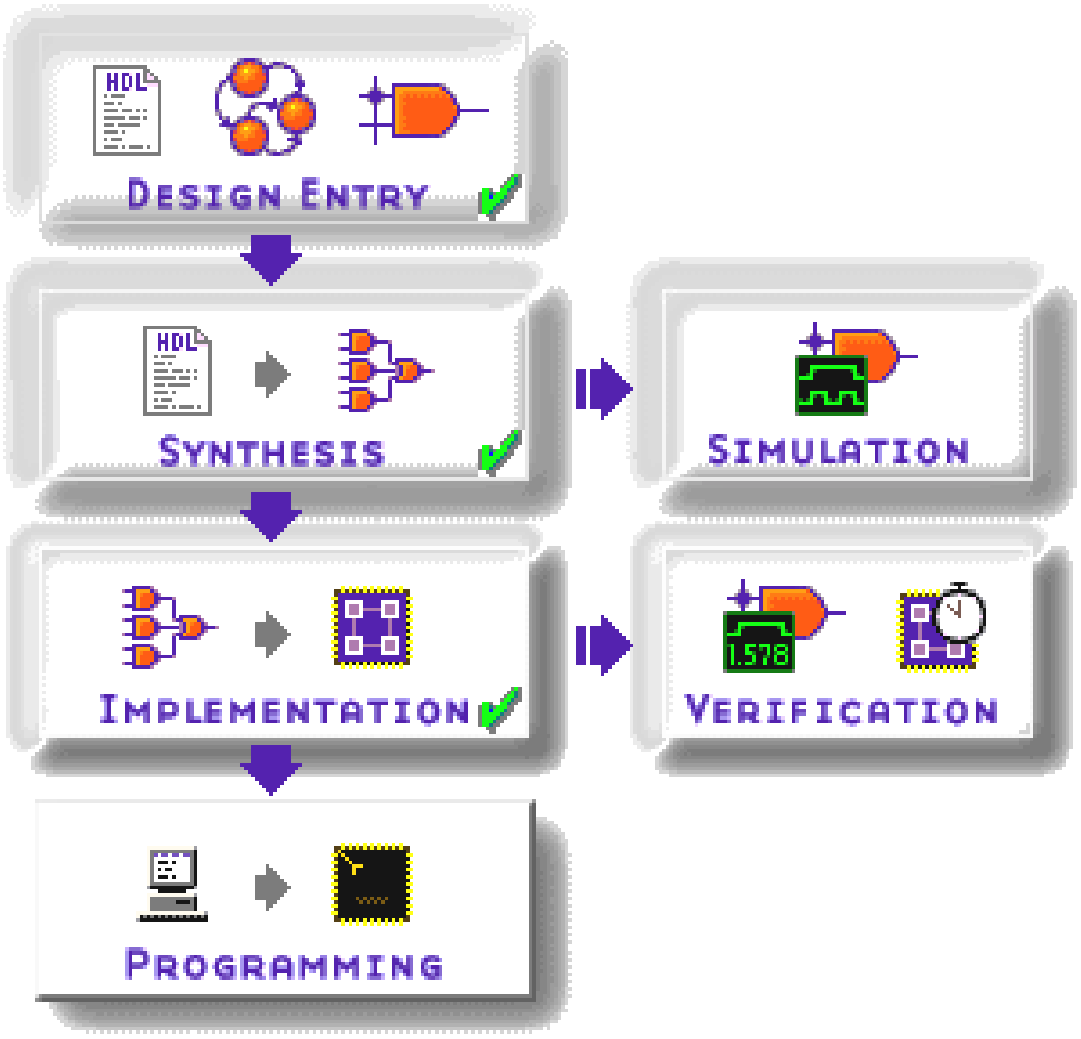
Pragmatic Logic Design

With XILINX Foundation 2.1i



David E. Vanden Bout

XESS Corp

Table of Contents

0

Introduction

There are numerous requests on Internet 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 cheap/free tools I can use to learn more?"

Xilinx has a student edition of their Foundation 2.1I software so anyone can get a low-cost set of tools for designing with programmable logic. I have written this book to give you a gentle introduction to using the Foundation 2.1I software. (Other programmable logic manufacturers have released low-cost or free toolsets for their devices. Someone else will have to write a book for them.)

This book tries to show you quickly how to use Foundation 2.1I to do the following:

- How to start a project and target it at a particular type of FPGA or CPLD.
- How to describe a logic circuit using VHDL, schematics, or both.
- How to simulate your circuit to test its functions.
- How to synthesize a netlist from your circuit.
- How to fit the netlist into an FPGA or CPLD.
- How to check the device utilization and timing.
- How to generate a bitstream for configuring the FPGA or CPLD.
- How to download the bitstream into the device.
- How to test the programmed device.

I will use combinatorial and sequential logic design examples to illustrate the design flow. Along the way you will see how to use many of the tools bundled into XILINX Foundation 2.1I. I will not go into a great deal of explanation on the theory of operation of the example circuits or the internal circuitry of the programmable logic device, but I will provide references to the appropriate texts and application notes on these subjects. In short, this is just a tutorial to get you started using the XILINX Foundation tools. After you go through this book you can move on to more advanced topics. The basic concepts introduced in each chapter are shown below:

Chapter 1 introduces the Foundation 2.1I software. In this chapter, you will use Foundation to describe a combinational circuit with VHDL, synthesize it,

simulate it, and then compile a configuration bitstream that you can download and test with an evaluation board.

Chapter 2 shows you how to design and implement a combinational circuit using the schematic editor included in the Foundation software.

Chapter 3 discusses how to do hierarchical and mixed-mode designs that include both schematics and VHDL.

Chapter 4 describes the I/O characteristics of FPGAs and CPLDs and shows you how to specify them in your designs.

Chapter 5 shows you how to design state machines using VHDL or the state machine editor included in the Foundation software.

Chapter 6 discusses how to perform timing analysis of your combinational and sequential logic designs.

Chapter 7 illustrates how to use the internal RAMs in a XILINX FPGA and how to interface an FPGA or CPLD to an external RAM chip.

Chapter 8 describes a simple microcontroller and shows how to implement it in an FPGA and a CPLD.

In addition to these chapters, there are two appendices in this text:

Appendix A1 steps you through the installation of the Foundation software, its service pack updates, and its FlexLM license.

Appendix A2 shows you how to configure and set-up your XS40 or XS95 Board so it can be used to run the design examples.

You should read both appendices before beginning on Chapter 1. That ensures you have the software and hardware installed correctly and reduces the chance of encountering a problem while doing the examples.

All the project files for each design example are provided on the CDROM. You should be able to re-create each design just from following the text, but you can use the project files in case you have problems or are in a hurry. Of equal importance, these working examples serve as a starting point for your own designs and explorations.

After you complete this book will you be an expert in using programmable logic and the Foundation 2.1I tools? The answer is a definite **NO**. This text is just a starting point. After you complete this text you will be able to:

- Understand the basic structure of the XILINX XC9500 CPLDs and XC4000 FPGAs and be able to further your understanding by reading their respective datasheets.
- Understand the basic functions of the Foundation 2.1I software tools.
- Use Foundation to create, synthesize, simulate, and compile logic circuits for CPLDs and FPGAs.

- Analyze the performance of your circuit with respect to a given XILINX CPLD or FPGA.
- Download and test your circuit on an FPGA or CPLD evaluation board.

1

VHDL-Based Design

In this chapter you will learn how to:

- Create a project with Foundation 2.1i.
- Enter VHDL code describing your logic circuit and check-and-correct any syntax errors.
- Pass the VHDL code to a synthesizer to generate a netlist for a particular FPGA or CPLD.
- Simulate the operations of the netlist to check the logical functions of your circuit.
- Transform the netlist into a configuration bitstream using the implementation tools.
- Download the bitstream into an FPGA or CPLD evaluation board.
- Test your circuit with the FPGA or CPLD evaluation board.

Overall Design Flow

Creating a logic circuit for a CPLD or FPGA usually consists of the following steps (depicted in Figure 1):

1. You enter a description of your logic circuit using a *hardware description language* (HDL) such as VHDL.
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 test the functions of your circuit by loading the netlist into a simulator, applying input patterns (known as *test vectors*), and observing the simulated outputs.

Look at the datasheet and XAPP097 for more details about the XC4000XL FPGAs. The XC9500 datasheet and XAPP073 have more information about the internal structure of CPLDs.

