Introduction to WebPACK

Using XILINX WebPACK Software to Create CPLD Designs

RELEASE DATE: 10/24/1999

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

All XC-prefix product designations are trademarks of Xilinx.

This document is placed in the public domain by XESS Corporation.

Table of Contents

What This Is and *Is Not* 1

CPLD Programming 3

Installing WebPACK 5

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

Our First Design 7

- An LED Decoder 7
- Starting WebPACK Project Navigator 8
- Naming Your Project 12
- Setting Your Chip Type 13
- Describing Your Design With VHDL 15
- Checking the VHDL Syntax 18
- Fixing VHDL Errors 20
- Synthesizing the Logic circuitry for Your Design 22
- Fitting the Logic Circuitry Into the CPLD 24
- Checking the Fit 27
- Constraining the Fit 29
- Viewing the Chip 34
- Generating the Bitstream 40
- Downloading the Bitstream 45
- Testing the Circuit 47

Hierarchical Design 49

- A Displayable Counter 49
- Starting a New Design 50

Adding the LED Decoder 51

Adding a Counter 54

Tying Them Together 57

Checking the VHDL Syntax 60

Constraining the Design..62

Synthesizing the Logic Circuitry for the Design 63

Fitting the Logic Circuitry Into the CPLD..64

Checking the Fit 65

Checking the Timing 67

Generating the Bitstream 69

Downloading the Bitstream 72

Testing the Circuit 73

Going Further… 74

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, then you can 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.

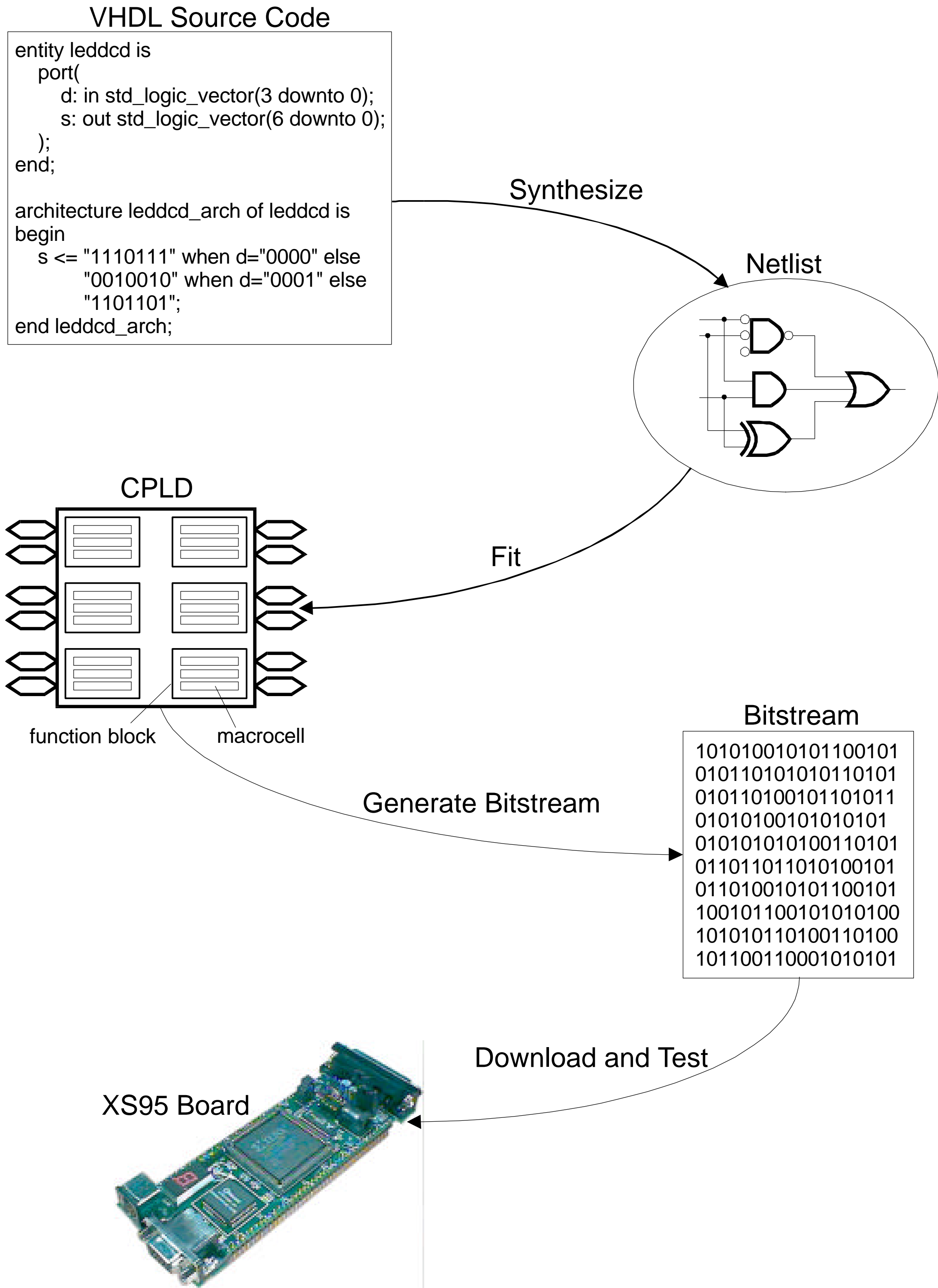
2. You use a *logic synthesizer* program to transform the HDL into a *netlist*. The netlist is just a description of the various logic gates in your design and how they are interconnected.

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.


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
CPLD
Fit
function block
macrocell
Bitstream
101010010101100101
010110101010110101
010110100101101011
01010100101010101
010101010100110101
011011011010100101
011010010101100101
100101100101010100
101010110100110100
101100110001010101
Generate Bitstream
Download and Test
XS95 Board

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:

1. XC9500 HDL-ABEL Synthesis Tools: This module provides the HDL editor and logic synthesizer. The software from module version 2.1WP2.1, released Sept 28, 1999 was used for this tutorial. The file for the module is webpack_hdl_abel.exe. The file size is 6.4 MBytes.

2. XC9500 Device Fitter Tools: This module provides the fitter for the XC9500 CPLDs. The software from module version 2.1WP2.1, released Sept. 28, 1999 was used for this tutorial. The file for the module is webpack_cpld_fitter.exe. The file size if 10.5 MBytes.

   There is another optional ChipViewer module that you can download. ChipViewer gives you a graphical view into the CPLD after you have fitted your logic into it. The software from module version 2.1WP2.1, released Sept. 28, 1999 was used for this tutorial. The file for the module is webpack_chipviewer.exe. The file size if 8.8 MBytes.

3. Device Programming Tools: This module provides the bitstream generator for the XC9500 CPLDs. The software from module version 2.1WP2.1, released Sept. 28, 1999 was used for this tutorial. The file for the module is webpack_programmer.exe. The file size if 6.7 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_cpld_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.

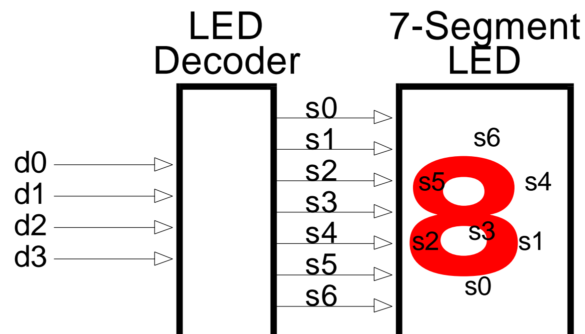
3 Our First Design

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:

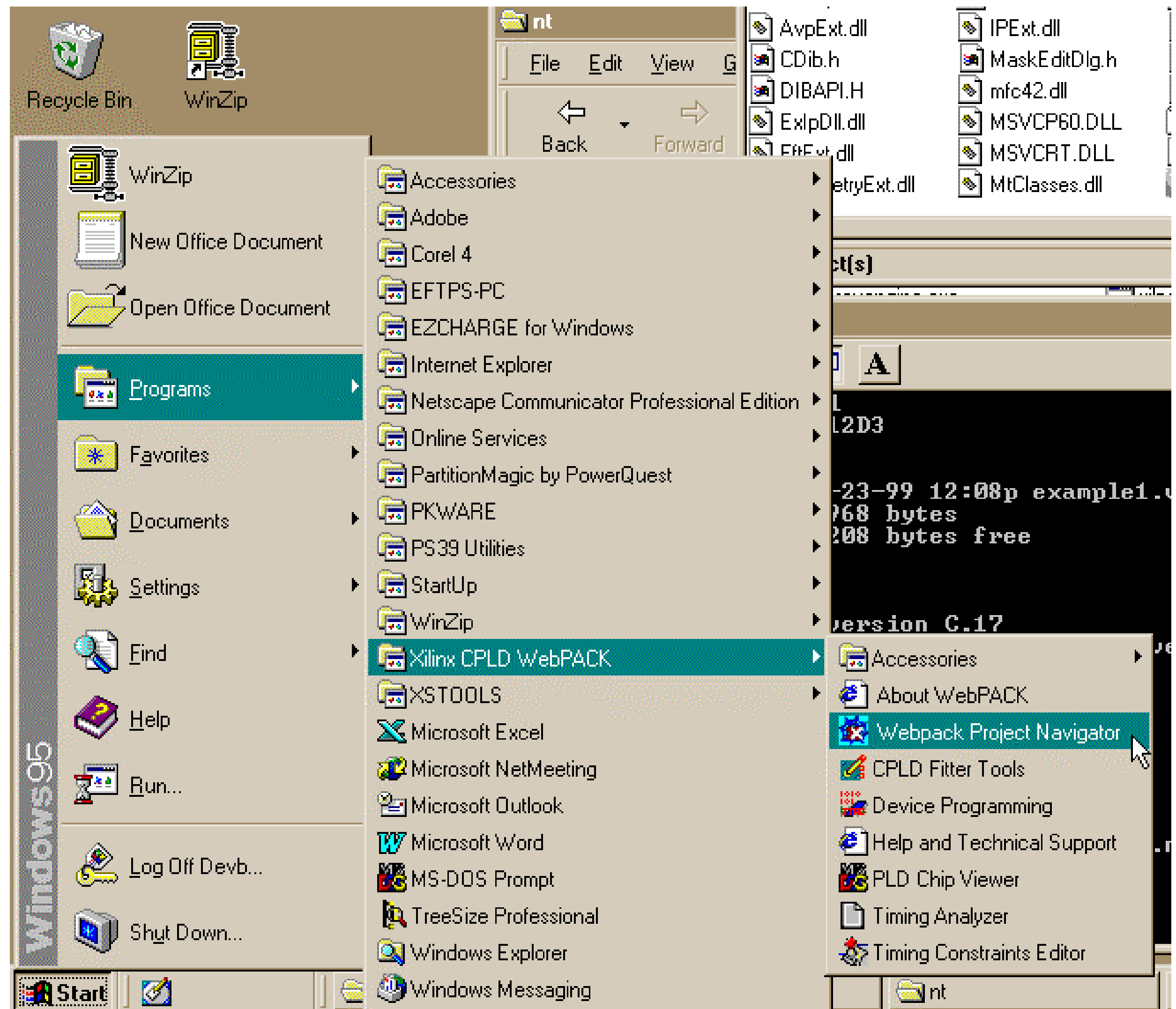
Four-bit	Hex Digit	LED Display
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	A
1011	B	b
1100	C	c
1101	D	d
1110	E	E
1111	F	F

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



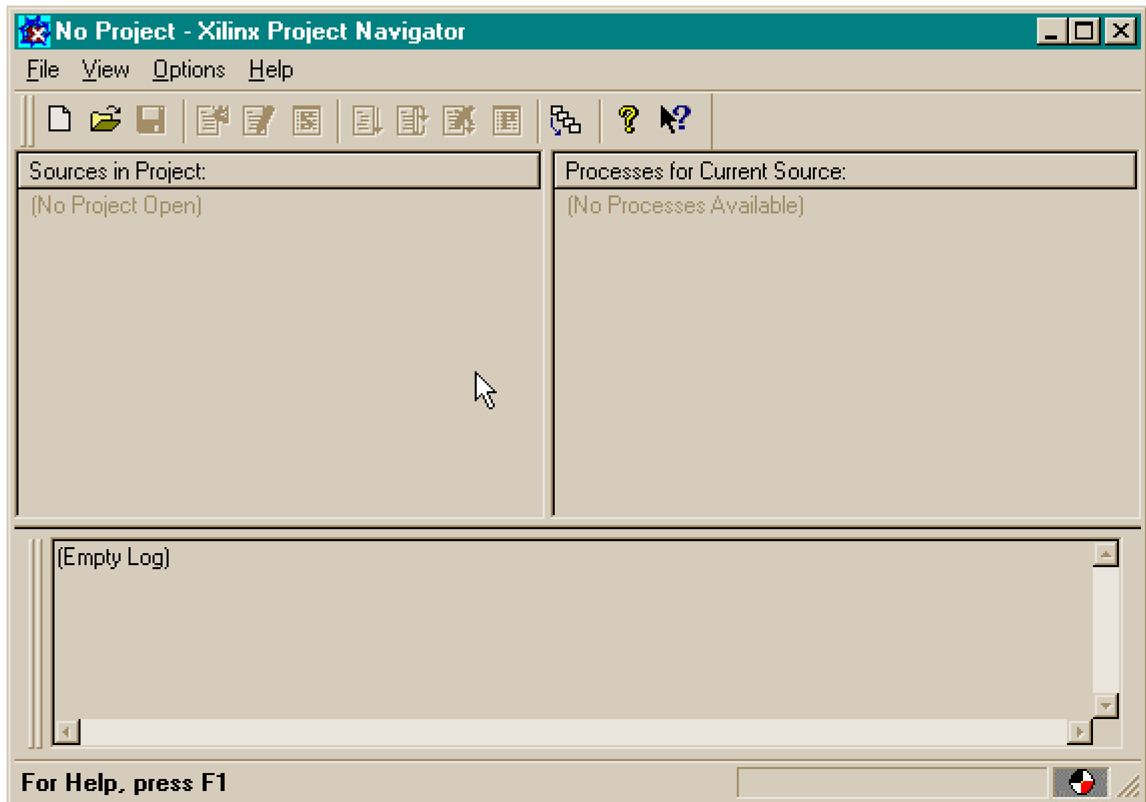
Starting WebPACK Project Navigator

We start WebPACK by selecting **Project Navigator** from the following menu:

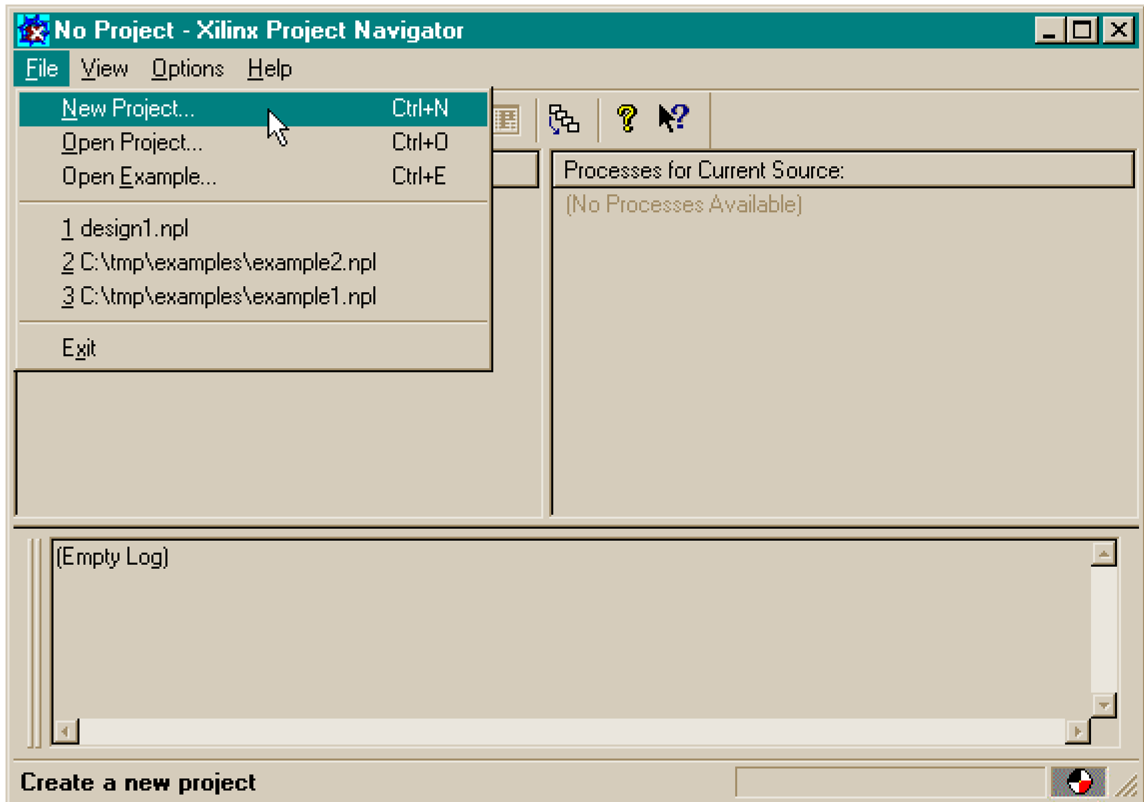


This will bring up an empty project window as shown below. The window has three panes:

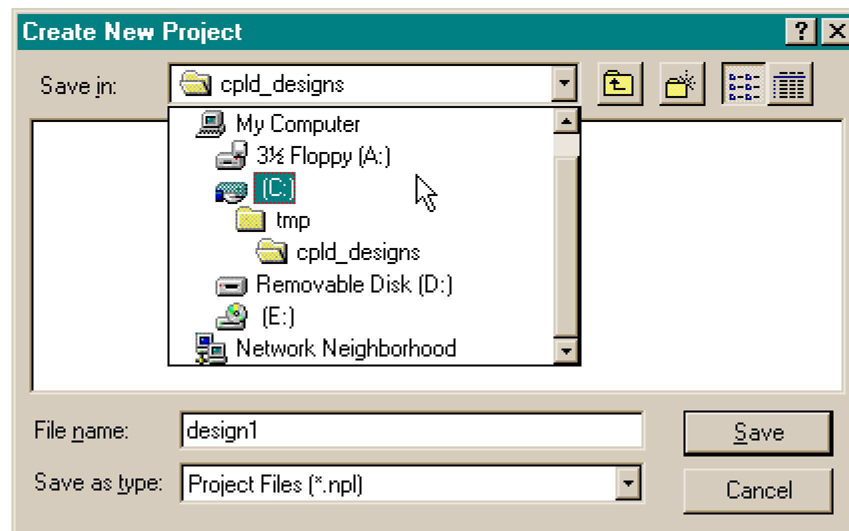
1. A **source pane** which keeps a hierarchical list of the HDL source files that make up our design.
2. A **process pane** which lists the various operations we can perform on a given object in the source pane.
3. A **log pane** which records the various messages from the currently running process.



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



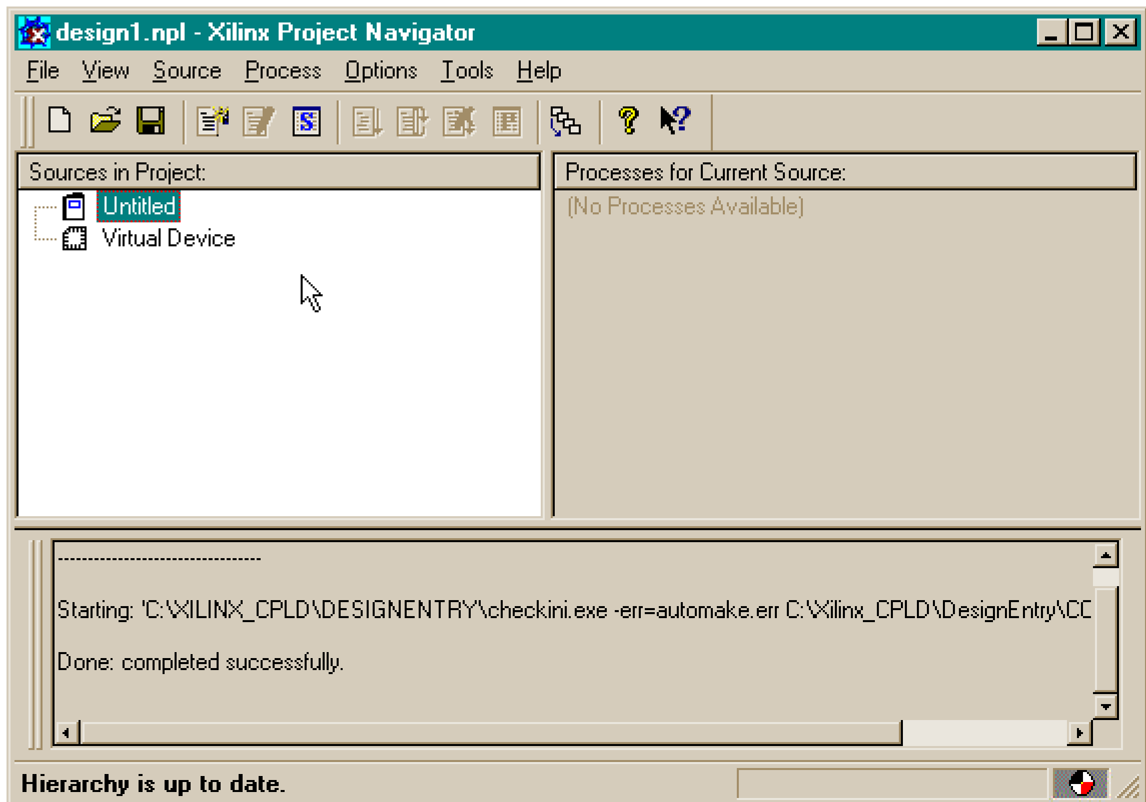
This brings up the **Create New Project** window. For these design examples, we will store everything in the **C:\tmp\cpld_designs** directory. The LED decoder design will be given the descriptive title of **design1**.



Clicking on **Save** closes the window and now the Sources pane in the Project Navigator window contains two items:

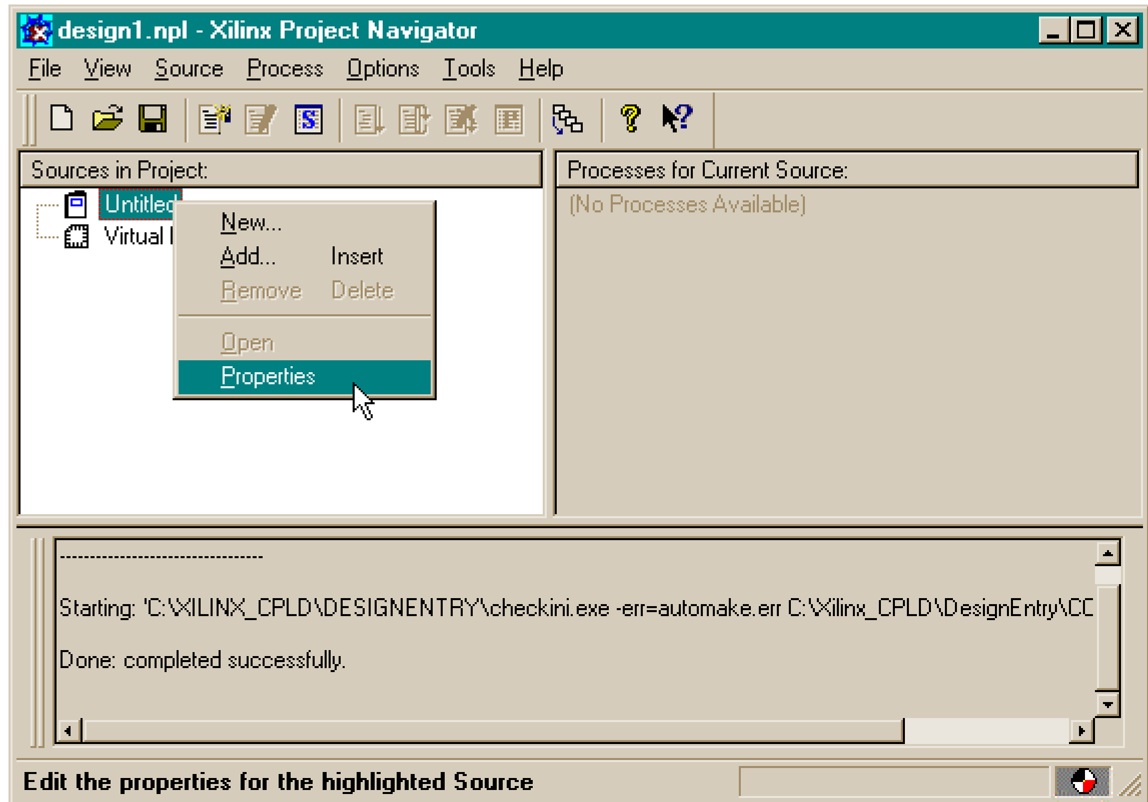
1. A **project object** called **Untitled**.

