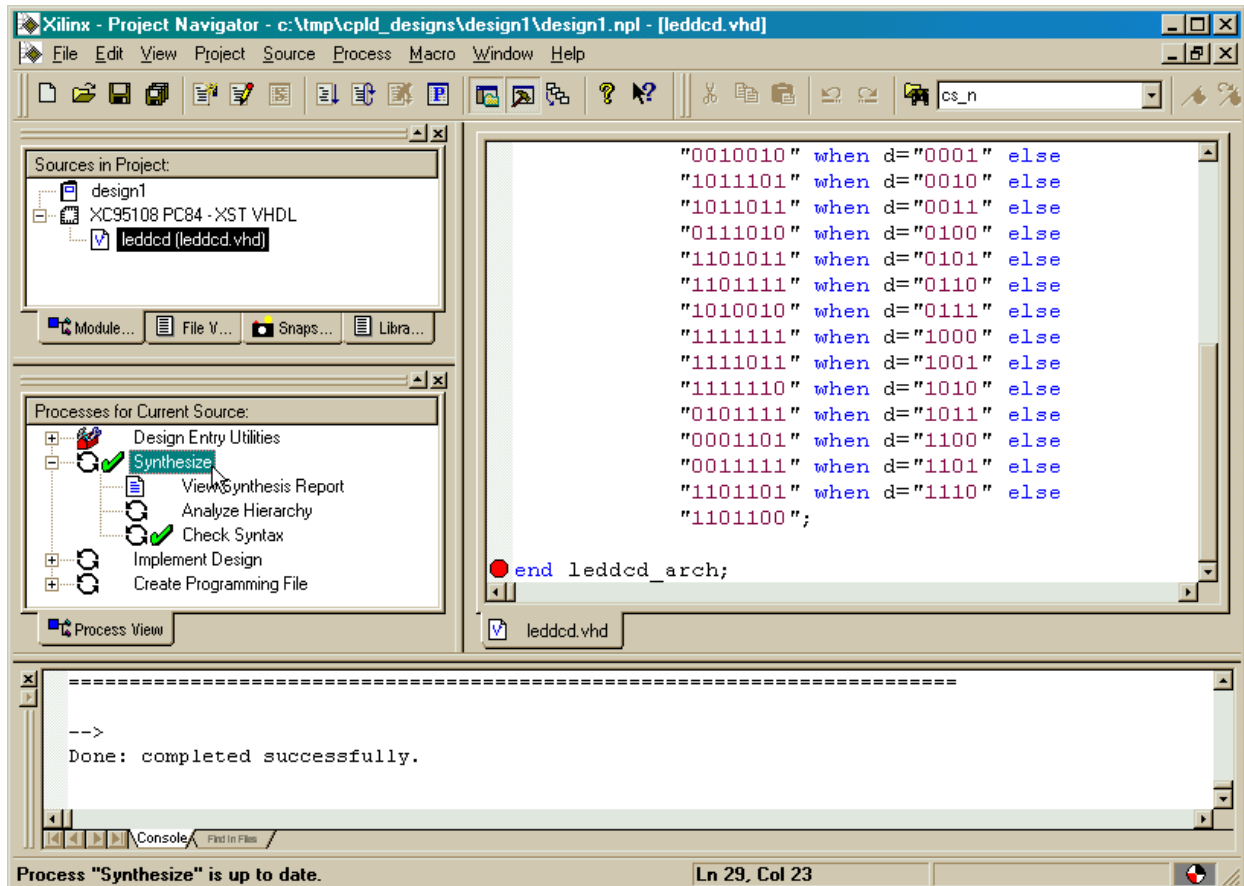
We can find the location of the error by scrolling the log pane at the bottom of the **Project Navigator** window until we find an error message. In this case, the error is located on line 31. You can manually scroll to line 31 in the HDL editor pane, or you can double-click on the error message in the log pane to go directly to the erroneous line.




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 re-running the syntax checker reveals another error on line 20: the keyword `else` has been misspelled as `els`. 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

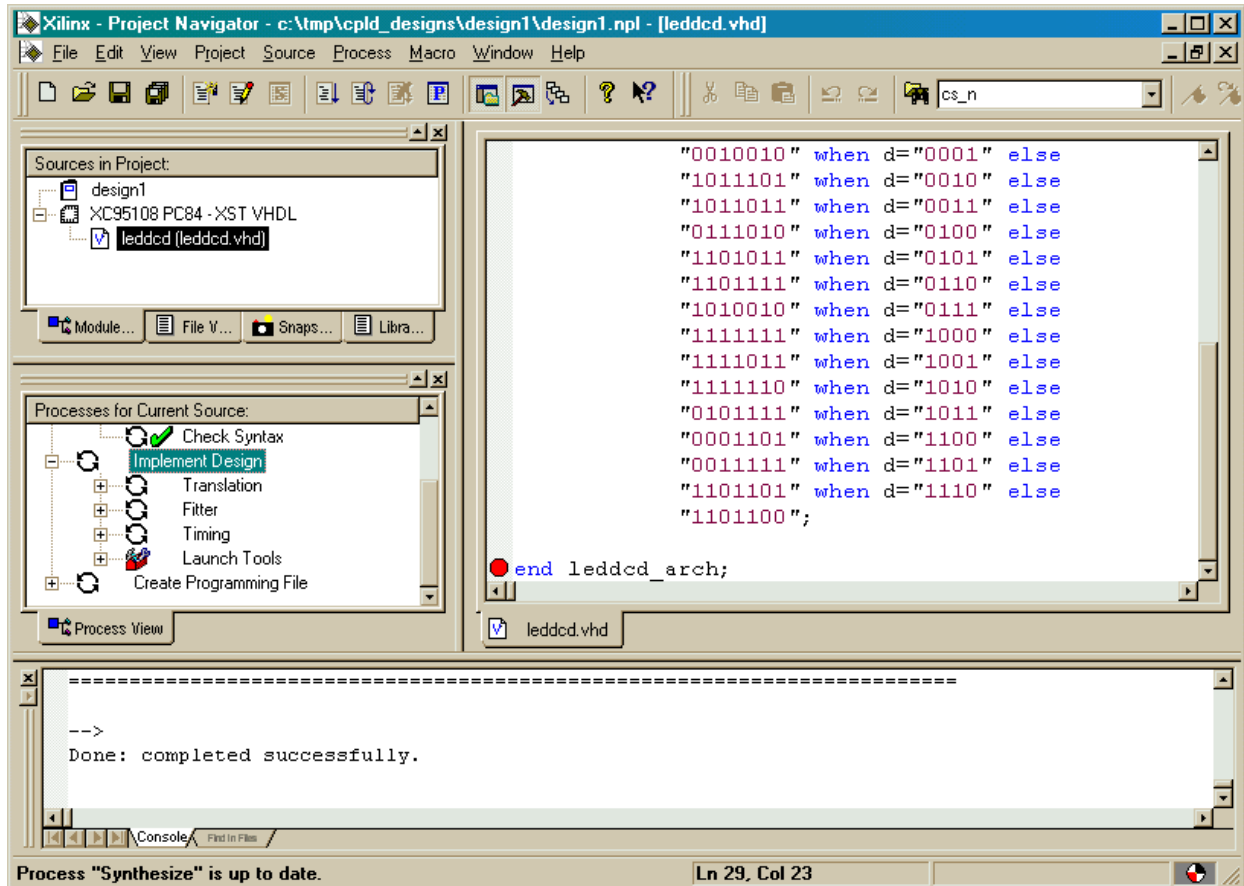
Now that we have valid VHDL for our design, we need to convert it into a logic circuit. This is done by highlighting the leddcd object in the Sources pane and then double-clicking on the Synthesize process as shown below.





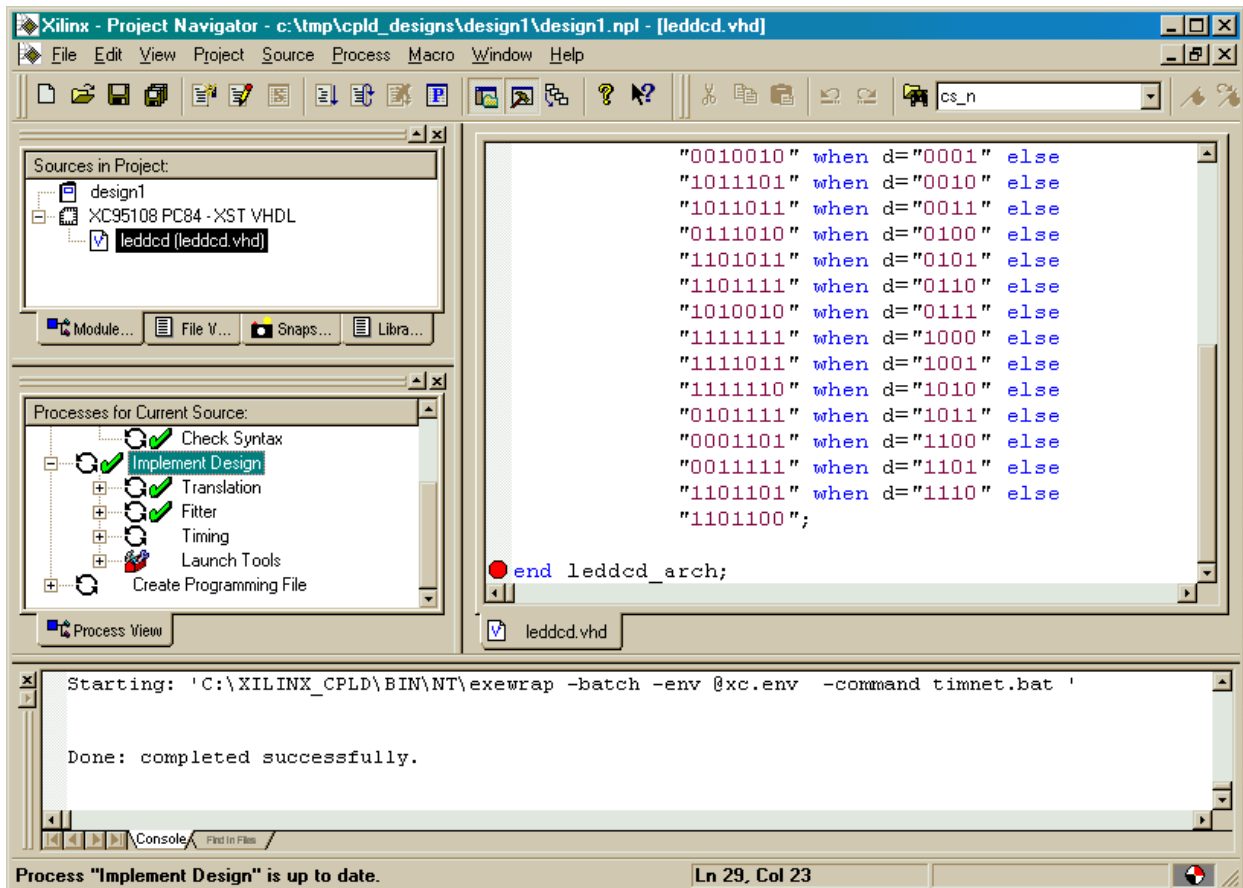
The synthesizer will read the VHDL code and transform it into a netlist of gates. This will take only a minute. If no problems are detected, a  will appear next to the Synthesize Tree process as shown above. You can double-click on the View Synthesis Report to see the various synthesizer options that 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-click on the Implement Design process.

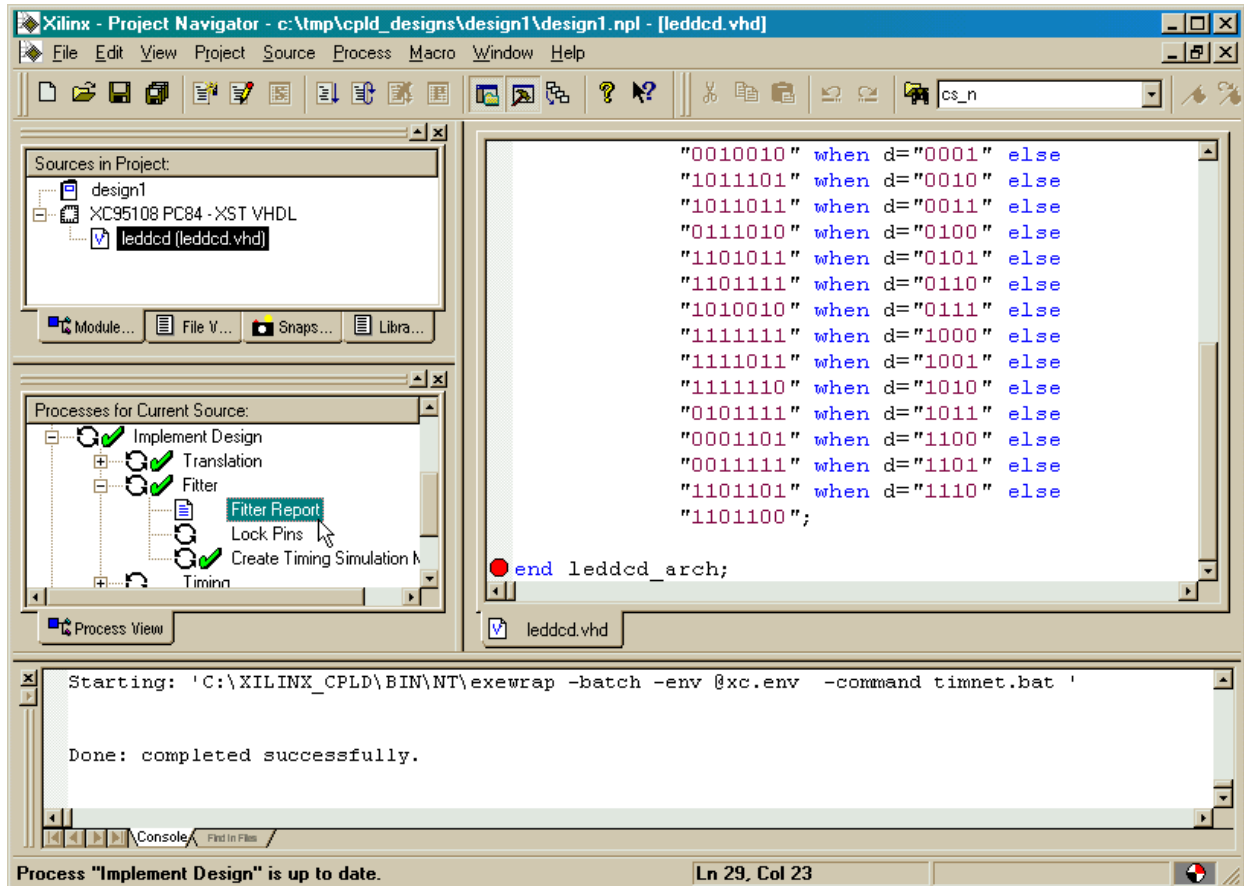


You can watch the progress of the fitting process in the status bar at the bottom of the Project Navigator window. For a simple design like the LED decoder, the fitting is completed in seconds (on a 450 MHz Celeron PC with 128 MBytes). A successful fit is indicated by the  next to the Implement Design process. You can expand the Implement Design process to see the subprocesses within it. The Translation 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 Fitter process actually maps the netlist into the circuitry elements contained in the device we selected. If the Implement Design process failed, a  would appear next to the subprocess where the error occurred.