4. You use implementation tools to map the logic gates and interconnections into the FPGA. The FPGA consists of an array of *slices* which can be further decomposed into *configurable logic blocks* (CLBs) that perform logic operations using a set of *look-up tables* (LUTs). The CLBs are interwoven with various local and global routing resources. The fitter places gates from your netlist in various CLBs in the slices and opens or closes switches in the routing resources to wire the gates together. (A similar process occurs when you use a CPLD except the slices are called *configurable function blocks* (CFBs), the CLBs are called *macrocells*, and the routing resources are called *routing matrices*.)

5. Once the implementation is complete, the state of the routing switches and CLBs (or macrocells) is extracted to create a *bitstream* where the ones and zeroes correspond to open or closed switches.

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

That's really all there is to it. XILINX Foundation 2.1i provides the HDL editor, logic synthesizer, simulator, implementation tools, and bitstream generator software. The GXSLOAD utility from XESS will download the bitstream into an XS40 FPGA Board or an XS95 CPLD Board.

See page 372 of *Digital Design Principles and Practices* by John Wakerly for a detailed explanation of LED decoders.

In the rest of this chapter we will follow the design flow of Figure 1 while designing a simple LED decoder circuit. The decoder will take a four-bit input that represents a hexadecimal number (0000…1111 → 0x0…0xF) and will output seven signals that drive a seven-segment LED display so it shows the corresponding hexadecimal digit (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, b, C, d, E, F).

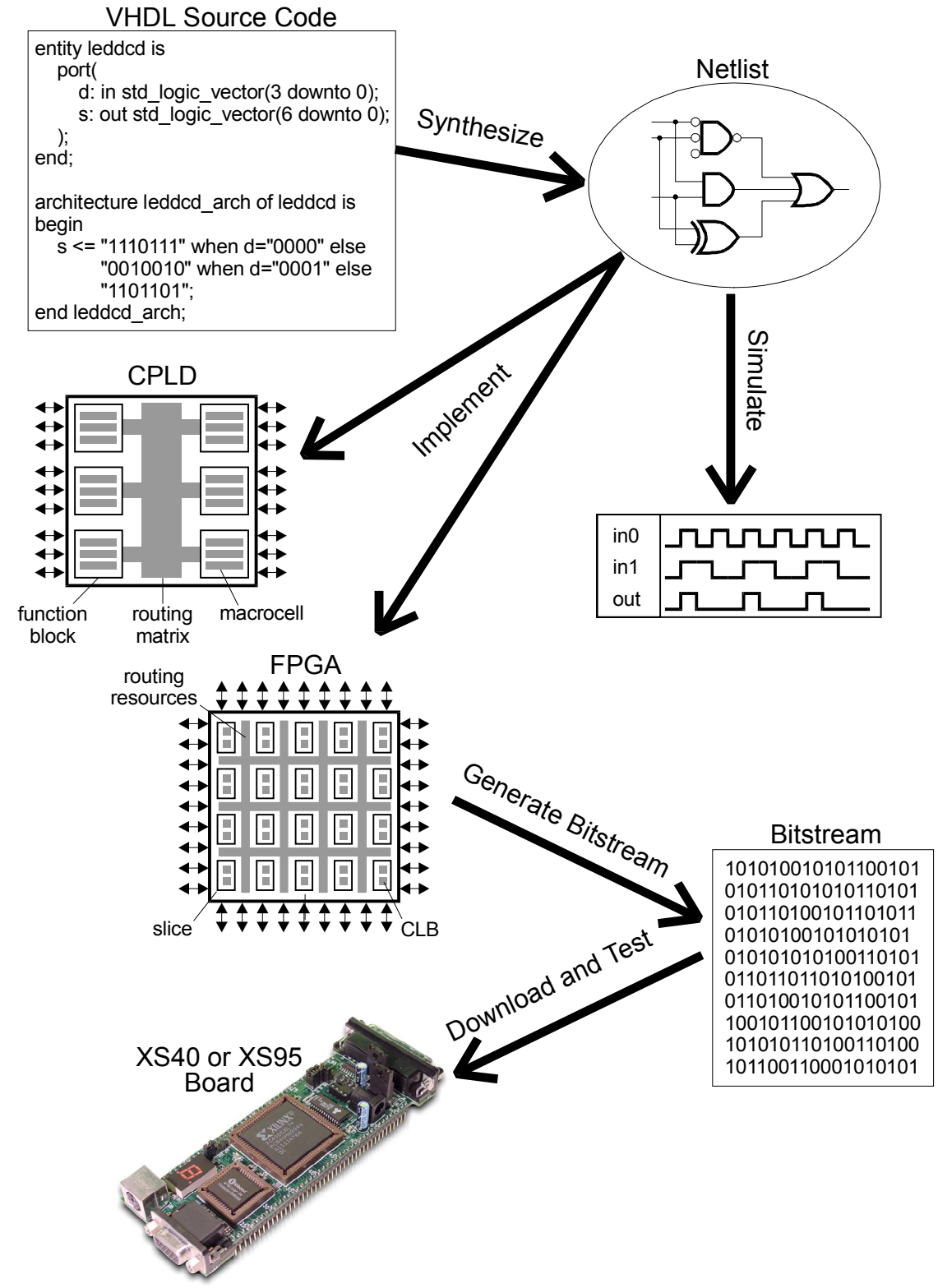
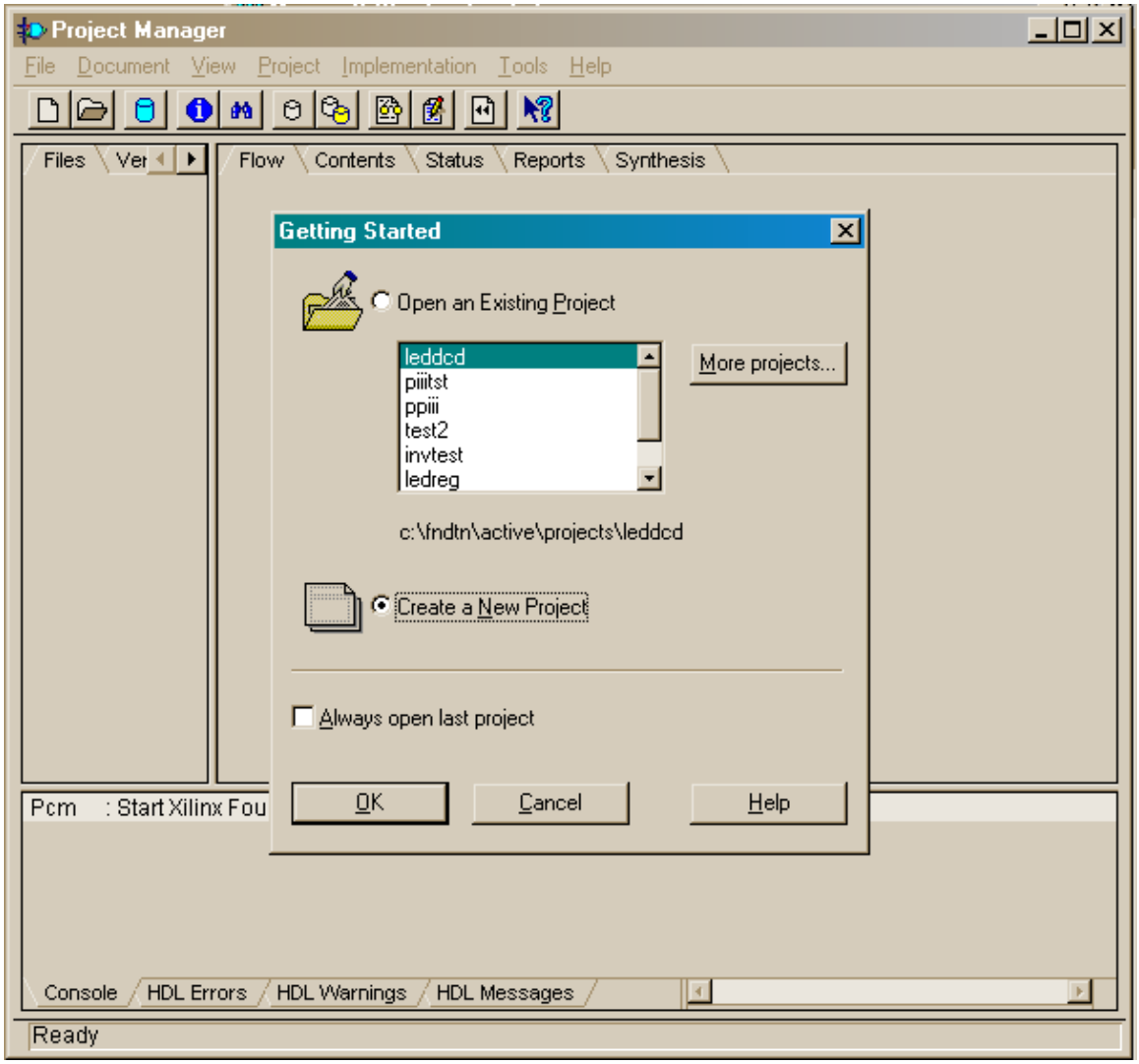