2. A **chip object** called **Virtual Device**.

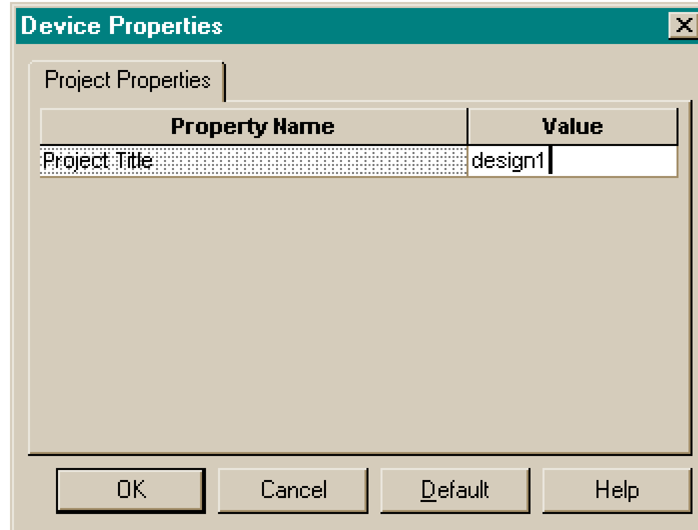


Naming Your Project

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



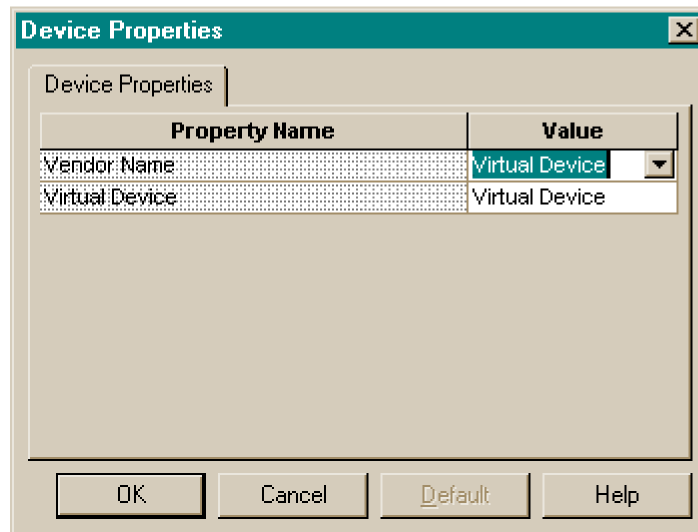
This displays the **Device Properties** window that contains a single **Project Properties** tab. The tab has a single slot where we can type the **Project Title**. I chose **design1** to keep it consistent with the project name we chose previously, but you can choose any name.



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

Setting Your Chip Type

After setting the project name, we need to set the type of device we are going to load our design into. Right-click on the Virtual Device object in the Sources pane and select the Properties item from the pop-up menu. The following window appears.

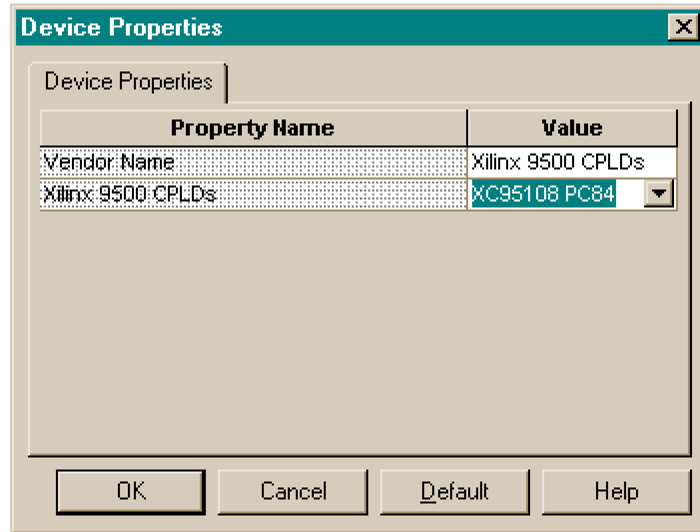


The Device Properties tab in the window has two slots:

1. The **Vendor Name** slot lets you select one of three main families of XILINX CPLDs: **9500**, **9500XL**, and **9500XV**. The XS95 Board has an XC9500 CPLD, so we will select the 9500 item from the list.

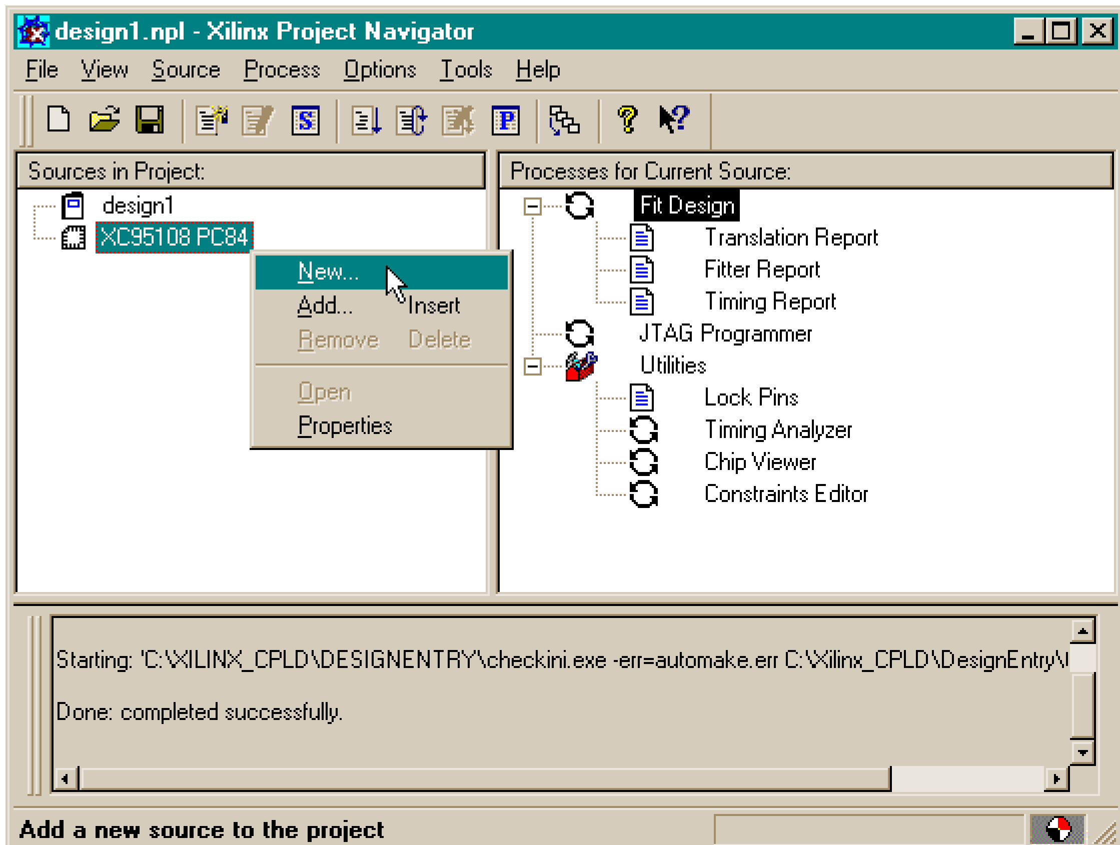
2. The **Virtual Device** slot lets us pick one of the members of the CPLD family that we just selected. (Note that the slot title changes to **Xilinx 9500 CPLDs** because we chose that

family of devices.) The XC9500 CPLD in the XS95 Board is an XC95108 in an 84-pin PLCC package, so we select **XC95108 PC84** from the drop-down list.

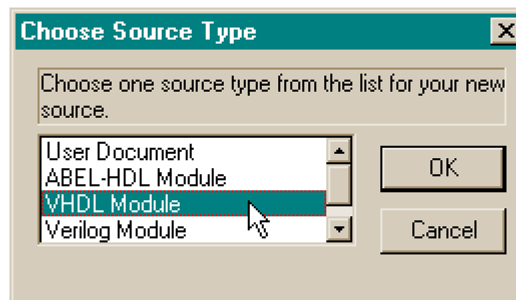


Describing Your Design With VHDL

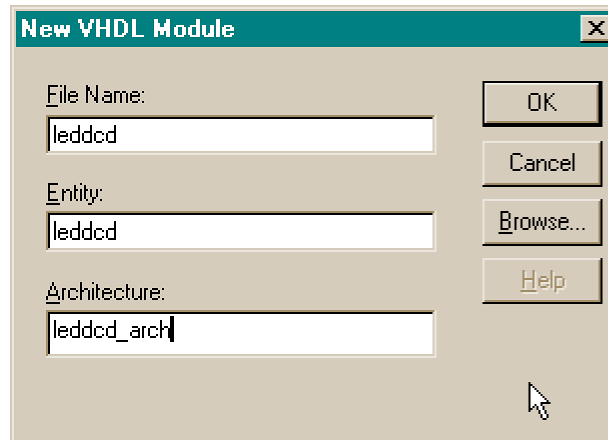
Now that the project is named and the chip type is selected, we can begin to actually do the design. We start by adding a VHDL file that describes the LED decoder to the design1 project. Right-click on the **XC95108 PC 84** object in the Sources pane and select **New...** from the pop-up menu as shown 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 OK.



Then we are prompted for a name for the VHDL file, the name of the module entity we will place in the file, and the name of the architecture section for the entity. In the window shown below, we have chosen the file name and entity name to both be **leddcd**, and the architecture section will be called **leddcd_arch**.



After clicking on OK, we are presented with a window containing the VHDL skeleton for our LED decoder. Lines 1-2 create a link to the IEEE library that contains various useful definitions for describing a design. We will place the description of the LED decoder input and output in the entity declaration between lines 4 and 6. And we will describe the logic operations of the decoder in the architecture section between lines 9 and 10.

Xilinx HDL_ABEL Text Editor - [leddcd.vhd]

File Edit View Templates Tools Options Window Help

```
 1 library ieee;
 2 use ieee.std_logic_1164.all;
 3
 4 entity leddcd is
 5
 6 end;
 7
 8 architecture leddcd_arch of leddcd is
 9 begin
10
11 end leddcd_arch;
12
```

Ln 1 Col 1 | 12 | WR | Rec Off | No Wrap | DOS | INS

The complete VHDL file for the LED decoder is shown below. The entity section lists two I/O ports:

d: an input bus which carries the 4-bit input to the LED decoder (line 6).

s: an output bus which drives the seven LED segments (line 7).

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 13-28).

Xilinx HDL_ABEL Text Editor - [leddcd.vhd]

File Edit View Templates Tools Options Window Help

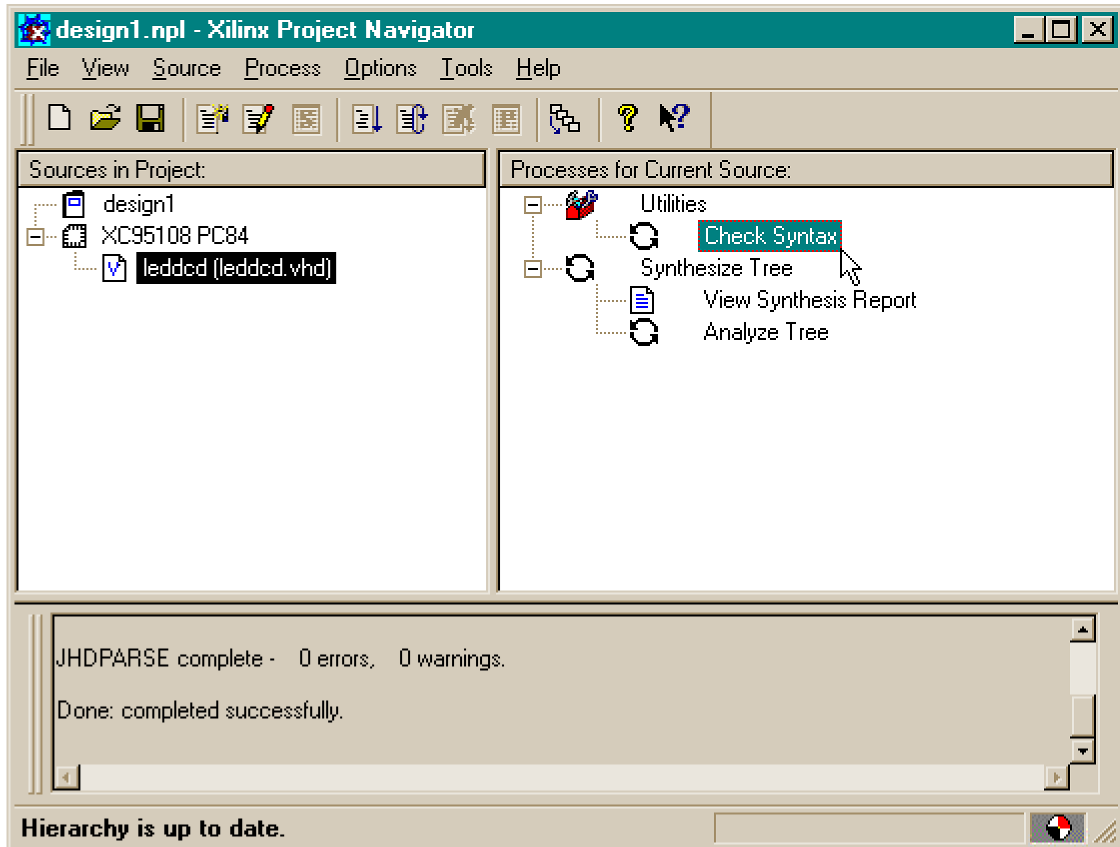
```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity leddcd is
5          port(
6                  d: in std_logic_vector(3 downto 0);
7                  s: out std_logic_vector(6 downto 0);
8          );
9  end;
10
11 architecture leddcd_arch of leddcd is
12 begin
13         s <=    "1110111" when d="0000" else
14                 "0010010" when d="0001" else
15                 "1011101" when d="0010" else
16                 "1011011" when d="0011" else
17                 "0111010" when d="0100" else
18                 "1101011" when d="0101" else
19                 "1101111" when d="0110" else
20                 "1010010" when d="0111" else
21                 "1111111" when d="1000" else
22                 "1111011" when d="1001" else
23                 "1111110" when d="1010" else
24                 "0101111" when d="1011" else
25                 "0001101" when d="1100" else
26                 "0011111" when d="1101" else
27                 "1101101" when d="1110" else
28                 "1101100";
29 end leddcd_arch;
```

Ln 7 Col 1 | 29 | WR | Rec Off | No Wrap | DOS | INS

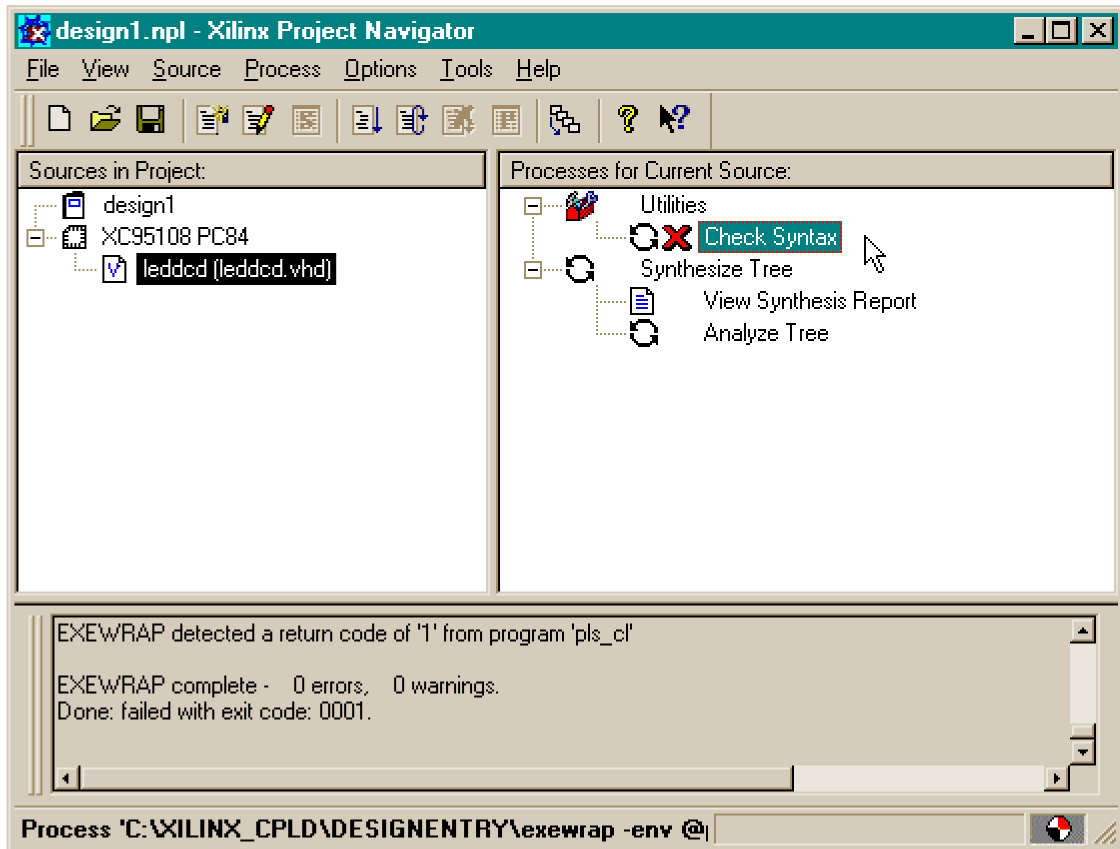
Once the VHDL source is entered, we click on the button to save it in the leddcd.vhd file.

Checking the VHDL Syntax

After creating the leddcd.vhd file, you will see it listed in the Sources pane in the Project Navigator window. We can check for errors in our VHDL by highlighting the **leddcd** object and then double-click on the **Check Syntax** process as shown below.

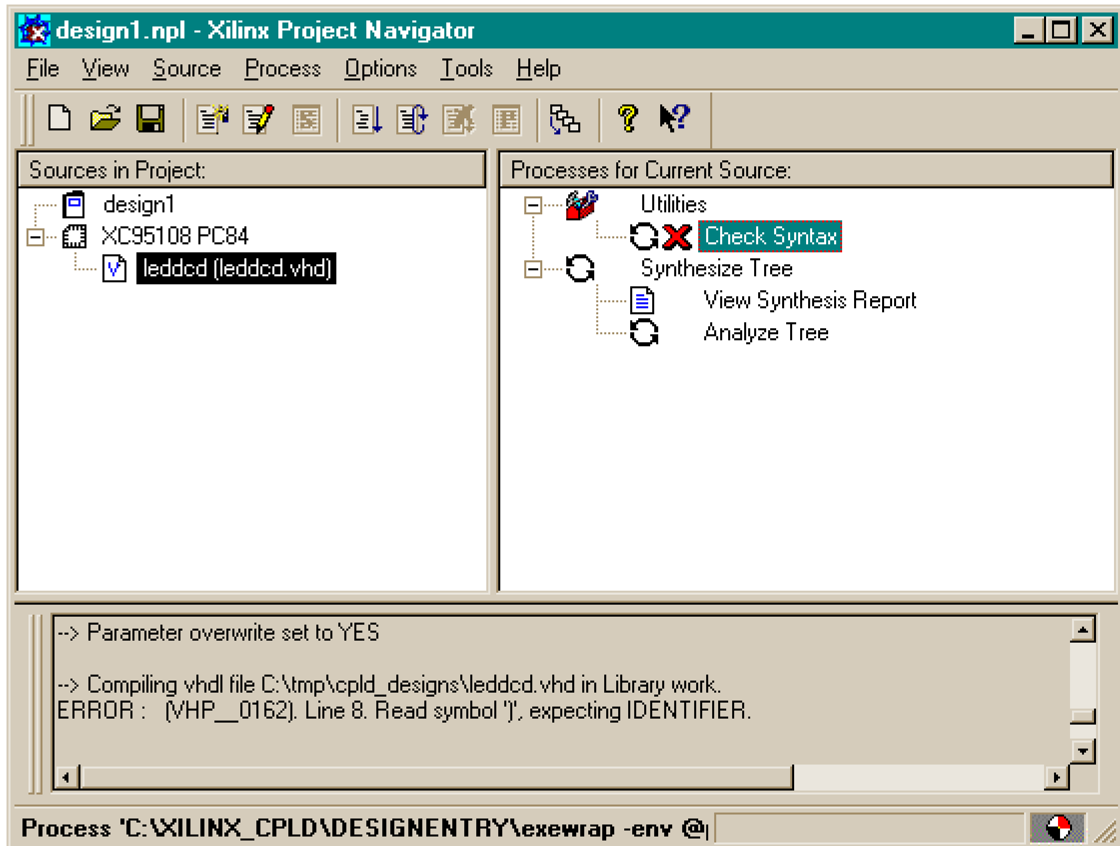


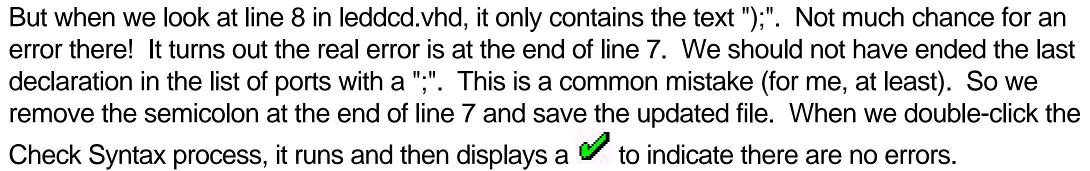
The syntax checking tool grinds away and then displays the result in the process window. In our case, an error was found as indicated by the next to the Check Syntax process. But what is the error and where is it?



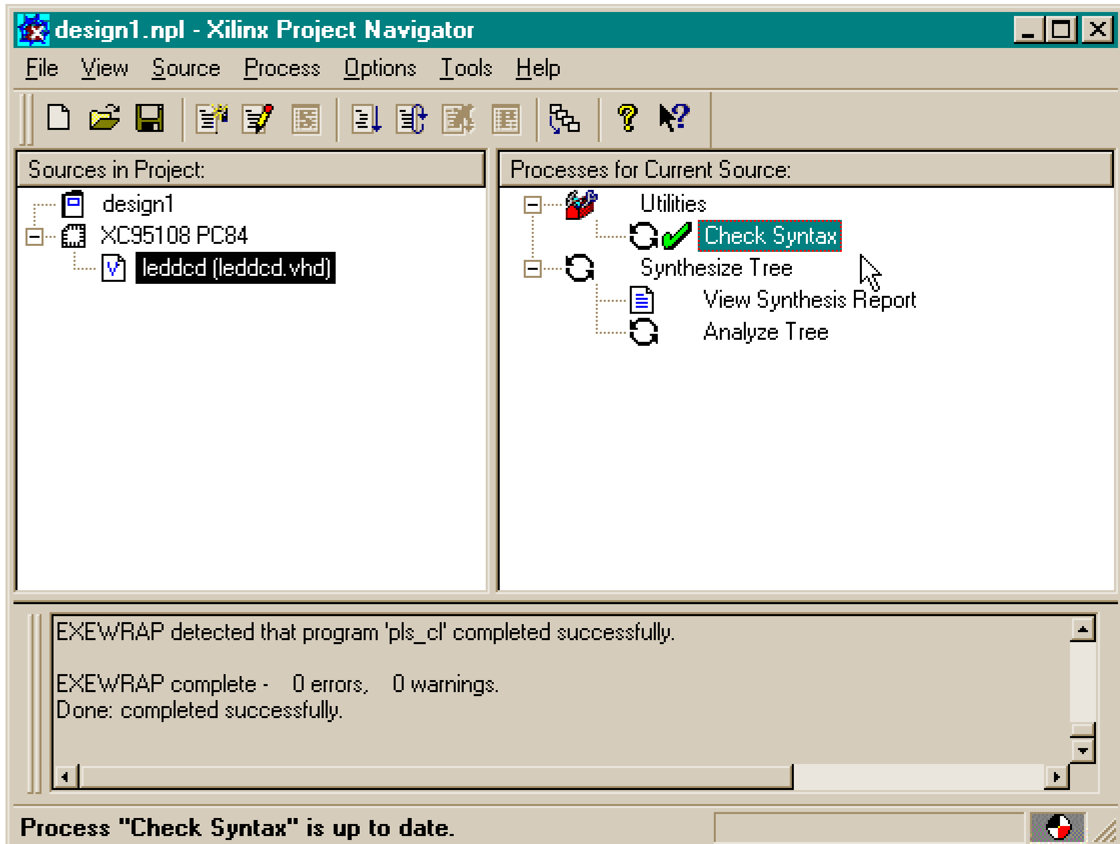
Fixing VHDL Errors

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 8.