## 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).

```
XACT: version D.20 Xilinx Inc.
Fitter Report
Design Name: leddcd Date: 8-25-2000, 9:11AM
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      0
s<1>            3        4      FB6_2   STD  FAST 45  I/O      0
s<2>            3        4      FB1_2   STD  FAST  1  I/O      0
s<3>            2        4      FB1_9   STD  FAST  6  I/O      0
s<4>            4        4      FB2_2   STD  FAST 71  I/O      0
s<5>            4        4      FB3_2   STD  FAST 14  I/O      0
s<6>            4        4      FB4_2   STD  FAST 57  I/O      0

** INPUTS **
Signal          Loc      Pin  Pin      Pin
Name            Loc      #    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

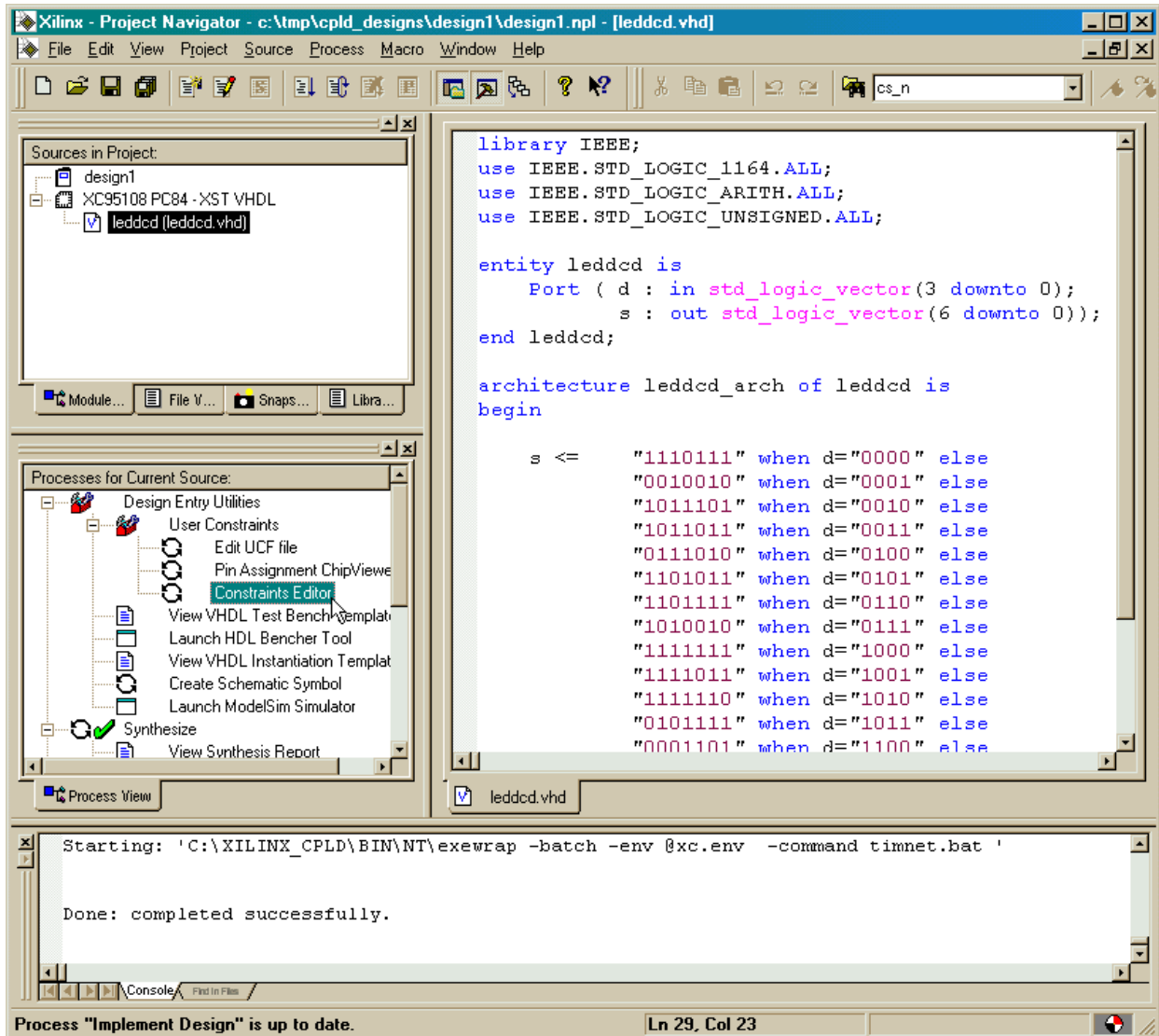
The problem we have now is that the inputs and outputs for the LED decoder were assigned to pins picked by the fitting process, but these are not the pins we actually want to use on the XS95 Board. The XS95 has eight inputs which are driven by the PC parallel port and we would like to assign the LED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

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

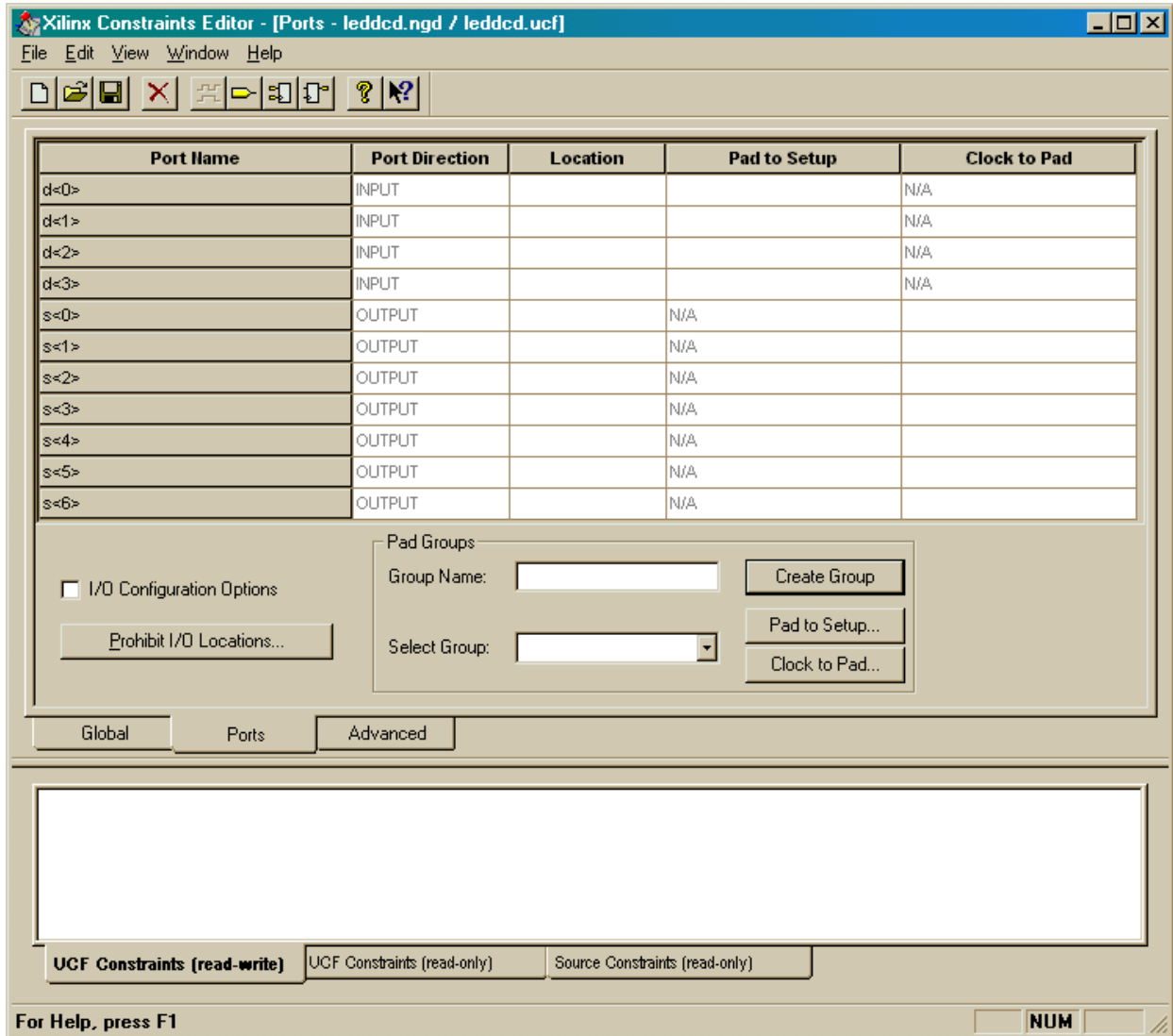
Likewise, the XS95 Board has a seven-segment LED attached to the following pins of the CPLD:

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

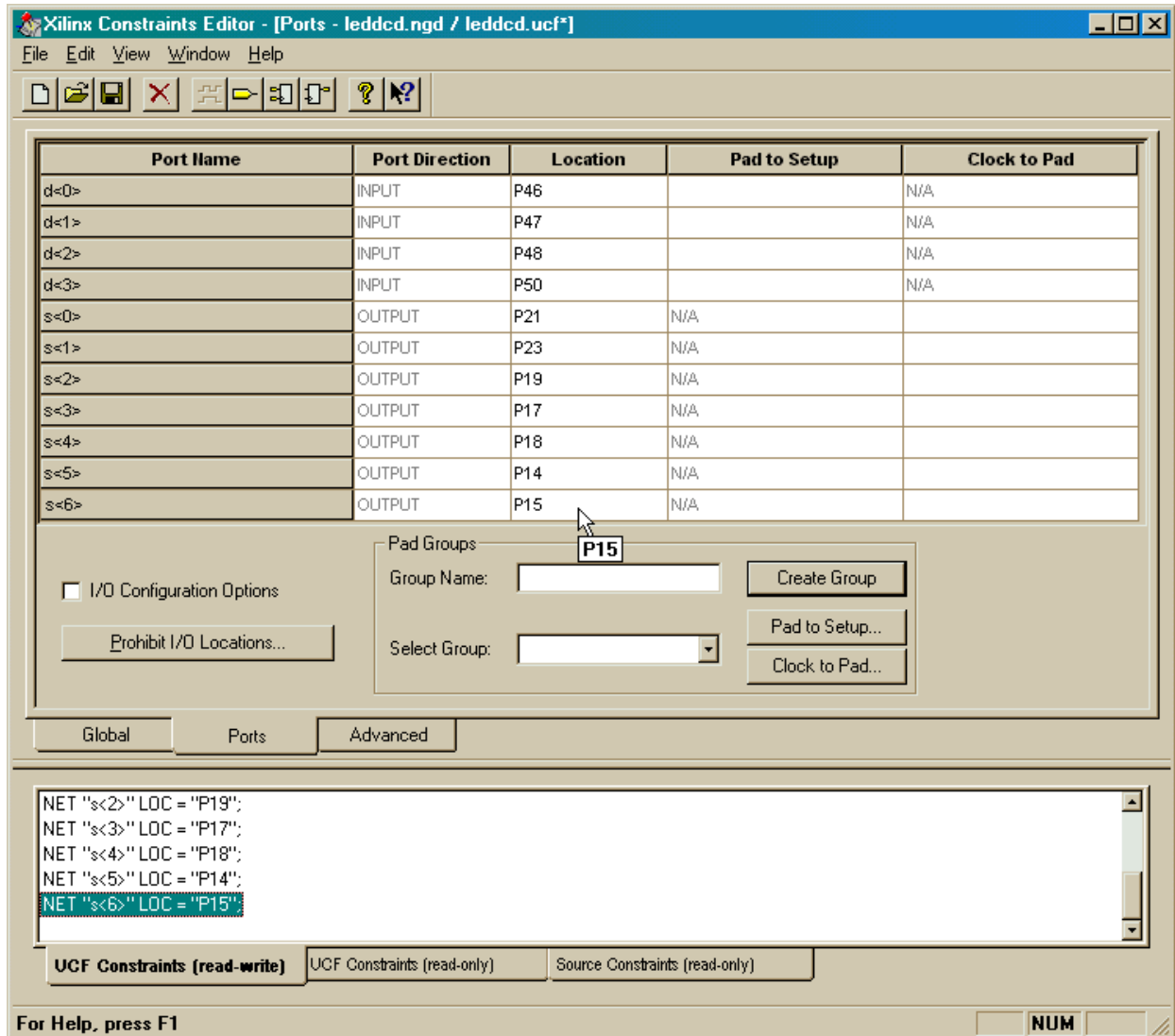
How do we constrain the fitting process so it assigns the inputs and outputs to the pins we want to use? We start by highlighting the XC95108 PC84 object in the Sources pane and then double-clicking the Constraints Editor process.




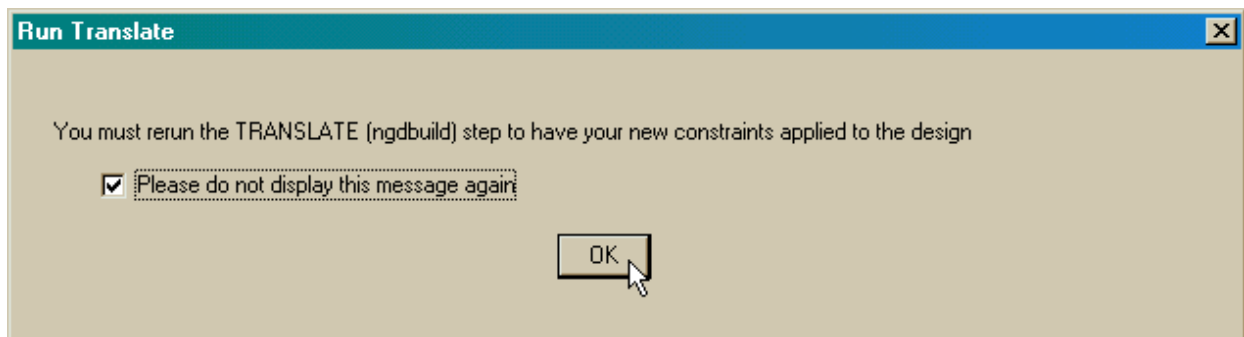
The **Constraints Editor** window appears. Click on the Ports tab in the upper pane. A list of the inputs and outputs for the LED decoder will appear. We can enter our pin assignments here.



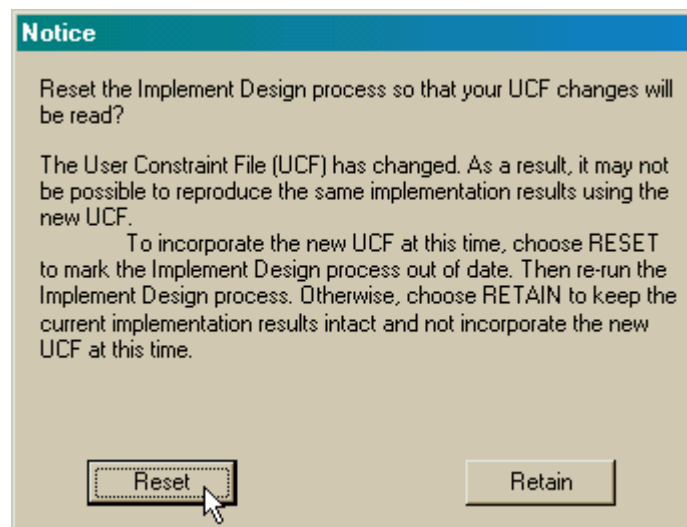
We start by clicking in the Location field for the **d<0>** input. Then just type in the pin assignment for this input: P46. Do this for each of the inputs and outputs using the pin assignments from the tables shown above. After doing this, the **Constraints Editor** window appears as follows.



After the pin assignments are entered, click on the  button to save the pin assignment constraints. The Constraint Editor will give us the following message:



Click OK and 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. Then 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     0
s<1>        3       4       FB3_12  STD  FAST 23  I/O     0
s<2>        3       4       FB3_8   STD  FAST 19  I/O     0
s<3>        2       4       FB3_5   STD  FAST 17  I/O     0
s<4>        4       4       FB3_6   STD  FAST 18  I/O     0
s<5>        4       4       FB3_2   STD  FAST 14  I/O     0
```

```
s<6>          4      4      FB3_3  STD  FAST 15  I/O      0

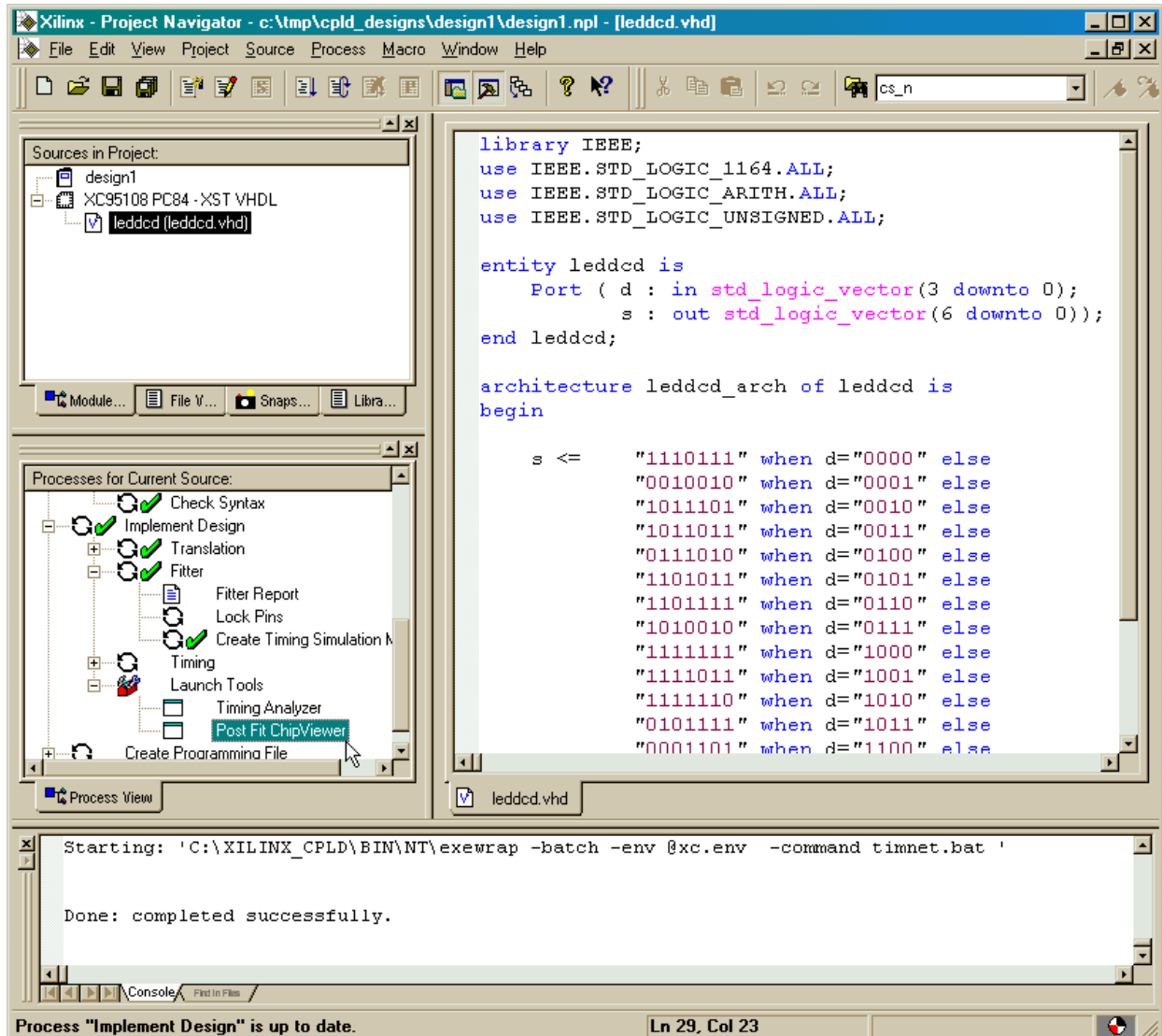
** INPUTS **
Signal          Loc          Pin  Pin      Pin
Name           #      Type  Use
d<0>          FB6_3          46  I/O      I
d<1>          FB6_5          47  I/O      I
d<2>          FB6_6          48  I/O      I
d<3>          FB6_8          50  I/O      I

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.

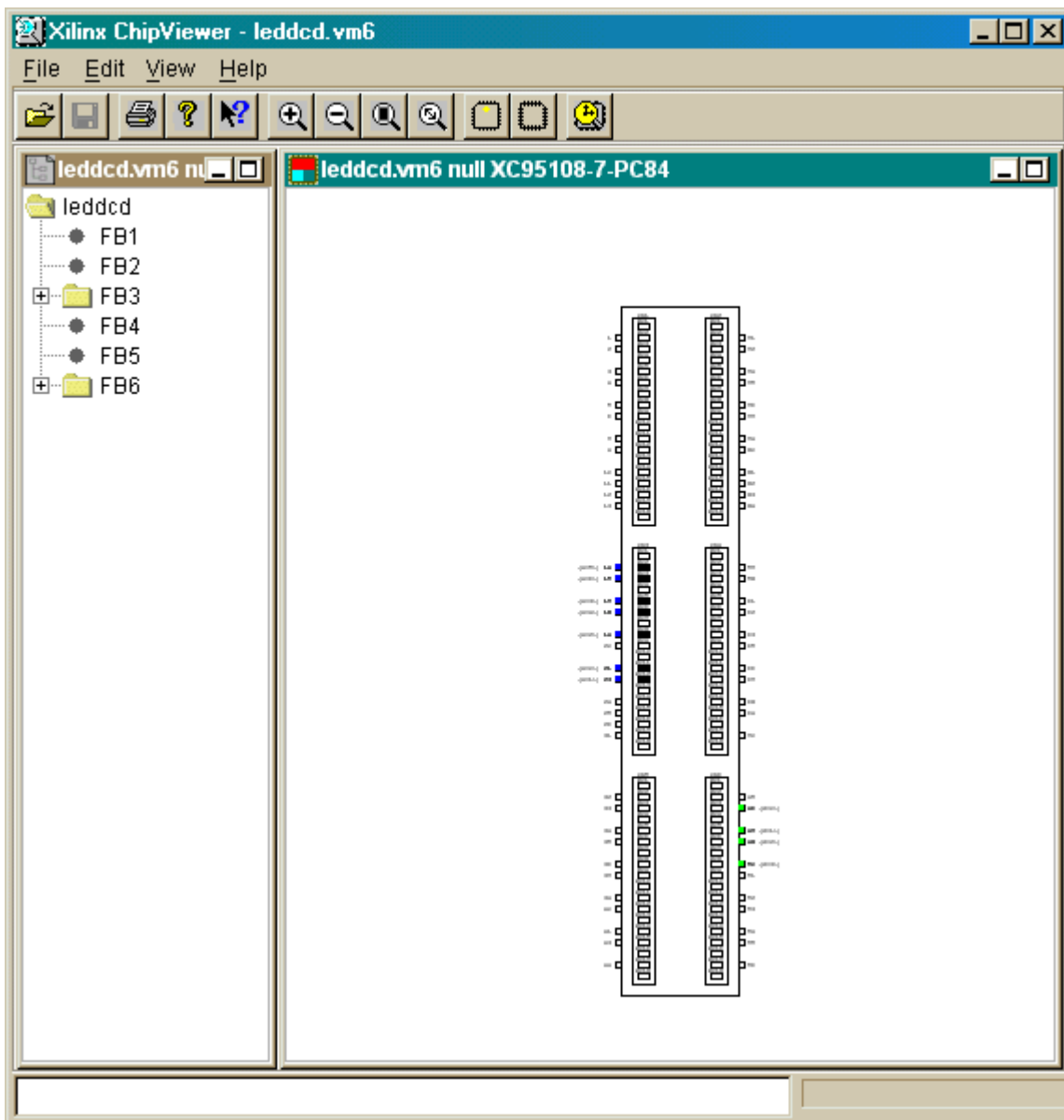
## Viewing the Chip



If you installed the ChipViewer with the rest of the WebPACK tools, then 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 XC95108 PC84 object in the Sources pane and then double-click the Post Fit ChipViewer process.

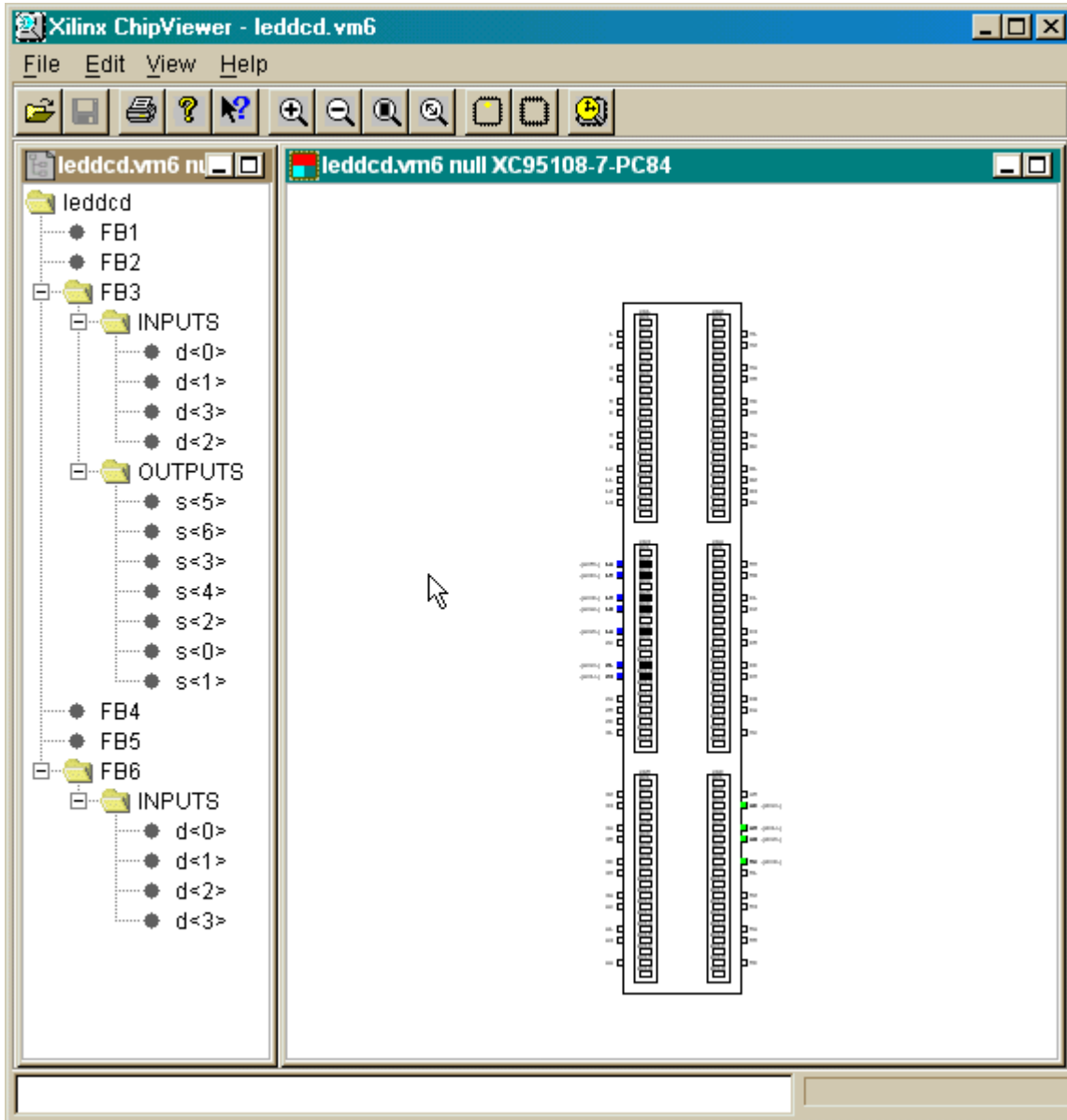


The **ChipViewer** window will appear containing two panes:

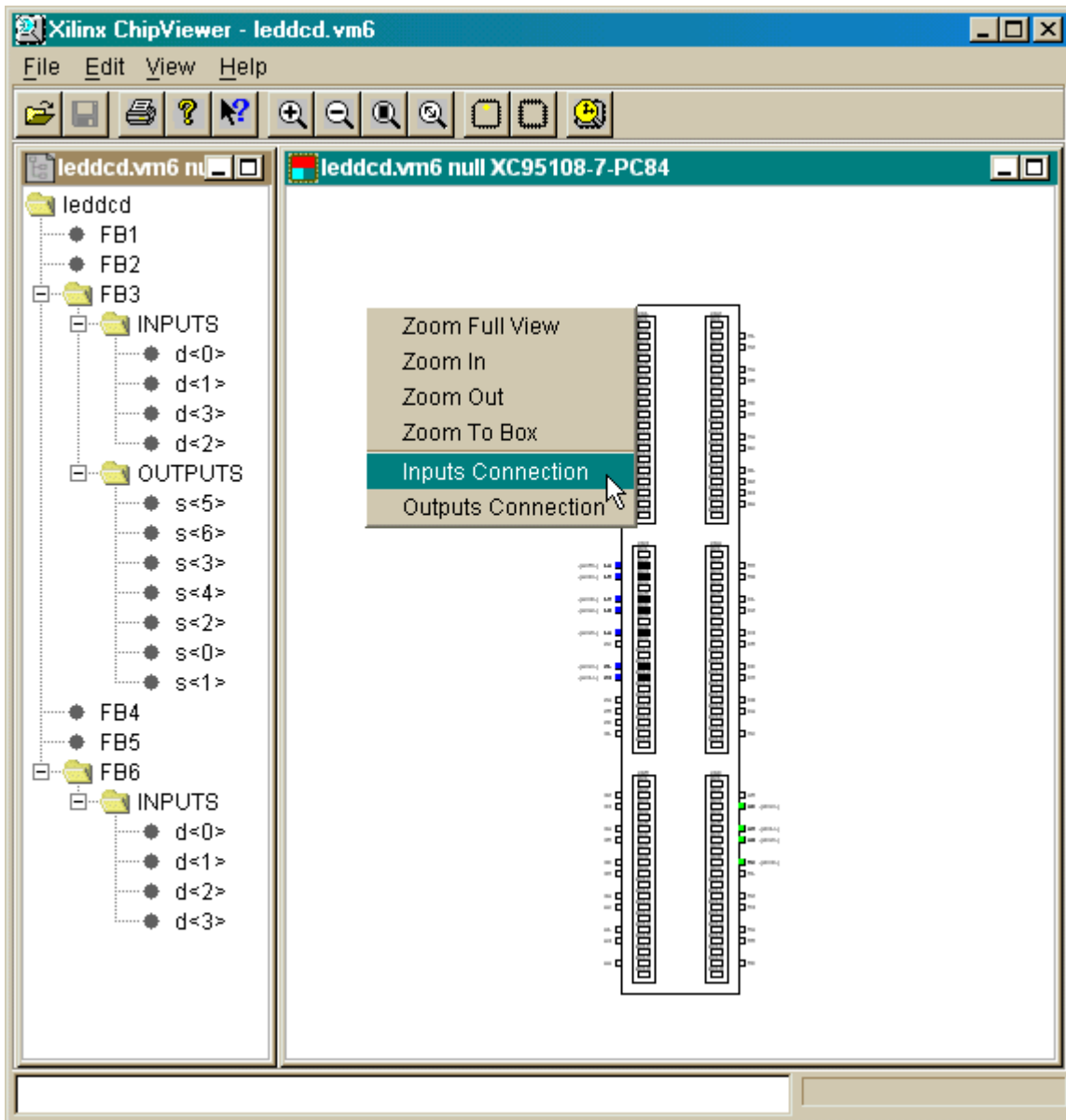
1. The left-hand pane lists the LED decoder inputs and outputs assigned to the various function blocks in the XC95108 PC84 CPLD.