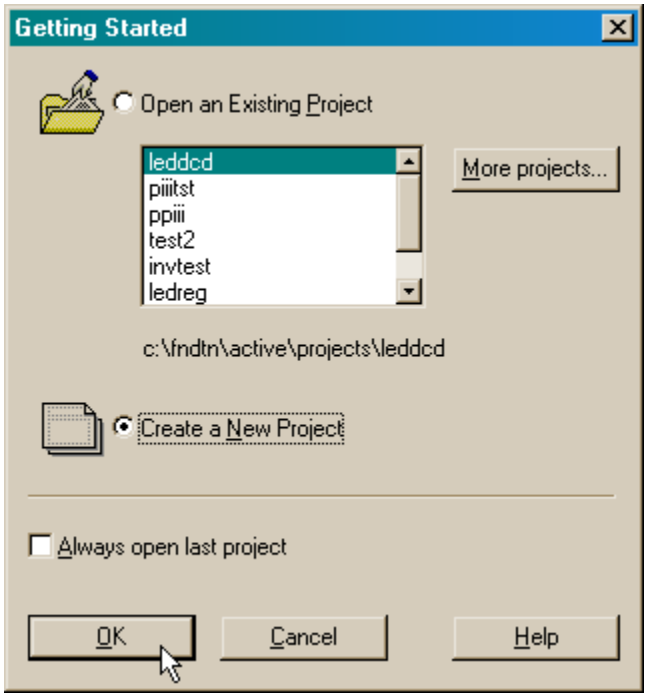
Figure 1: Steps in creating and testing an FPGA or CPLD-based design.

Starting a Project

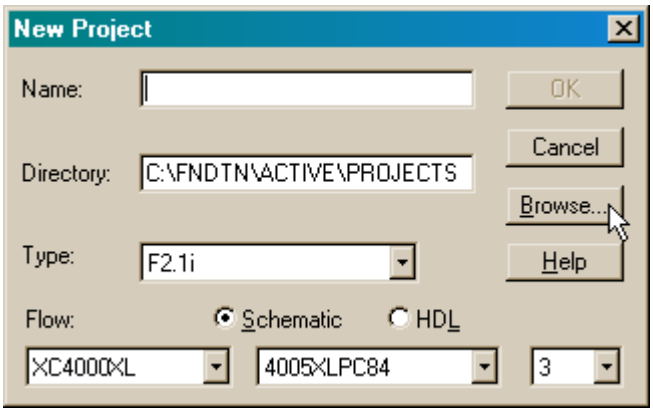
To begin the LED decoder project, double-click the start-up icon for Foundation 2.1i which will bring up the **Project Manager** window.



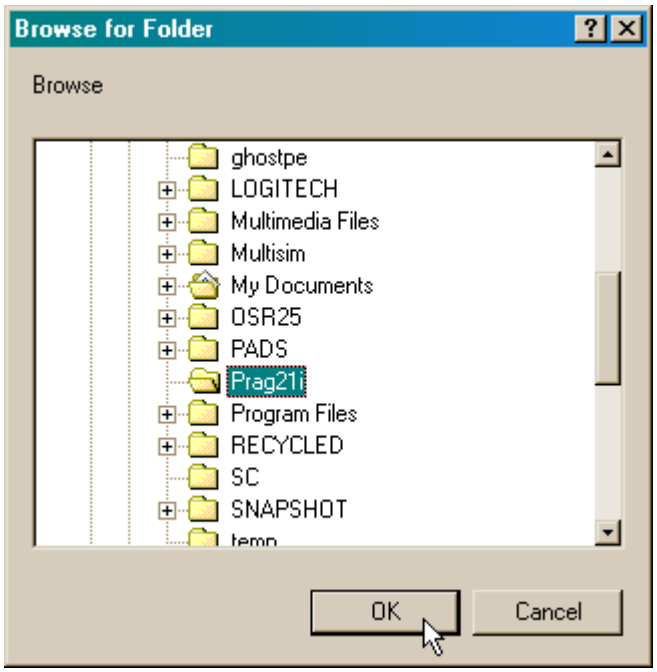
In the **Getting Started** dialog window, click on the Create a New Project radio button and then click on the OK button.



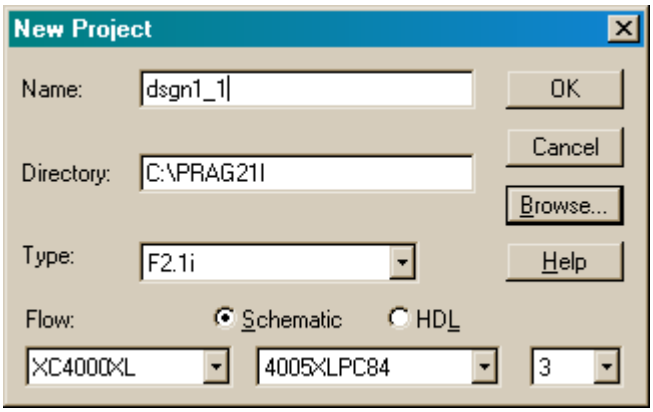
The **New Project** window appears where you setup the project. First, you must specify the directory where the project design files will be stored. The Directory field is initialized with the default directory within the Foundation directory tree. It is OK to place your projects there, but in this book I will place all the examples in a separate directory called Prag21i. To do the same, begin by clicking on the Browse... button.



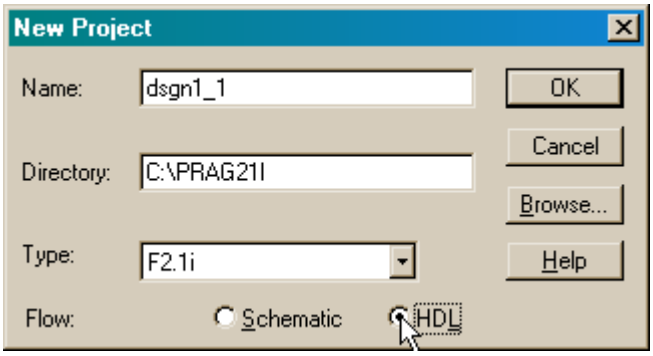
Scroll down in the list of directories in the **Browse for Folder** window and highlight the Prag21i folder. Then click on OK.



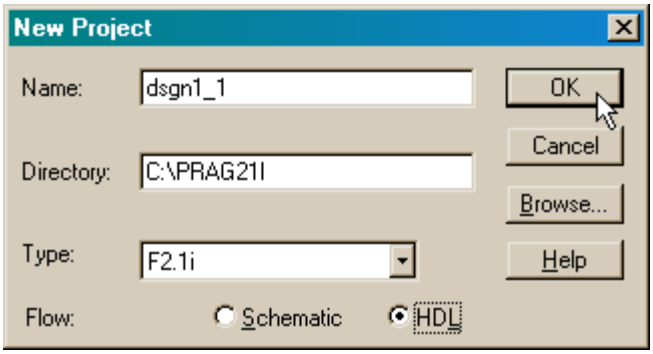
The path to the selected project folder will now appear in the Directory field. Next you must name your project. I placed the string dsgn1_1 in the Name field to indicate this is the first design of Chapter 1 of this book, but you can use any name you like.