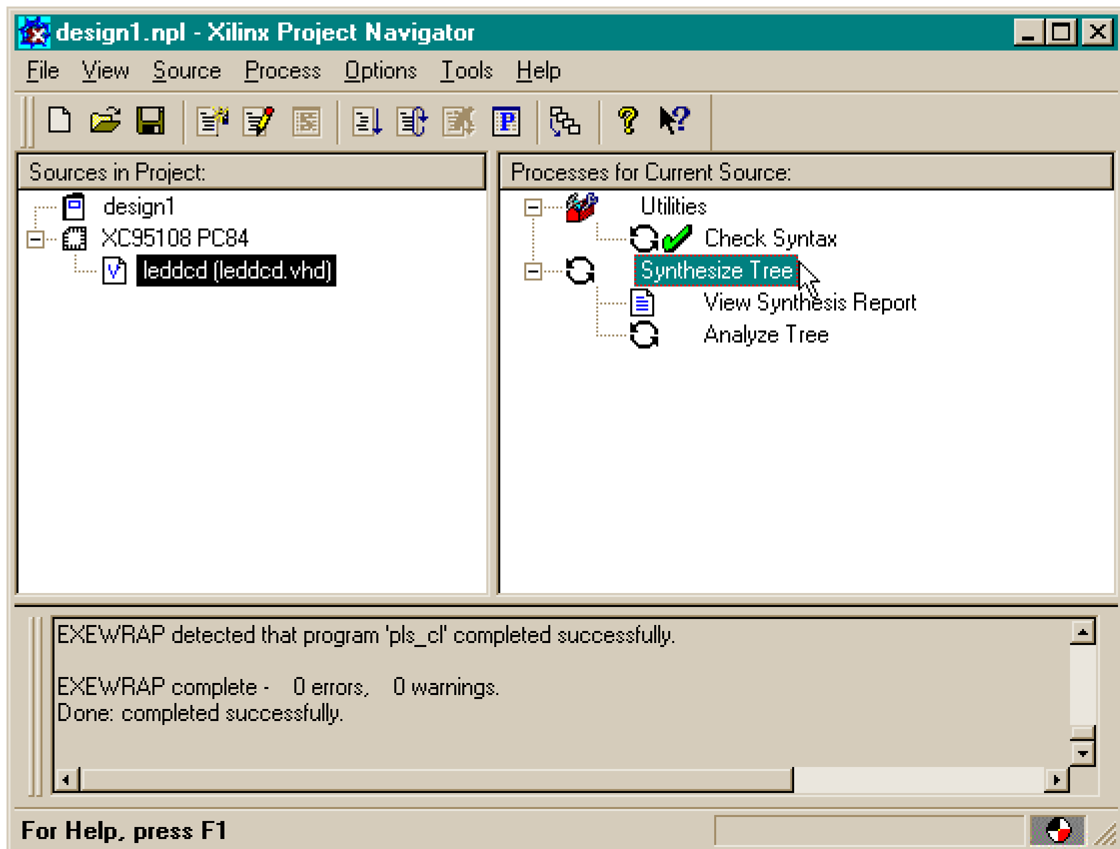


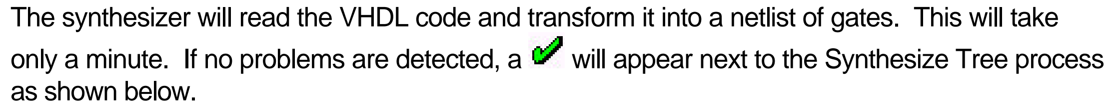
But when we look at line 8 in leddcd.vhd, it only contains the text ");". Not much chance for an error there! It turns out the real error is at the end of line 7. We should not have ended the last declaration in the list of ports with a ";". This is a common mistake (for me, at least). So we remove the semicolon at the end of line 7 and save the updated file. When we double-click the Check Syntax process, it runs and then displays a ✔ to indicate there are no errors.

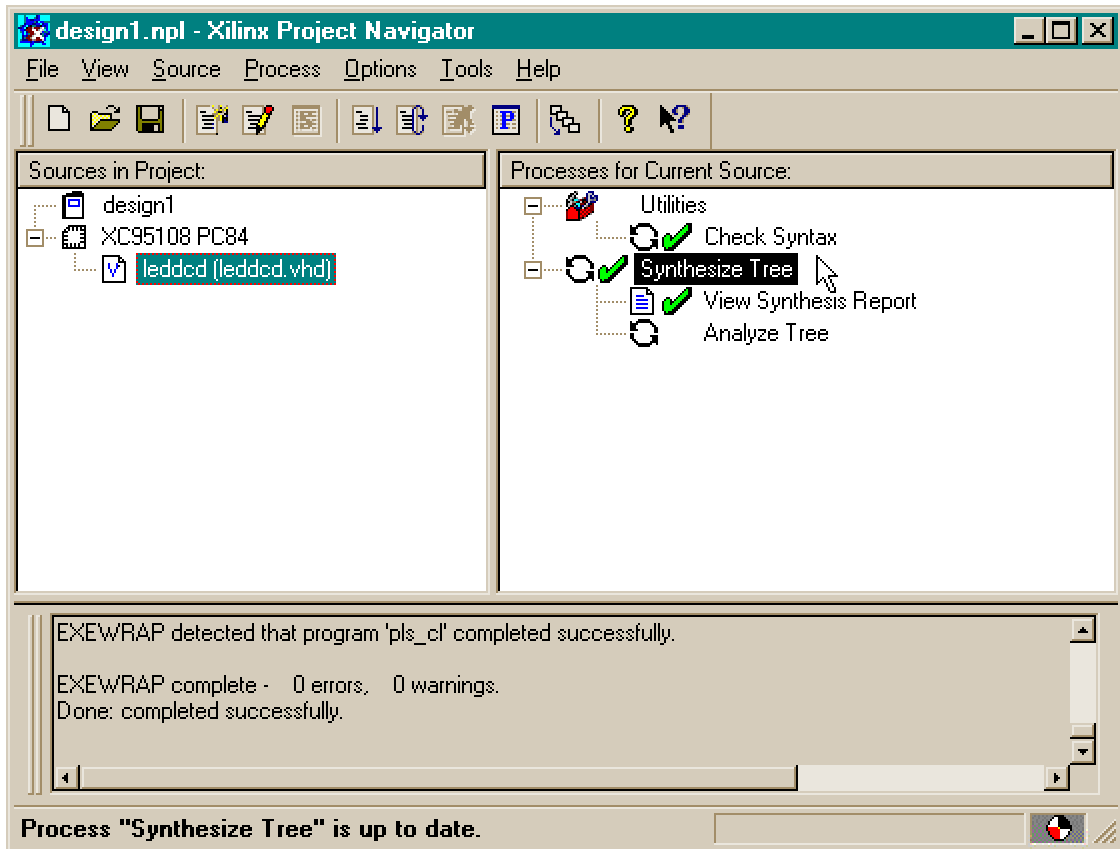


Synthesizing the Logic circuitry for Your Design

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 Tree** process as shown below.

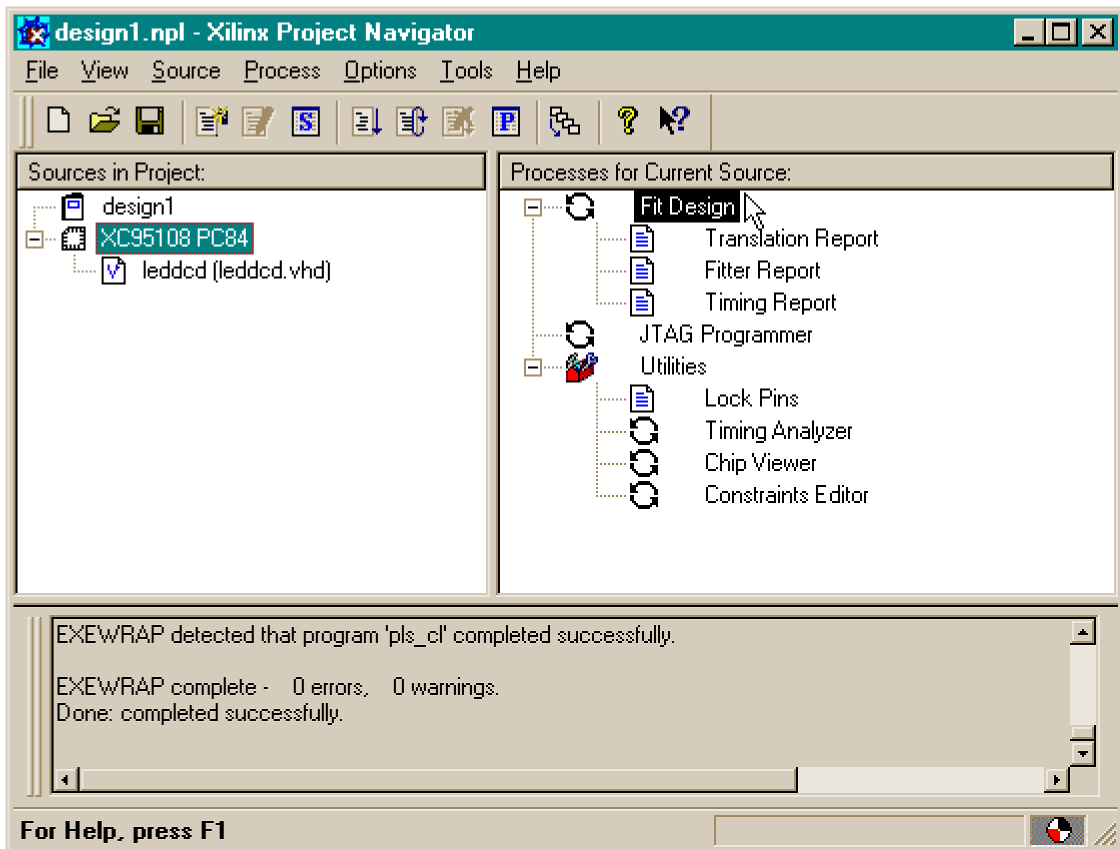


The synthesizer will read the VHDL code and transform it into a netlist of gates. This will take only a minute. If no problems are detected, a  will appear next to the Synthesize Tree process as shown below.

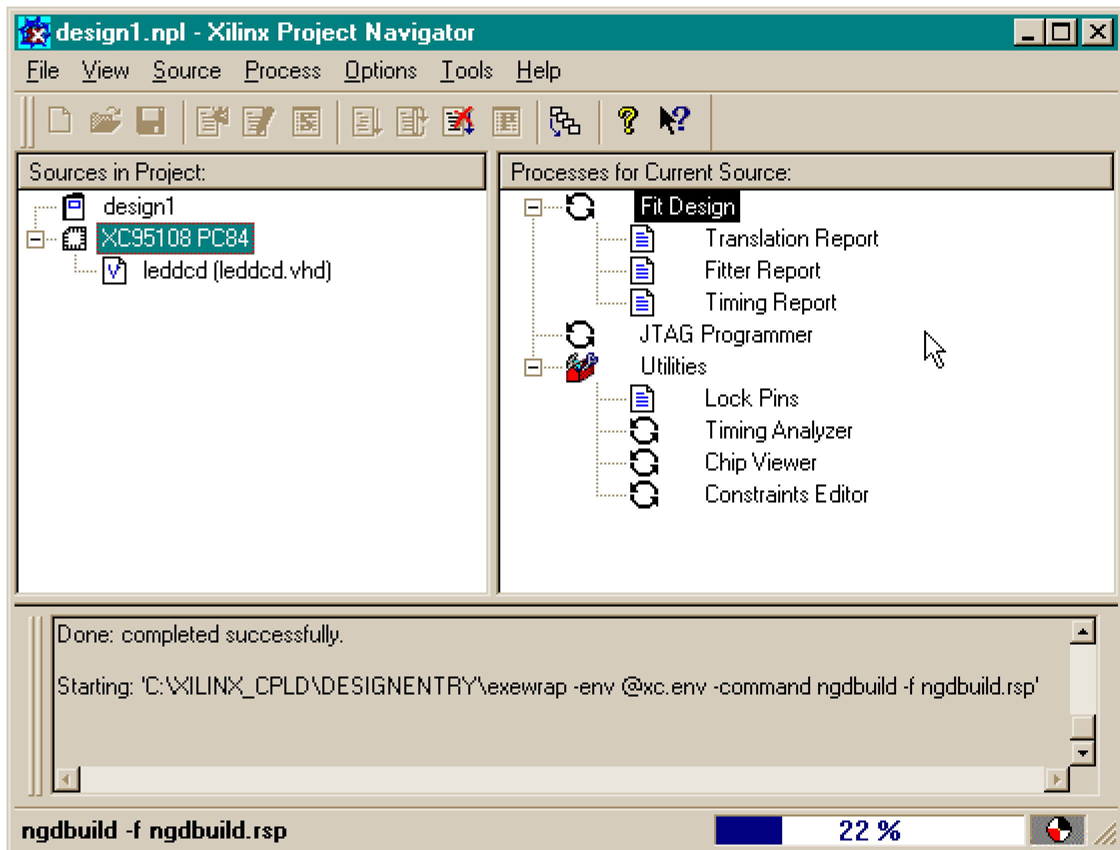


Fitting the Logic Circuitry Into the CPLD

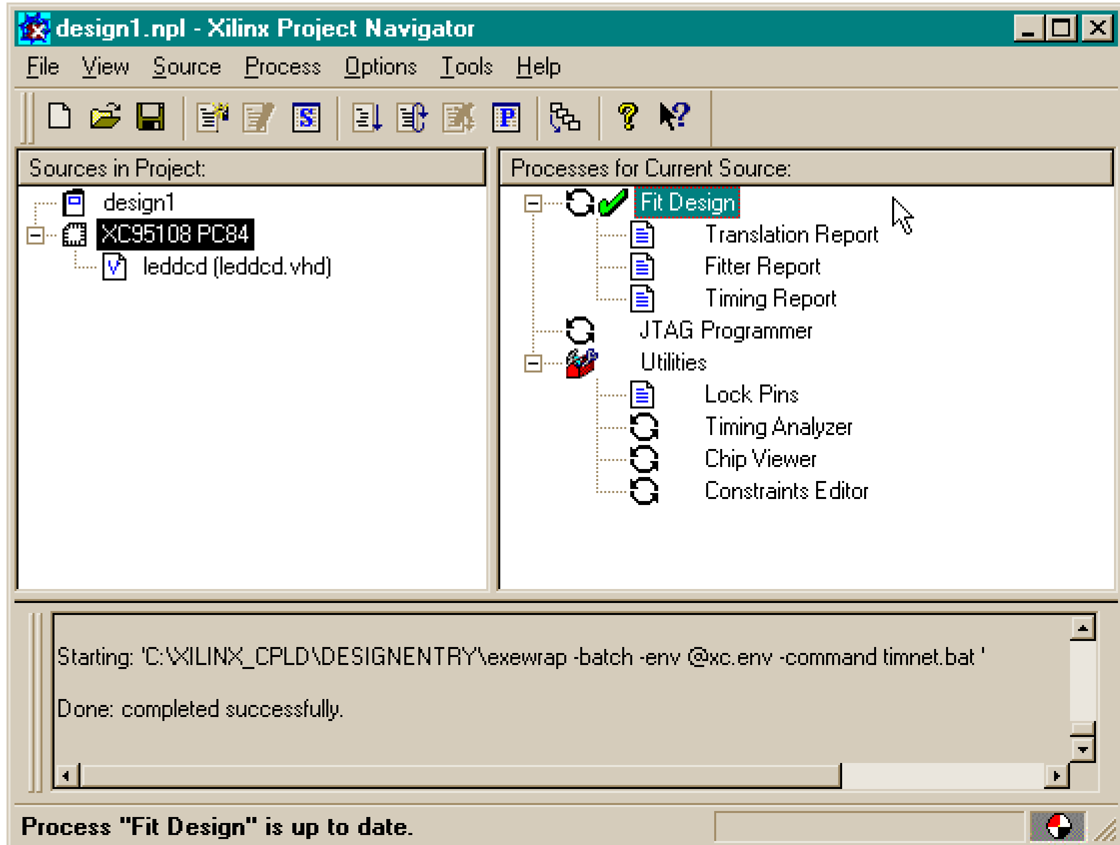
We now have a synthesized logic circuit for the LED decoder, but we need to fit it into the logic resources of the CPLD in order to actually use it. We start this process by highlighting the XC95108 PC84 object in the Sources pane and then double-click on the **Fit Design** process.



You can watch the progress of the fitting process in the status bar at the bottom of the Project Navigator window as shown below. For a simple design like the LED decoder, the fitting is completed in under a minute.

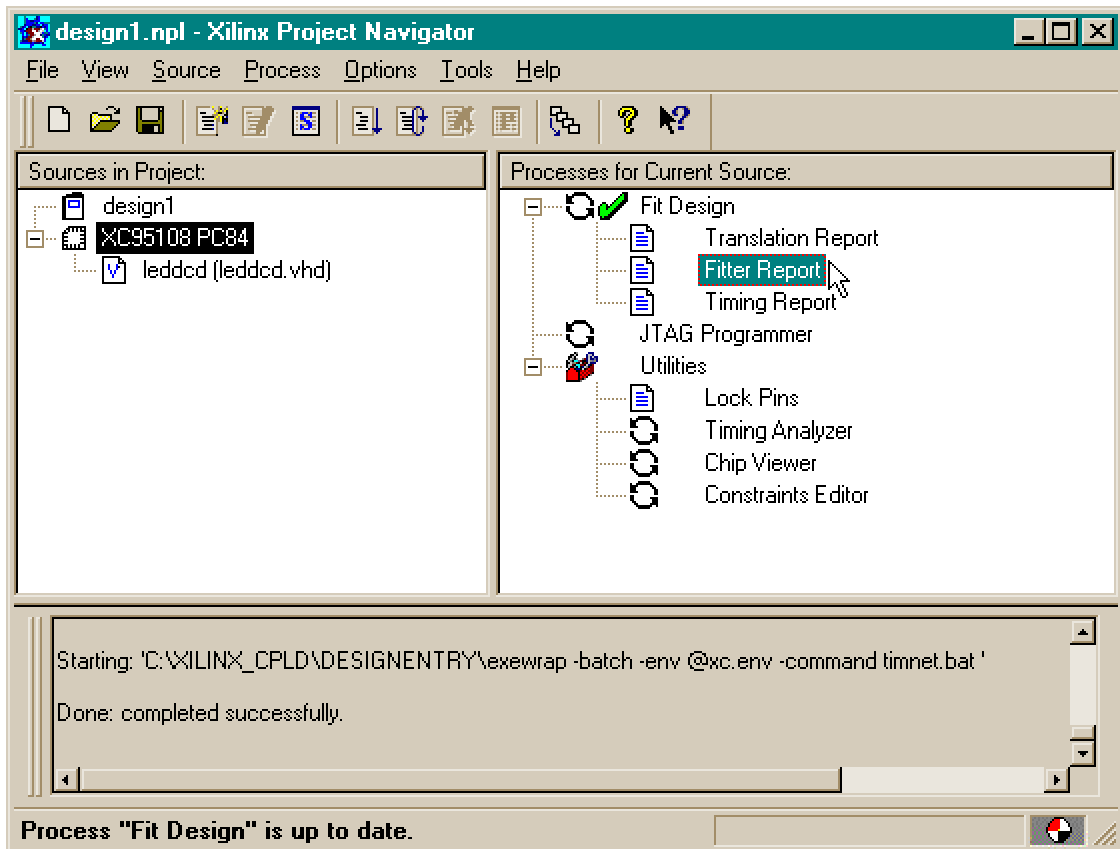


A successful fit is indicated by the ✔ next to the Fi t Design process.



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** process.



This brings up a window containing the fitting statistics for the LED decoder. The top of the file contains the lines:

```
XACT:  version C.17                                Xilinx Inc.
                                   Fitter Report
   Design Name: design1
Fitting Status: Successful                            Date: 10-23-1999, 10:04PM

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

Design         Device            Macrocells      Product Terms   Pins
Name           Used              Used            Used            Used
design1        XC95108-7-PC84    7  /108 (   6%) 24 /540 (   4%) 11 /69  ( 15%)
```

This shows the LED decoder only uses 7 of the 108 available macrocells in the XC95108 CPLD. And it only uses 7 I/O pins (4 for input, 7 for output).

Further down in the fitting report we can see what pins the inputs and outputs use. The d inputs have been assigned to pins 83, 17, 72, and 44. The outputs which drive the LED segments have been routed through pins 14, 1, 45, 6, 71, 32, 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       FB3_2   STD  FAST 14   I/O       O
s_1                 3       4       FB1_2   STD  FAST 1    I/O       O
s_2                 3       4       FB6_2   STD  FAST 45   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       FB5_2   STD  FAST 32   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_16            83   I/O       I
d_1                                 FB3_5             17   I/O       I
d_2                                 FB2_3             72   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_4  =  d_2 * d_3 * /d_0
   + d_2 * d_1 * /d_0
   + d_3 * d_1 * d_0
   + d_2 * /d_3 * /d_1 * d_0

/s_3  =  /d_2 * /d_3 * /d_1
   + d_2 * /d_3 * d_1 * d_0

/s_2  =  /d_3 * d_0
   + d_2 * /d_3 * /d_1
   + /d_2 * /d_1 * d_0

/s_1  =  d_2 * d_3 * d_1
   + d_2 * d_3 * /d_0
   + /d_2 * /d_3 * d_1 * /d_0

/s_0  =  d_2 * d_1 * d_0
   + d_2 * /d_3 * /d_1 * /d_0
   + /d_2 * d_3 * d_1 * /d_0
   + /d_2 * /d_3 * /d_1 * d_0
```

```
/s_6  =  d_2 * d_3 * /d_1
   + d_2 * /d_1 * /d_0
   + /d_2 * d_3 * d_1 * d_0
   + /d_2 * /d_3 * /d_1 * d_0