2. The right-hand pane shows the 108 macrocells of the CPLD arranged into six groups of 18 cells each. The 69 I/O pins are also shown. (The 15 pins used for Vcc, GND, and programming are not shown.)



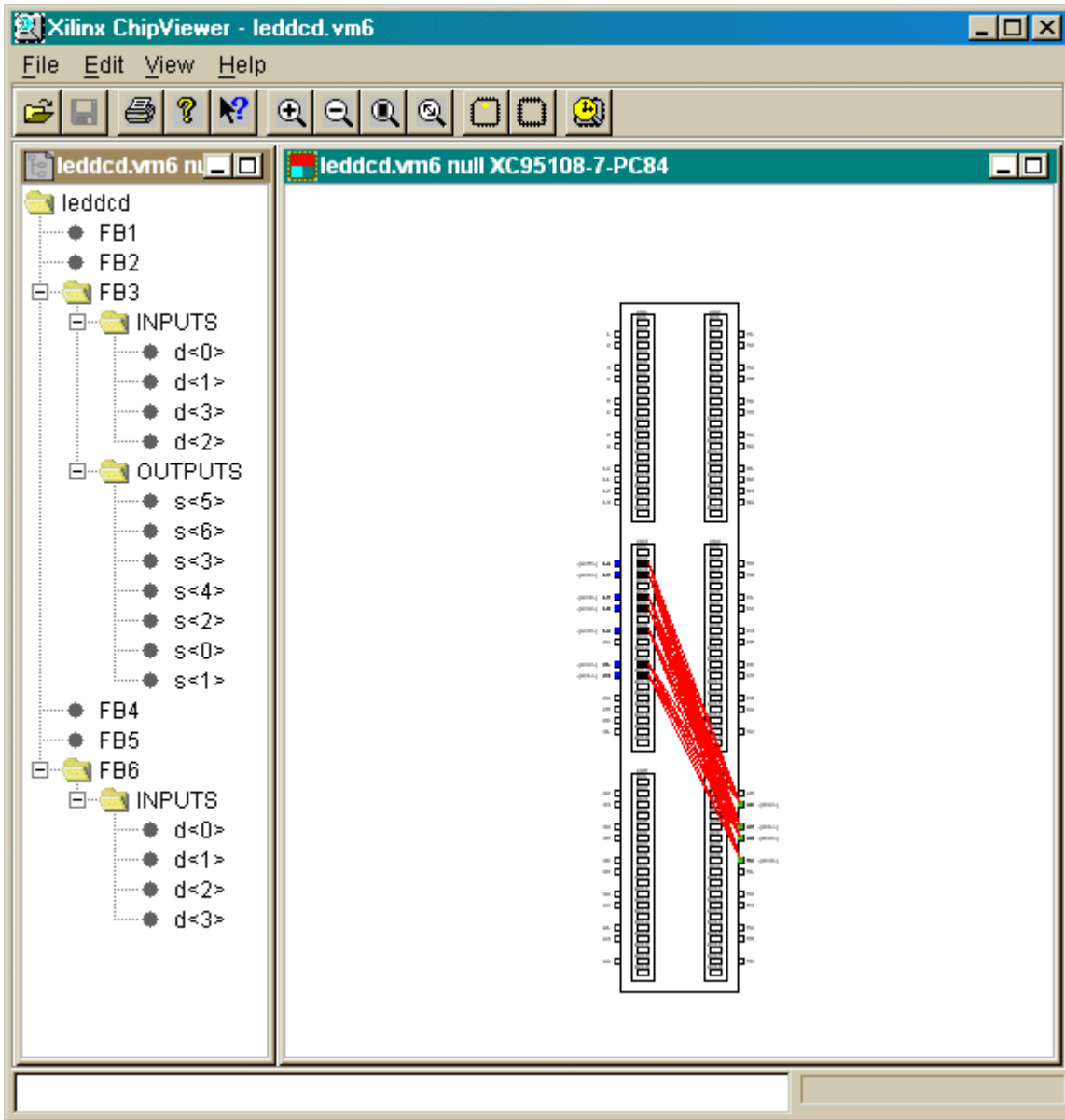
The function blocks with  icons next to them contain macrocells that are used in our design. (The remaining function blocks are not used by our design.) Clicking on the  next to FB6, we can see that the inputs enter through pins attached to function block 6. Meanwhile, all the outputs are generated by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




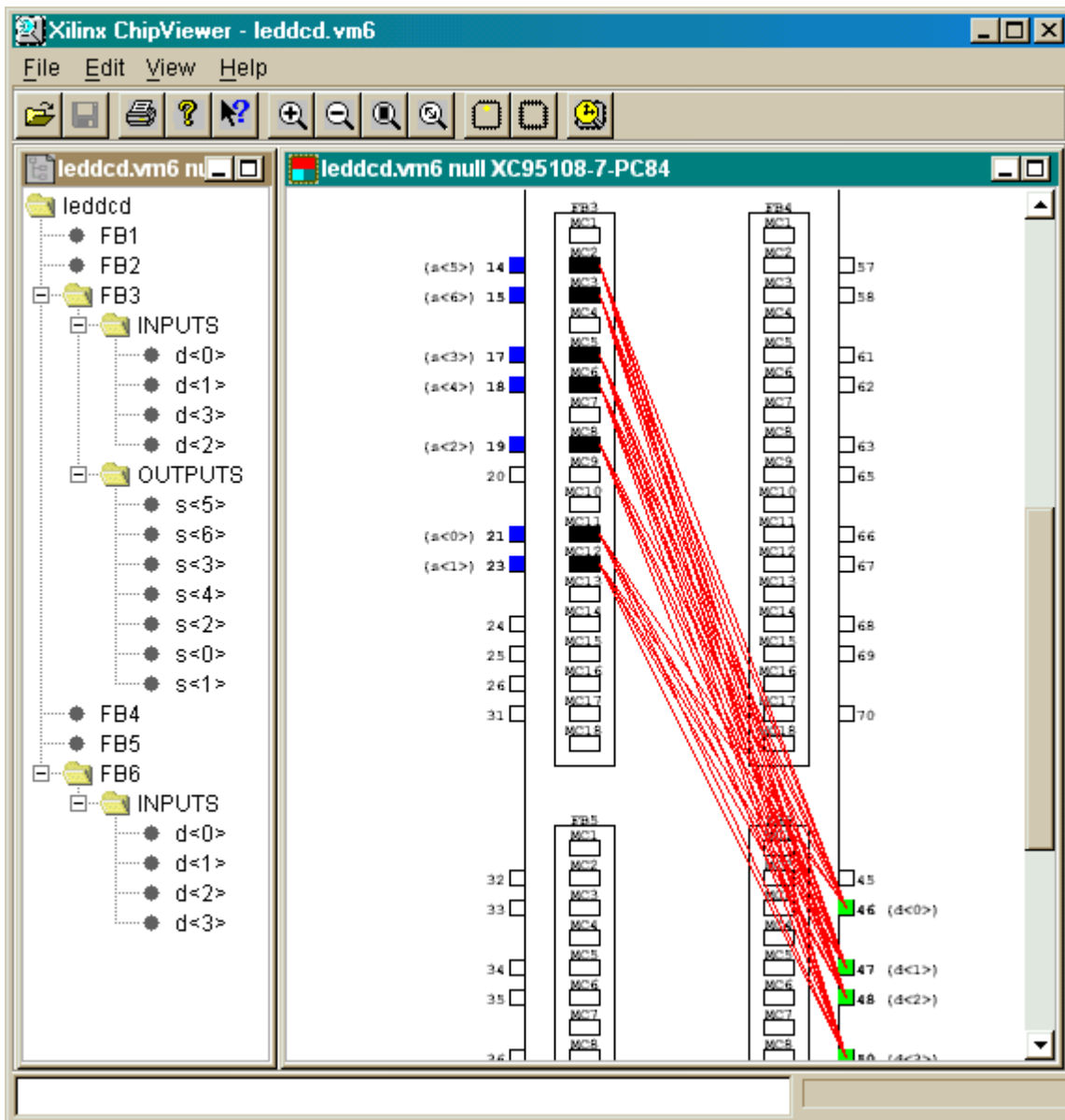
We can see which inputs affect each output by right-clicking in the right-hand pane and selecting the Inputs Connection item from the pop-up menu.



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.

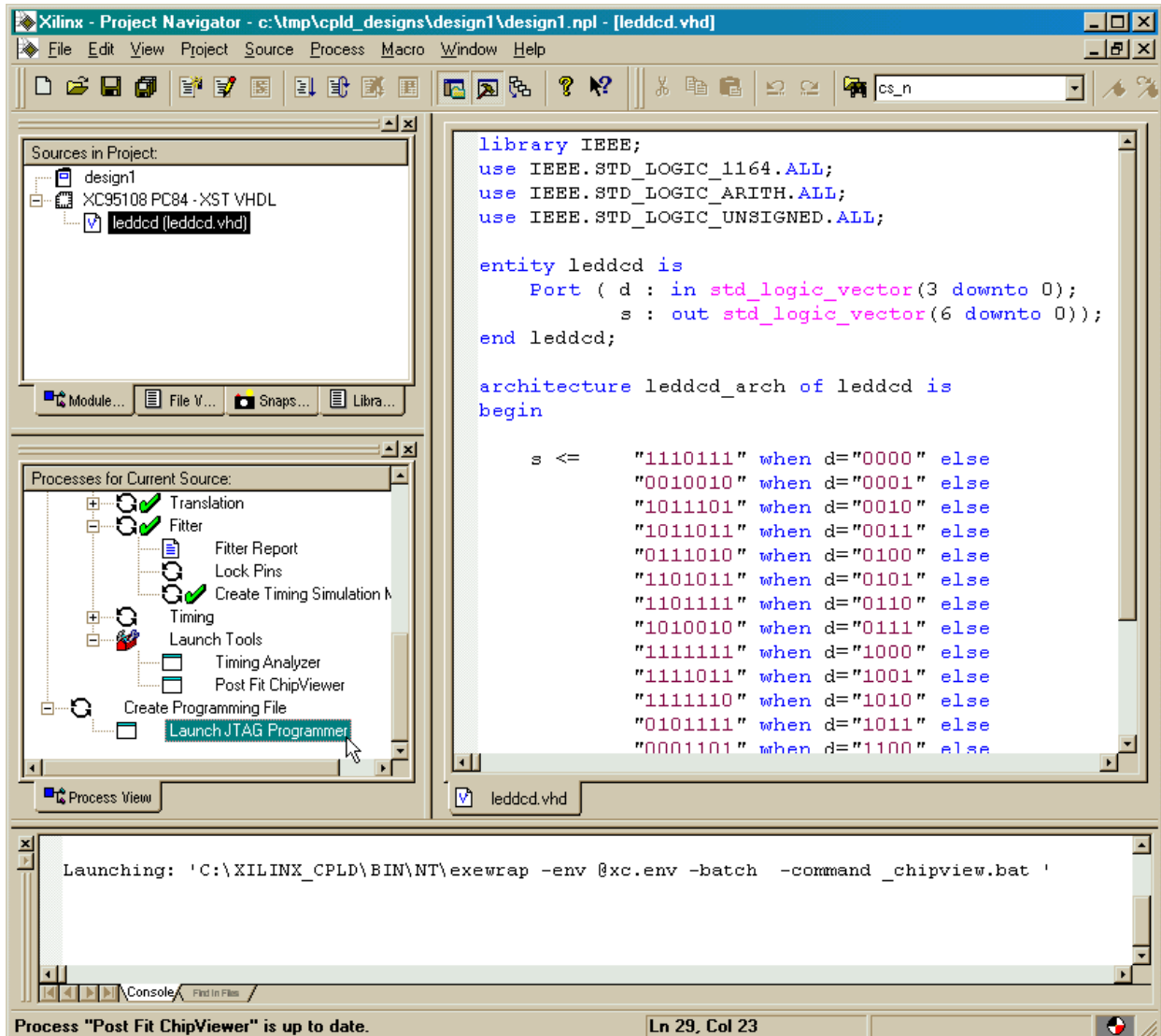


We can see more detail by clicking on the  button several times to expand the right-hand pane. We can see the name of each output and the pin it is assigned to. By placing the mouse pointer over a particular pin, we get some information on the settings for the configuration options for the pin and the 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.

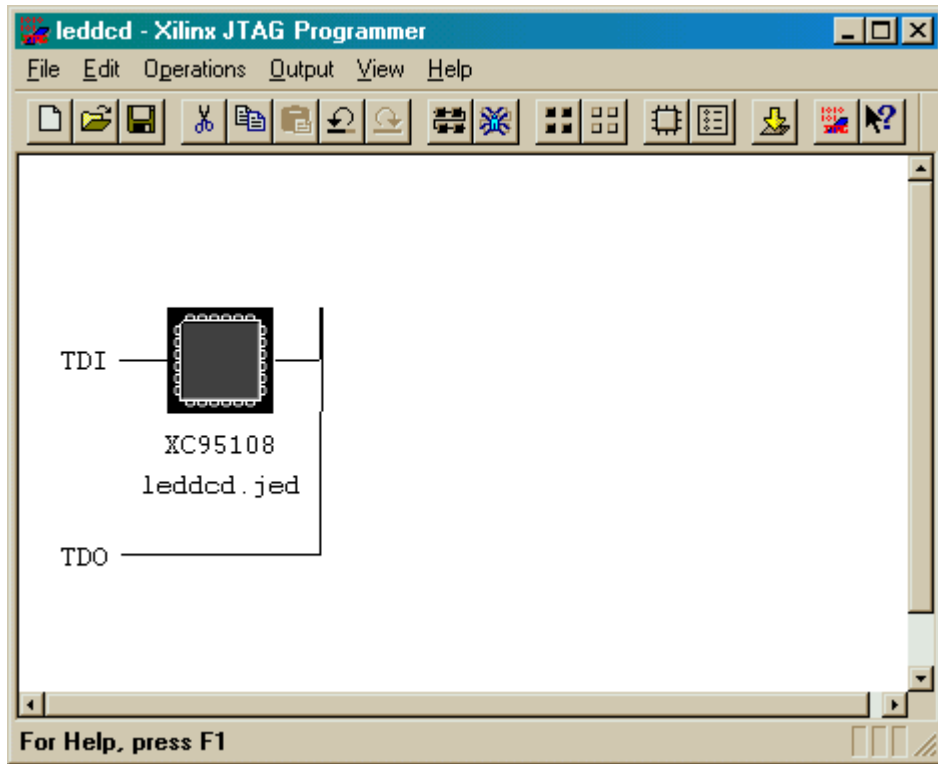


## Generating the Bitstream

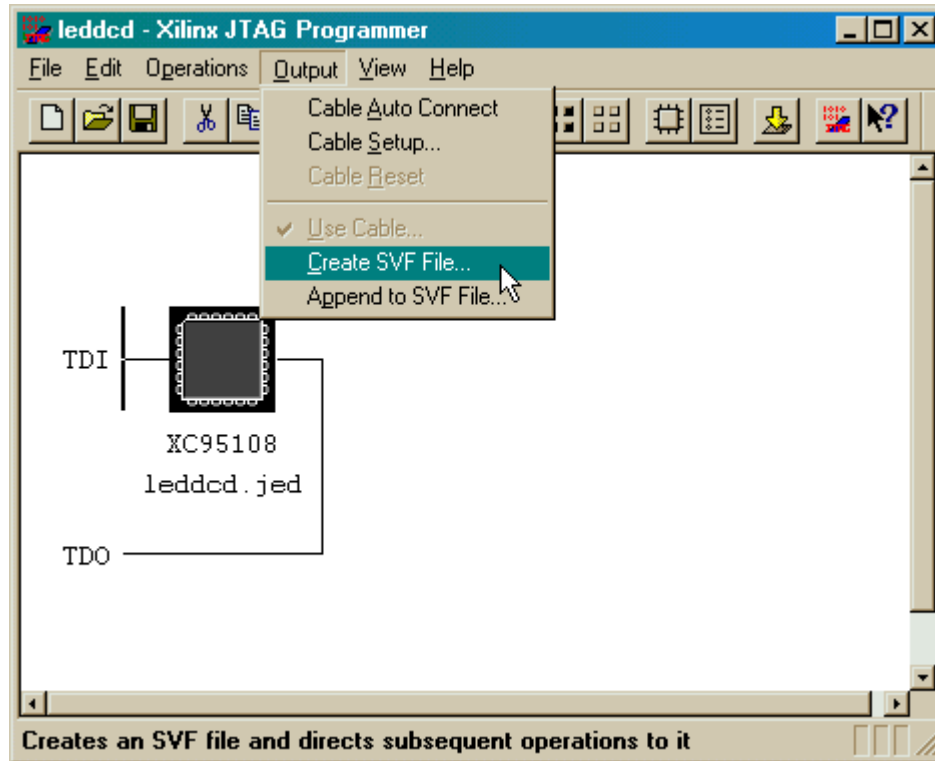
Now that we have synthesized our design and fitted it to the CPLD with the correct pin assignments, we are ready to generate the bitstream that is used to program the actual chip. To initiate the programmer, we highlight the XC95108 PC84 object in the Source pane and double-click on the JTAG Programmer process.



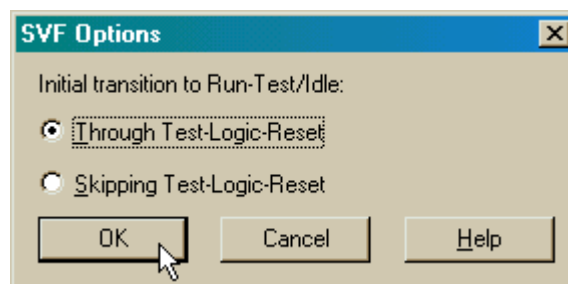
The **JTAG Programmer** window will appear with the chain of chips that are to be programmed. We only have one chip in our LED decoder design, so only one XC95108 CPLD is shown.



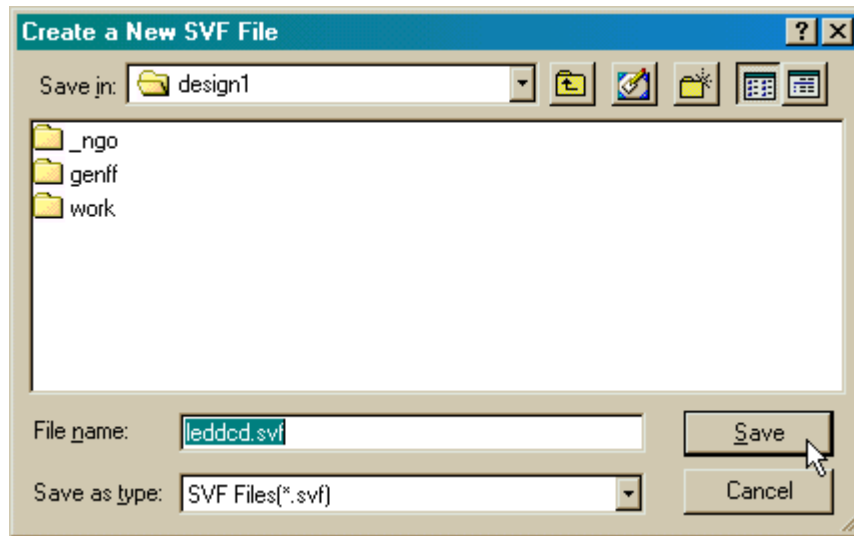
We proceed by selecting the destination for the bitstream. The XS95 Board has a separate utility called GXLOAD for programming the CPLD, so we need to store the bitstream into a file that GXLOAD can read. To do this, select Output⇒Create SVF File... as follows.



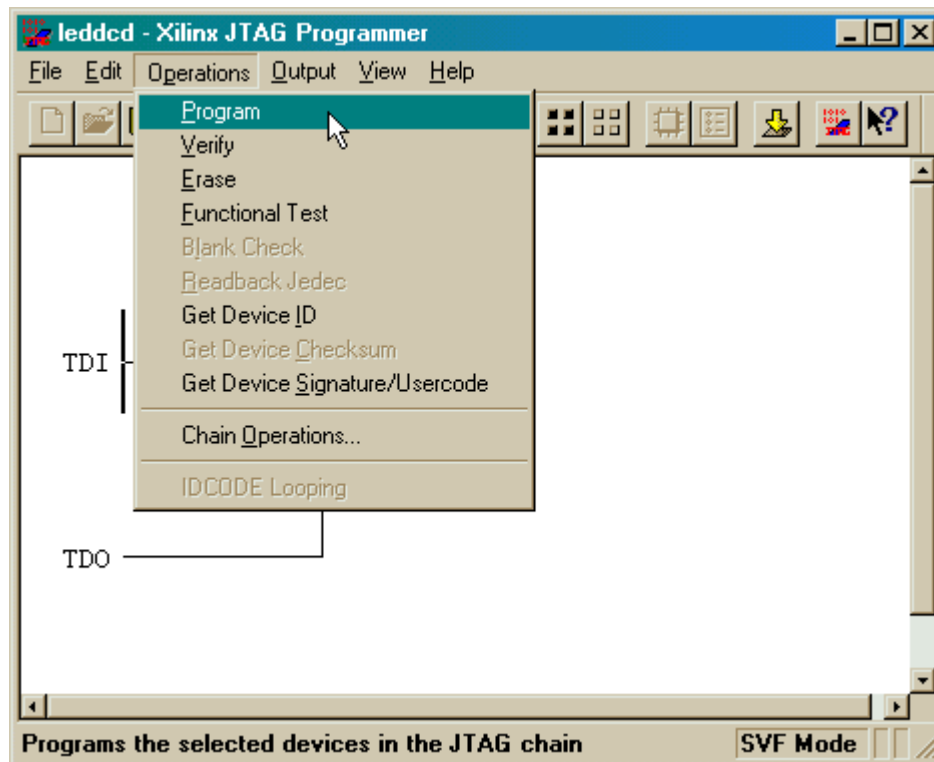
This brings up the **SVF Options** window which asks us to make a choice about the initialization sequence of the JTAG state machine that controls the programming process. Don't even worry about this. Just accept the default and click on OK.



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 leddcd.svf.



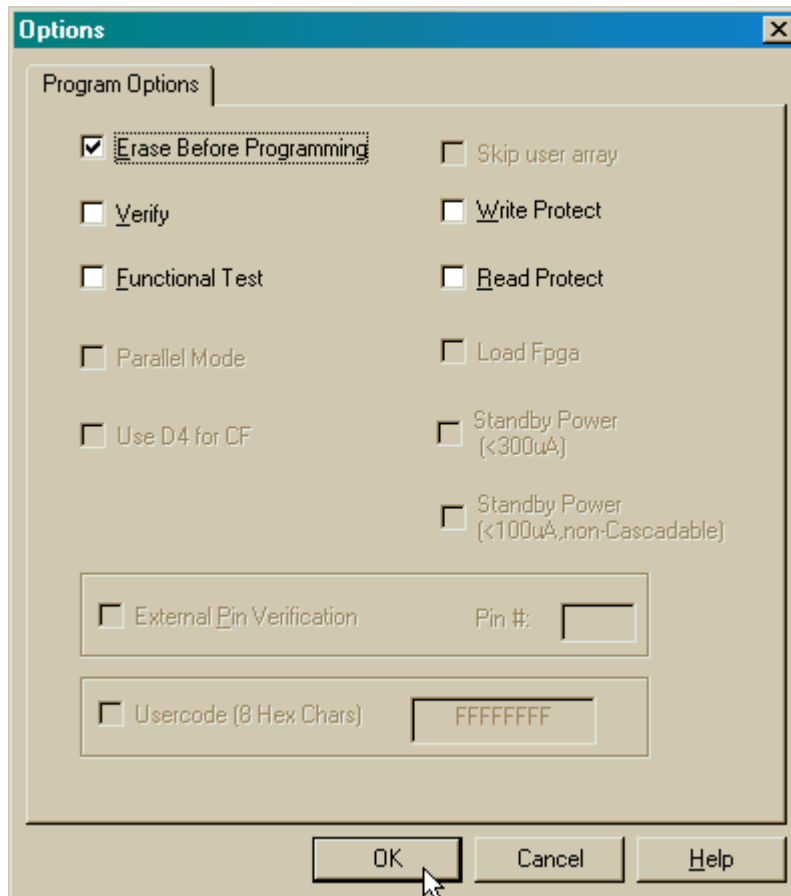
At this point we can generate the bitstream that will be stored in the leddcd.svf file. Click on the Operations menu item and select Program from the drop-down list. This initiates the actual bitstream generation process.



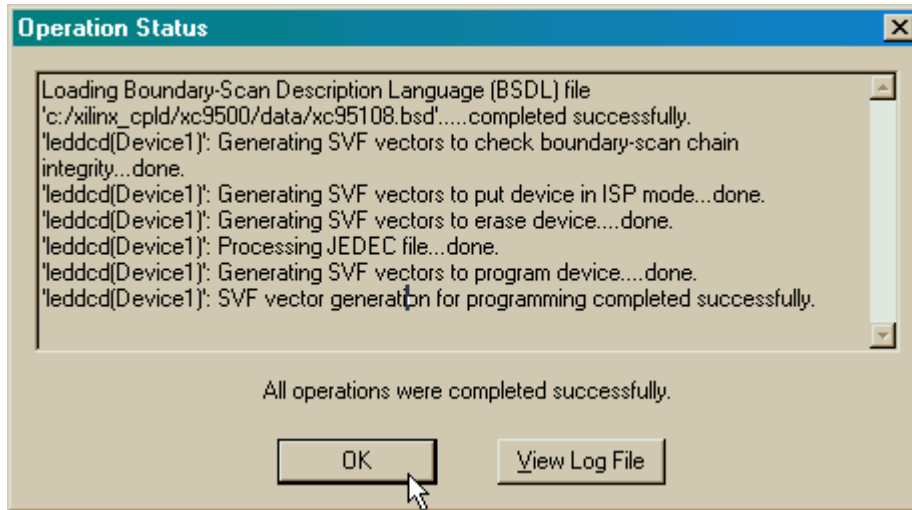
An **Options** window appears where we can set some options for the generated bitstream. We usually check the Erase Before Programming box so that the Flash storage in the CPLD will be erased before we start loading a new design. (The only time we

can leave this box unchecked is when we are programming a CPLD which we know is already erased.)

The other two options which 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.



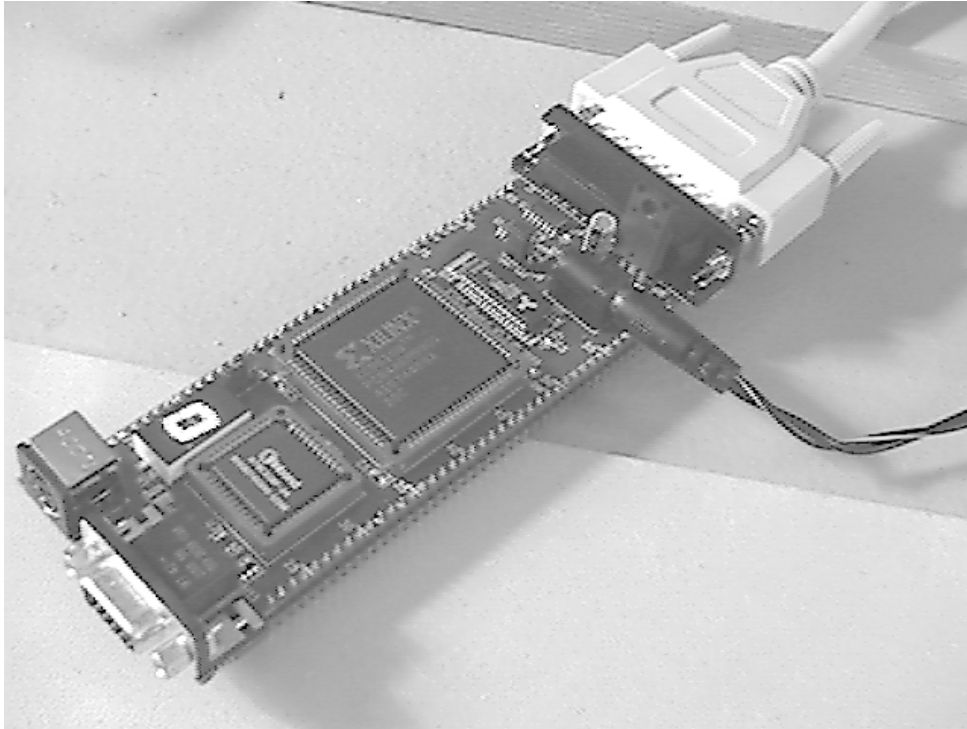
Once we click OK in the **Options** window, the bitstream generation process begins. The progress is reported in the **Operation Status** window:



Once the bitstream file is generated, we click on OK to close the window. Then we close the JTAG Programmer 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.



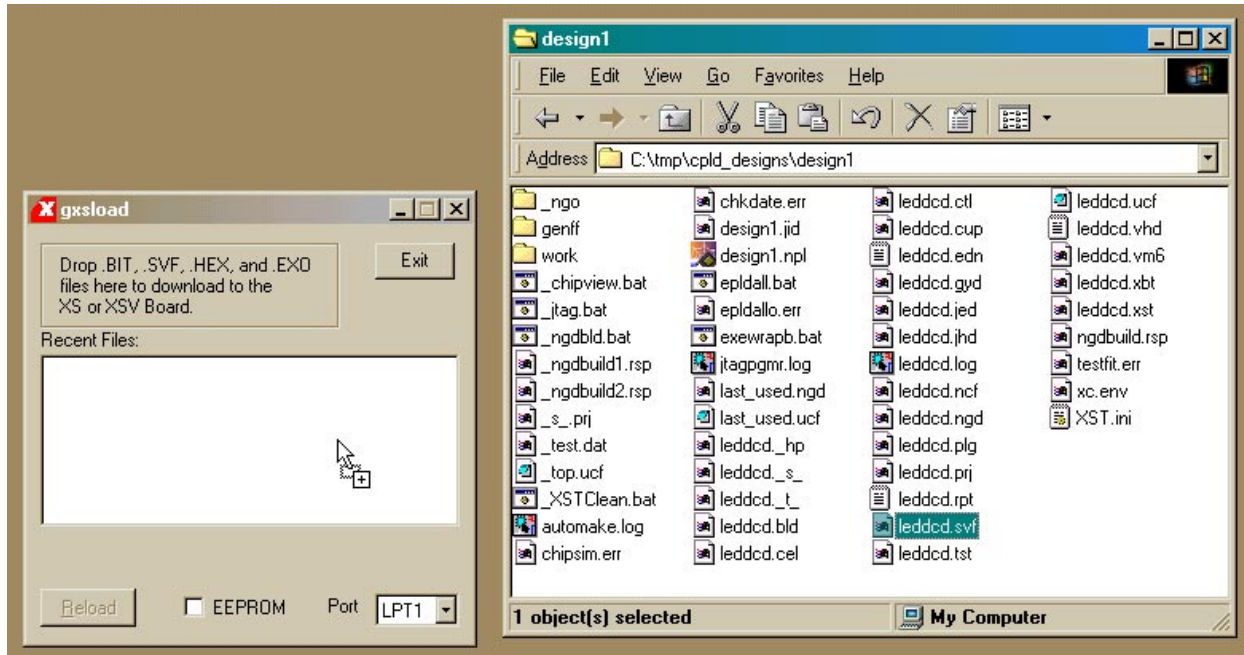
The XS95 Board is programmed using the GXSLD utility. We double click the



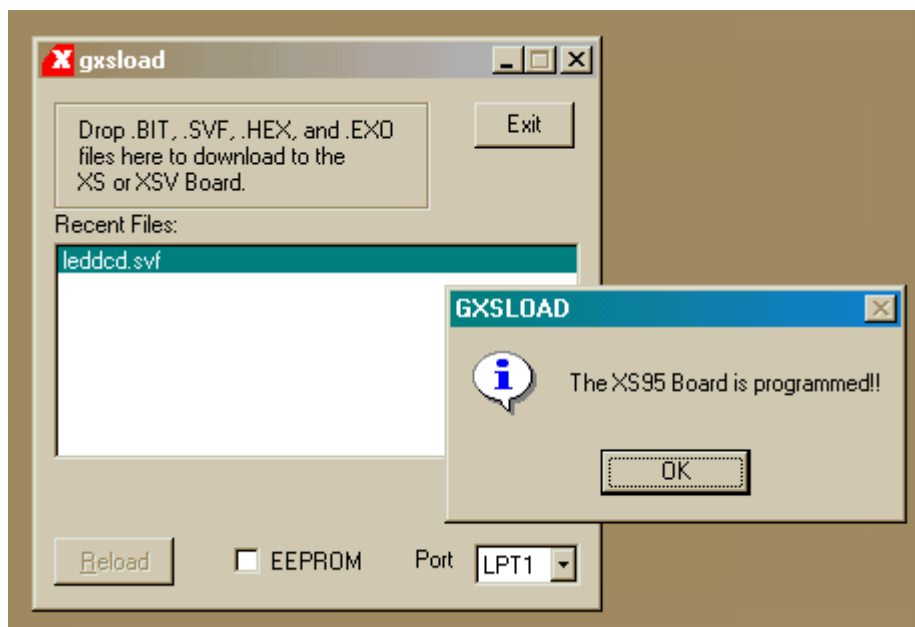
GXSLD icon to bring up the following window:



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).