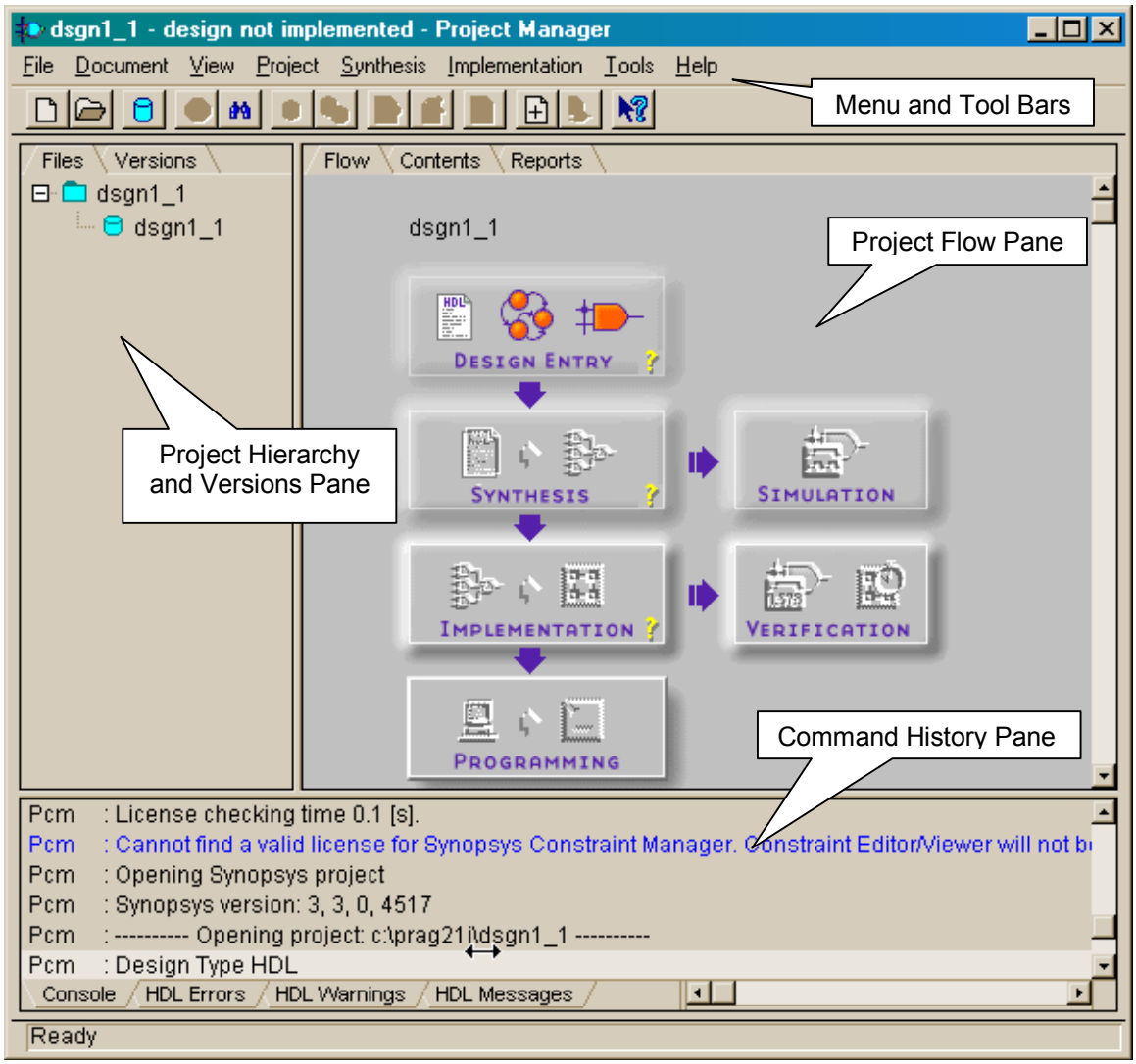
Next, choose to create an HDL-based design rather than a schematic-based design by clicking on the HDL radio button. Once you do this, the fields for selecting the programmable device family will disappear from the bottom of the window. That's OK. We will select the target device for the LED decoder circuit when we run the synthesizer later on.



This completes the initial setup for the LED decoder project, so click on the OK button.

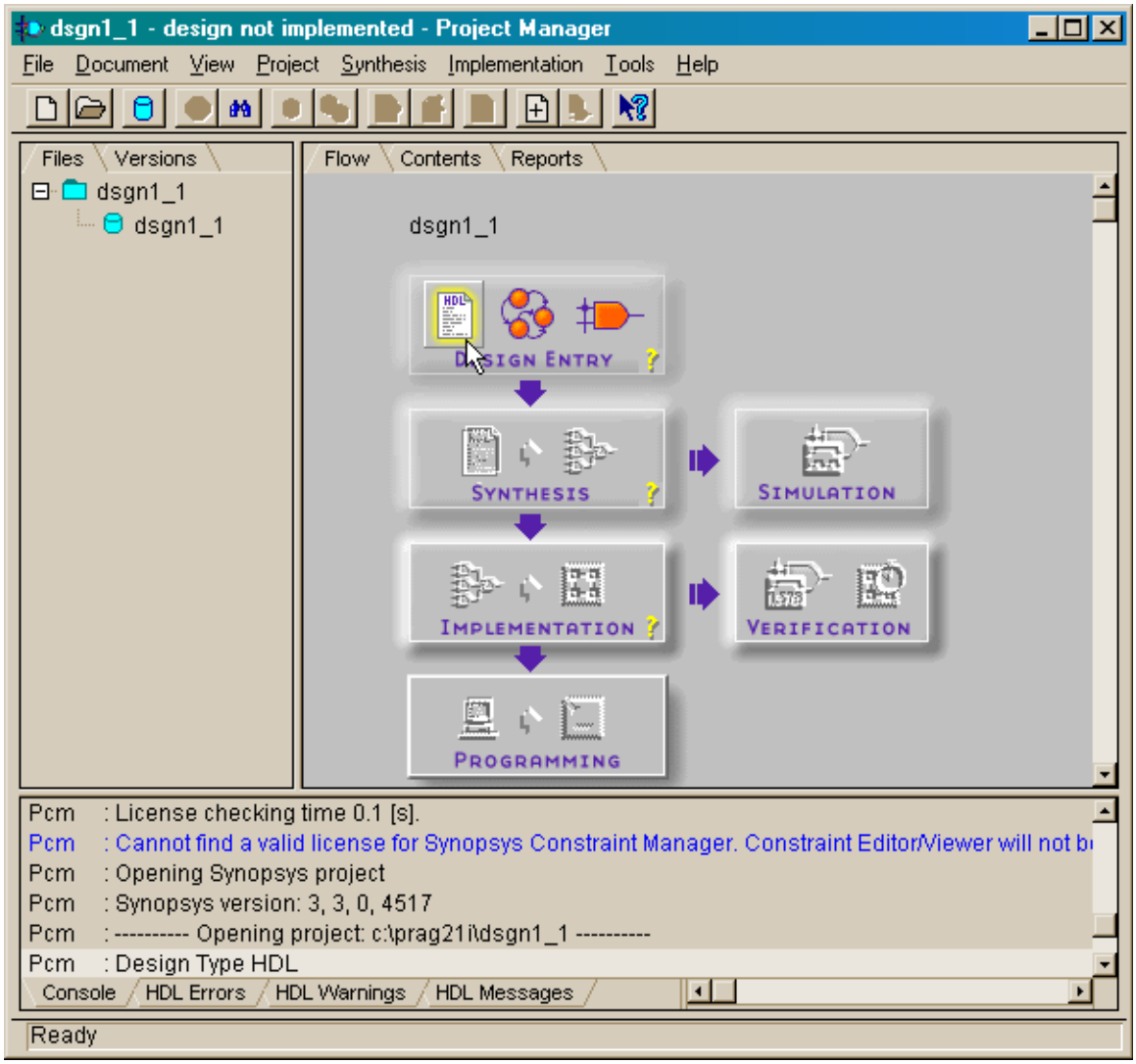


Now the **Project Manager** window appears as shown below. You can execute operations using the menu items and buttons in the menu and toolbars across the top of the window. The results of these commands are reported in the **Command History** pane at the bottom of the window. Your current point within the process of creating your design is indicated within the **Project Flow** pane. And the various files that make up each iteration of your design are listed in the **Project Hierarchy and Version** pane.