/s_5  =  d_2 * d_3 * /d_1
   + /d_2 * /d_3 * d_1
   + /d_2 * /d_3 * d_0
   + /d_3 * d_1 * d_0
```

Constraining the Fit

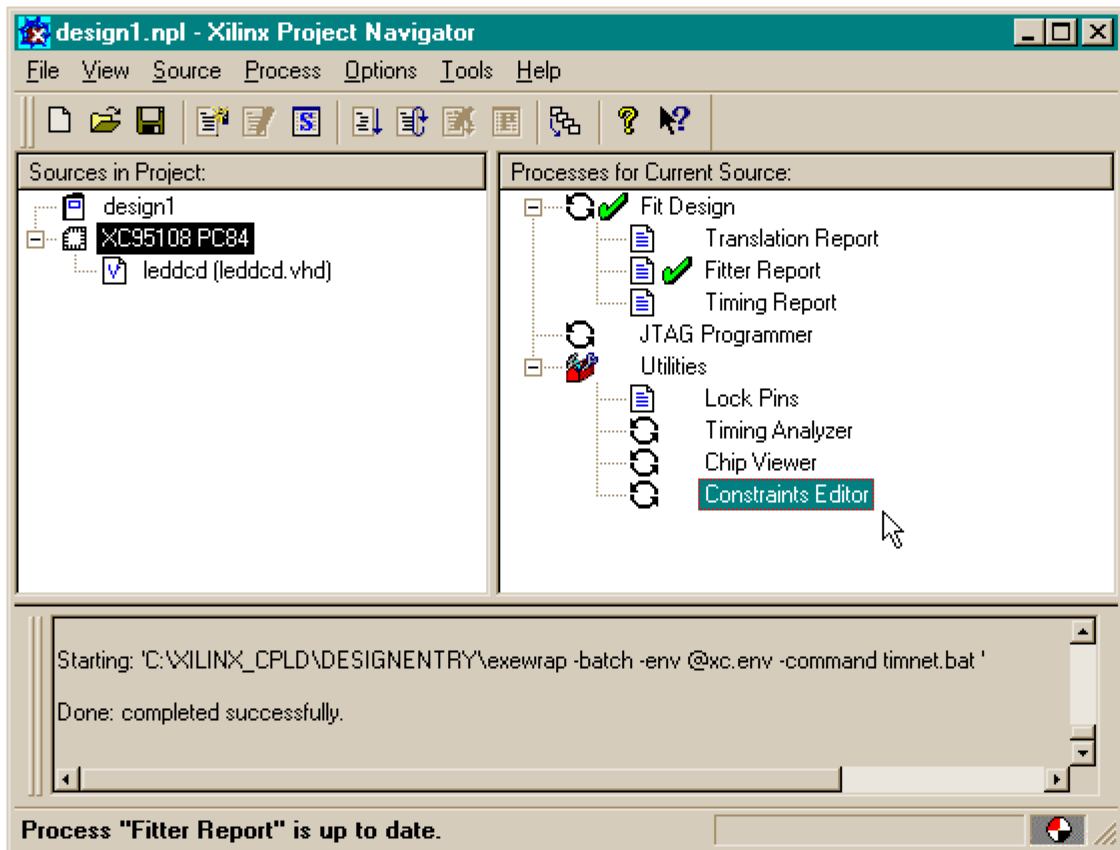
The problem we have now is that the inputs and outputs for the LED decoder were assigned to pins picked by the fitting process, but these are not the pins we actually want to use on the XS95 Board. The XS95 has eight inputs which are driven by the PC parallel port and we would like to assign the LED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

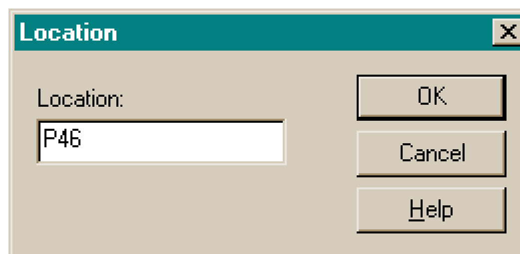
LED Decoder Output	XS95 XC95108 CPLD Pin
s0	P21
s1	P23
s2	P19
s3	P17
s4	P18
s5	P14
s6	P15

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.

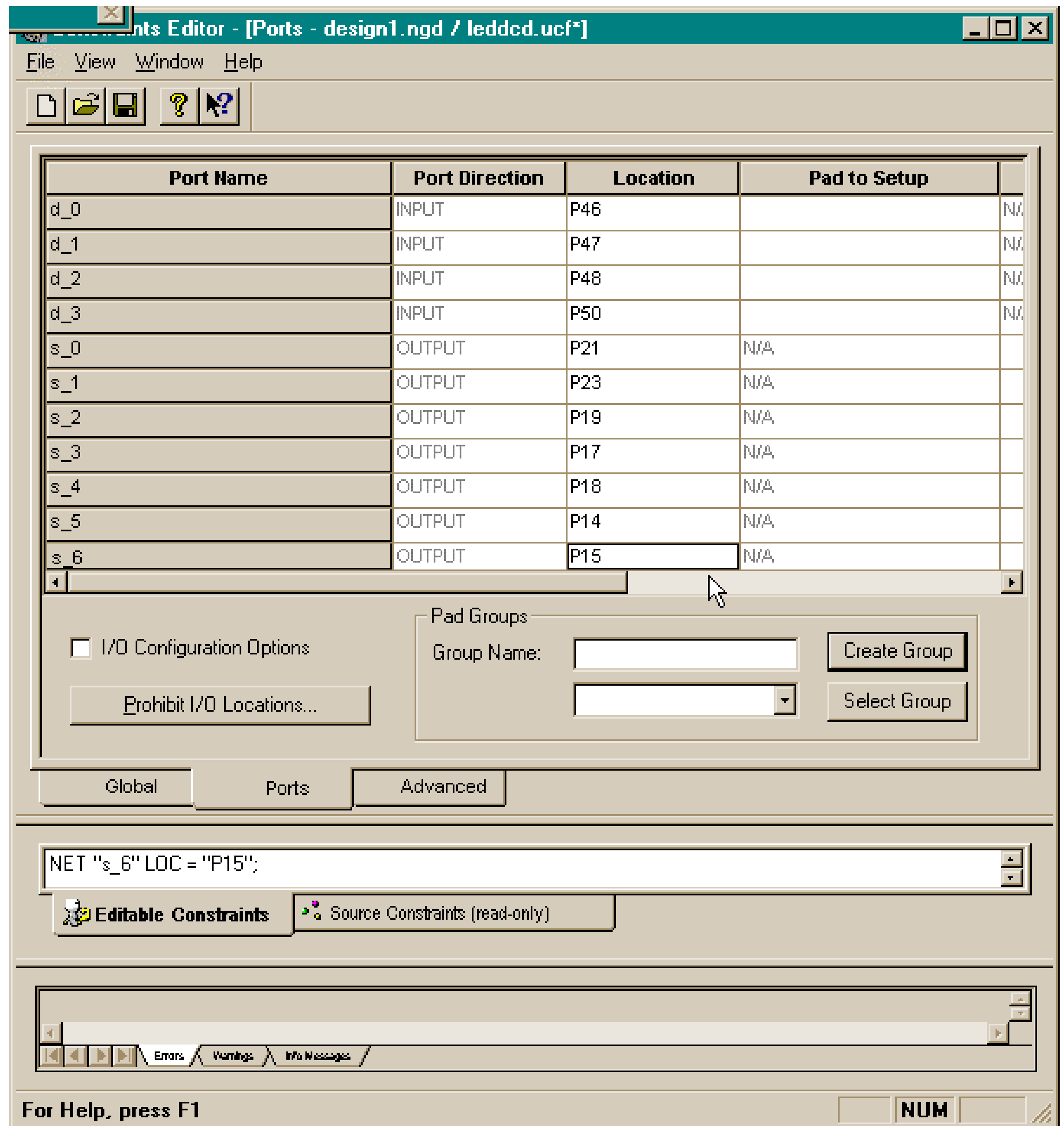


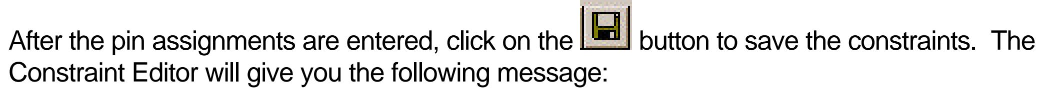
The Constraints Editor window appears. Click on the **Ports** tab in the upper pane. A list of the inputs and outputs for the LED decoder will appear. We can enter our pin assigments here.

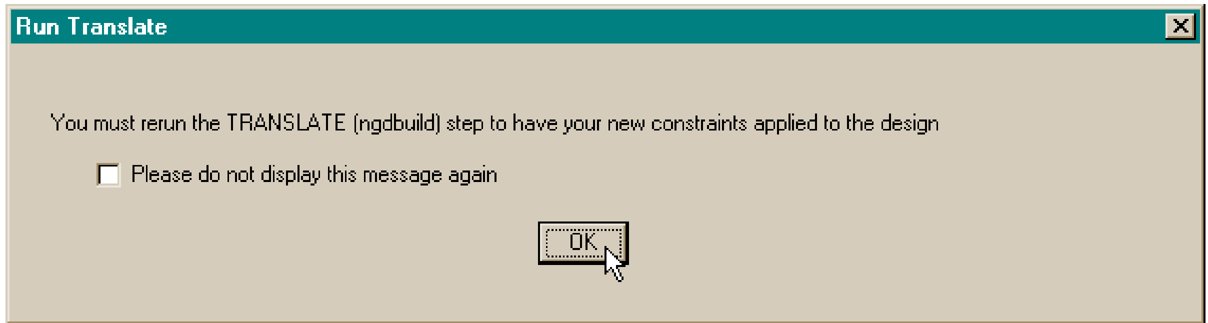
We start by double-clicking in the **Location** slot for the **d_0** input. This causes the following window to appear. Type **P46** into the slot since this is the pin we want d0 assigned to.



After clicking OK, the pin assignment will appear in the Location slot for pin d0. We can repeat the process for the rest of the inputs and outputs. After doing this, the Constraints Editor window appears as follows:


...nts Editor - [Ports - design1.ngd / leddcd.ucf*]
File View Window Help
Port Name
Port Direction
Location
Pad to Setup
d_0 INPUT P46
d_1 INPUT P47
d_2 INPUT P48
d_3 INPUT P50
s_0 OUTPUT P21 N/A
s_1 OUTPUT P23 N/A
s_2 OUTPUT P19 N/A
s_3 OUTPUT P17 N/A
s_4 OUTPUT P18 N/A
s_5 OUTPUT P14 N/A
s_6 OUTPUT P15 N/A
I/O Configuration Options
Prohibit I/O Locations...
Pad Groups
Group Name:
Create Group
Select Group
Global
Ports
Advanced
NET "s_6" LOC = "P15";
Editable Constraints
Source Constraints (read-only)
For Help, press F1
NUM

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



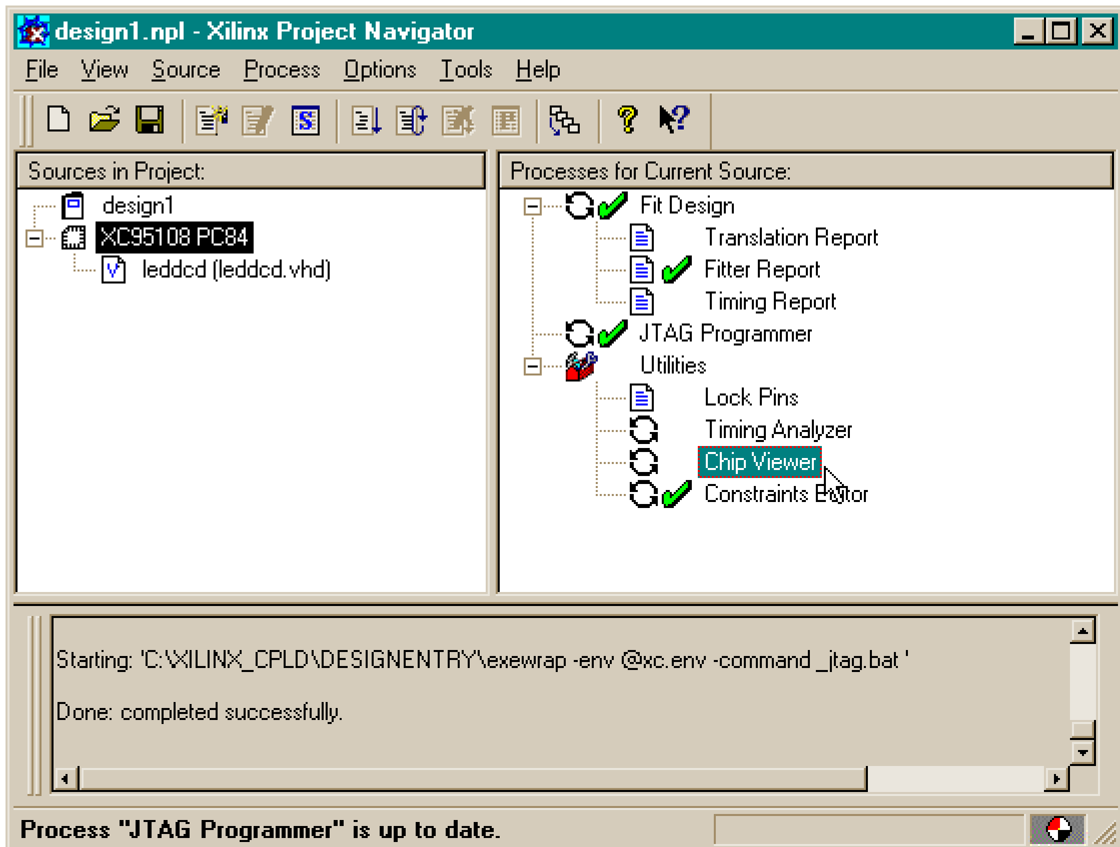
This just means we have to re-run the fitting process again if we want the design to use the pin assignments we just made. Click on OK and then double-click the Fit Design process to re-fit the design with the new pin assignments. Then double-click on the Fitter Report process to view the pin assignments 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       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 so it appears we accomplished what we wanted.

Viewing the Chip

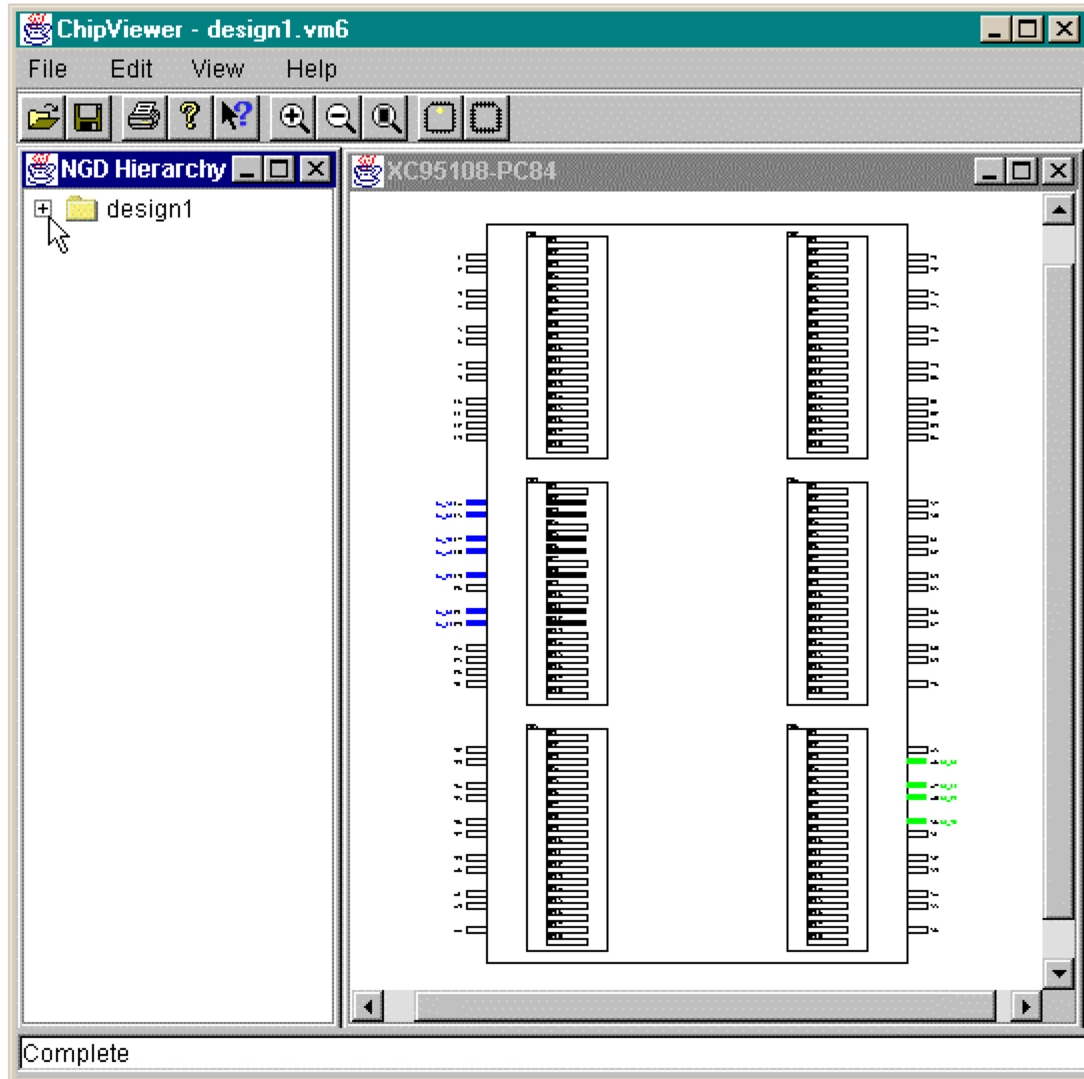
If you installed the ChipViewer with the rest of the WebPACK tools, then you can get a graphical depiction of how the logic circuitry and I/O are assigned to the CPLD macrocells and pins. Just highlight the XC95108 Pc84 object in the Sources pane and then double-click the **Chip Viewer** process.

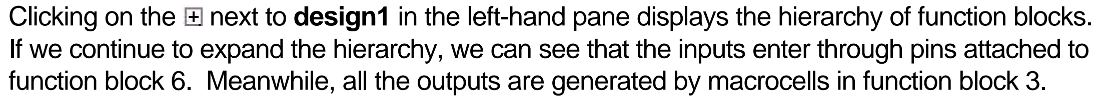


The ChipViewer window will appear containing two panes:

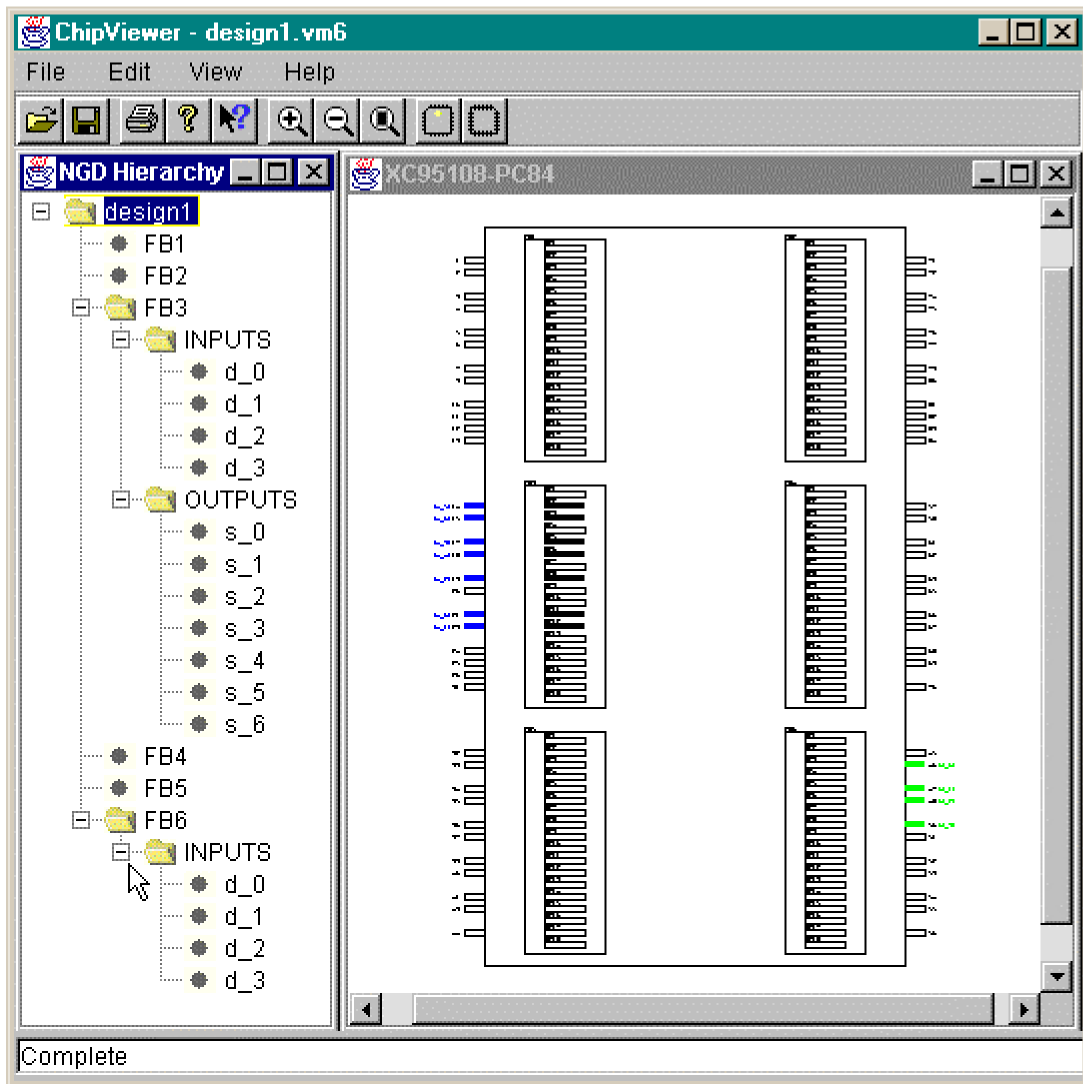
1. The left-hand pane lists the inputs and outputs assigned to the various function blocks in the XC95108 PC84 CPLD.

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.)

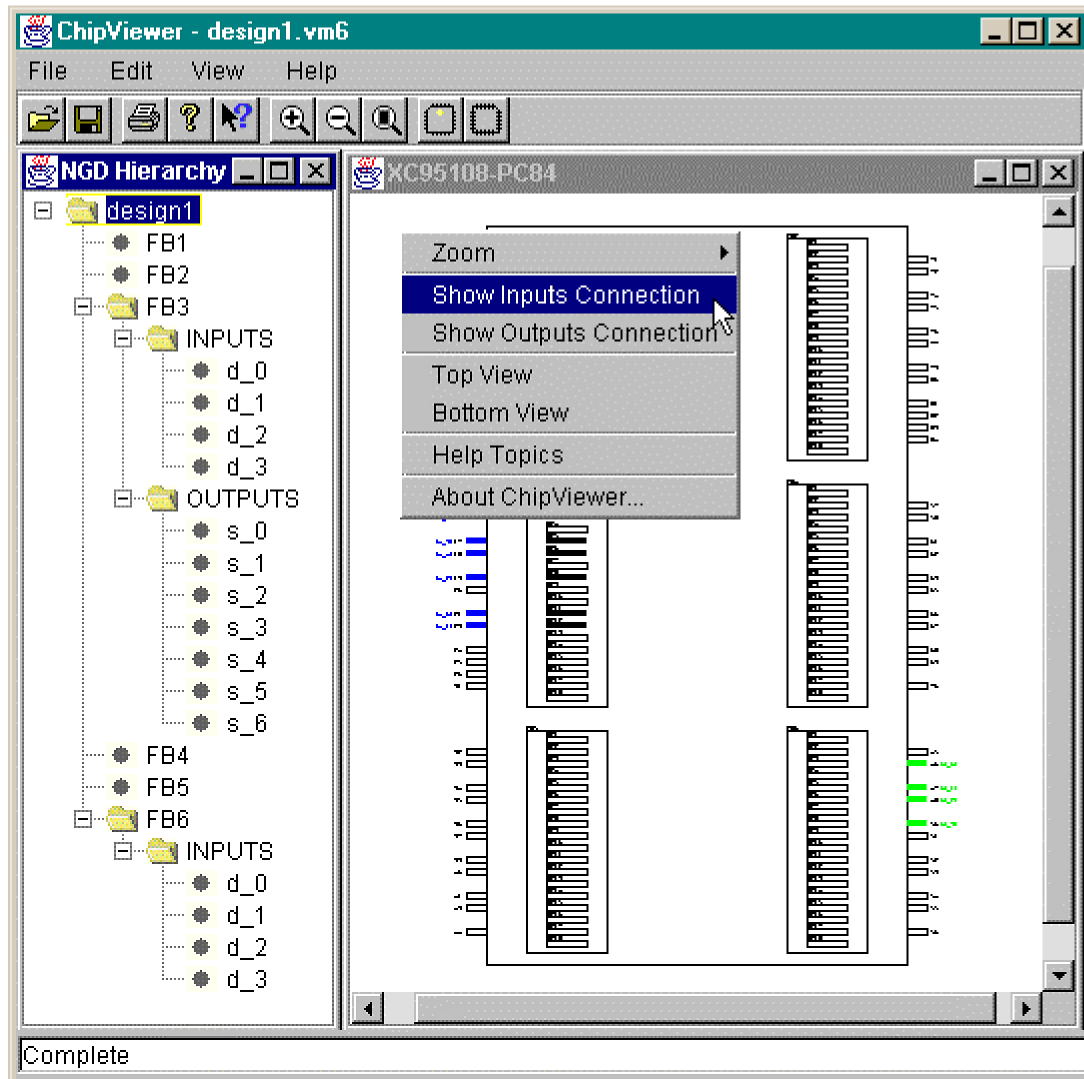




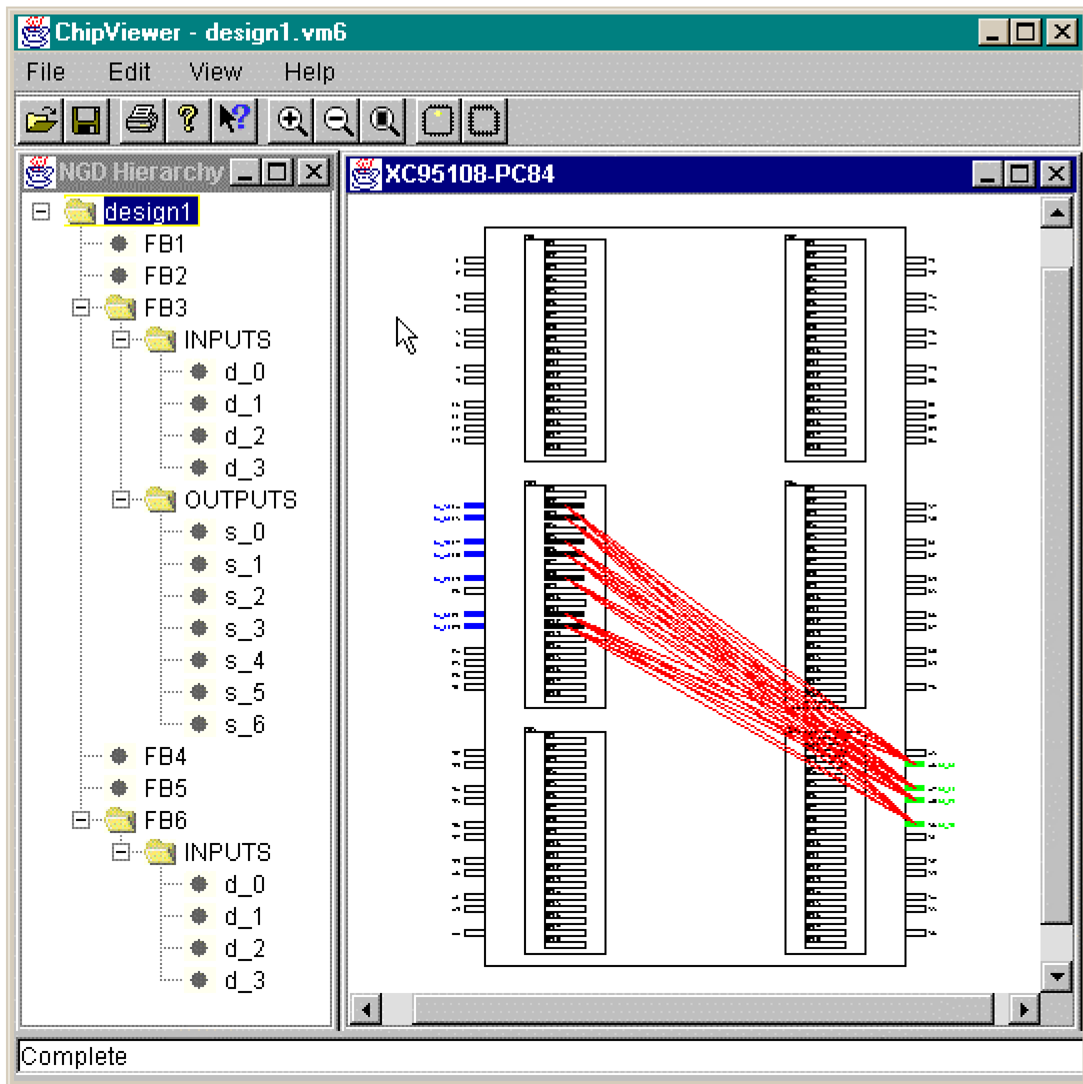
Clicking on the ⊞ next to **design1** in the left-hand pane displays the hierarchy of function blocks. If we continue to expand the hierarchy, we can see that the inputs enter through pins attached to function block 6. Meanwhile, all the outputs are generated by macrocells in function block 3.

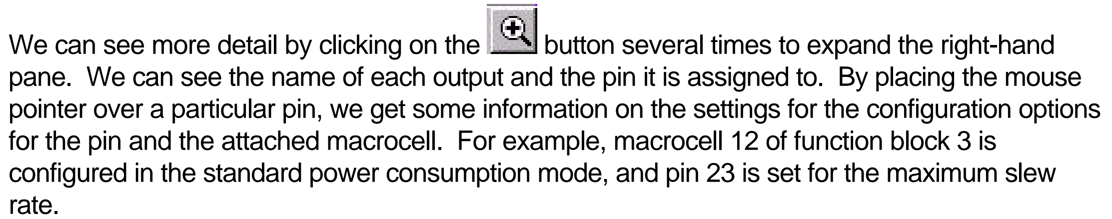


We can see which inputs affect each output by right-clicking in the right-hand pane and selecting the **Show Inputs Connection** item from the pop-up menu.

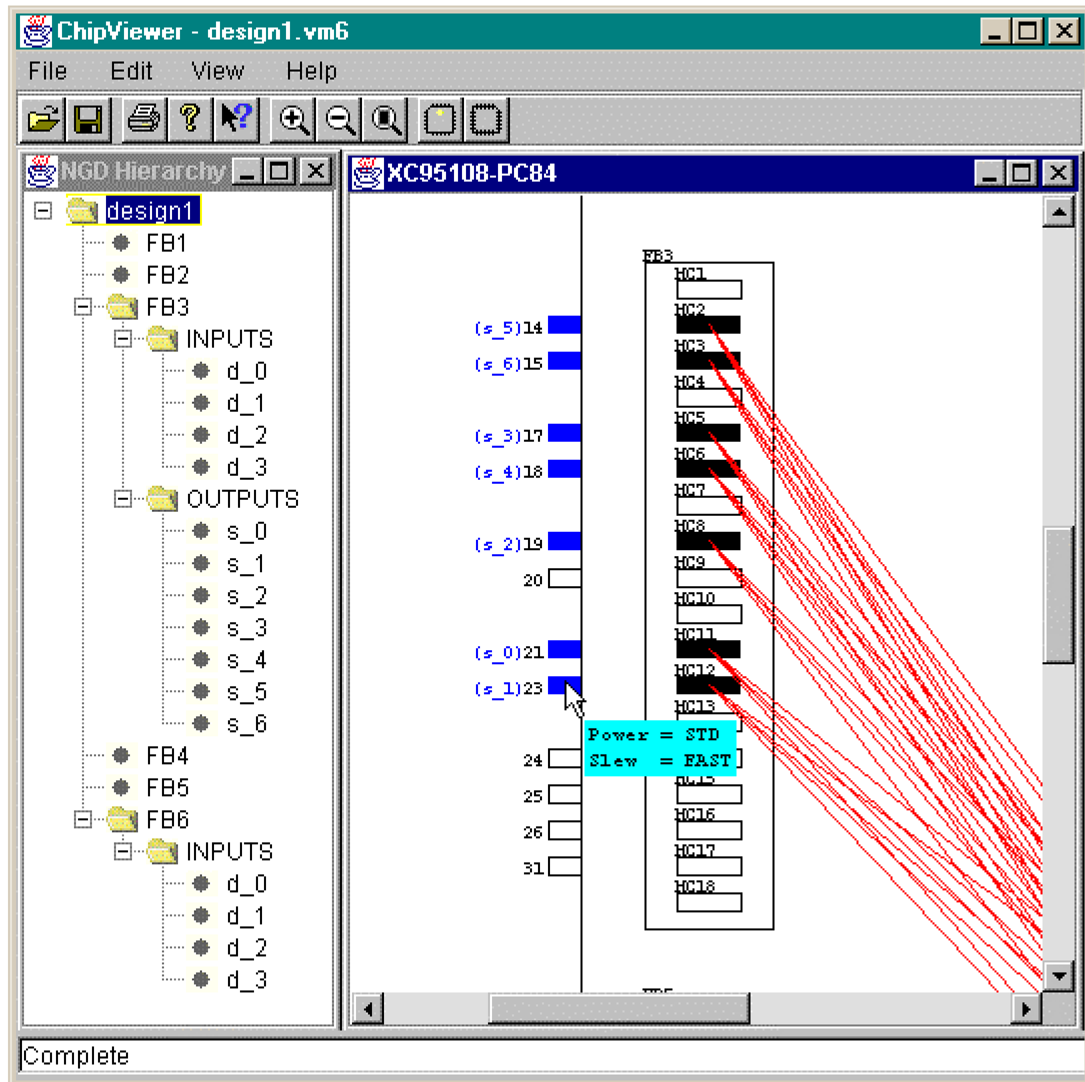


This causes red connecting lines to be drawn from each input (colored green) to the outputs it affects (colored blue) as shown below. For the LED decoder, every input affects every output so there are seven lines connecting each input to every output.



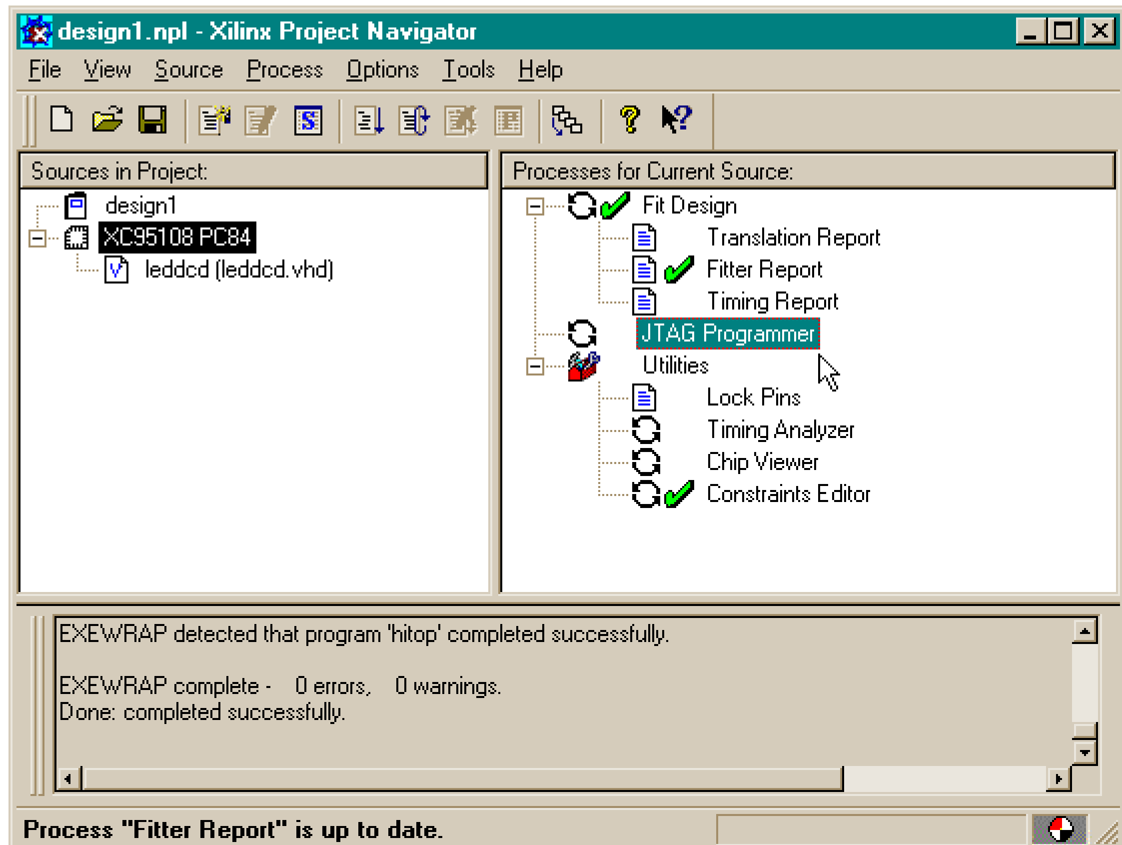


We can see more detail by clicking on the [zoom-in button] button several times to expand the right-hand pane. We can see the name of each output and the pin it is assigned to. By placing the mouse pointer over a particular pin, we get some information on the settings for the configuration options for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

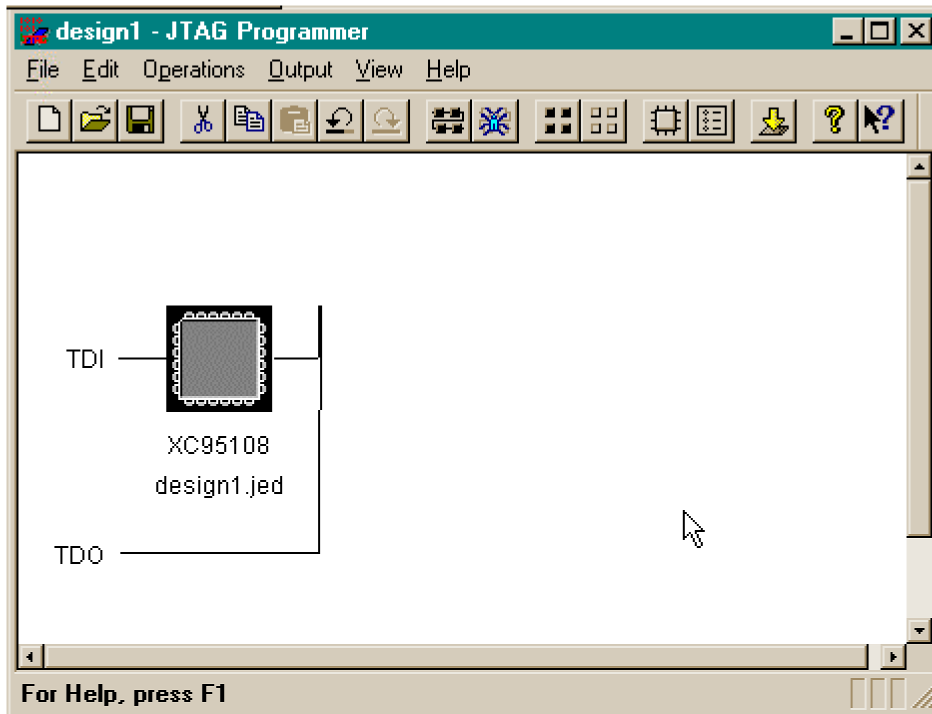


Generating the Bitstream

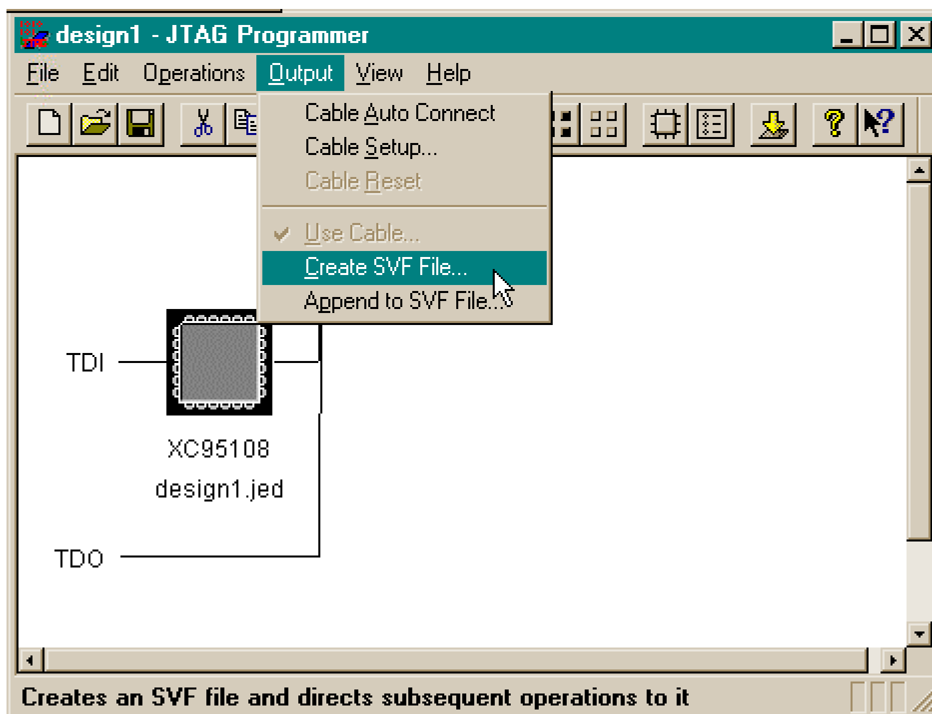
Now that we have synthesized our design, fitted it to the CPLD with the correct pin assignments, we are ready to generate the bitstream that is used to program the actual chip. To initiate the programmer, we highlight the XC95108 PC84 object in the Source pane and double-click on the **JTAG Programmer**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



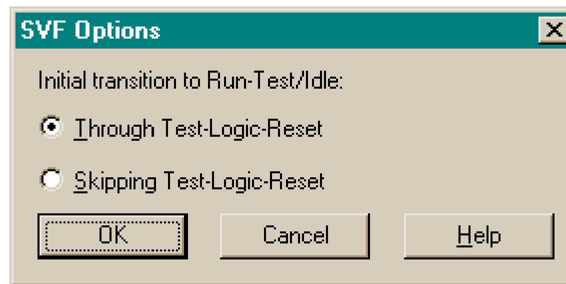
The JTAG Programmer window will appear with the chain of chips that are to be programmed. We only have one chip in our LED decoder design, so only one XC95108 CPLD is shown.