To initiate the programming of the XS95 Board, we just drag the leddcd.svf file from the **design1** window into the **gxslod** window. Programming the Flash of the XC95108 CPLD takes about a minute.

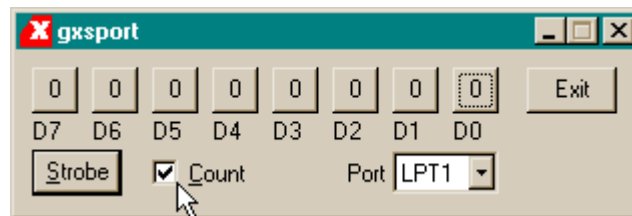


## Testing the Circuit

Once the XC95108 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 **GXSPORT** 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.



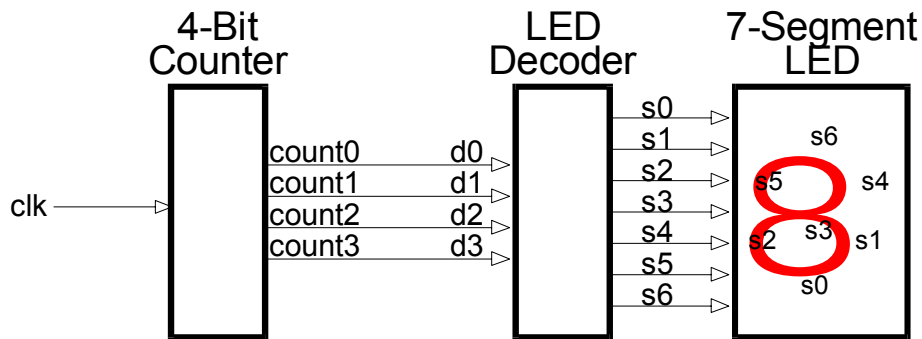
If you check the Count box in the **gxsport** window, then each click on the Strobe button increments the eight-bit value represented by D7-D0. This makes it easy to check all sixteen input combinations.

## 4

# Hierarchical Design

## A Displayable Counter

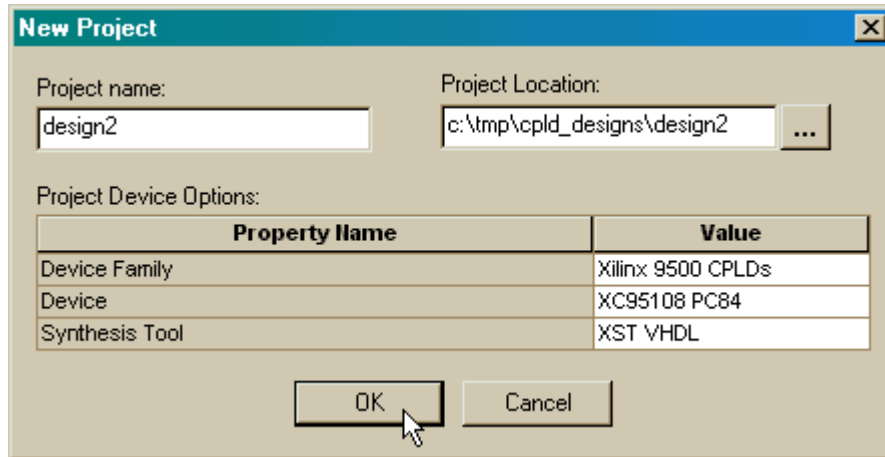
We went through a lot of work for our first CPLD design, so we will reuse it in this design: a four-bit counter whose value is displayed on a seven-segment display. The counter will increment on every rising edge of the clock. The four-bit output from the counter enters the LED decoder whereupon the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



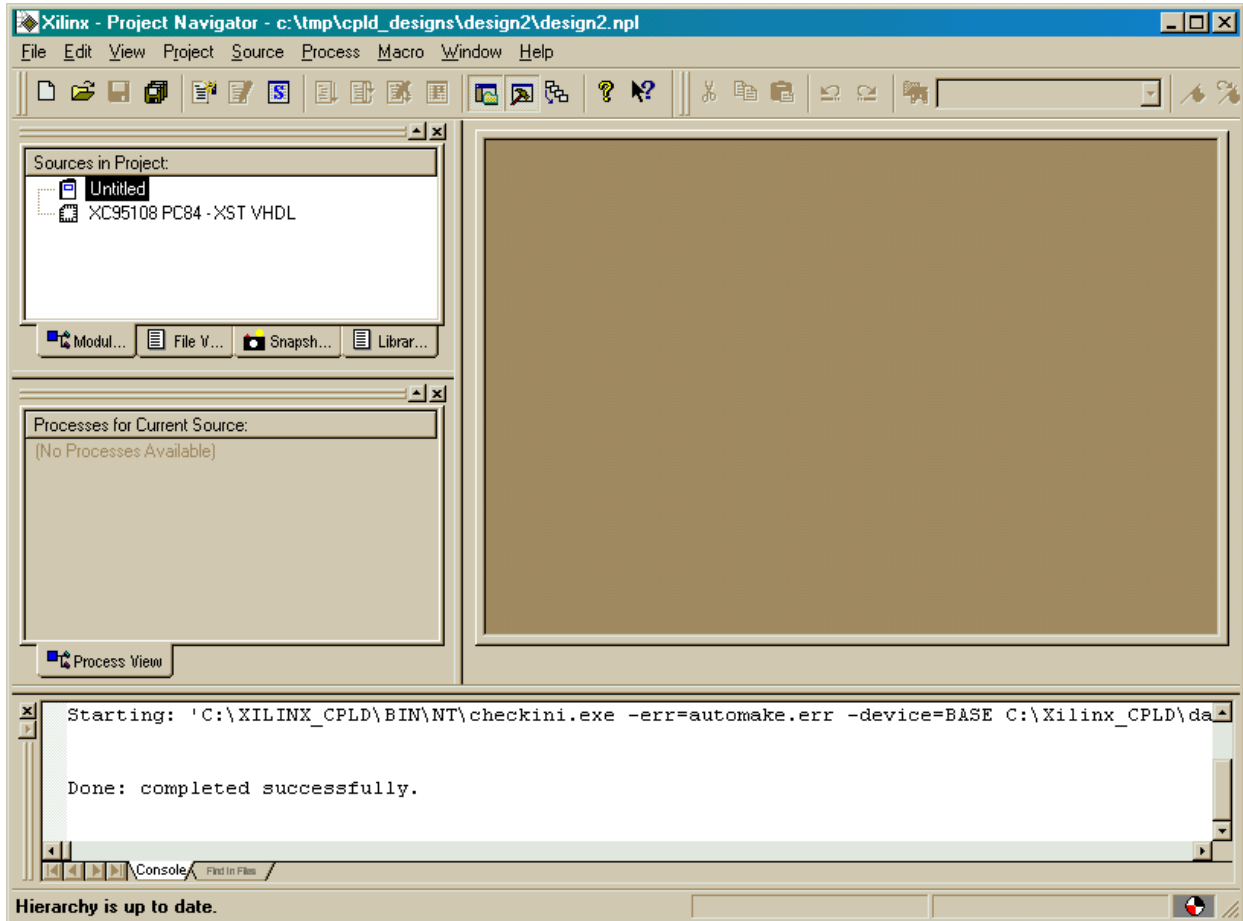
This design is hierarchical in nature. The LED decoder and counter are modules which are interconnected within a top-level module.

## 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.

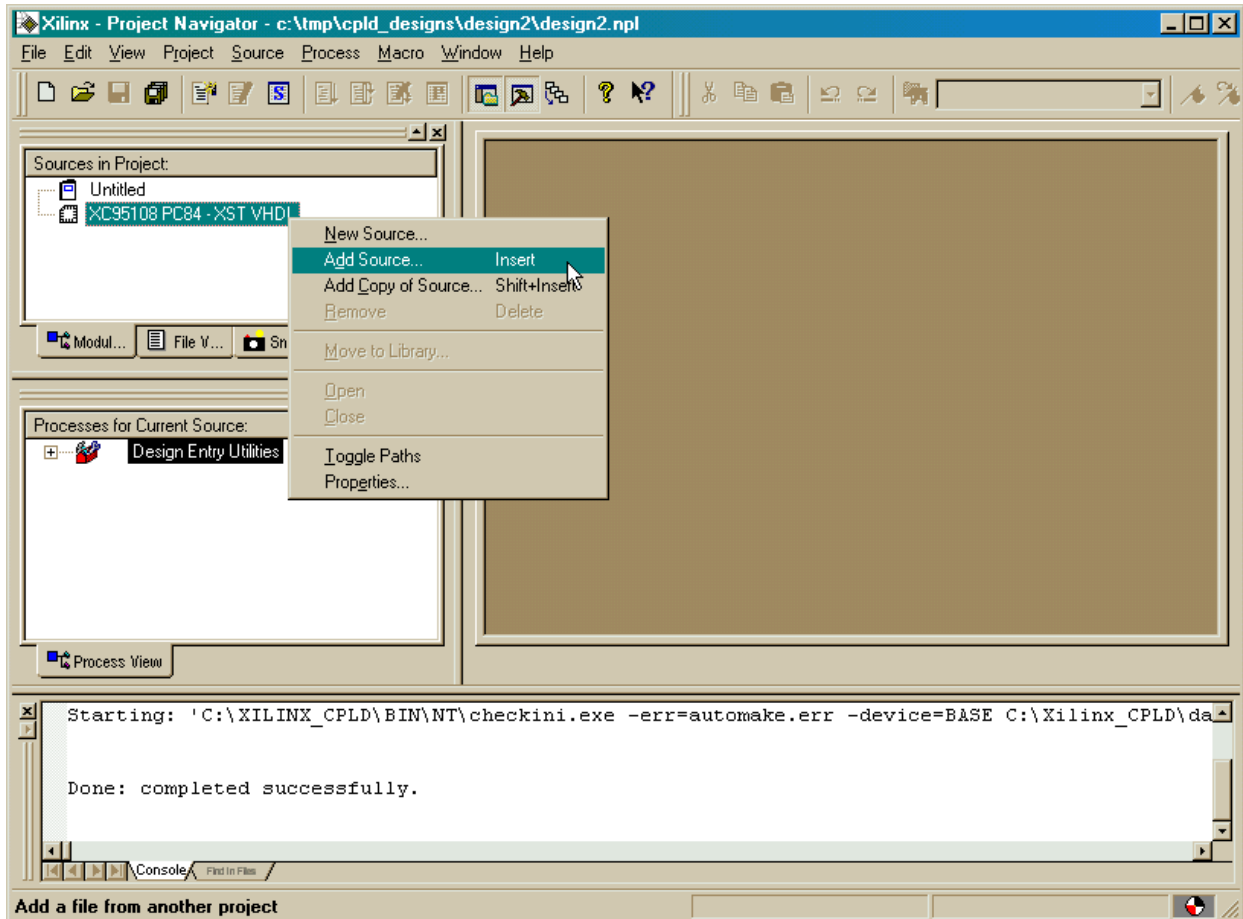


Once we click on OK in the **New Project** window, the **Project Navigator** window appears as shown below.

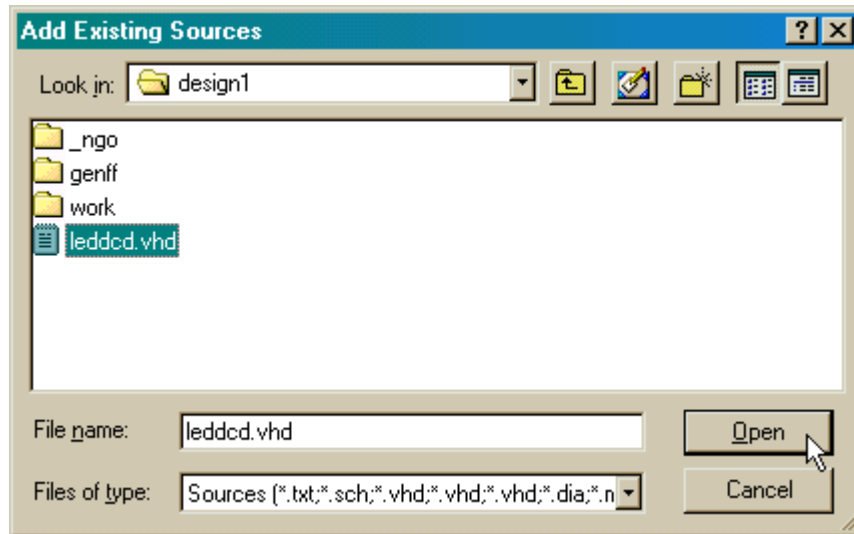


## Adding the LED Decoder

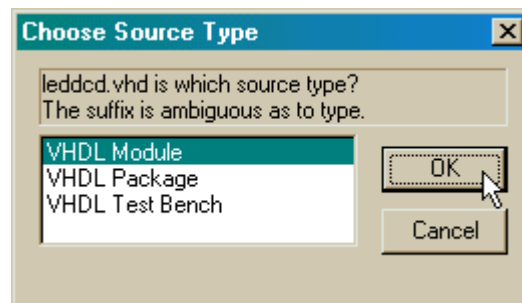
The first thing we do after getting the **design2** project started is to add the LED decoder module. We do this by right-clicking on the XC95108 PC84 object in the Source pane and selecting Add Source ... from the pop-up menu.



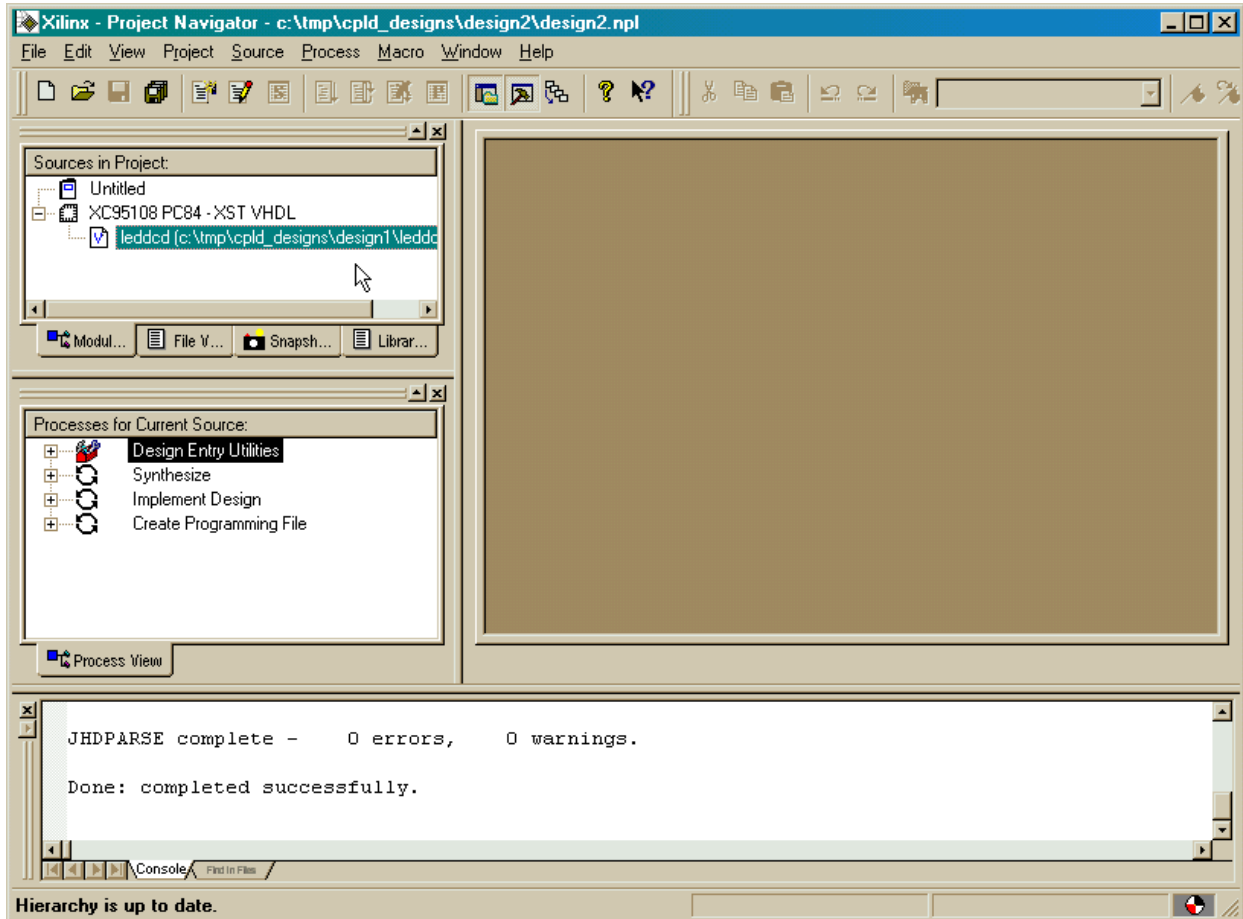
The **Add Existing Sources** window appears and we move to the C:\tmp\cpld\_designs folder. Then we highlight the leddcd.vhd file that contains the VHDL source code for the LED decoder.



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

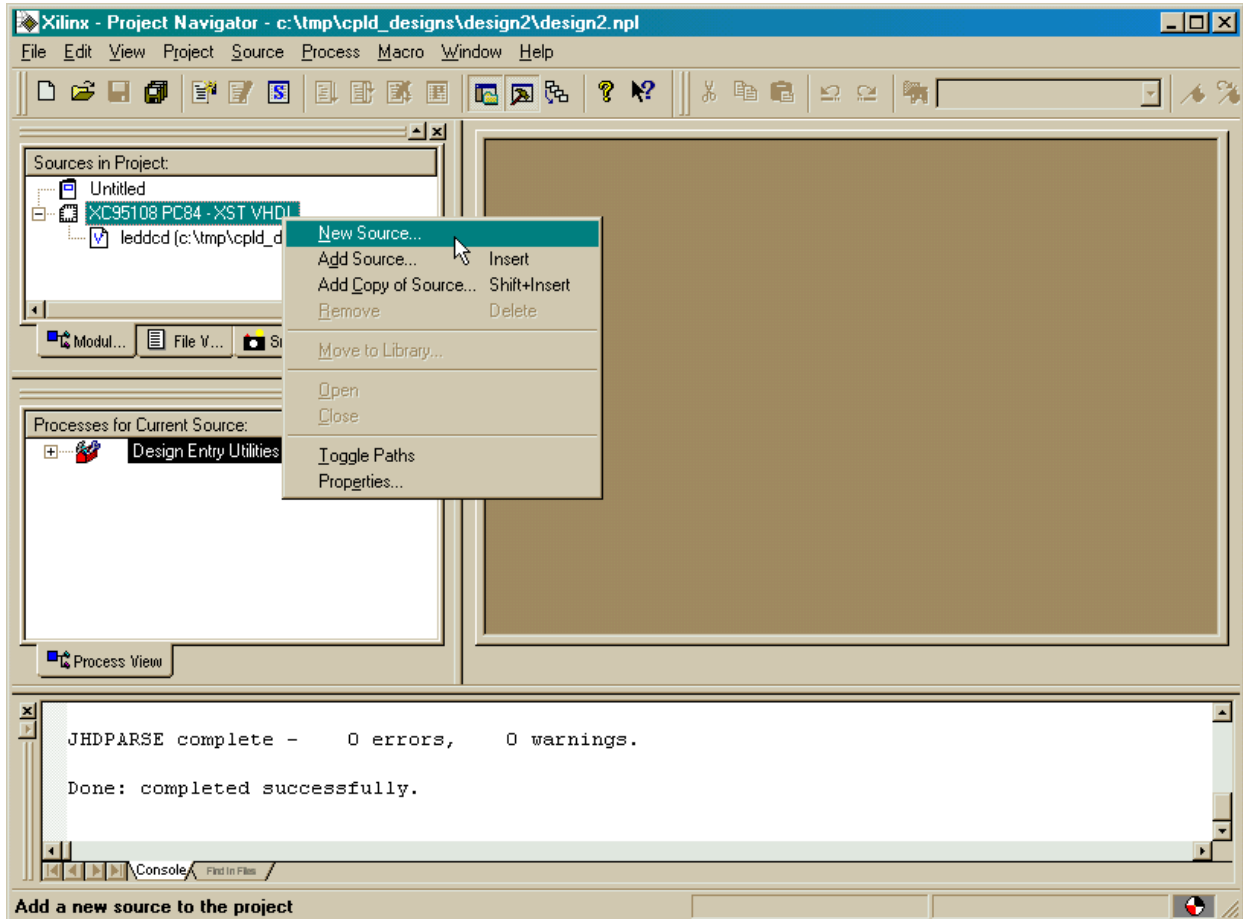


We select VHDL Module since the leddcd.vhd file contains a standard VHDL description of a circuit. (Packages contain extra syntactical elements for modules meant to be used as a library. Test benches contain VHDL code that exercises other VHDL modules through a sequence of tests.) After clicking OK,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

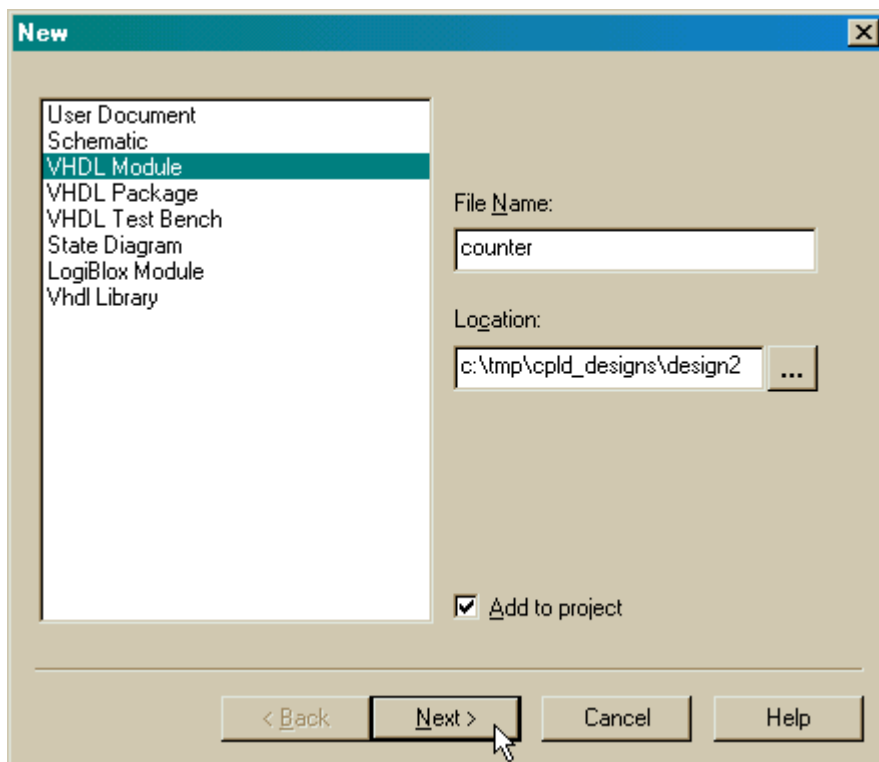


## Adding a Counter

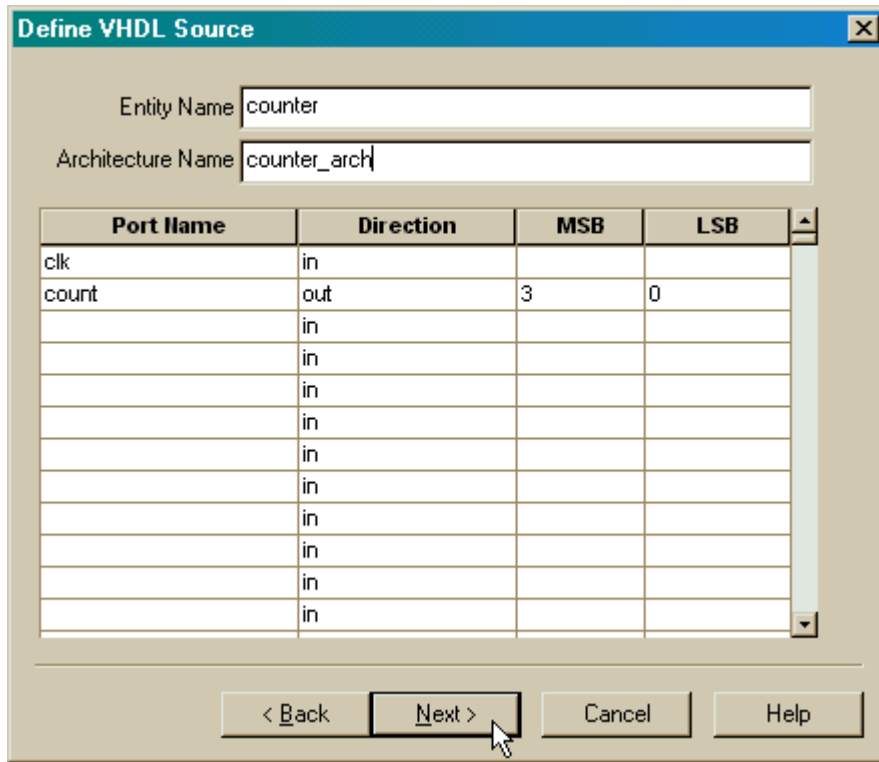
Now we have to add the counter to our design. We don't have a counter module yet, so we have to build one with VHDL. Right-click on the XC95108 PC84 object and select New Source... from the pop-up menu.



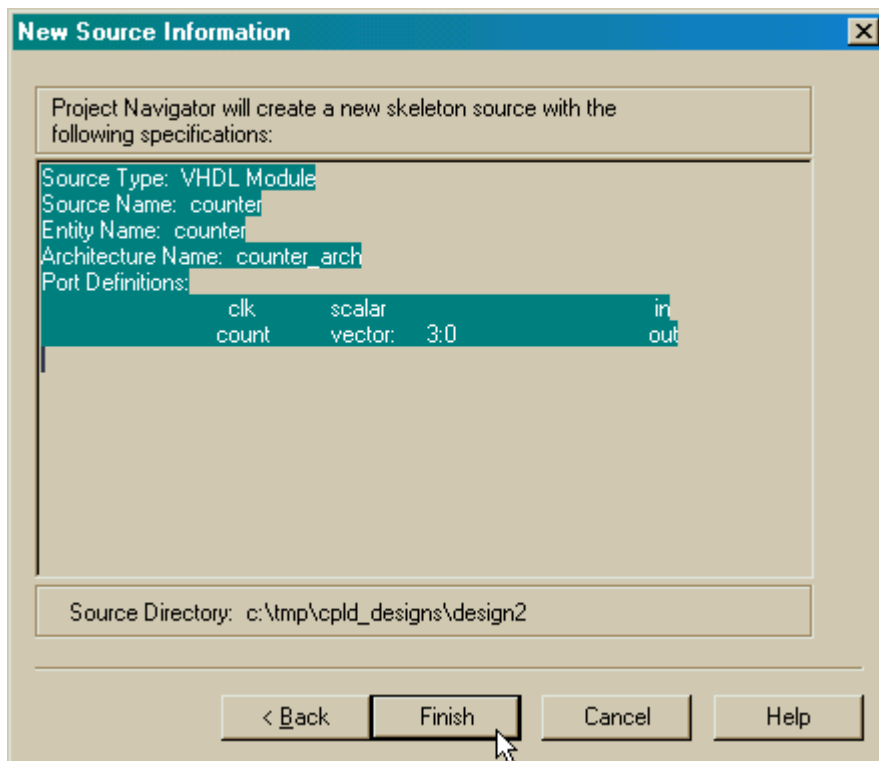
As in the previous example, we are prompted for the type of file we want to add to the project. Once again, we select the VHDL Module menu item. Then we type `counter` into the File Name field and click on the Next button.



Then we type in the entity and architecture names into the **Define VHDL Source** window as shown below. The **counter** module receives a single input, **clk**, and has a four-bit 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 four-bit 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;

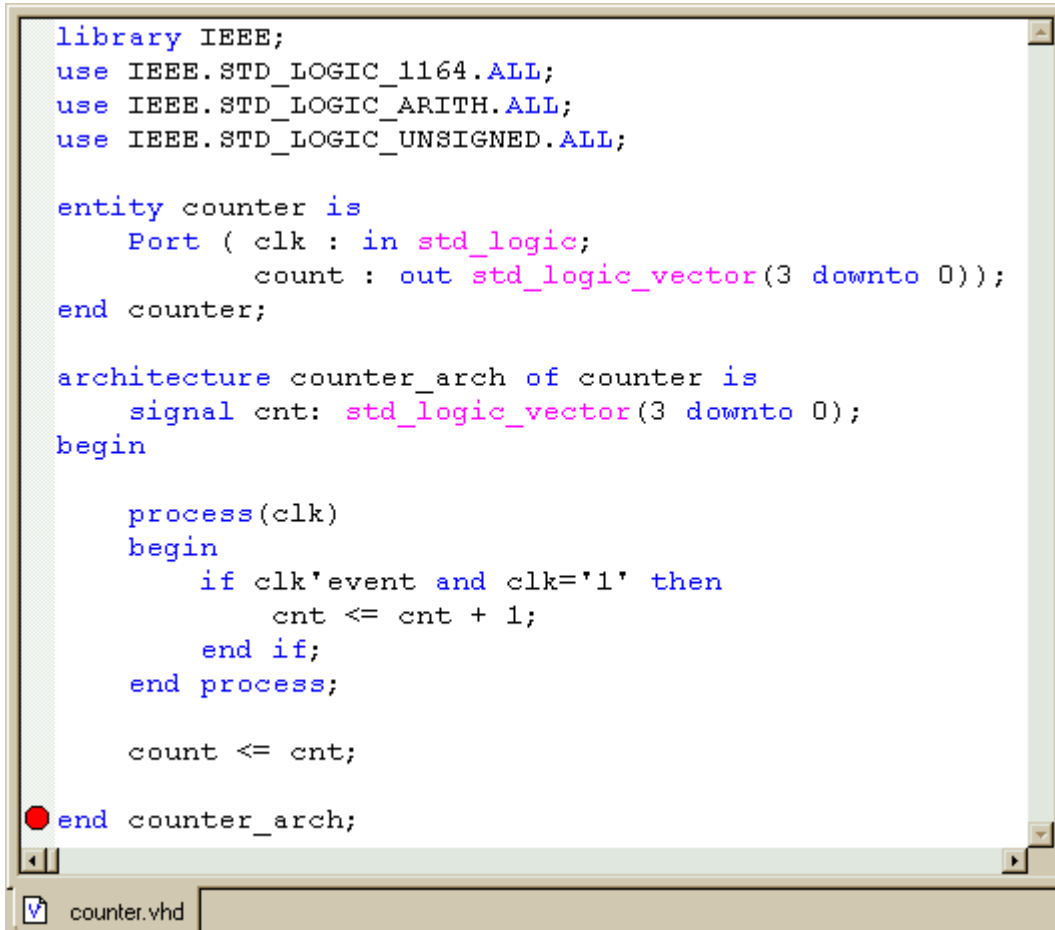
entity counter is
    Port ( clk : in std_logic;
          count : out std_logic_vector(3 downto 0));
end counter;

architecture counter_arch of counter is
    signal cnt: std_logic_vector(3 downto 0);
begin

    process(clk)
    begin
        if clk'event and clk='1' then
            cnt <= cnt + 1;
        end if;
    end process;

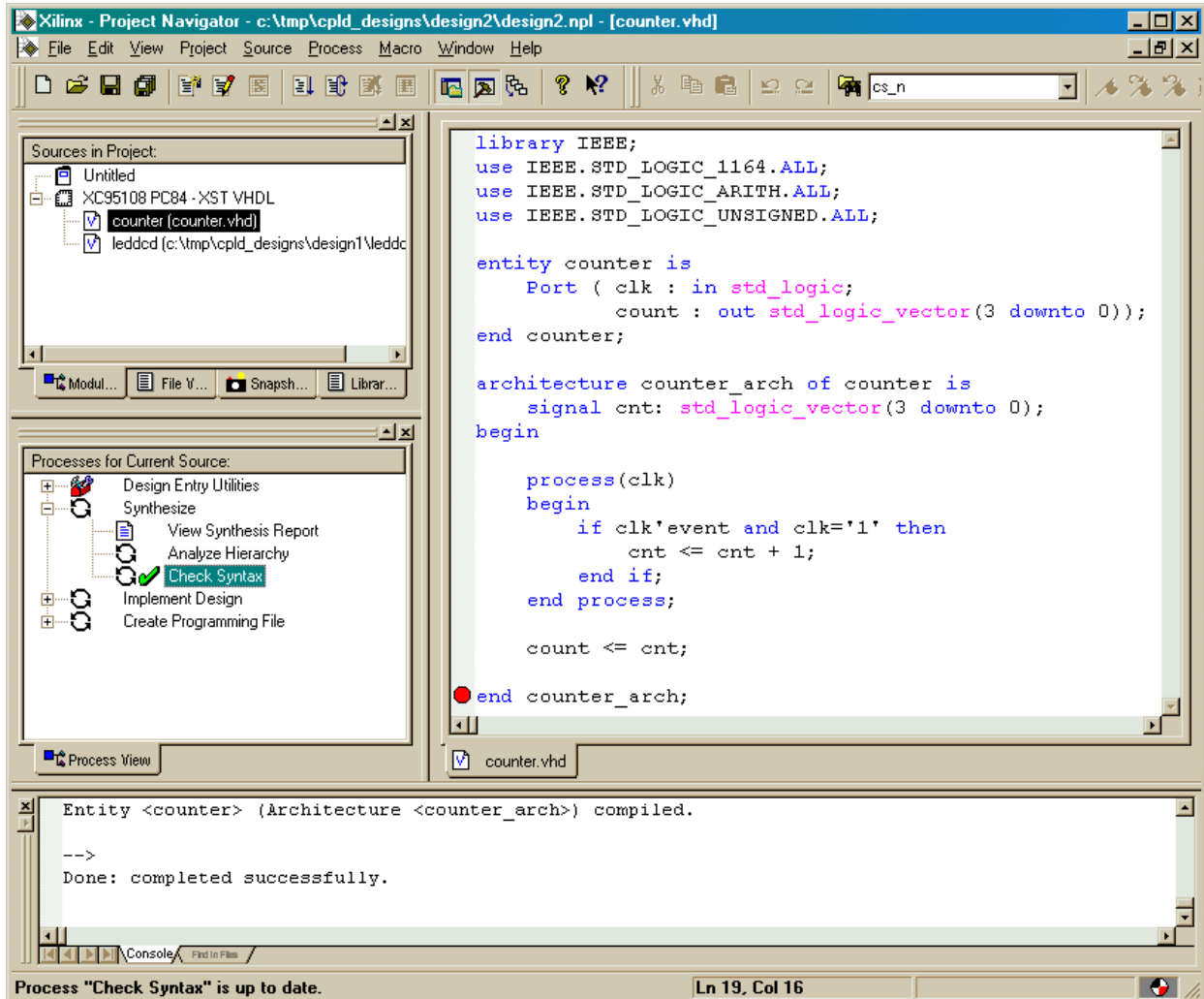
    count <= cnt;

end counter_arch;
```