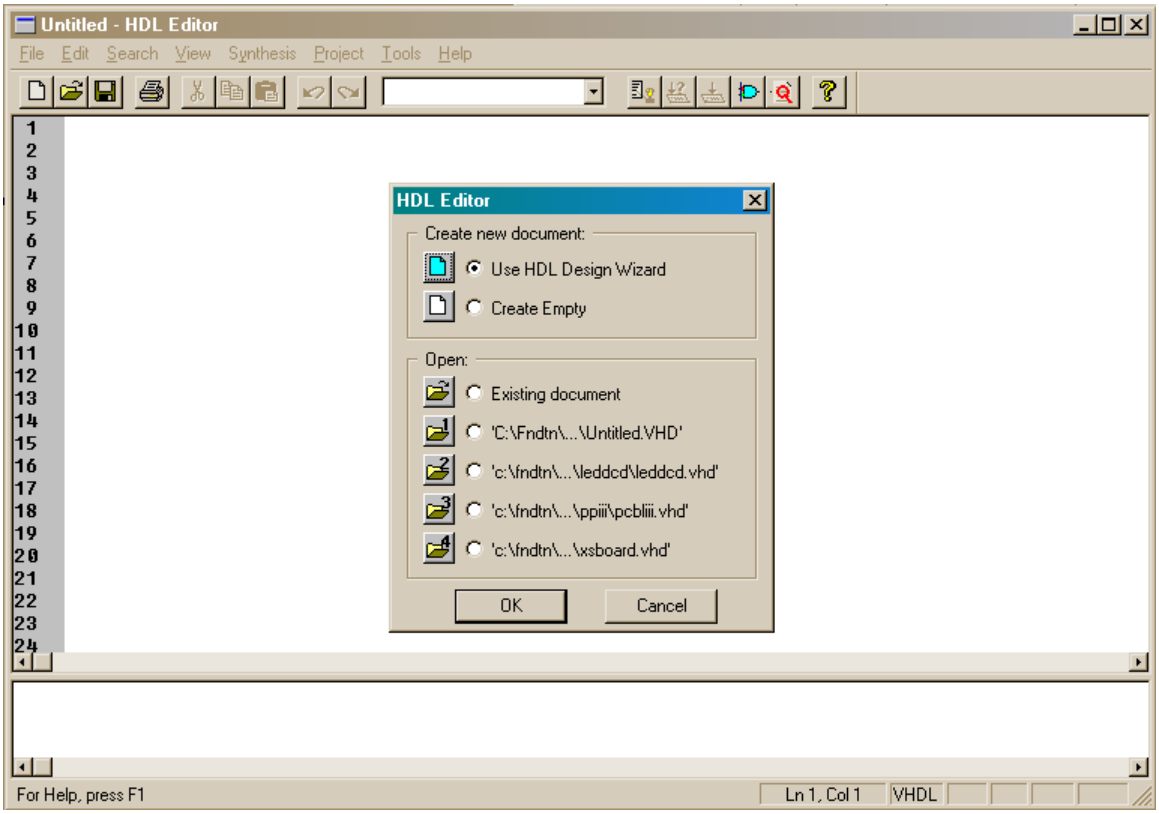


Creating the VHDL Source Code

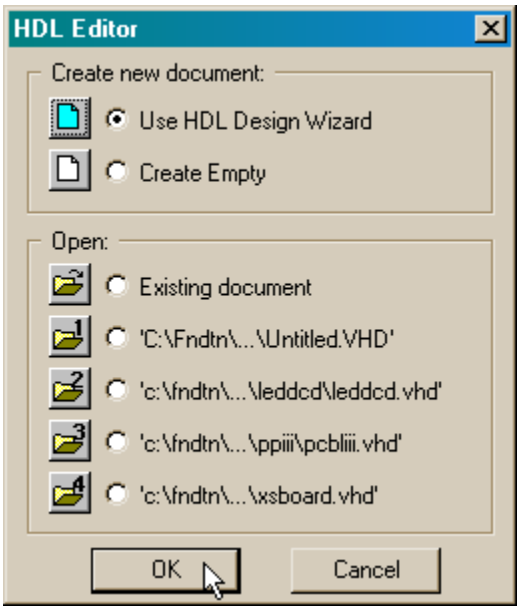
The first step is to enter your VHDL code for the LED decoder into a file. Start the HDL editor by clicking on the left-most icon of the Design Entry block in the **Project Flow** pane.



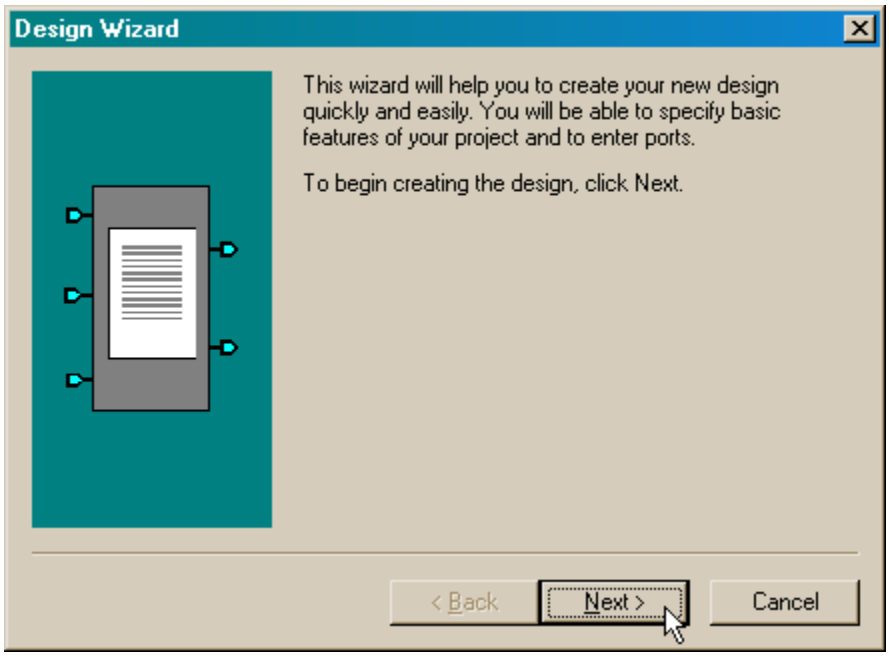
This brings up an HDL Editor window and a dialog box in which you can select how you want to enter your new code.



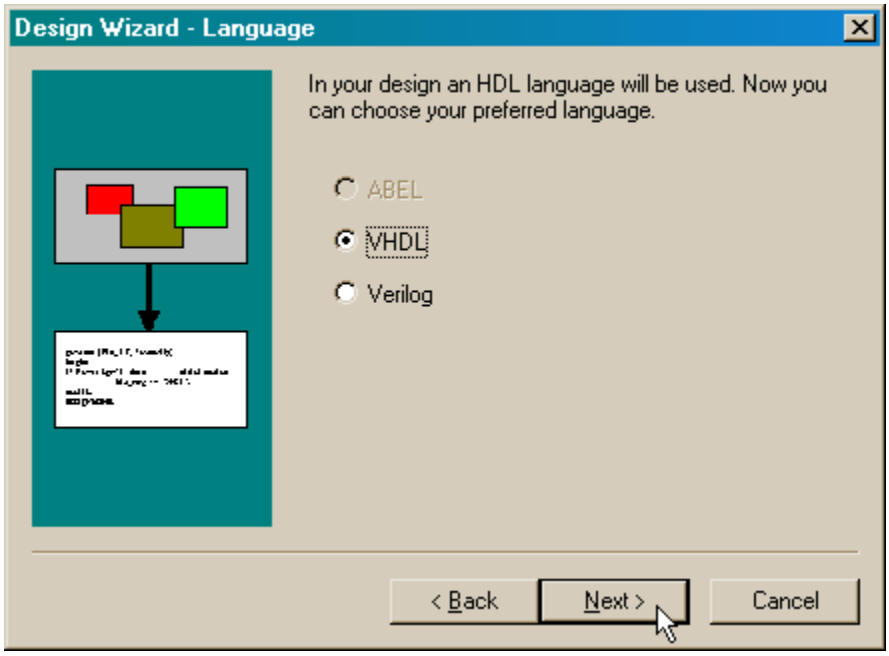
If you are an experienced HDL coder, you will probably select the Create Empty option to start with an empty editor window where you can type-in your source directly. In this example, we will select the Use HDL Design Wizard option which will lead us through the creation of a code skeleton that we can modify to describe the LED decoder circuit.



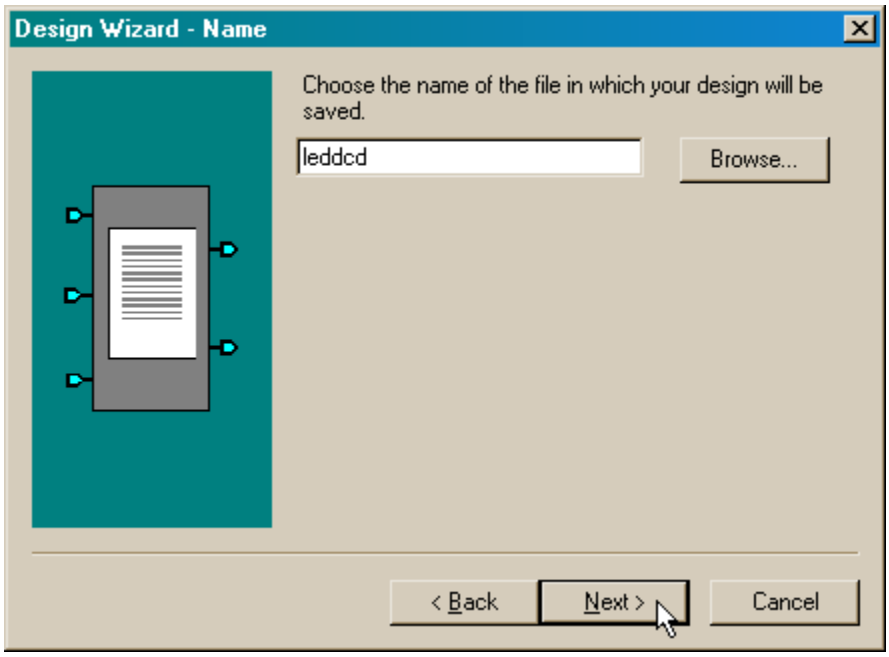
After clicking OK in the previous window, the **Design Wizard** window appears. Click Next> to move on.