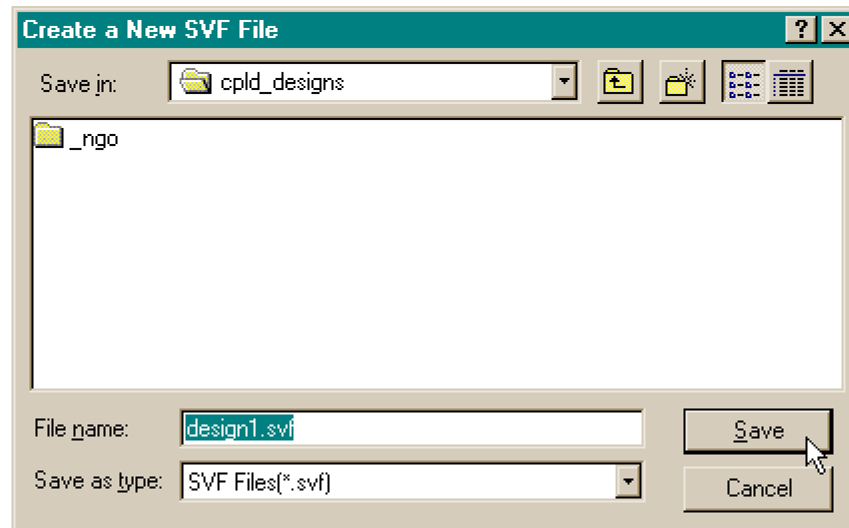
We proceed by selecting the destination for the bitstream. The XS95 Board has a separate utility called GXSLOAD for programming the CPLD, so we need to store the bitstream into a file that GXSLOAD can read. To do this, select **Output⇒Create SVF File...** as follows:



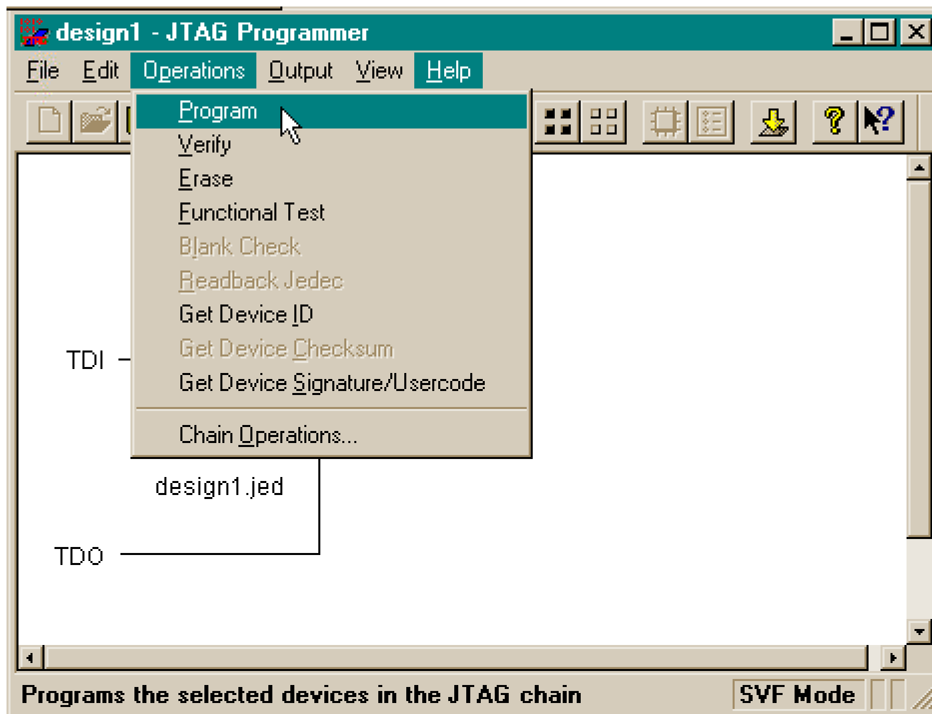
This brings up a window which asks us to make a choice about the initialization sequence of the JTAG state machine that controls the programming process. Don't even worry about this. Just accept the default and click on OK.



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 **design1.svf**.

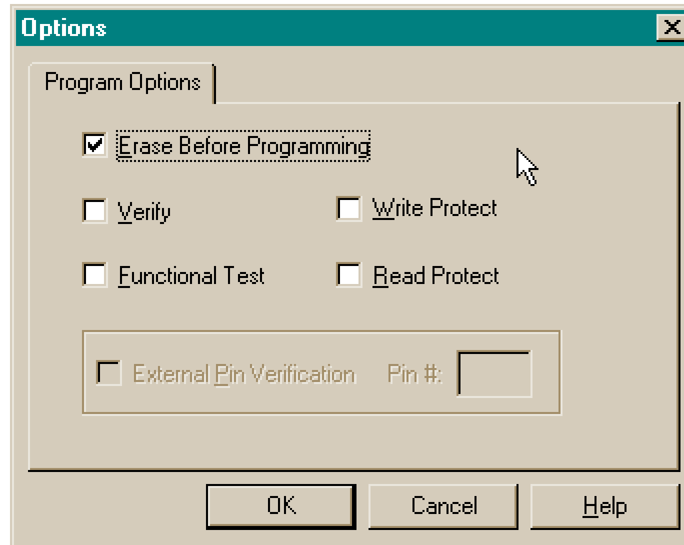


Now that we have specified the destination file for the bitstream, we can actually generate that bitstream. Click on the **Operations** menu item and select **Program** from the drop-down list. This initiates the actual bitstream generation process.

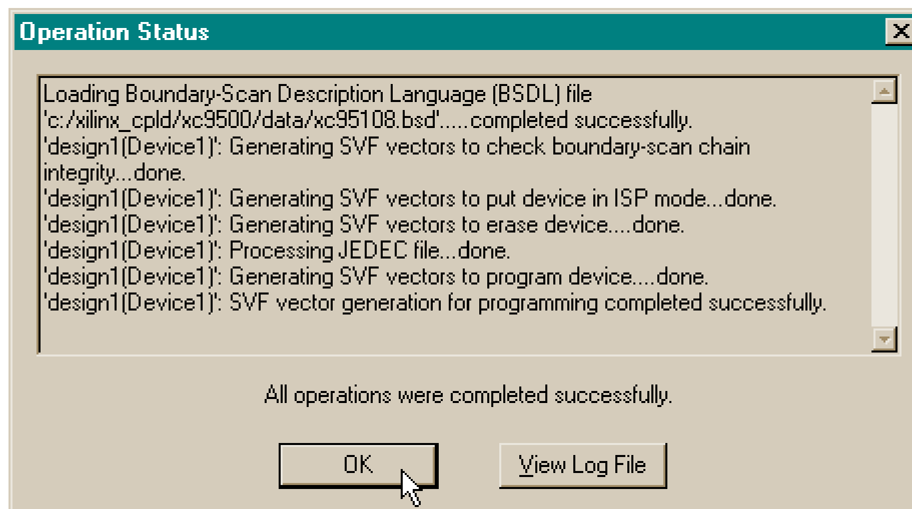


A window appears where we can set some options for the generated bitstream. We usually check the **Erase Before Programming** box so that the Flash storage in the CPLD will be erased before we start loading a new design. (The only time we can leave this box unchecked is when we are programming a CPLD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



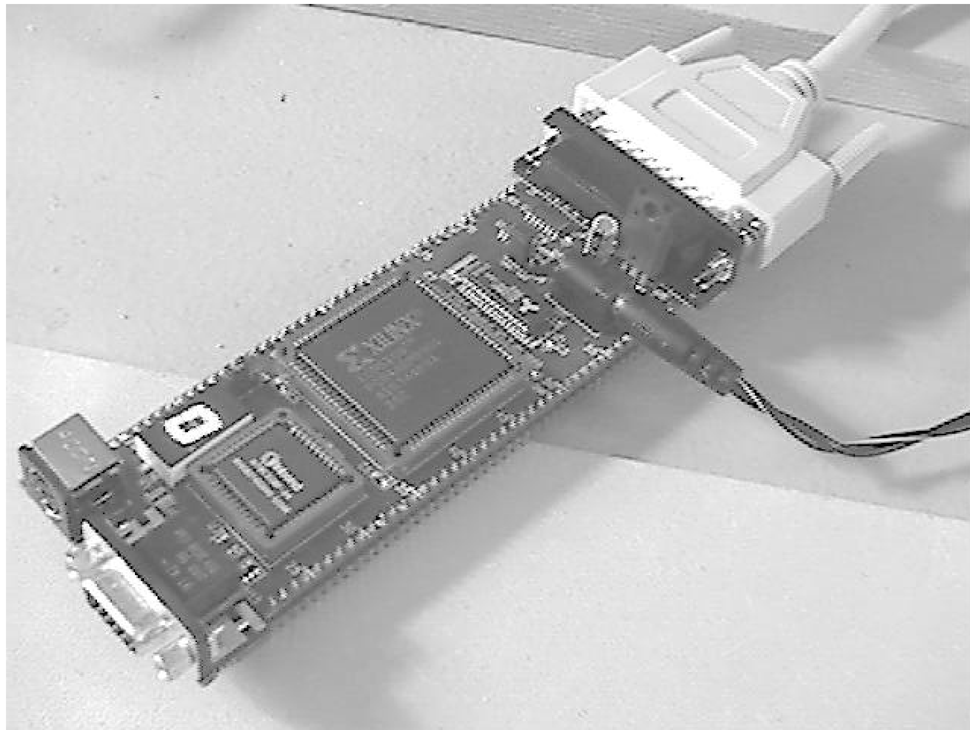
Once we click OK in the Options window, the bitstream generation process begins. The progress is reported in the **Operation Status** window:



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.)

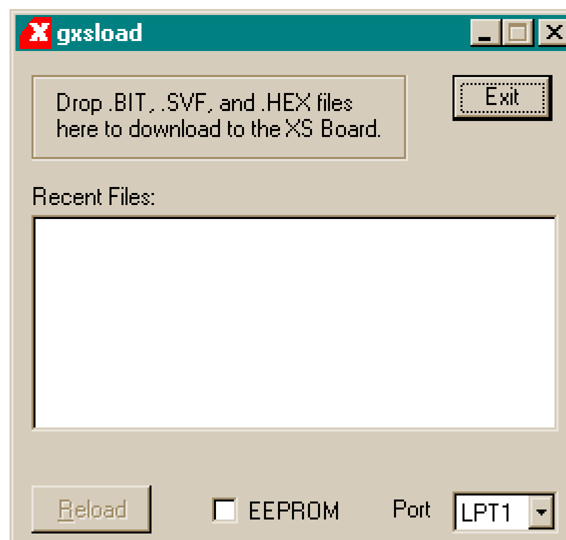
Downloading the Bitstream

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.

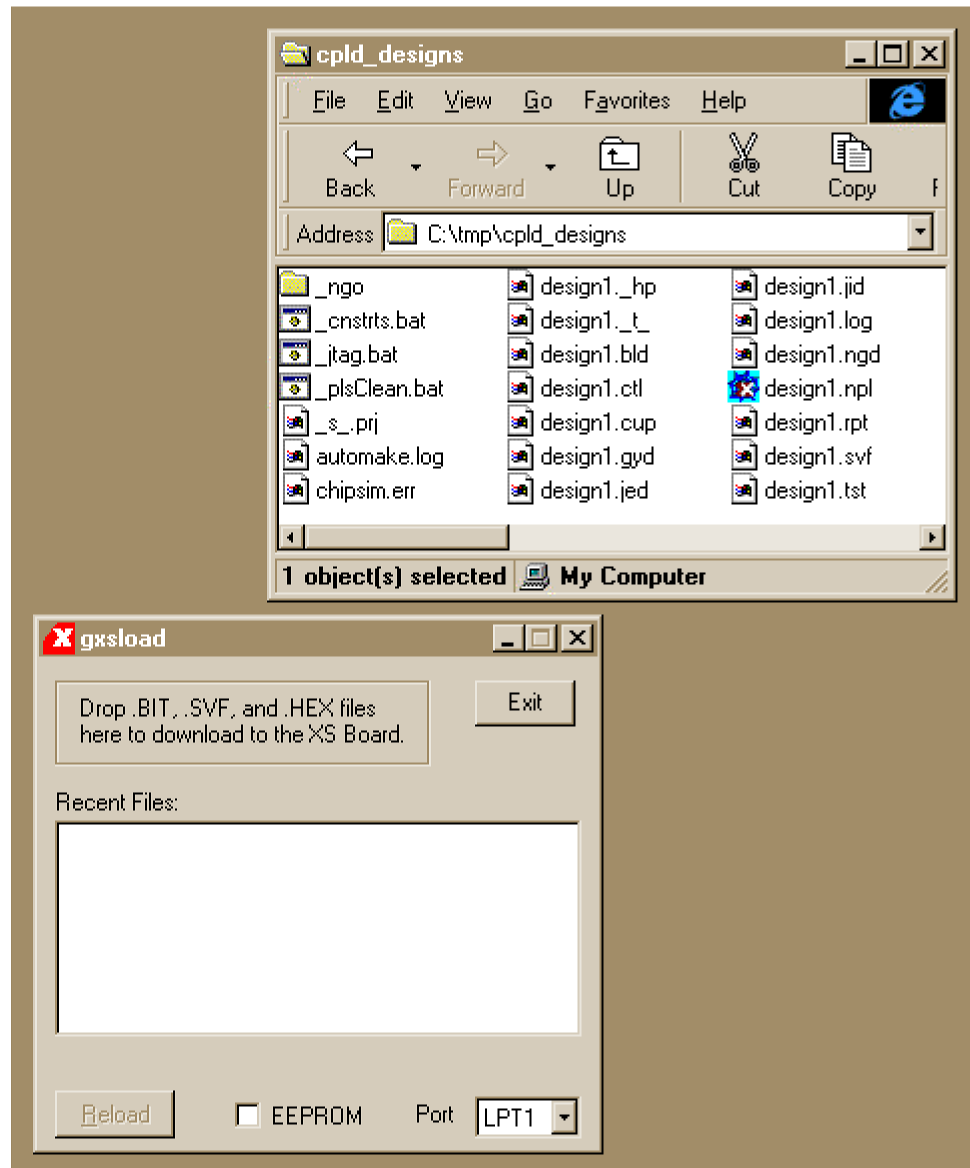


The XS95 Board is programmed using the GXSLOAD utility. We double click the

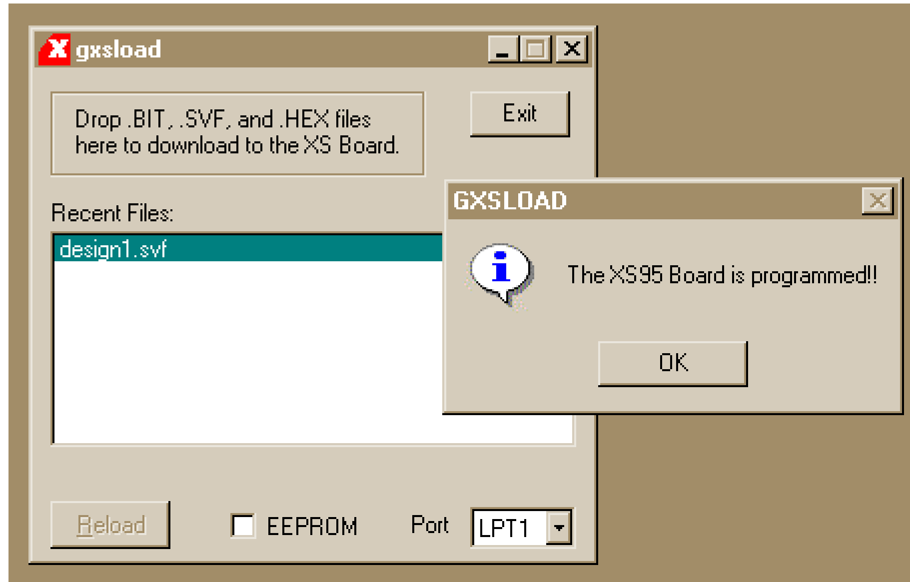
icon to bring up the following window:



Then we open a window that shows the contents of the directory where we have stored our LED decoder design (C:\tmp\cpld_designs in this case).



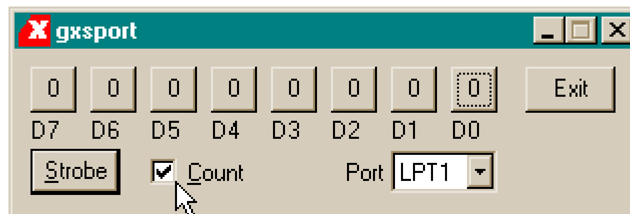
To initiate the programming of the XS95 Board, we just drag the design1.svf file from the directory window into the gxsload window. Programming the Flash of the XC95108 CPLD takes about a minute.



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 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 GXSPORT. 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.



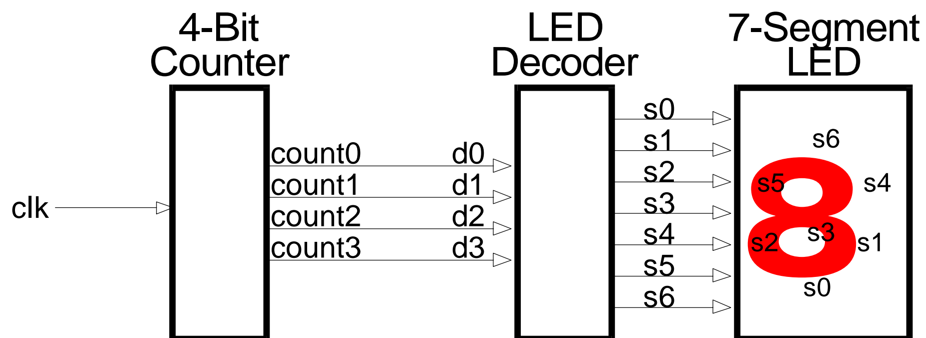
If you check the **Count** box in the gxsport window, then each click on the Strobe button increments the eight-bit value represented by D7-D0. This makes it easy to check all sixteen input combinations.

4

Hierarchical Design

A Displayable Counter

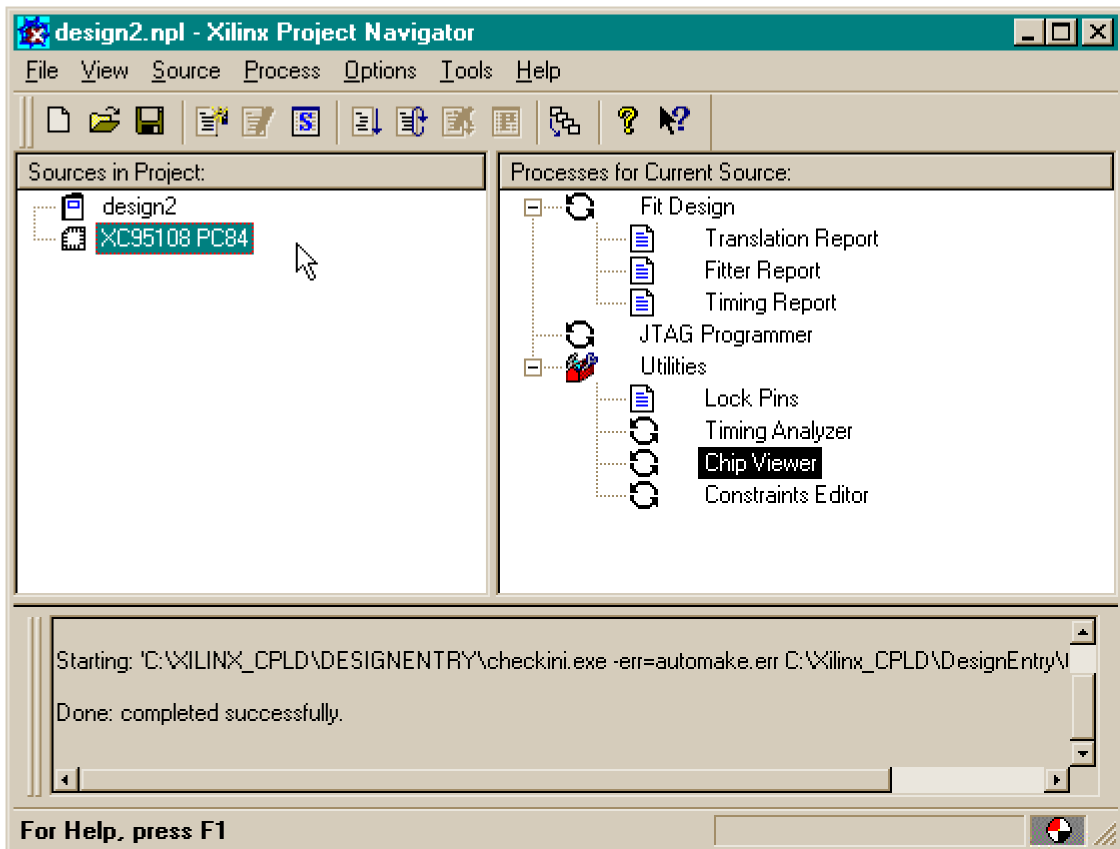
We went through a lot of work for our first CPLD design, so we will reuse it in this design: a four-bit counter whose value is displayed on a seven-segment display. The counter will increment on every rising edge of the clock. The four-bit output from the counter enters the LED decoder whereupon the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

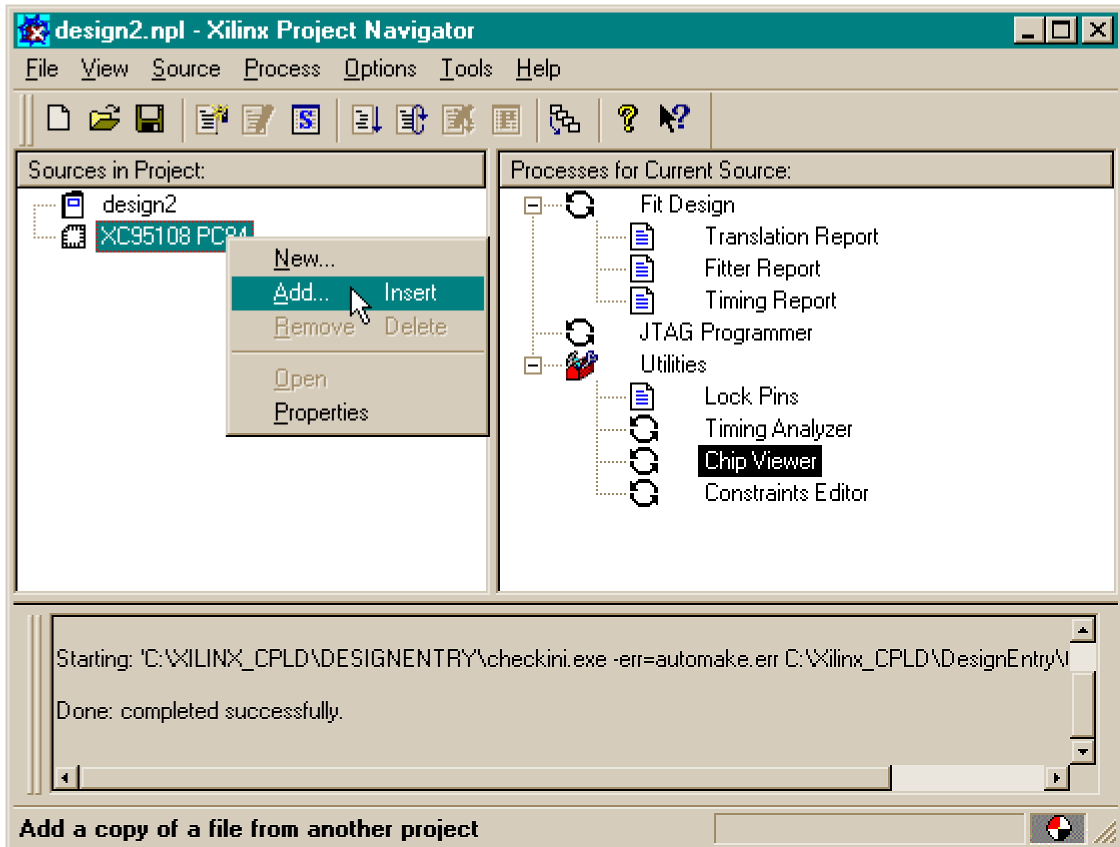
Starting a New Design

We can start a new project using the File⇒New Project... menu item. The **design2** project is placed in the same directory as the previous design: **C:\tmp\cpld_designs**. Then we set the project name property to design2 and set the device properties to target the XC95108 PC84 device just like we did in the previous example.

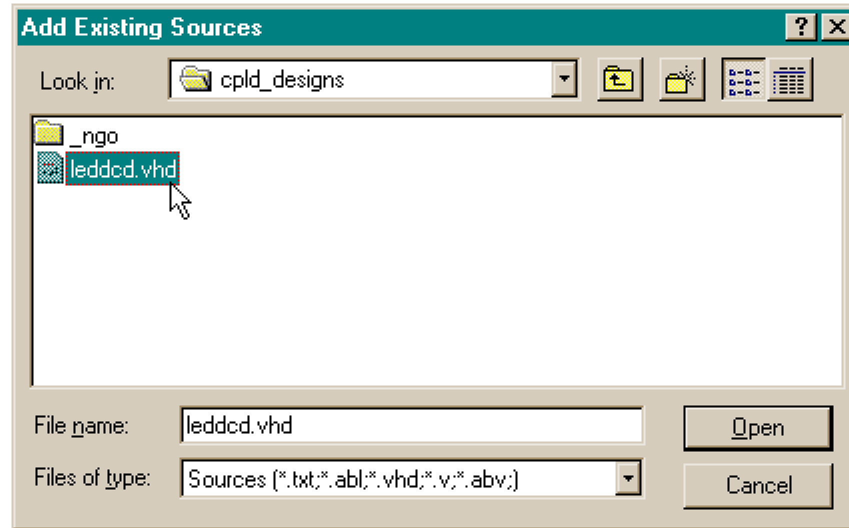


Adding the LED Decoder

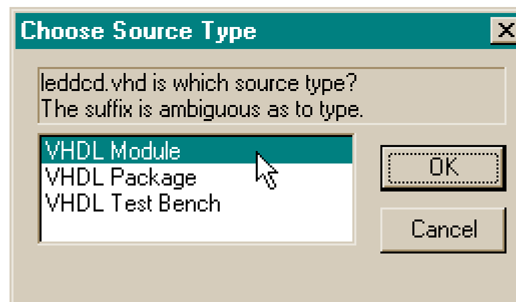
The first thing we do after getting the project started is to add the LED decoder module. We do this by right-clicking on the XC95108 PC84 object in the Source pane and selecting **Add...** from the pop-up menu.



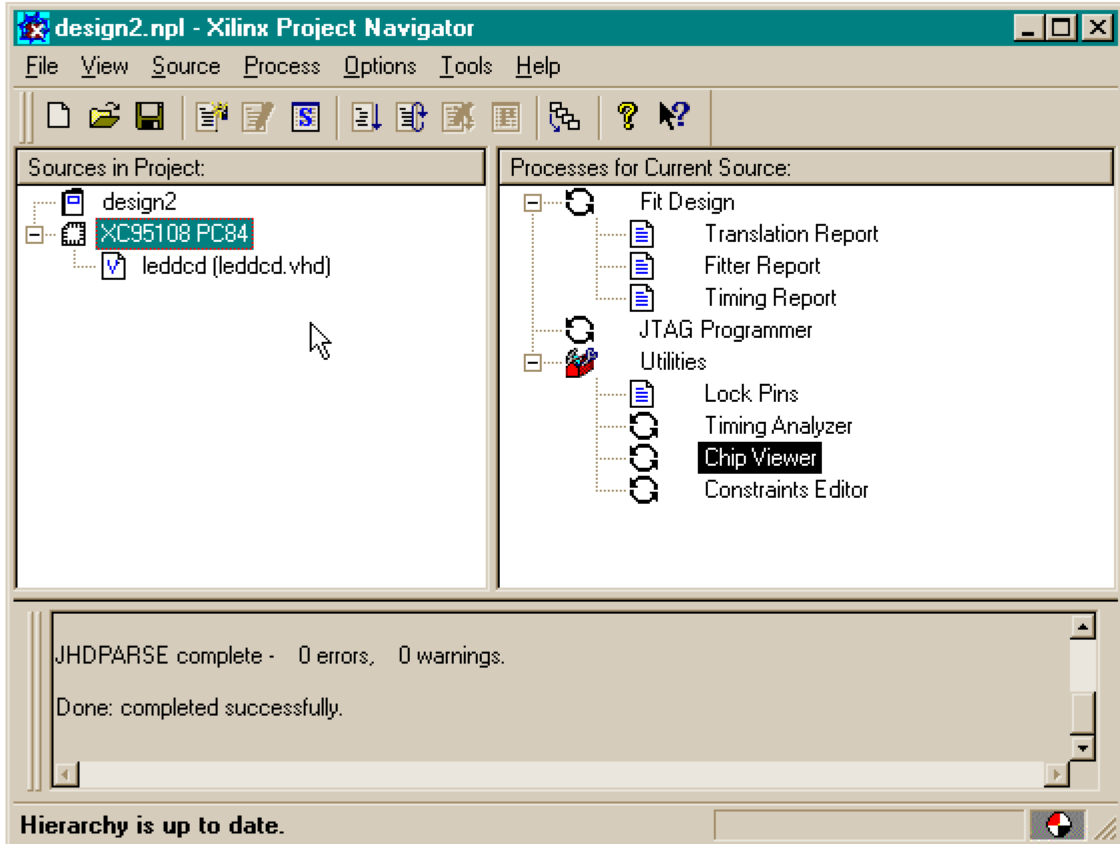
The Add Existing Sources window appears and we select the leddcd.vhd file that contains the VHDL source code for the LED decoder.



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

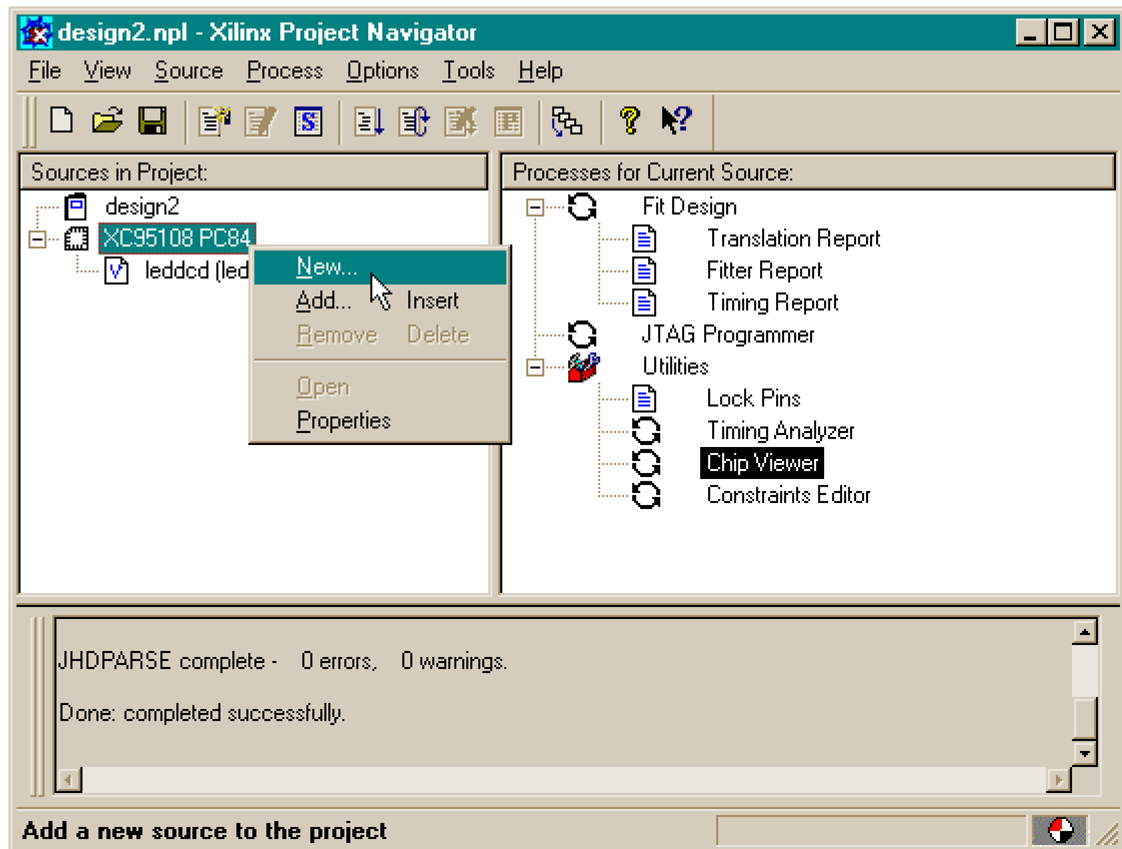


We select **VHDL Module** since the leddcd.vhd file contains a standard VHDL description of a circuit. (Packages contain extra syntactical elements for modules meant to be used as a library. Test benches contain VHDL code that exercises other VHDL modules through a sequence of tests.) After clicking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

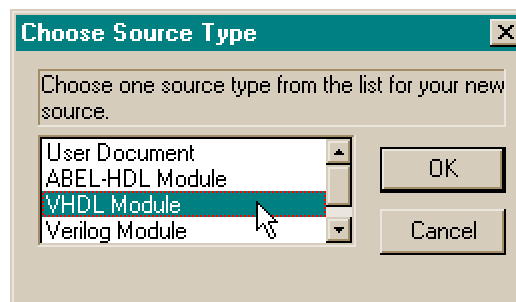


Adding a Counter

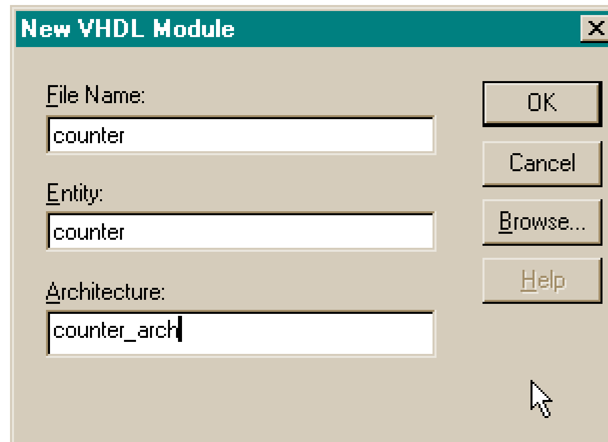
Now we have to add the counter to our design. We don't have a counter module yet, so we have to build one with VHDL. Right-click on the XC95108 PC84 object and select New... from the pop-up menu.



As in the previous example, we are prompted for the type of file we want to add to the project. Once again, we are using VHDL.



Then we are rpompted for the file name, entity name, and architecture section name which we fill-in as follows:



After clicking OK in the New VHDL Module window, we are presented with a VHDL skeleton for the four-bit counter. We flesh-out that skeleton as follows:

Xilinx HDL_ABEL Text Editor - [counter.vhd]

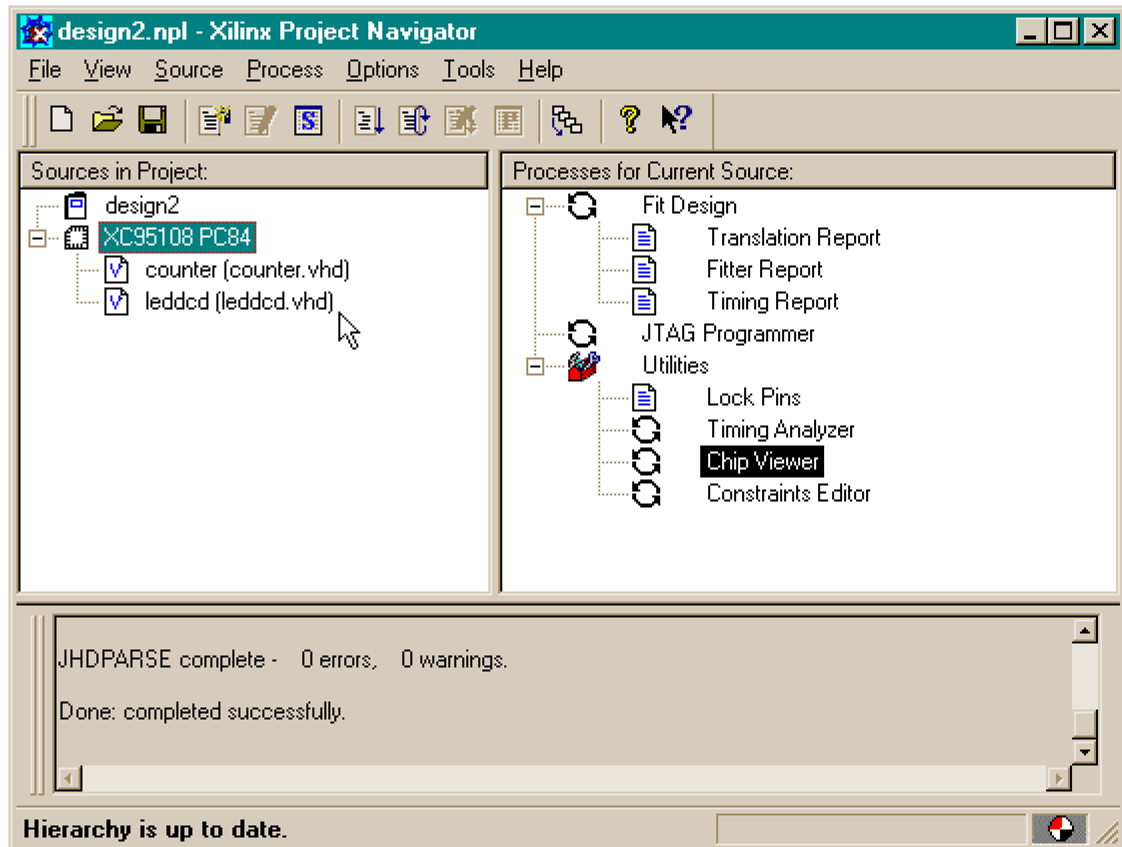
```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4
5  entity counter is
6      port(
7          clk: in std_logic;
8          count: out std_logic_vector(3 downto 0)
9      );
10 end;
11
12 architecture counter_arch of counter is
13     signal cnt, next_cnt: std_logic_vector(3 downto 0);
14 begin
15
16     next_cnt <= cnt + 1;
17
18     process(clk)
19     begin
20         if (clk'event and clk='1') then
21             cnt <= next_cnt;
22         end if;
23     end process;
24
25     count <= cnt;
26
27 end counter_arch;
```

Ln 1 Col 1 | 28 | # | WR | Rec Off | No Wrap | DOS | INS | NUM

On lines 7 and 8 we declare the clock input (**clk**) and the four-bit output from the counter (**count**). Line 13 declares two 4-bit signals: **cnt** is the current value of the counter, and **next_cnt** is the value that will be loaded into the counter on the next rising clock edge. The next_cnt value is computed by the addition of 1 to the cnt value on line 16. (We can use the high-level addition operator instead of having to describe a four-bit adder because on line 3 we have linked into the **ieee.std_logic_unsigned.all** package which supports unsigned arithmetic.)

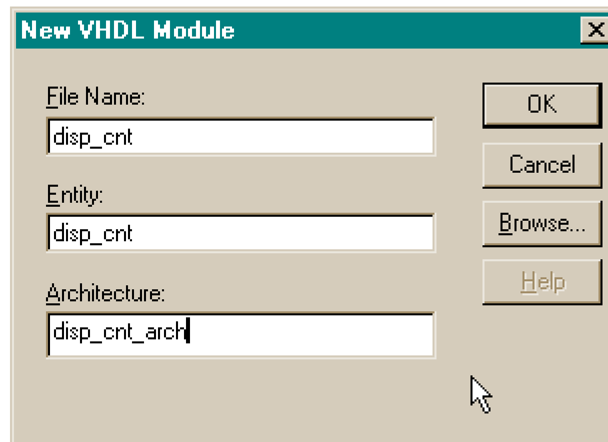
The process on lines 18-23 controls when the next_cnt value becomes the current value of the counter. The condition clause of line 20 is only true when the value on the clk input goes from 0 to 1. Then the statement on line 21 is activated which loads the value from the next_cnt signal into the current count signal. Finally, line 25 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.

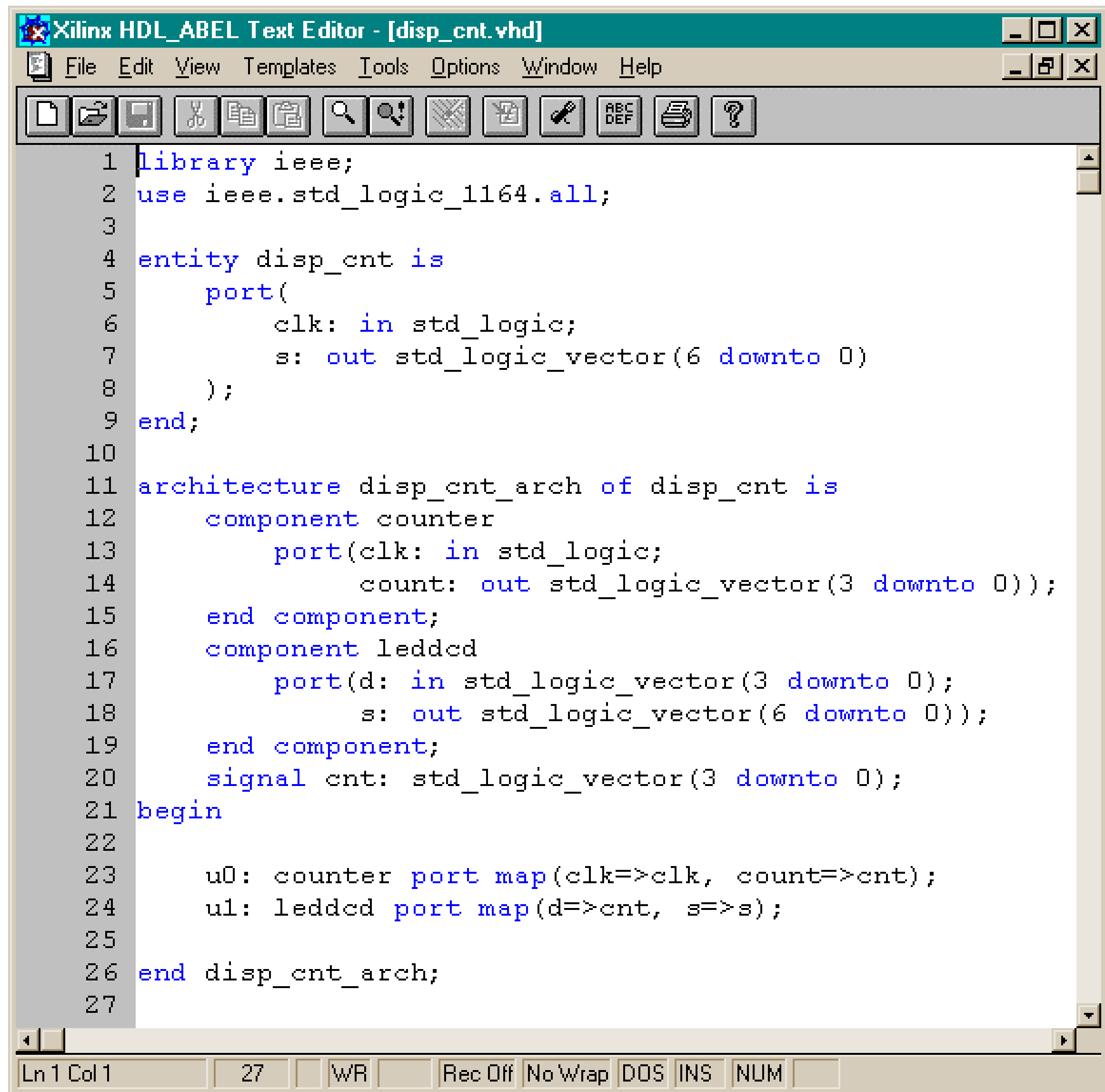


Tying Them Together

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 with the following file, entity, and architecture names:



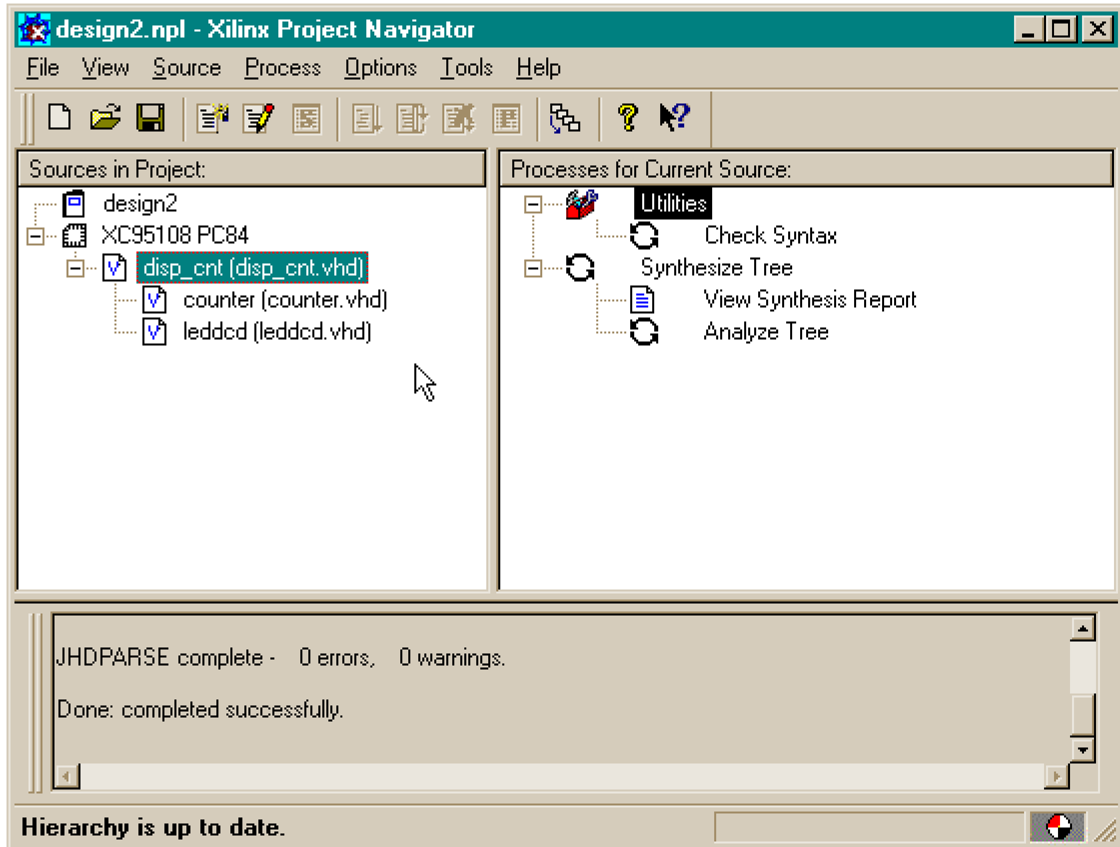
Then we fill-out the resulting VHDL skeleton so it reads as follows:



The **disp_cnt** module has a single clock input (**clk**) and seven outputs to drive an LED digit (**s**) declared on lines 6 and 7. Within the architecture section, we declare the I/O structure of the counter and LED decoder modules (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.

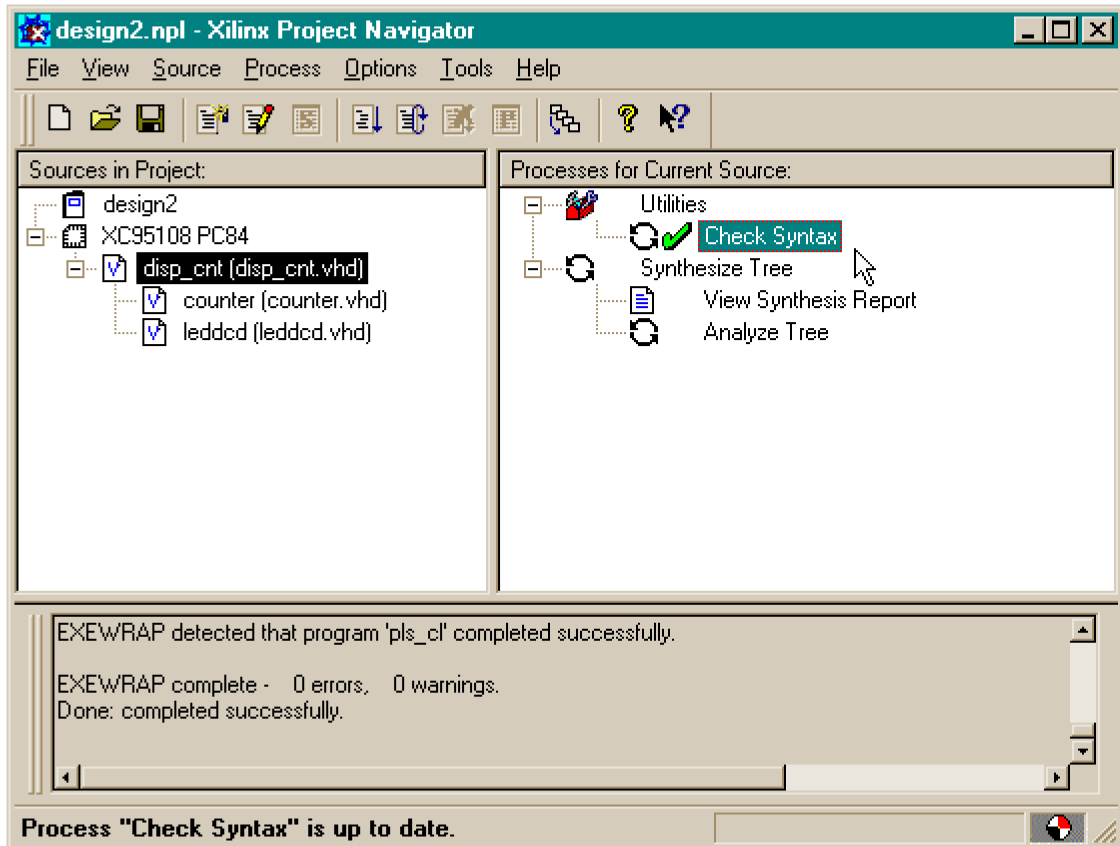
On line 23, we instantiate the counter module. The clock input to the disp_cnt module is attached to the clock input of the counter, and the output from the counter is attached to the four-bit cnt signal. The cnt signal is attached to the input of the instantiated LED decoder, and the decoder outputs 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.

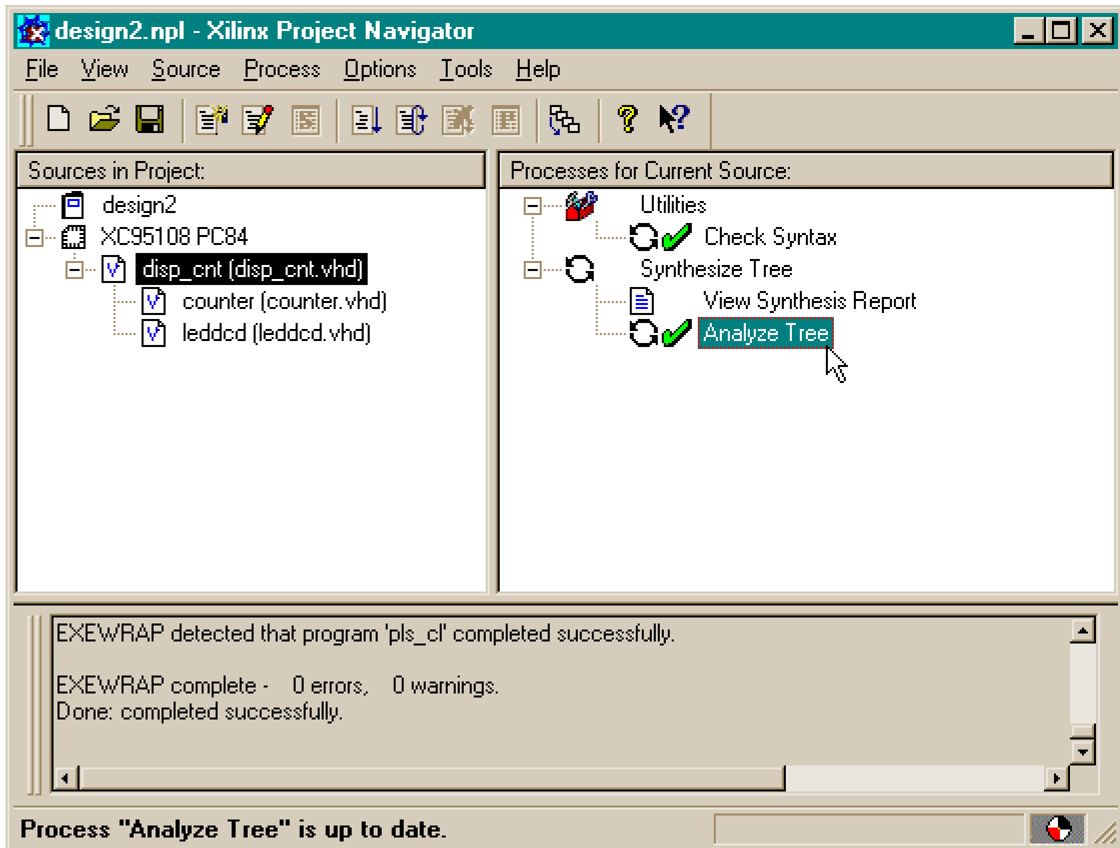


Checking the VHDL Syntax

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.

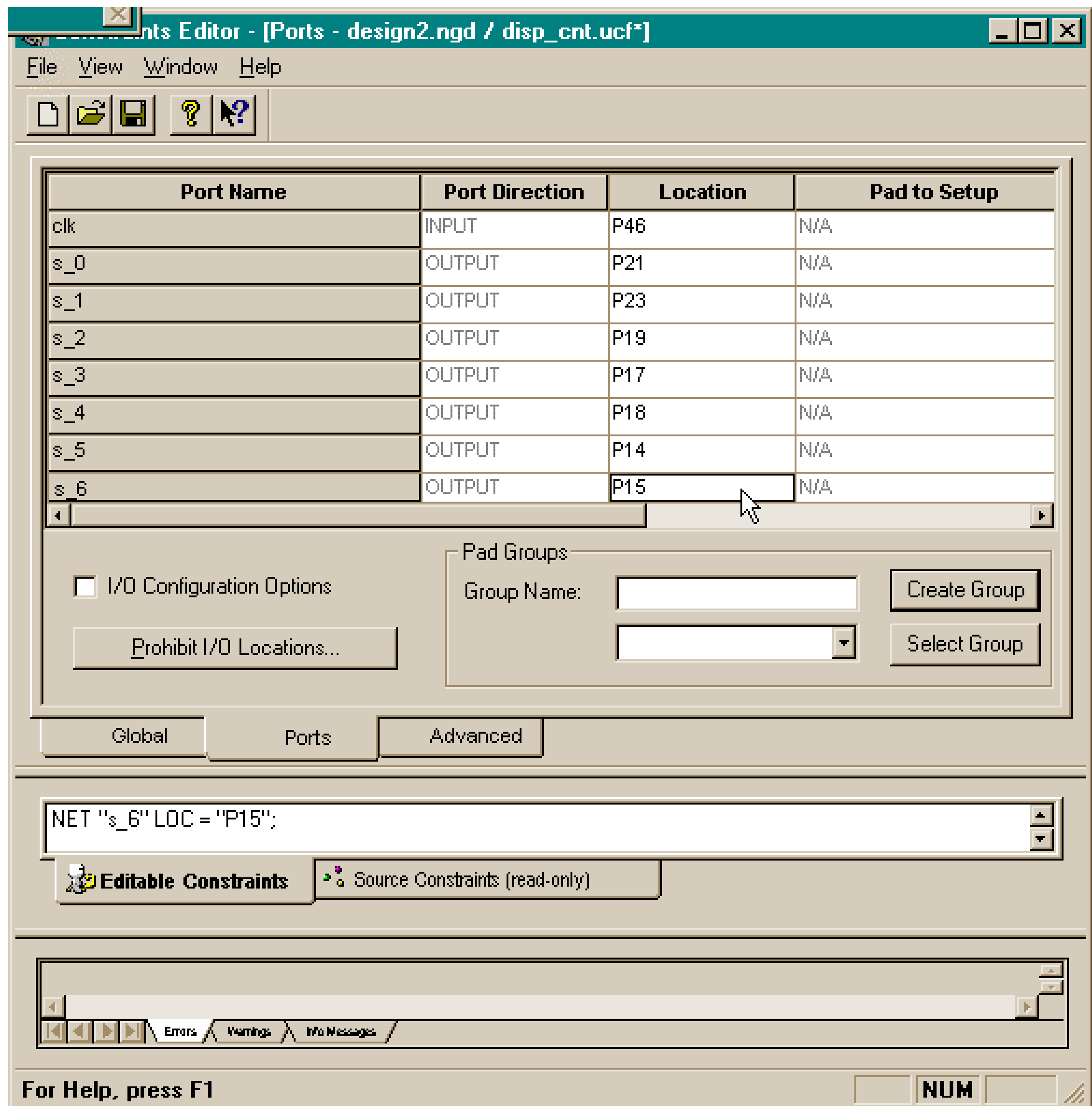


The disp_cnt module is syntactically correct as evidenced by the , but this does not mean the VHDL of the counter and LED decoder modules is correct. We can check the entire design by double-clicking the **Analyze Tree** process. This runs a process that checks the VHDL for each module and their interconnections with each other. The that appears after the analysis process completes shows we have no syntax problems in our modules.



Constraining the Design

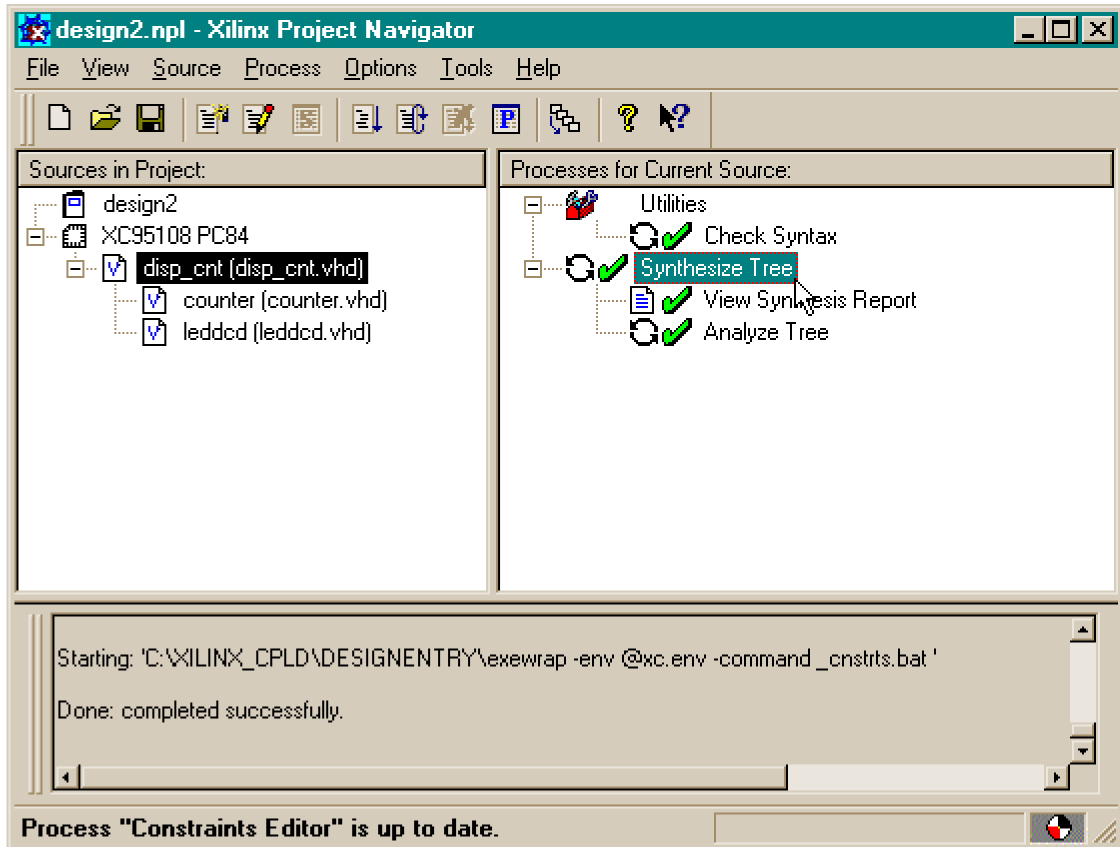
Before synthesizing the displayable counter, we need to assign the pins which the inputs and outputs will use. Highlight the XC95108 PC84 object in the Source pane and then double-click the Constraint Editor process. In the Ports tab of the Constraint Editor window, set the pin assignments for the clock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



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

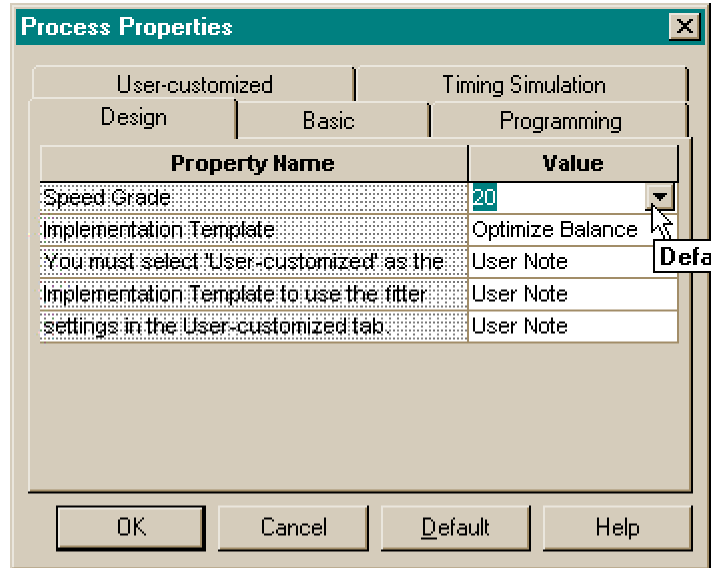
Synthesizing the Logic Circuitry for the Design

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

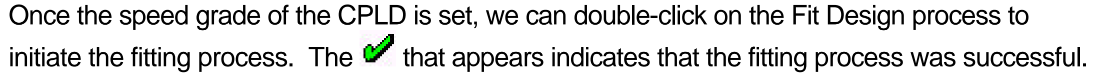


Fitting the Logic Circuitry Into the CPLD

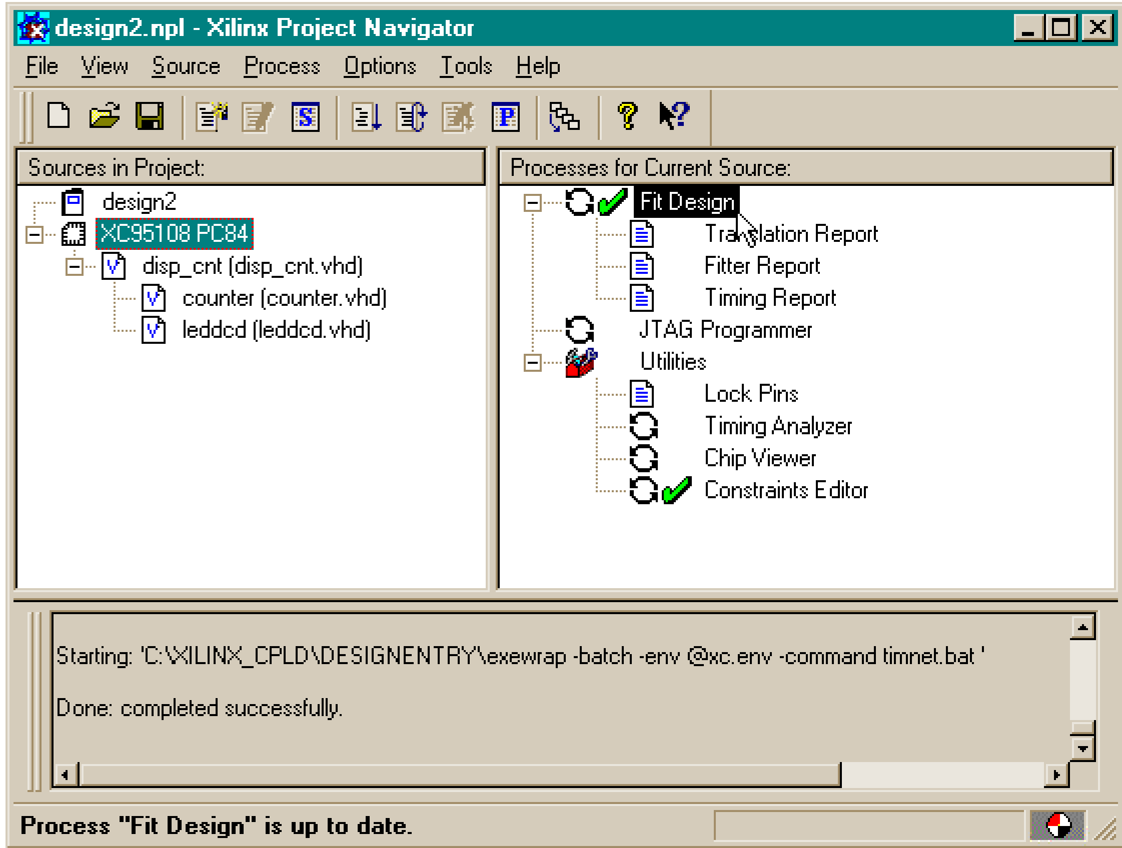
Once the netlist is synthesized, we can begin the process of fitting the it into the CPLD. Before activating the fitting process, however, we will give the fitter some information on the speed of the CPLD we are targeting. The XC95108 on the XS95 Board is a -20 part which means that the pad-to-pad delay through a single macrocell is 20 ns. Right-clicking on the Fit Design process and selecting the Properties item in the pop-up menu brings up the following window:



In the **Speed Grade** slot of the **Design** tab, we set the speed to 20. 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 Fit 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 C.17                                Xilinx Inc.
                                        Fitter Report
   Design Name: design2
Fitting Status: Successful                           Date: 10-24-1999,  8:52AM

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

Design        Device            Macrocells     Product Terms    Pins
Name          Used              Used           Used             Used
design2       XC95108-7-PC84    11 /108 ( 10%) 32 /540 (   5%)  8  /69  ( 11%)
```