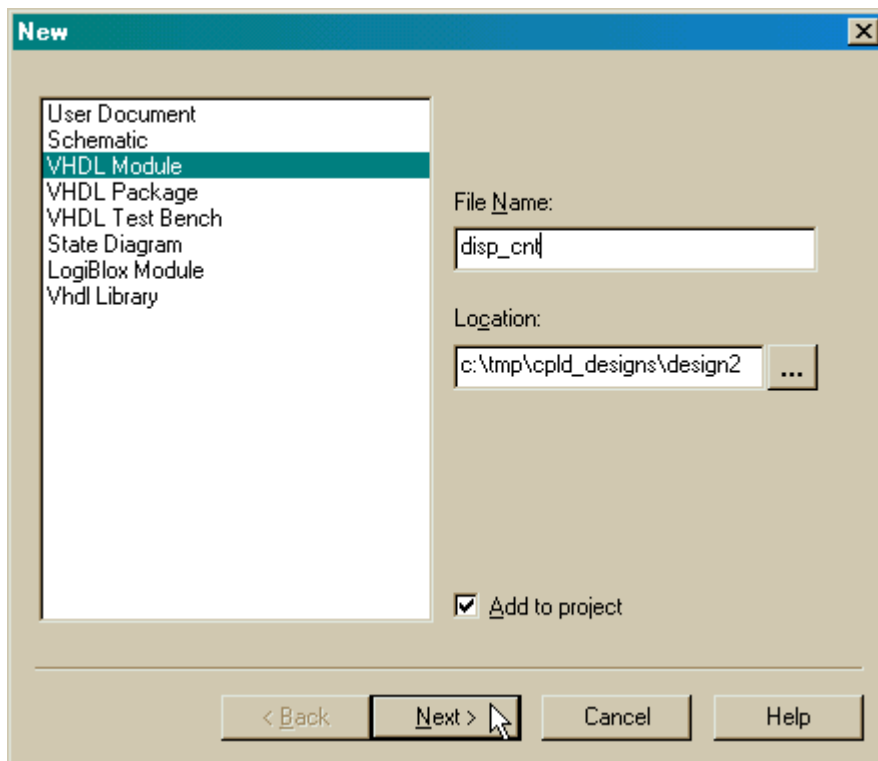
Line 12 declares a 4-bit signal, **cnt**, that is the current value of the counter. The process on lines 15-20 controls when counter increments. The condition clause of line 17 is only true when the value on the **clk** input goes from 0 to 1. Then the statement on line 18 replaces the value in **cnt** with its incremented value. (We can use the high-level addition operator instead of having to describe a four-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 current counter value onto the outputs of the module.