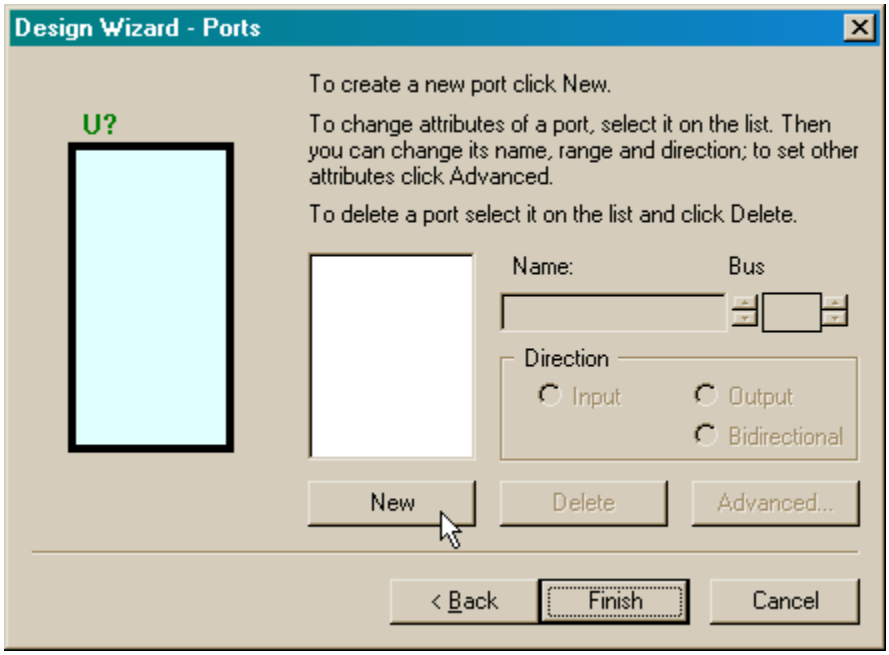
In the **Design Wizard – Language** window, select the particular HDL you plan to use (VHDL in our case). Then click on Next>.



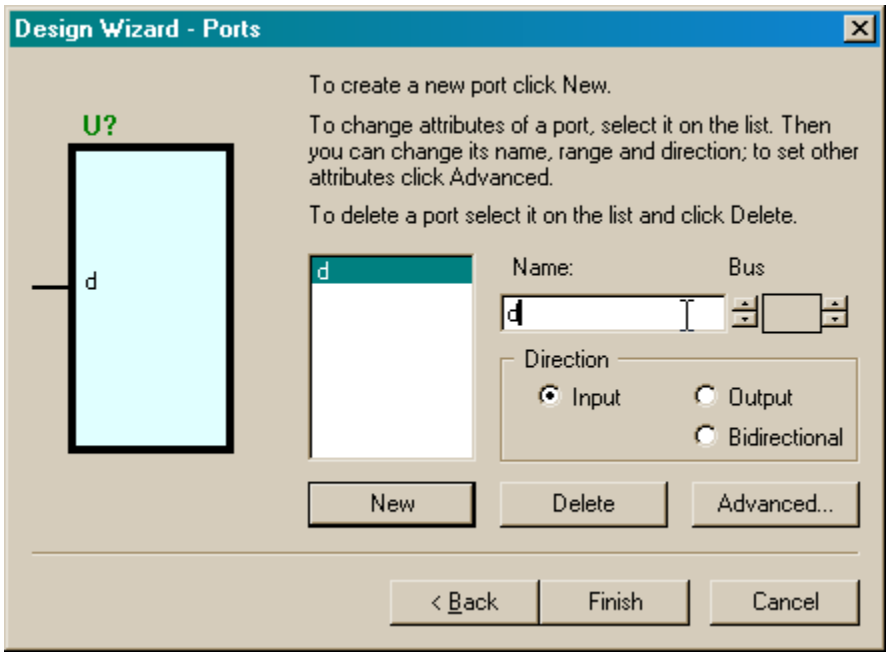
Type-in the name of the file in which you want to store the VHDL code. This does not have to be the same name as the entire project. I used leddcd for this example. After specifying the file name, click on Next>.



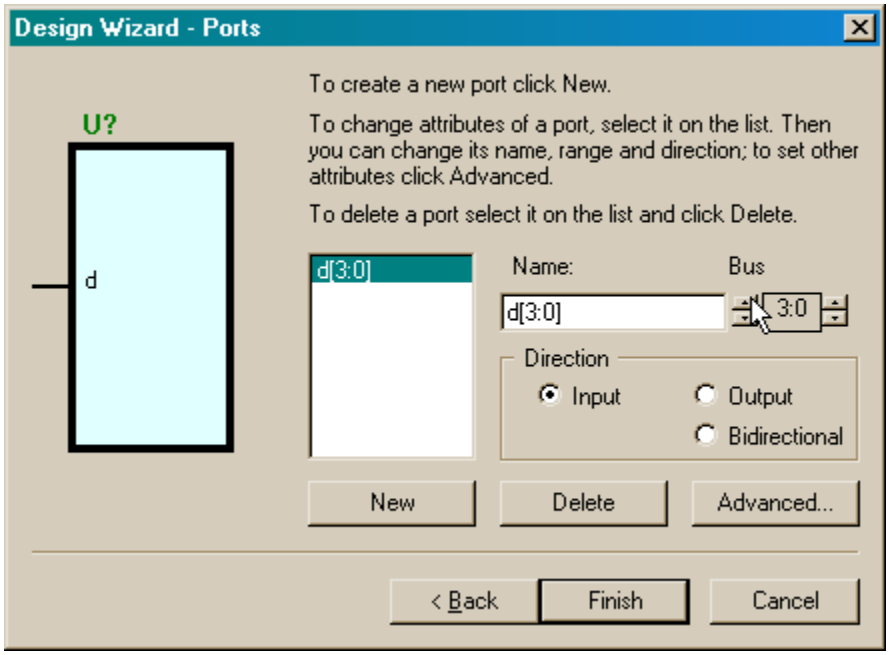
The window for specifying the inputs and outputs of the LED decoder circuit now appears. Begin by clicking on the New button.



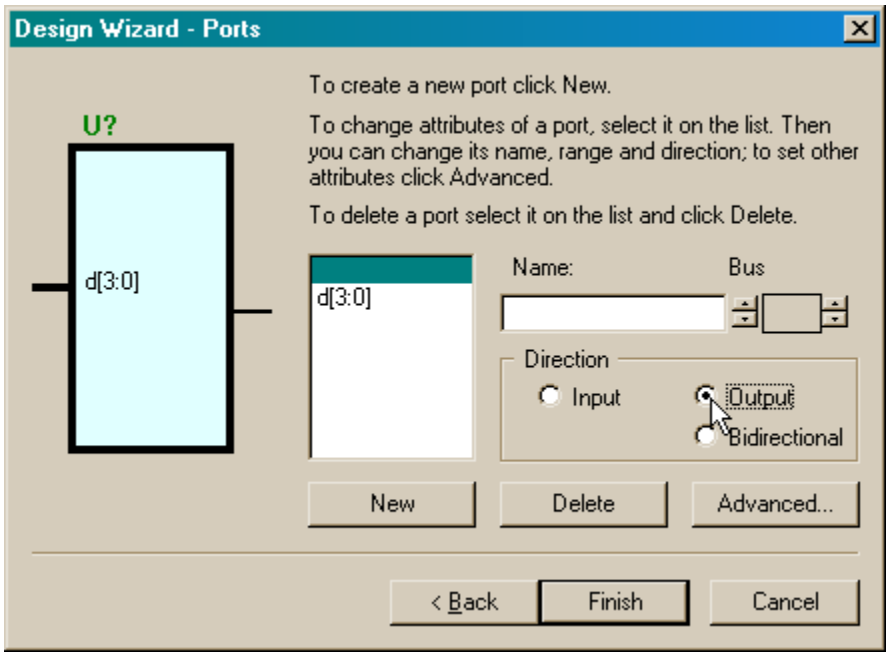
Next, type the name of the input port to the LED decoder. (I have named it **d** in this example.) The type of the port is selected by clicking one of the radio buttons in the Direction portion of the window. The Input button is selected by default so no action is needed in this case.