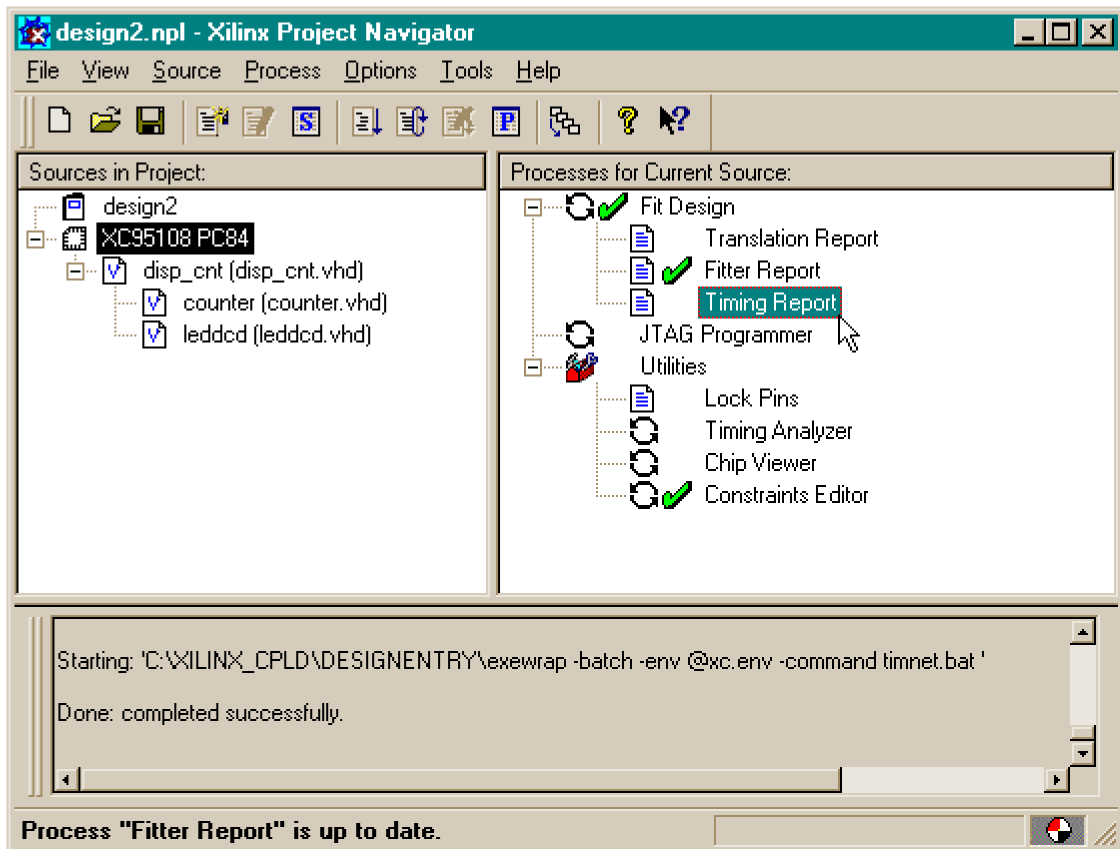
The displayable counter consumes 11 of the 108 macrocells: four for the four-bit counter and seven for the LED decoder. Looking further, we find the pin assignments for the clock input and LED decoder outputs match the assignments we made in the Constraint Editor:

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

** LOGIC **
Signal              Total   Signals Loc     Pwr  Slew Pin  Pin       Pin
Name                Pt      Used            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       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
clk                                 FB6_3             46   I/O       I
```

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:

```
------------------------------------------------------------------------
                      Clock Pad to Output Pad (tCO) (nsec)

\ From      c
 \          l
  \         k
   \
    \
     \
      \
  To   \------

s_0      28.0
s_1      28.0
s_2      28.0
s_3      28.0
s_4      28.0
s_5      28.0
s_6      28.0
```

This table says that there is a 28 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.)

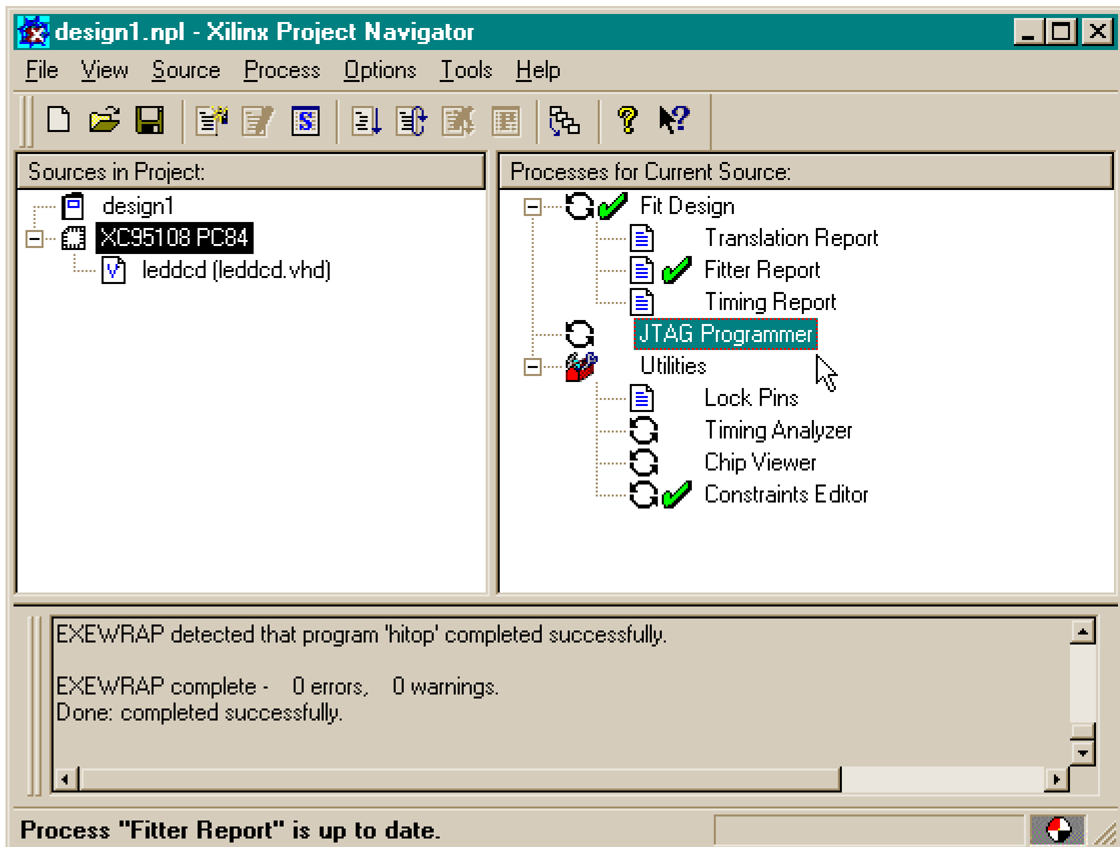
Looking further into the file we see the following table.

```
-------------------------------------------------------------------------
                          Clock to Setup (tCYC) (nsec)
                                  (Clock: clk)

\ From      N     N     N
 \          2     2     2
  \         7     8     9
   \        .     .     .
    \       Q     Q     Q
     \
      \
  To   \------------------

N26.D    12.0  12.0  12.0
N27.D          12.0  12.0
N28.D                12.0
N29.D                12.0
```

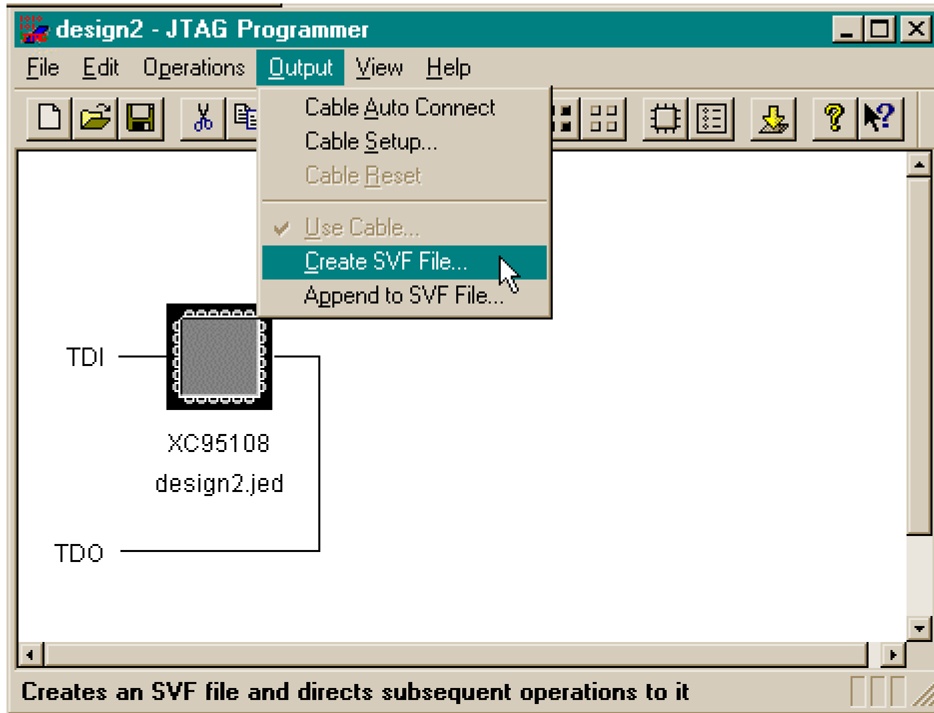
This states that a minimum of 12 ns is needed from the time the counter flip-flops are clocked until they can be clocked again. The time delay results from the sum of the clock-to-output delay of each flip-flop, internal wiring delays, and flip-flop setup times.

Generating the Bitstream

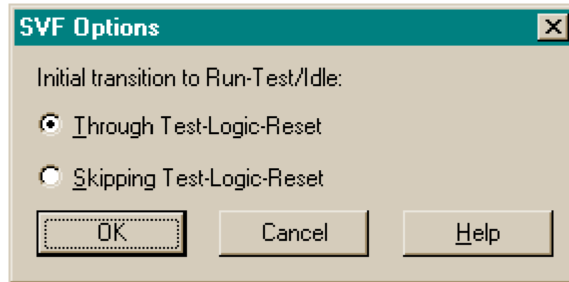
Now we are ready to generate the bitstream for the displayable counter. To initiate the programmer, we highlight the XC95108 PC84 object in the Source pane and double-click on the JTAG Programmer process.



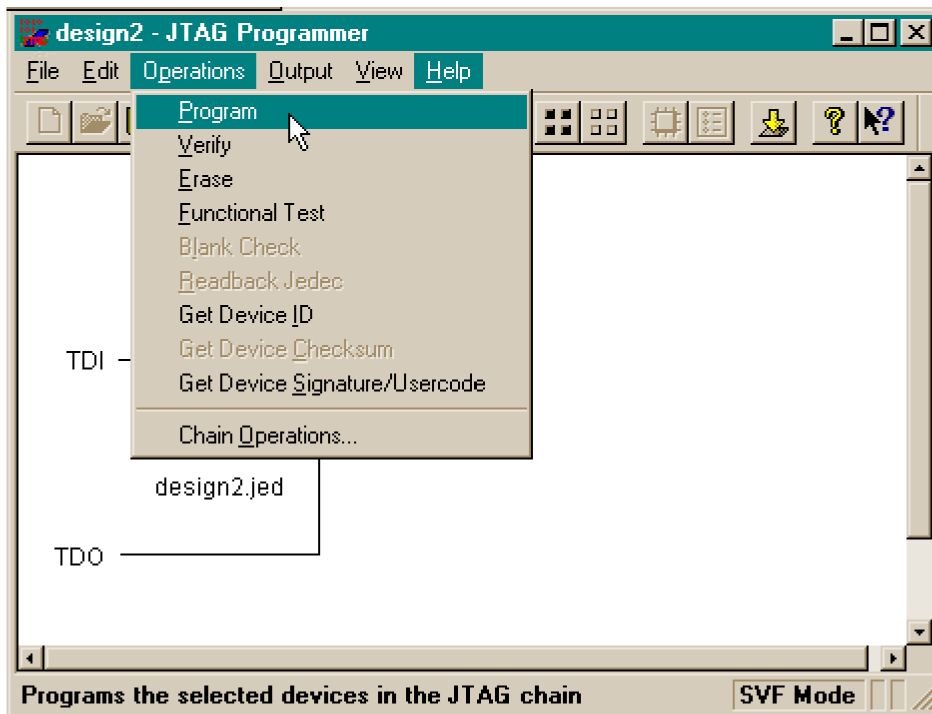
The JTAG Programmer window will appear and again we only have one chip in our design, so only one XC95108 CPLD is shown. We proceed by selecting a file as the destination for the bitstream so we can use GXSLOAD for programming the CPLD in the XS95 Board.



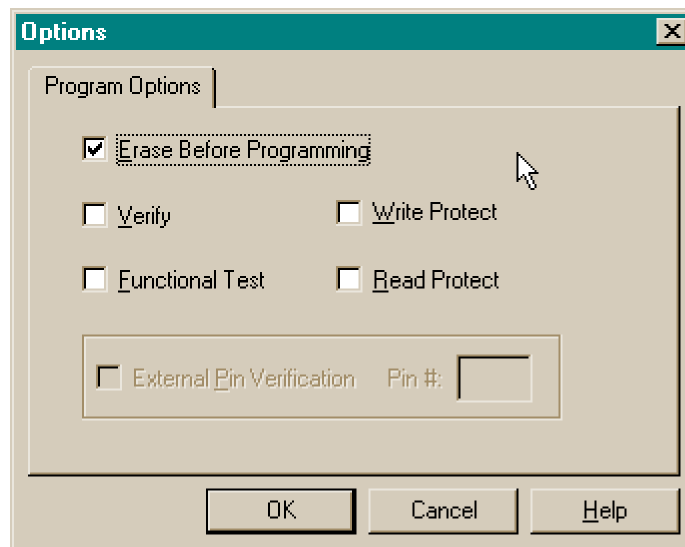
Once again we choose the default for the initialization sequence of the JTAG state machine that controls the programming process.



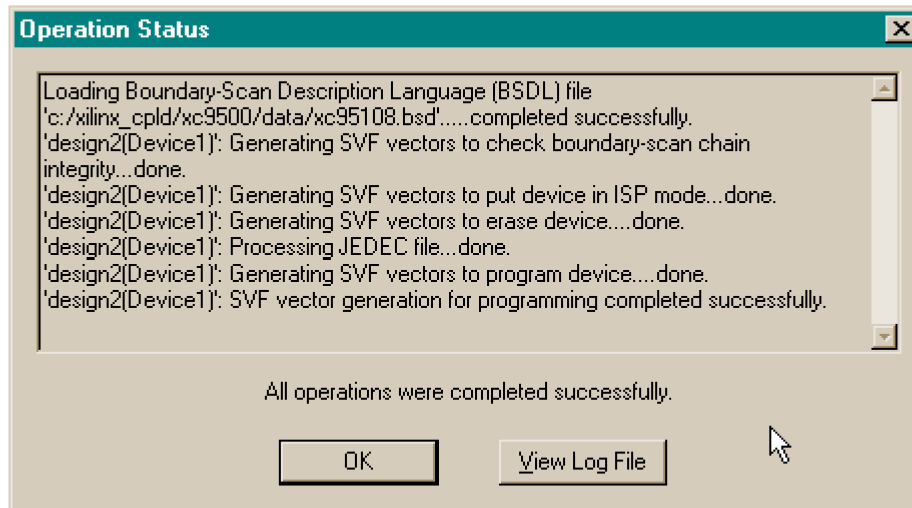
Then the window appears where we can select the default name of design2.svf as the name for the file that will hold the bitstream. We click on Save and then proceed to generate the bitstream by selecting the Operations⇒Program menu item..



A 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:

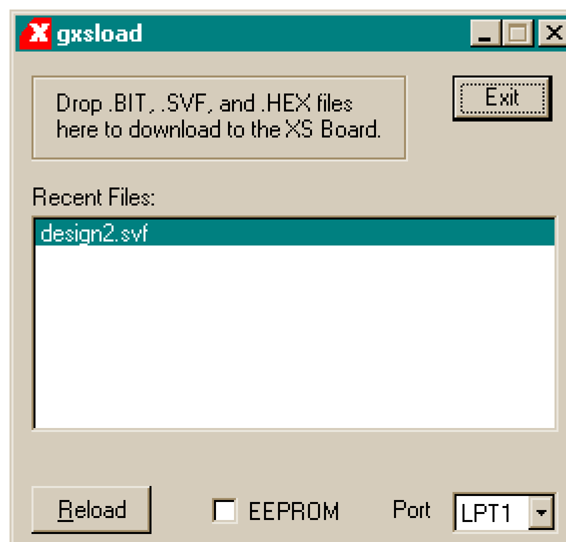


Once the bitstream file is generated, we click on OK to close the window. Then we close the JTAG Programmer window

Downloading the Bitstream

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

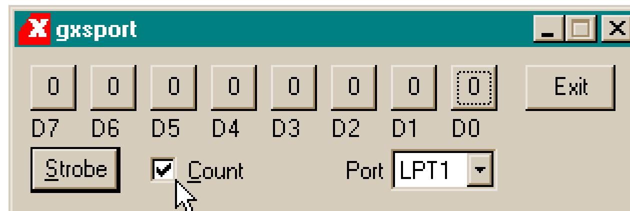
icon to bring up the GXSLOAD window. Then we drag the design2.svf file from the C:\tmp\cpld_designs directory window and drop it into the gxsload window.



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. To strobe the clock:

1. Click on the D0 button.
2. Click on Strobe
3. Click on D0 to toggle the clock bit value
4. Click on Strobe again.

This increments the value shown on the XS95 LED display

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 *The Practical Xilinx Designer Lab Book* (ISBN: 0-13021-617-8) from Prentice Hall. It presents a series of design examples of increasing complexity that are targeted toward both CPLDs and FPGAs.
- 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.