After entering the VHDL shown above and saving it, we see that the counter module has been added to the Source 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. To do this, we will create a new top-level VHDL module in the `disp_cnt.vhd` file.



The module will have the entity and architecture names shown below. The top-level module only needs a single clock input and seven outputs to drive the LED digit.

The dialog box titled "Define VHDL Source" contains the following information:

Entity Name:

Architecture Name:

Port Name	Direction	MSB	LSB
clk	in		
s	out	6	0
	in		
	in		
	in		
	in		
	in		
	in		
	in		
	in		
	in		
	in		

Buttons: < Back, Next >, Cancel, Help

Then we fill-out the resulting VHDL skeleton so it reads as follows:

```

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

entity disp_cnt is
    Port ( clk : in std_logic;
          s : out std_logic_vector(6 downto 0));
end disp_cnt;

architecture disp_cnt_arch of disp_cnt is
    component counter
        port(clk: in std_logic;
             count: out std_logic_vector(3 downto 0));
    end component;
    component leddec
        port(d: in std_logic_vector(3 downto 0);
             s: out std_logic_vector(6 downto 0));
    end component;
    signal cnt: std_logic_vector(3 downto 0);
begin

    u0: counter port map(clk=>clk, count=>cnt);
    u1: leddec port map(d=>cnt, s=>s);

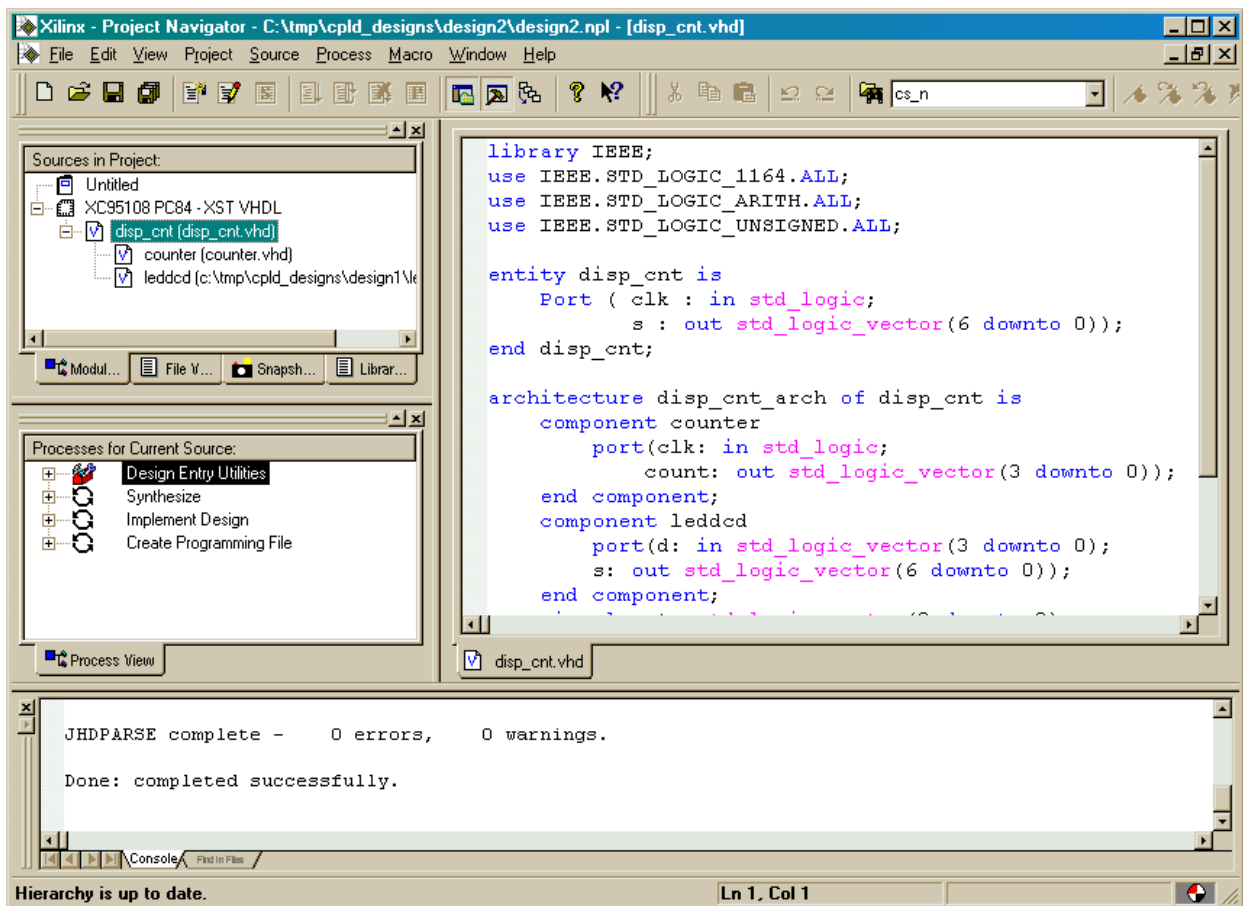
end disp_cnt_arch;

```

Within the architecture section of the **disp\_cnt** module, we declare the I/O interface of the counter and LED decoder components (lines 12-15 and 16-19, respectively). And on line 20 we declare an internal four-bit signal that carries the value from the counter module to the LED decoder.

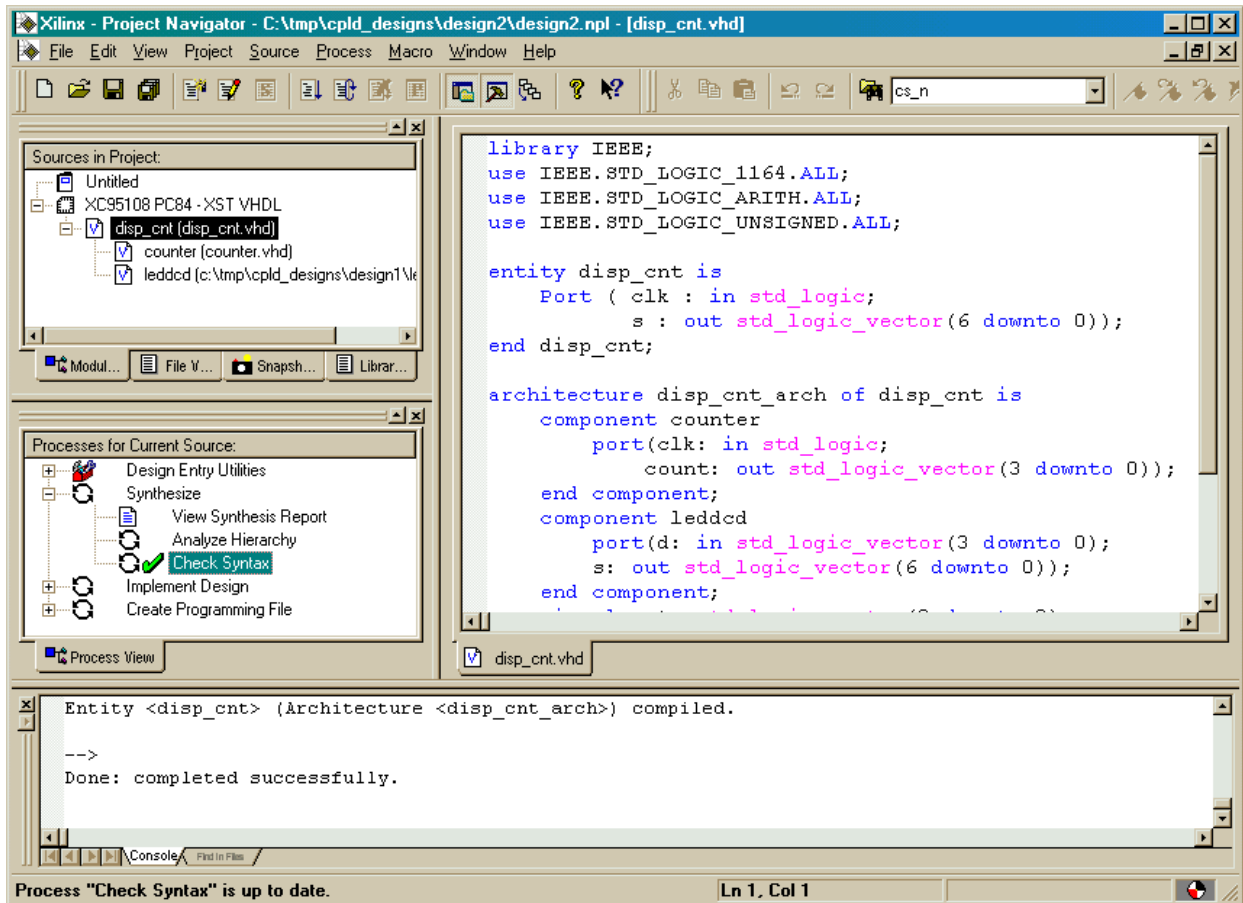
On line 23, we instantiate the **counter** module. The clock input to the **disp\_cnt** module is attached to the clock input of the counter, and the output from the counter is attached to the four-bit **cnt** signal. The **cnt** signal is attached to the input of the LED decoder, and the decoder outputs are connected to the outputs of the **disp\_cnt** module on line 24.



Once we save the VHDL for the top-level module, we see the updated hierarchy in the Source pane of the **Project Navigator** window. Now the **counter** and **leddcd** modules are shown as modules that are included in the **disp\_cnt** module.

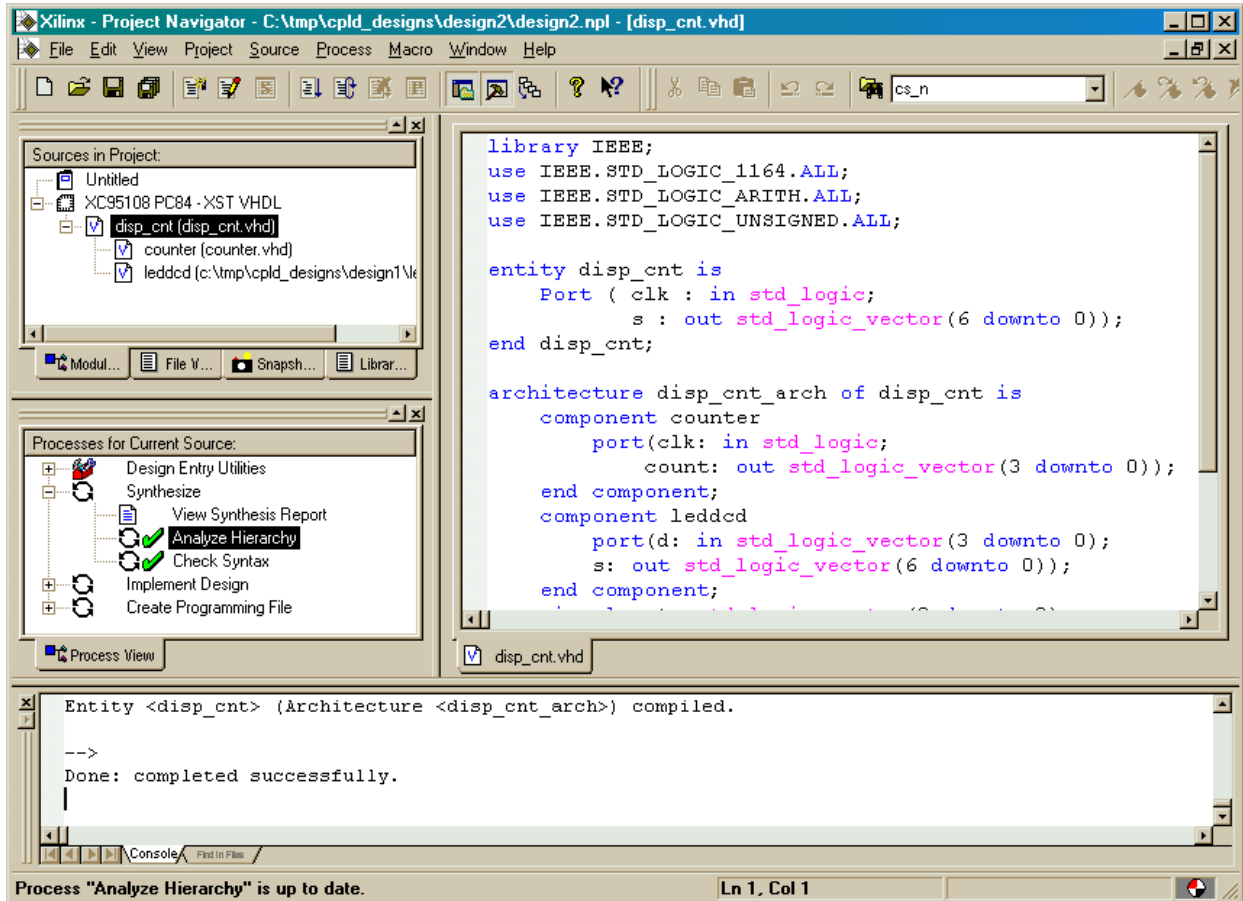


## Checking the VHDL Syntax

We can check the VHDL syntax for the top-level module by highlighting the **disp\_cnt** module in the Source pane and then double-clicking the Check Syntax process.

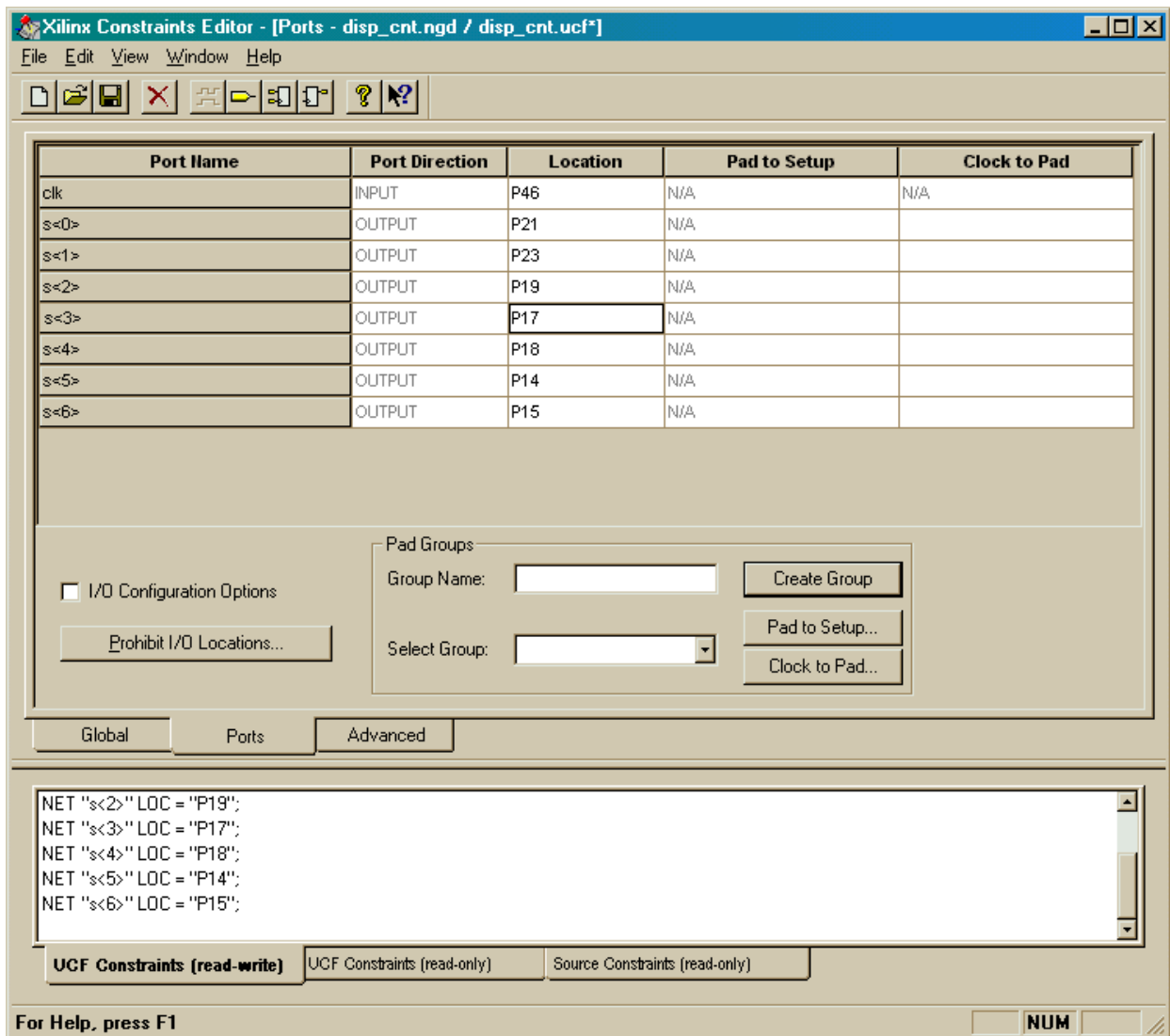


The **disp\_cnt** module is syntactically correct as evidenced by the  next to the Check Syntax process, but this does not mean the VHDL of the **counter** and **leddcd** modules is correct. We can check the entire design by double-clicking the Analyze Hierarchy process. This checks the VHDL for each module and their interconnections with each other. The  that appears after the analysis process completes shows we have no syntax problems in our modules.



## Constraining the Design

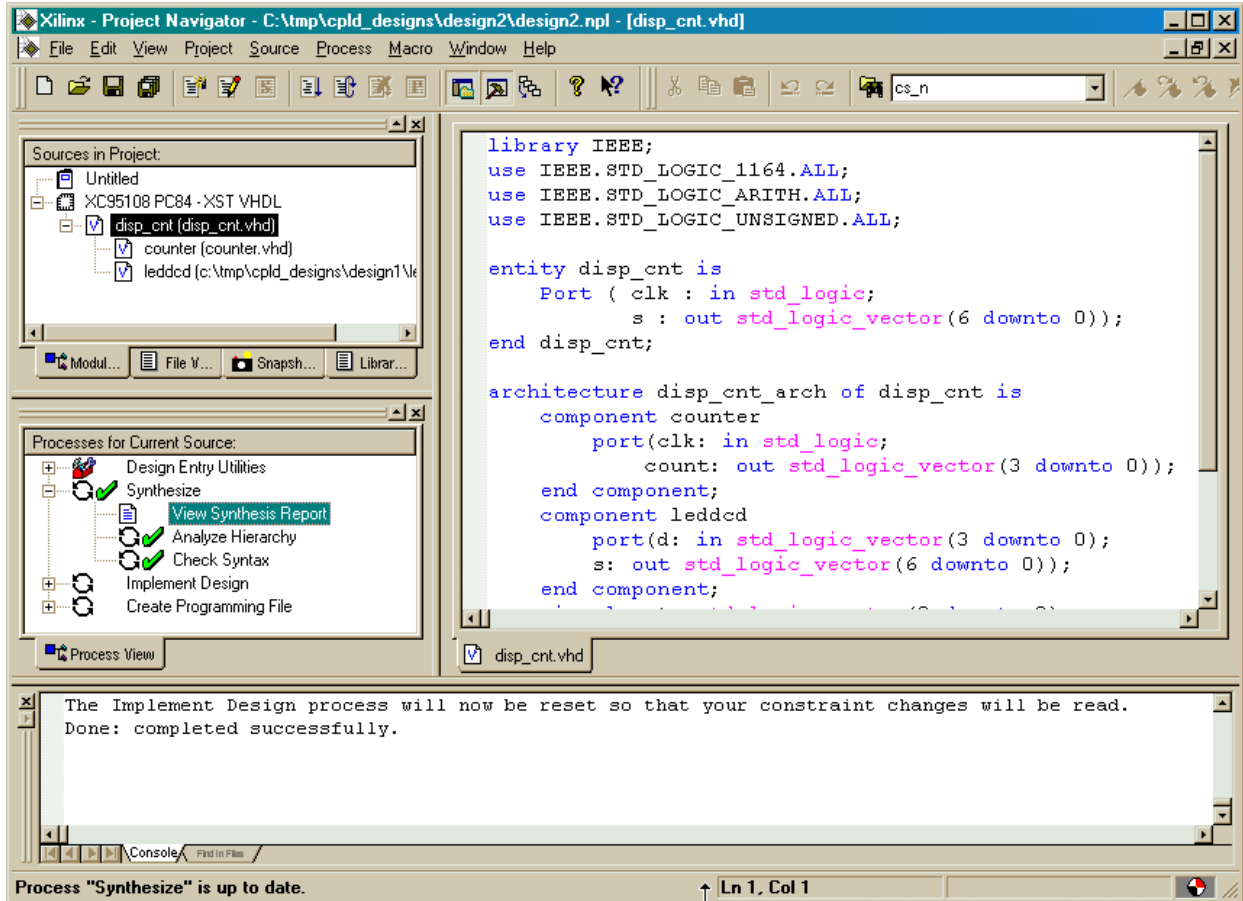
Before synthesizing the displayable counter, we need to assign the pins which the inputs and outputs will use. Highlight the XC95108 PC84 object in the Source pane and then double-click the Constraint Editor process. In the Ports tab of the **Constraint Editor** window, set the pin assignments for the clock input and LED segment drivers as follows:



Assigning the **clk** input to pin P46 lets us use GXSPORT to control the clock through the PC parallel port attached to the XS95 Board. The output assignments connect the displayable counter to the seven-segment LED on the XS95 Board as in the previous design example.

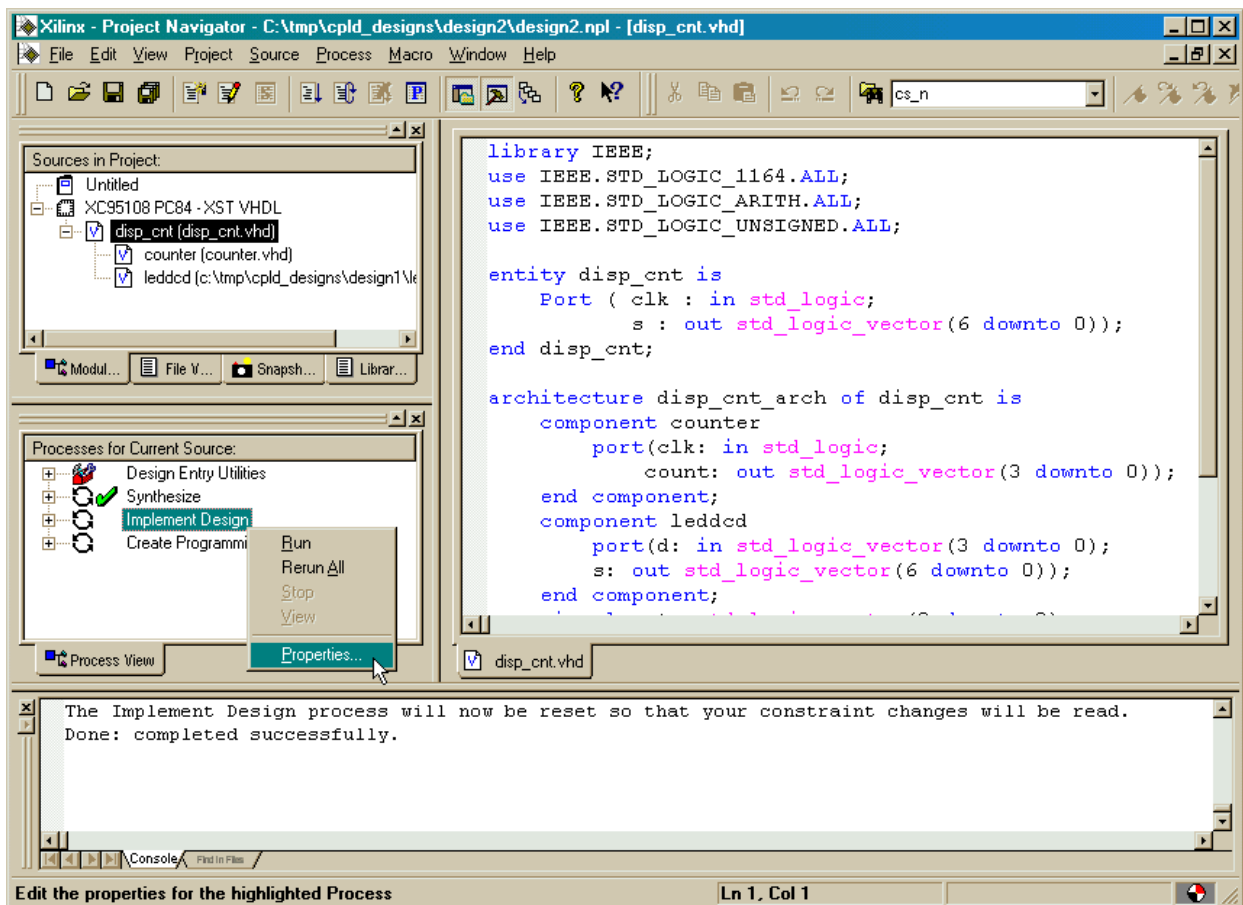
## Synthesizing the Logic Circuitry for the Design

Now we can synthesize the logic circuit netlist by highlighting the top-level `disp_cnt` module in the Source pane and double-clicking the Synthesize process.

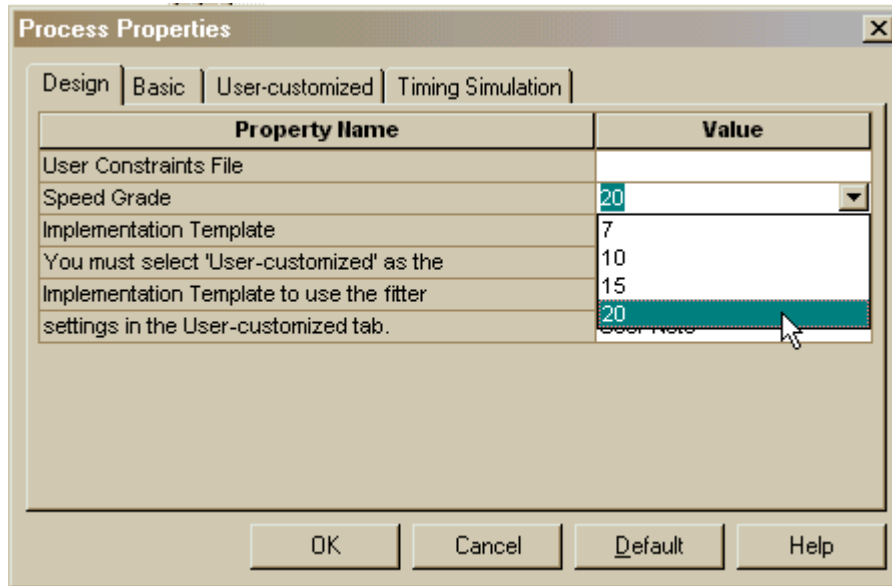



## Fitting the Logic Circuitry Into the CPLD

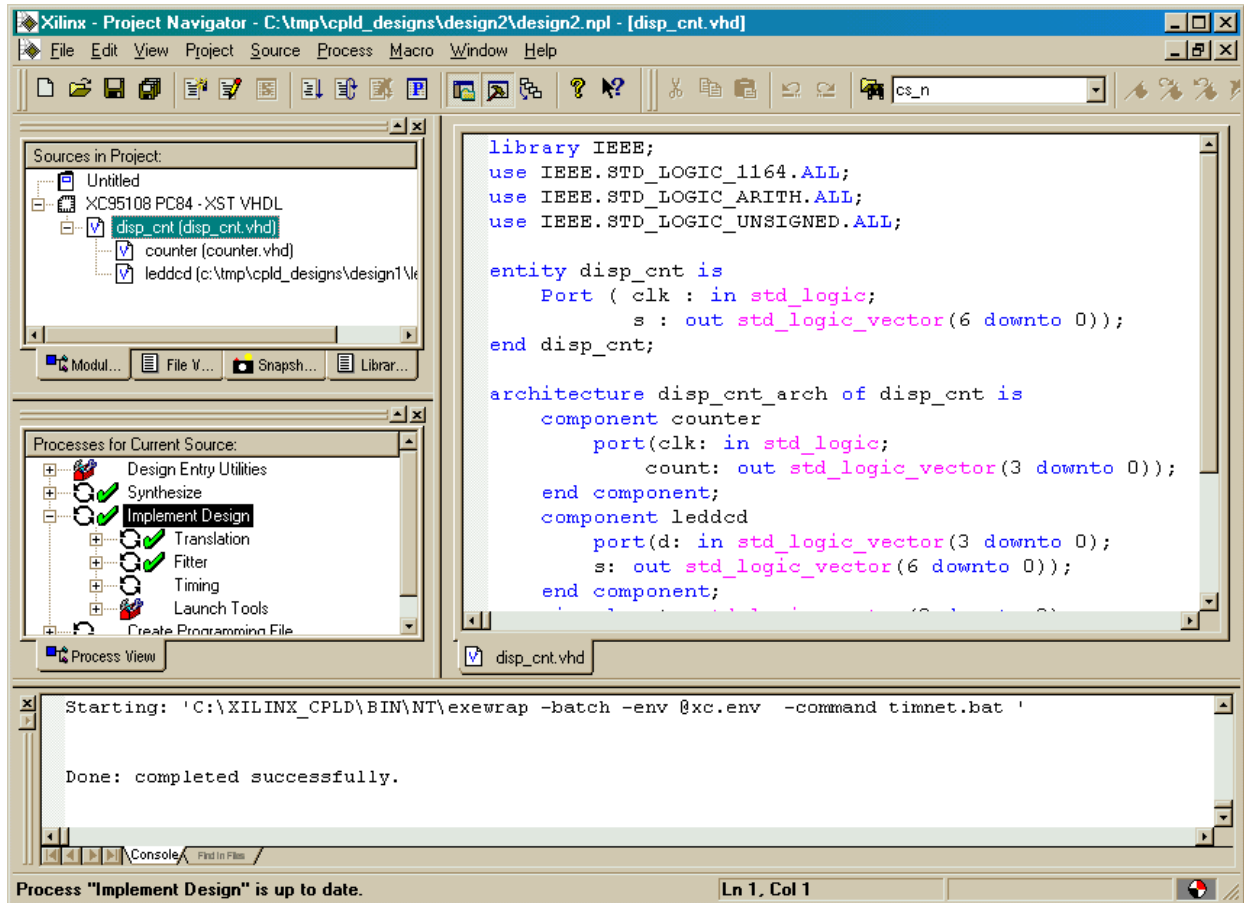
Once the netlist is synthesized, we can begin the process of fitting it into the CPLD. Before activating the fitting process, however, we will give the fitter some information on the speed of the CPLD we are targeting. The XC95108 on the XS95 Board has a -20 speed grade which means that the pad-to-pad 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 move 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:

```
XACT: version D.20                               Xilinx Inc.
                                           Fitter Report
Design Name: disp_cnt                           Date: 8-27-2000, 1:00PM
Device Used: XC95108-20-PC84
Fitting Status: Successful

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

Macrocells      Product Terms   Registers      Pins           Function Block
Used            Used            Used           Used           Inputs Used
11 /108 ( 10%) 32 /540 ( 5%)  4 /108 ( 3%)  8 /69 ( 11%)  5 /216 ( 2%)
```

The displayable counter consumes 11 of the 108 macrocells: four for the four-bit counter and seven 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     Loc      Mode Rate #    Type    Use
N26         2       4       FB3_18  STD                   (b)     (b)
N27         2       3       FB3_17  STD                   31 I/O     (b)
N28         2       2       FB3_16  STD                   26 I/O     (b)
N29         2       2       FB3_15  STD                   25 I/O     (b)
s<0>        4       4       FB3_11  STD  FAST  21  I/O     0
s<1>        3       4       FB3_12  STD  FAST  23  I/O     0
s<2>        3       4       FB3_8   STD  FAST  19  I/O     0
s<3>        2       4       FB3_5   STD  FAST  17  I/O     0
s<4>        4       4       FB3_6   STD  FAST  18  I/O     0
s<5>        4       4       FB3_2   STD  FAST  14  I/O     0
s<6>        4       4       FB3_3   STD  FAST  15  I/O     0

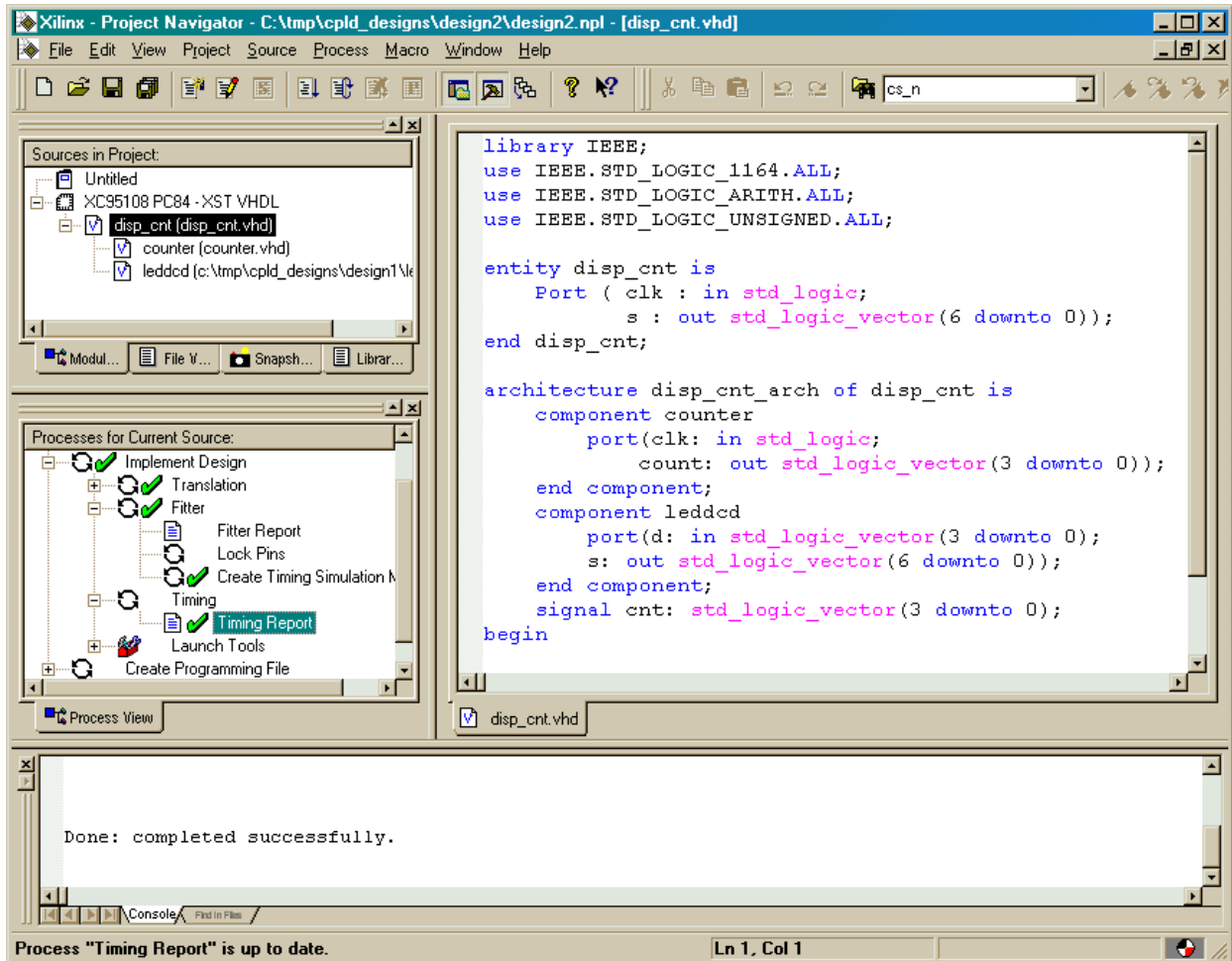
** INPUTS **
Signal      Loc      Pin   Pin      Pin
Name        Loc      #    Type    Use
clk         FB6_3   46   I/O     I

End of Resources Used by Successfully Mapped Logic
```

Also note that there are four signals in addition to the input and outputs. These are the four outputs from the counter (**N26**, **N27**, **N28**, and **N29**). 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 Timing Report process.



The timing report contains the following information:

```

Minimum Clock Period: 16.0ns
Maximum Internal Clock Speed: 62.5Mhz
(Limited by Clock Pulse Width)
-----
Clock Pad to Output Pad (tCO) (nsec)
-----
\ From      c
           l
           k
-----
To  \-----
s<0> 26.5
s<1> 26.5
s<2> 26.5
s<3> 26.5
s<4> 26.5
s<5> 26.5
s<6> 26.5
    
```

This table says that there is a 26.5 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.)

Looking further into the file we see the following table.

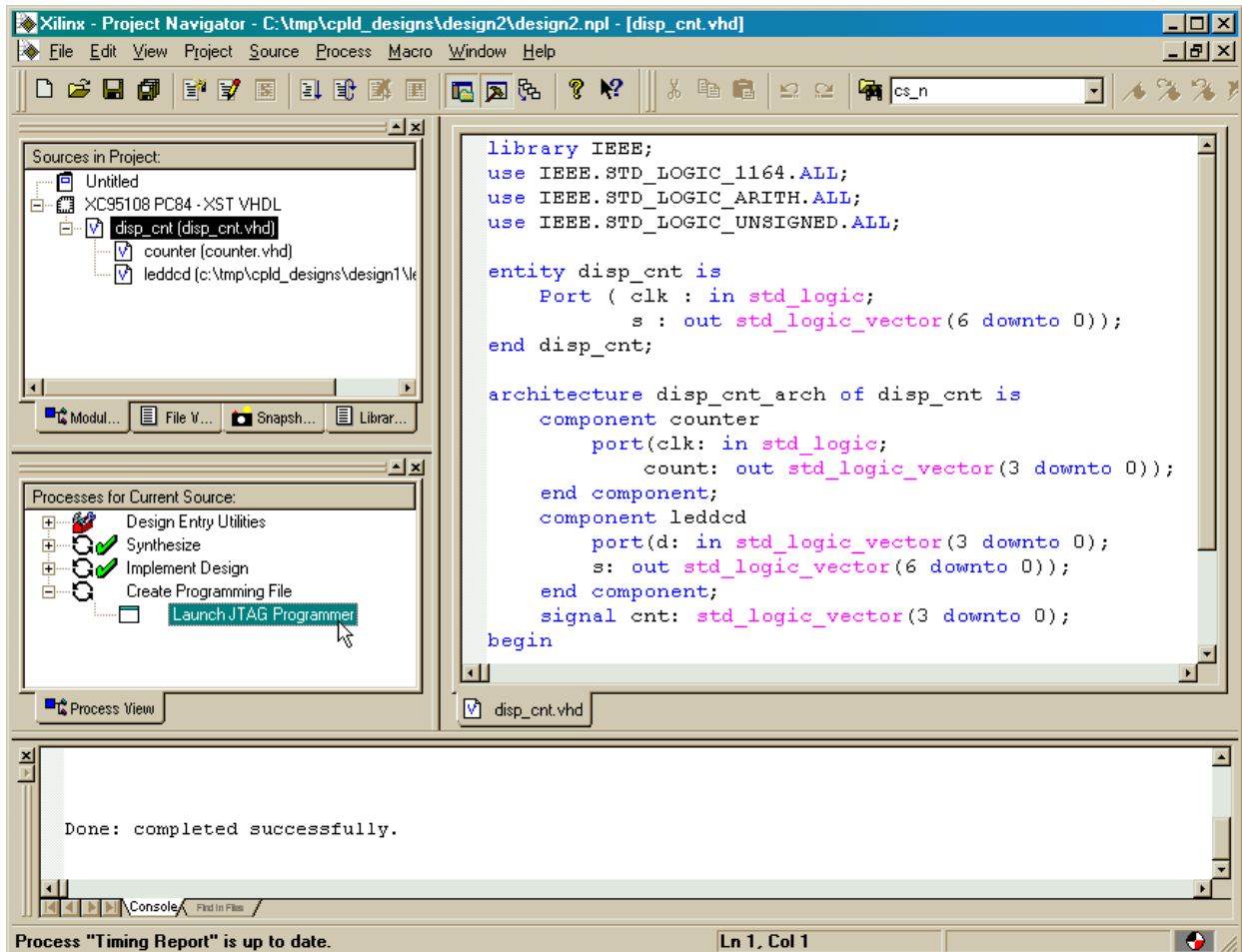
```

-----
Clock to Setup (tCYC) (nsec)
(Clock: clk)
-----
\ From      N      N      N
           2      2      2
           7      8      9
           .      .      .
           Q      Q      Q
-----
To  \-----
N26.D 10.5  10.5  10.5
N27.D      10.5  10.5
N28.D           10.5
N29.D           10.5
    
```

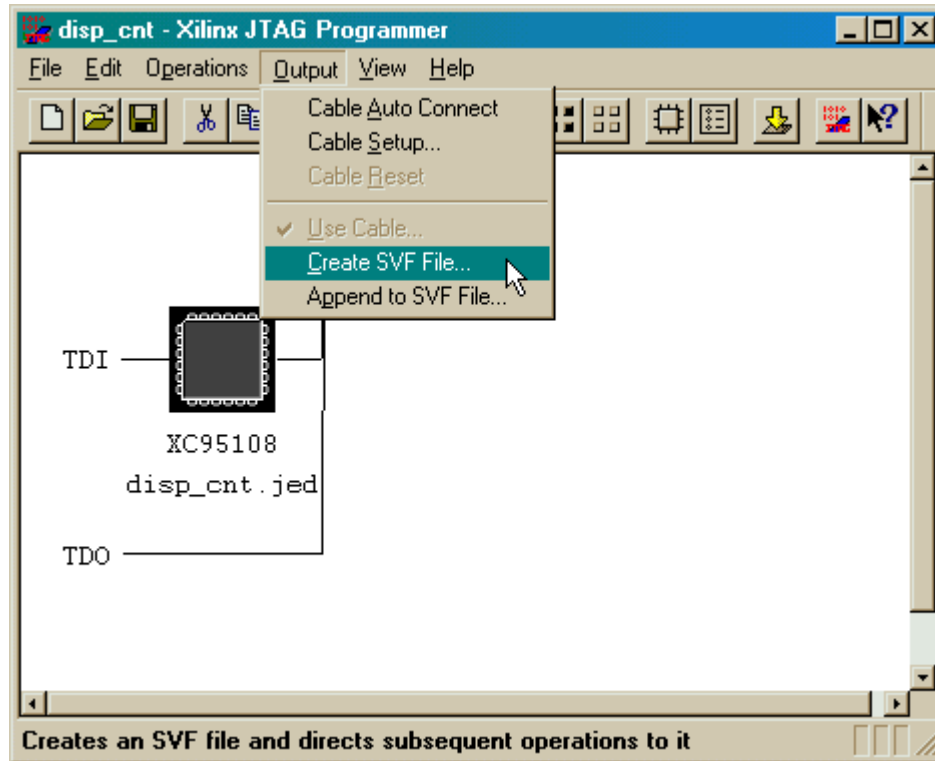
This states that a minimum of 10.5 ns is needed from the time the counter flip-flops are clocked until they can be clocked again. The time delay results from the sum of the clock-to-output delay of each flip-flop, internal wiring delays, and flip-flop setup times.

## Generating the Bitstream

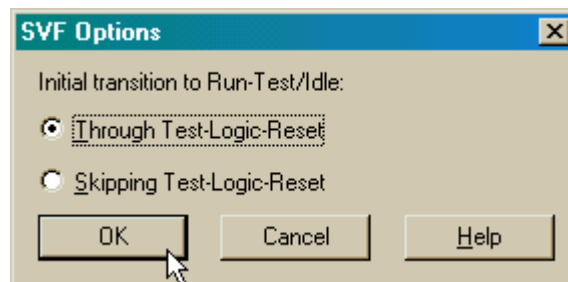
Now we are ready to generate the bitstream for the displayable counter. To initiate the programmer, we highlight the XC95108 PC84 object in the Source pane and double-click on the JTAG Programmer process.



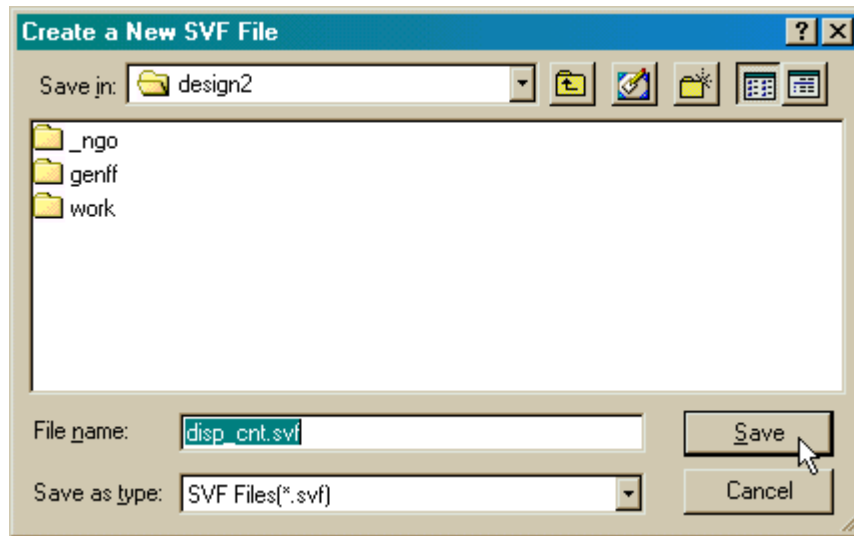
The **JTAG Programmer** window will appear and again we only have one chip in our design, so only one XC95108 CPLD is shown. We proceed by selecting an SVF file as the destination for the bitstream so we can use GXSLD for programming the CPLD in the XS95 Board.



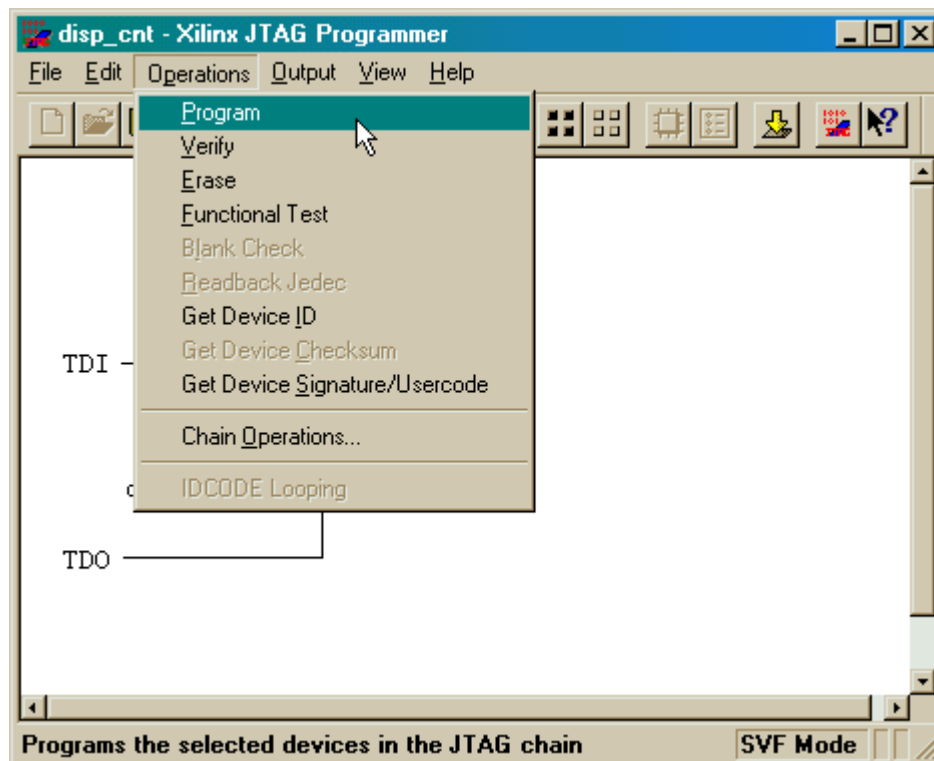
Once again we choose the default for the initialization sequence of the JTAG state machine that controls the programming process.



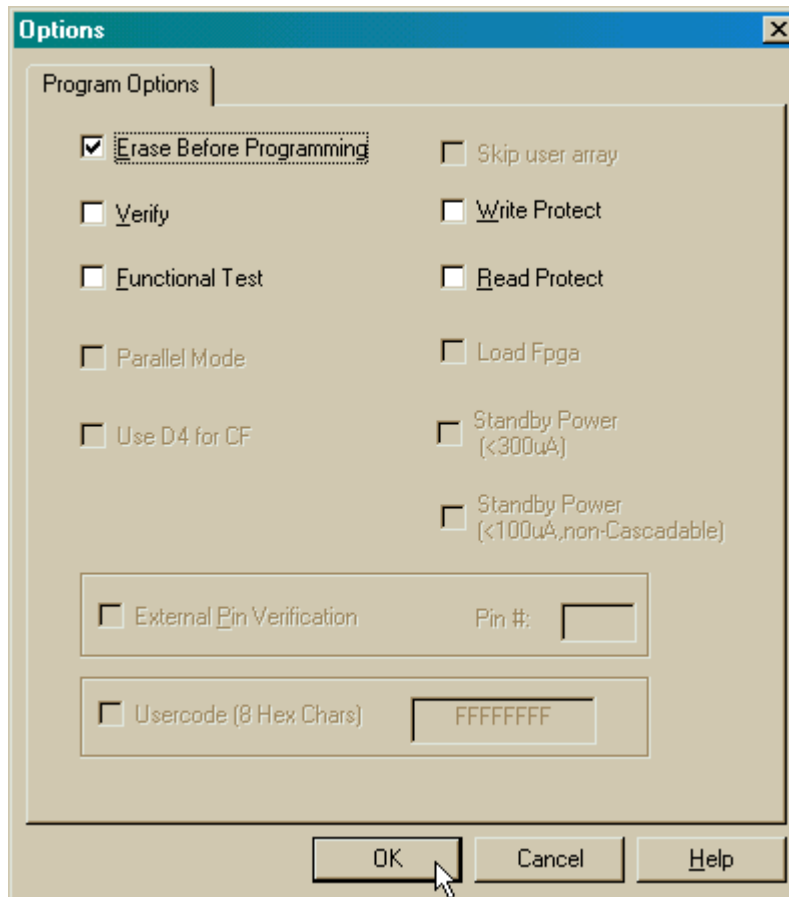
Now a window appears where we can enter the name for the file that will hold the bitstream. We can click on Save to accept the default name of disp\_cnt.svf.



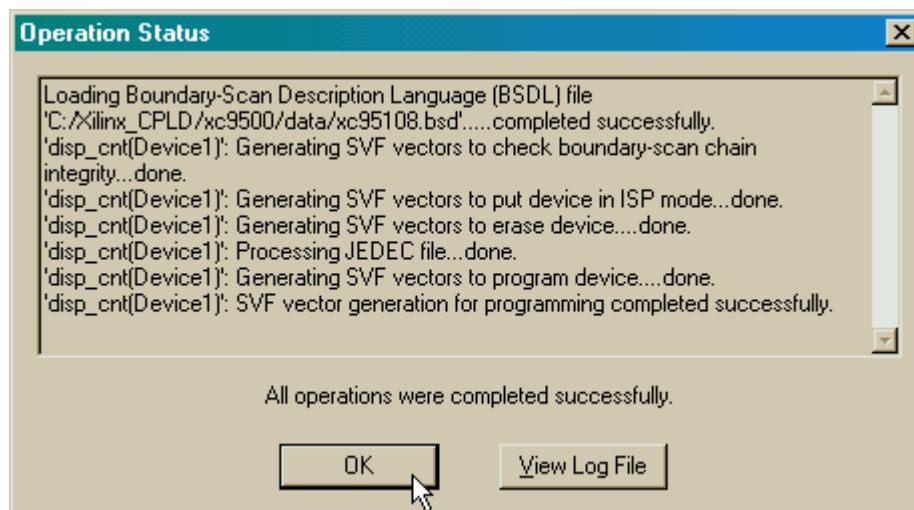
We then proceed to generate the bitstream by selecting the Operations⇒Program menu item.



An **Options** window appears where we can set the Erase Before Programming option for the generated bitstream.



Once we click OK in the **Options** window, the bitstream generation process begins. The progress is reported in the **Operation Status** window:



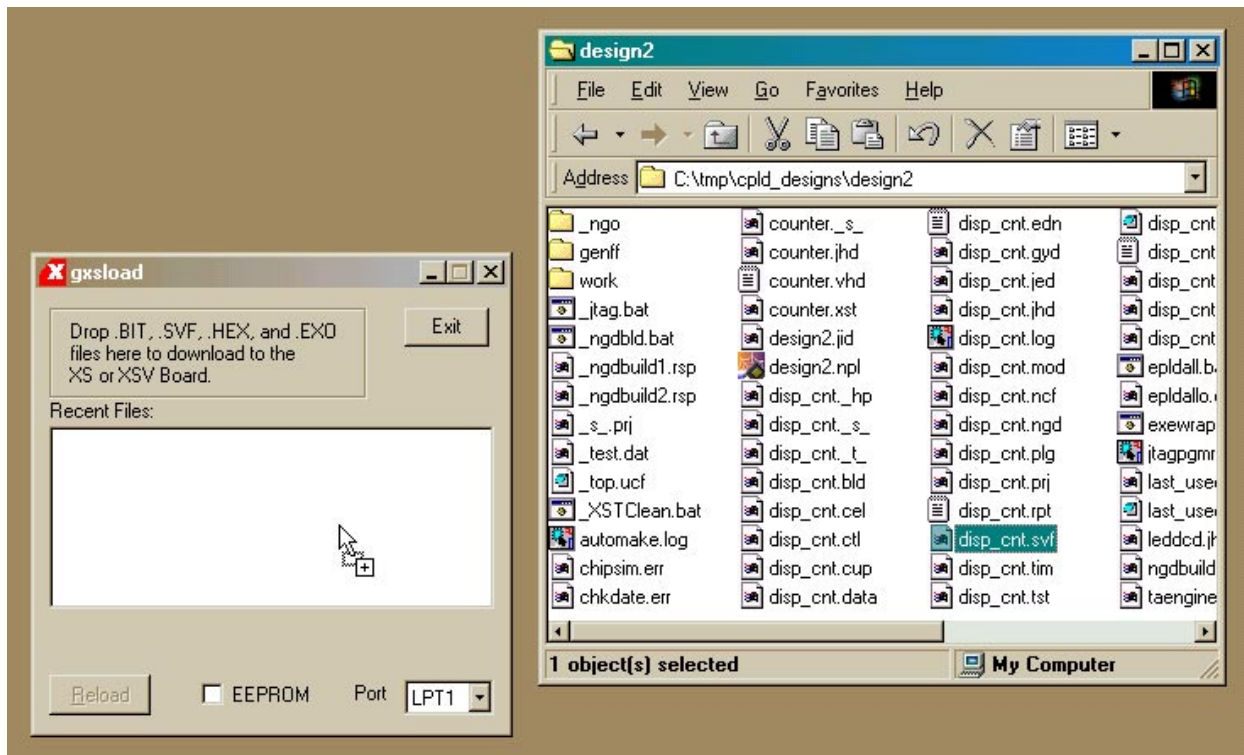
Once the bitstream file is generated, we click on OK to close the window. Then we close the **JTAG Programmer** window

## Downloading the Bitstream

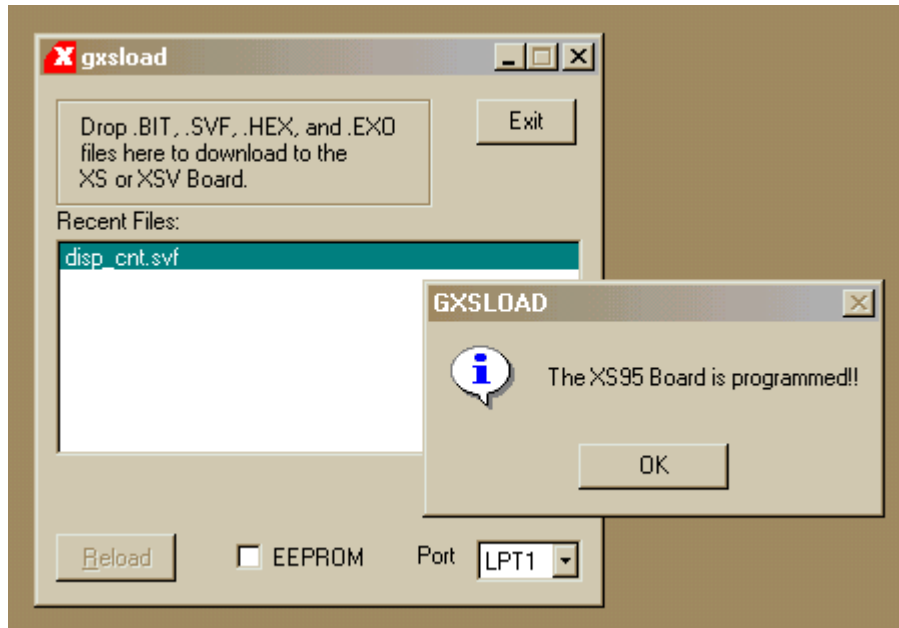
Now we program the bitstream file into the CPLD of the XS95 Board. We double click



the **GXSLOAD** icon to bring up the **gxslload** window. Then we drag the disp\_cnt.svf file from the C:\tmp\cpld\_designs\design2 folder and drop it into the **gxslload** window.



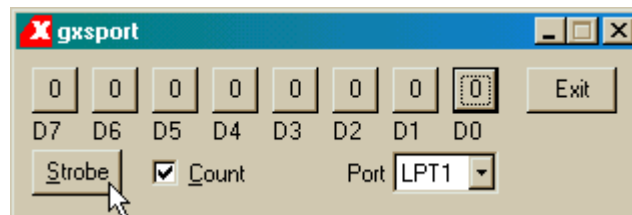
The XC95108 CPLD programming starts and completes in about a minute.



## Testing the Circuit

Once the XC95108 CPLD on the XS95 Board is programmed, we can begin testing

displayable counter. Double-clicking the  icon initiates the GXSPORT utility.



The **clk** input of the counter is assigned to the pin controlled by the D0 button of GXSPORT. We place a check in the Count box so that the D0 value will toggle every time we press the Strobe button. To generate a clock pulse, we just click twice on Strobe. Each clock pulse will increment the value shown on the LED digit of the XS95 Board.

# 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⇒Technical Support. You will be presented with a browser window full of 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.