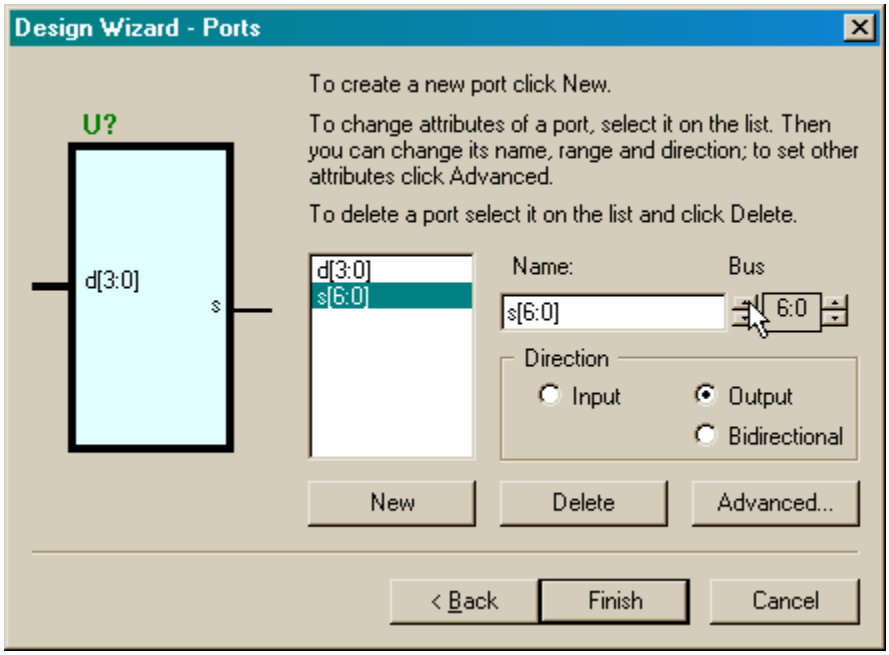
As mentioned previously, the LED decoder takes a four-bit input so we need to set the width of the **d** input bus. Click three times on the upper-left button of the Bus field to set the **d** input to four bits as shown below. The d input bus will appear on the left-hand side of the design block and in the list of I/O ports.



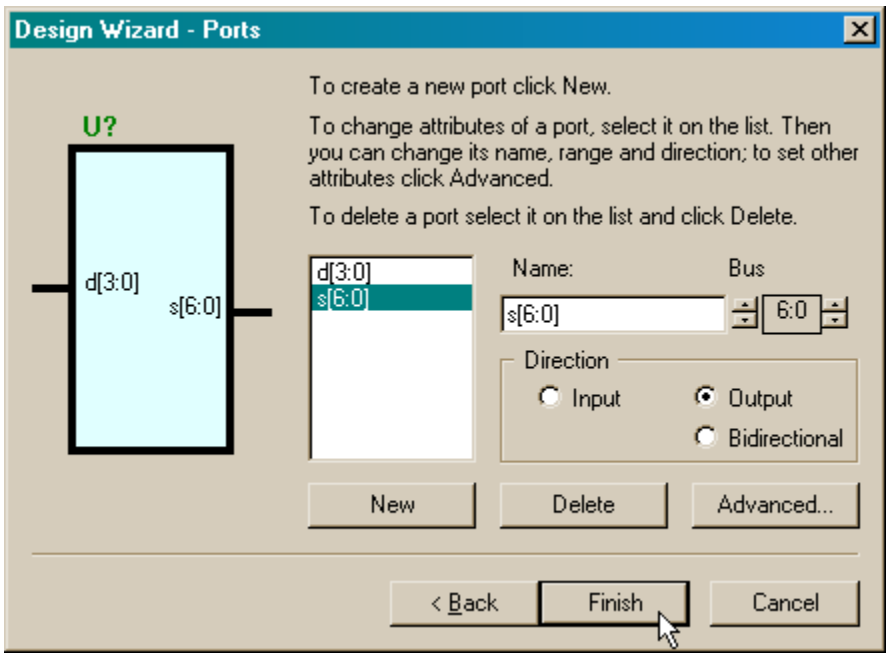
Now that the description of the input port is completed, click on the New button to start describing the output port. Then click on the Output button to set the direction of the port.



Type-in the name of the output port (**s** in this case) and then click on the upper-left Bus button six times. This sets the number of outputs to the seven required to drive a seven-segment LED.



Now that the input and output ports have been described, click on the Finish button.

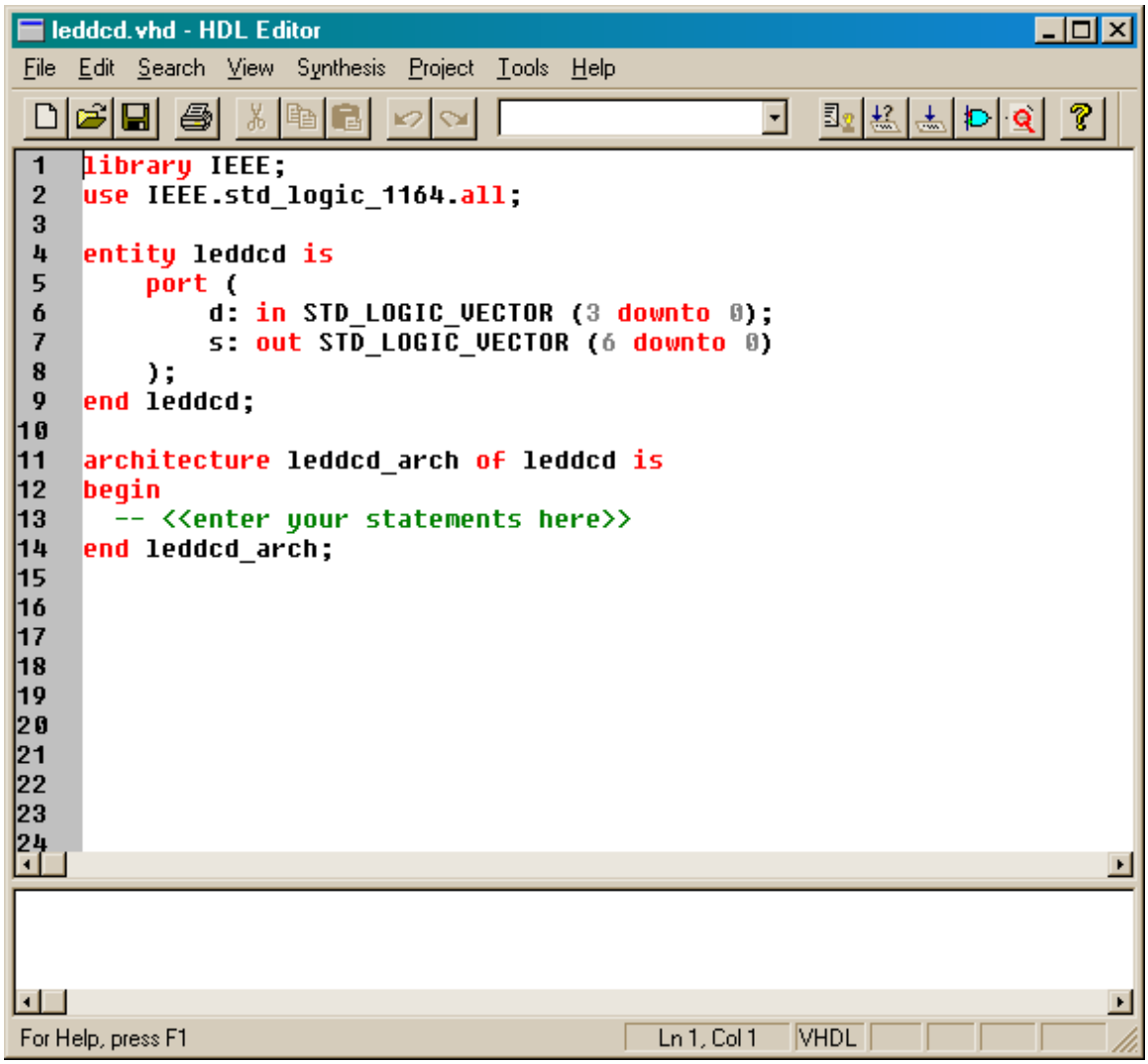


A VHDL code skeleton for the LED decoder circuit now appears in the **HDL Editor** window. The skeleton is composed of the following parts:

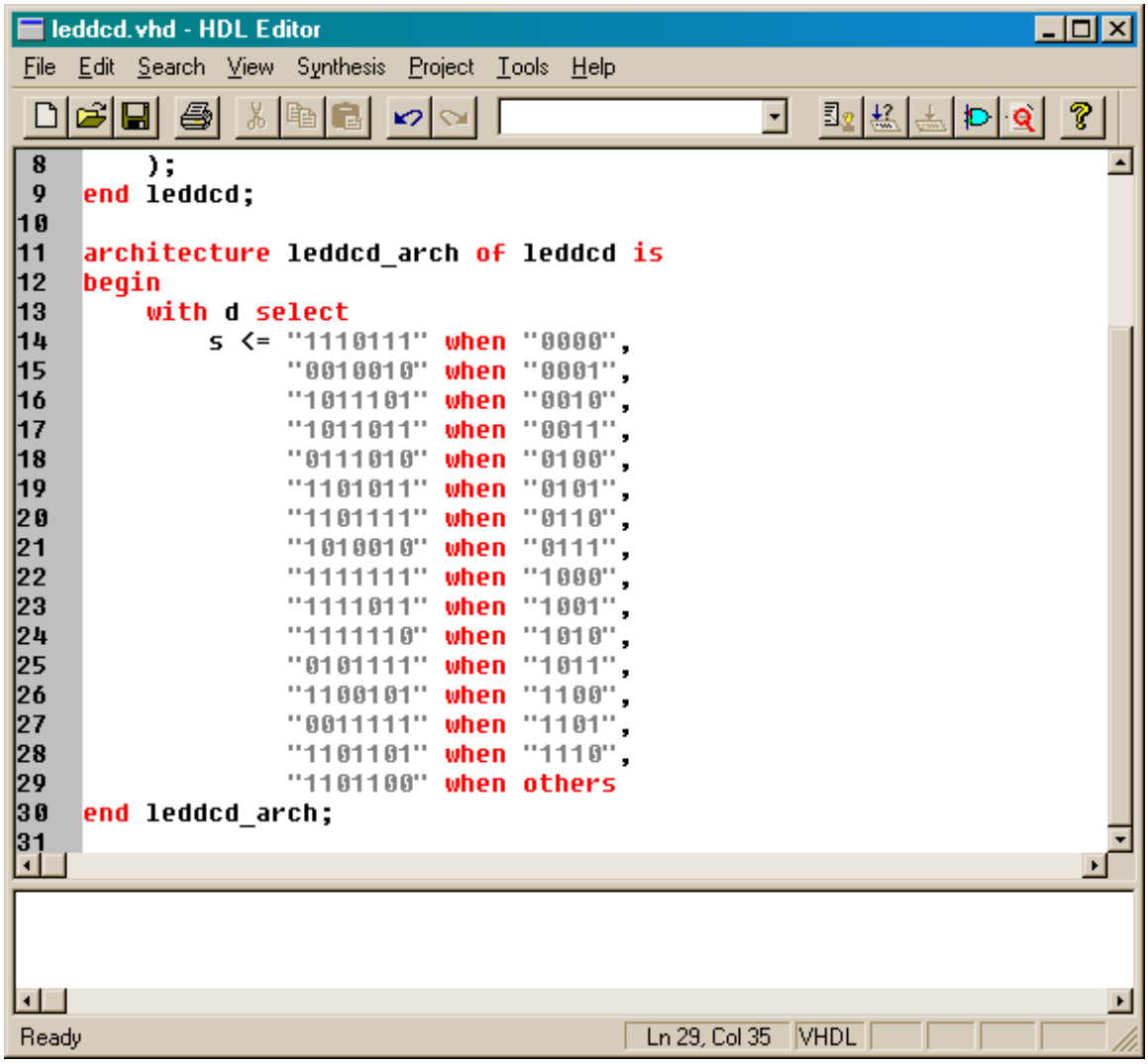
Lines 1–2: The standard IEEE library is linked in so our VHDL code can make use of the various functions and type declarations it contains.

Lines 4–9: The ports through which the circuit inputs and outputs signals are defined in the entity section.

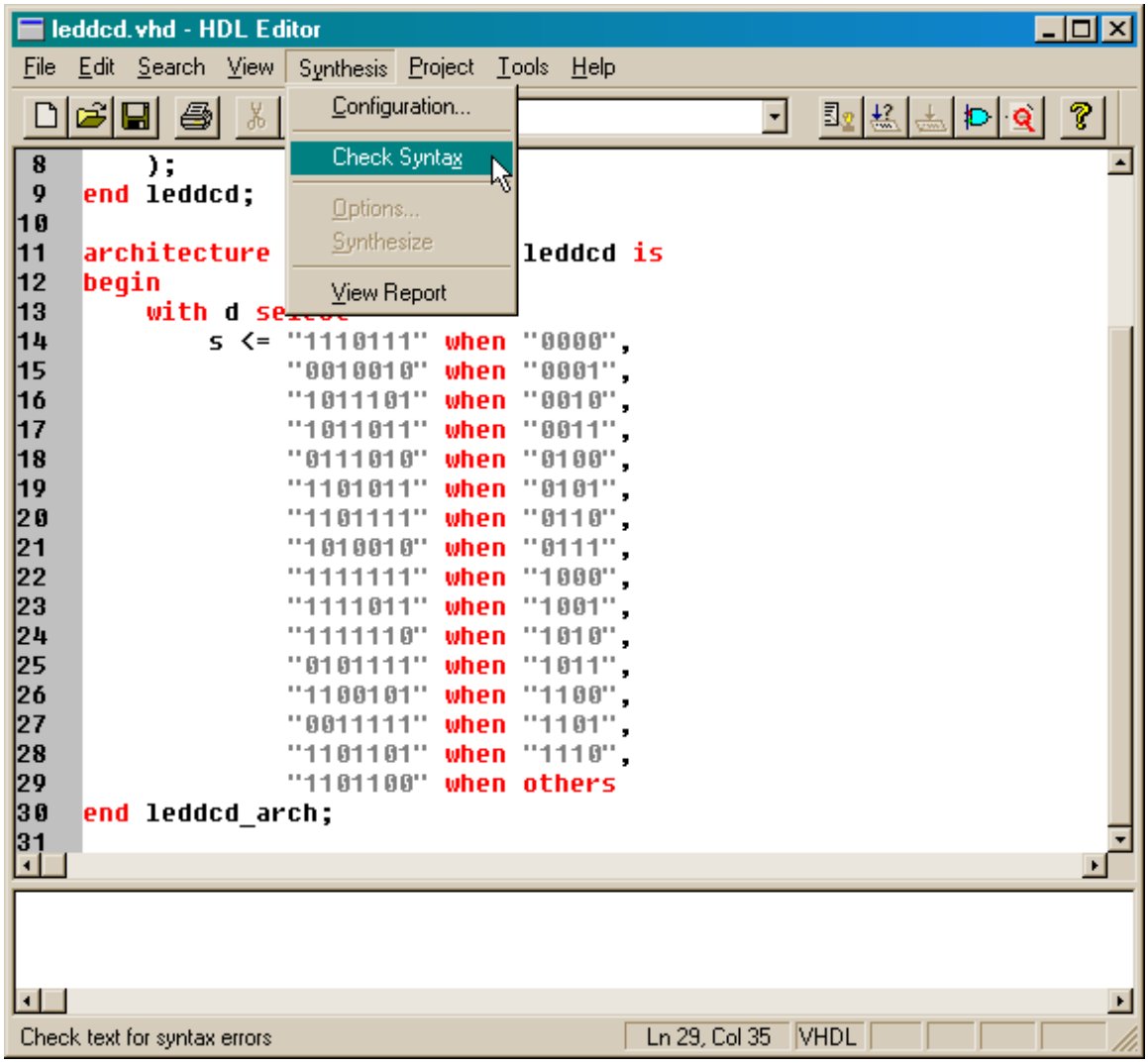
Lines 11–14: The statements that describe the operations of the LED decoder circuit are placed in the architecture section. These statements will replace line 13.


leddcd.vhd - HDL Editor
File Edit Search View Synthesis Project Tools 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 leddcd;
10
11 architecture leddcd_arch of leddcd is
12 begin
13 -- <<enter your statements here>>
14 end leddcd_arch;
15
16
17
18
19
20
21
22
23
24
For Help, press F1
Ln 1, Col 1
VHDL

The skeleton code on line 13 is replaced with the VHDL code on lines 13–29 shown in the window below. This code specifies the particular seven-bit value that will appear on the **s** outputs for each possible four-bit value that is driven into the **d** inputs. A high level on an output will illuminate the LED segment to which it is attached. (See Figure 2 for the correspondence between the **s** outputs and the individual segments that make up the LED digit.)

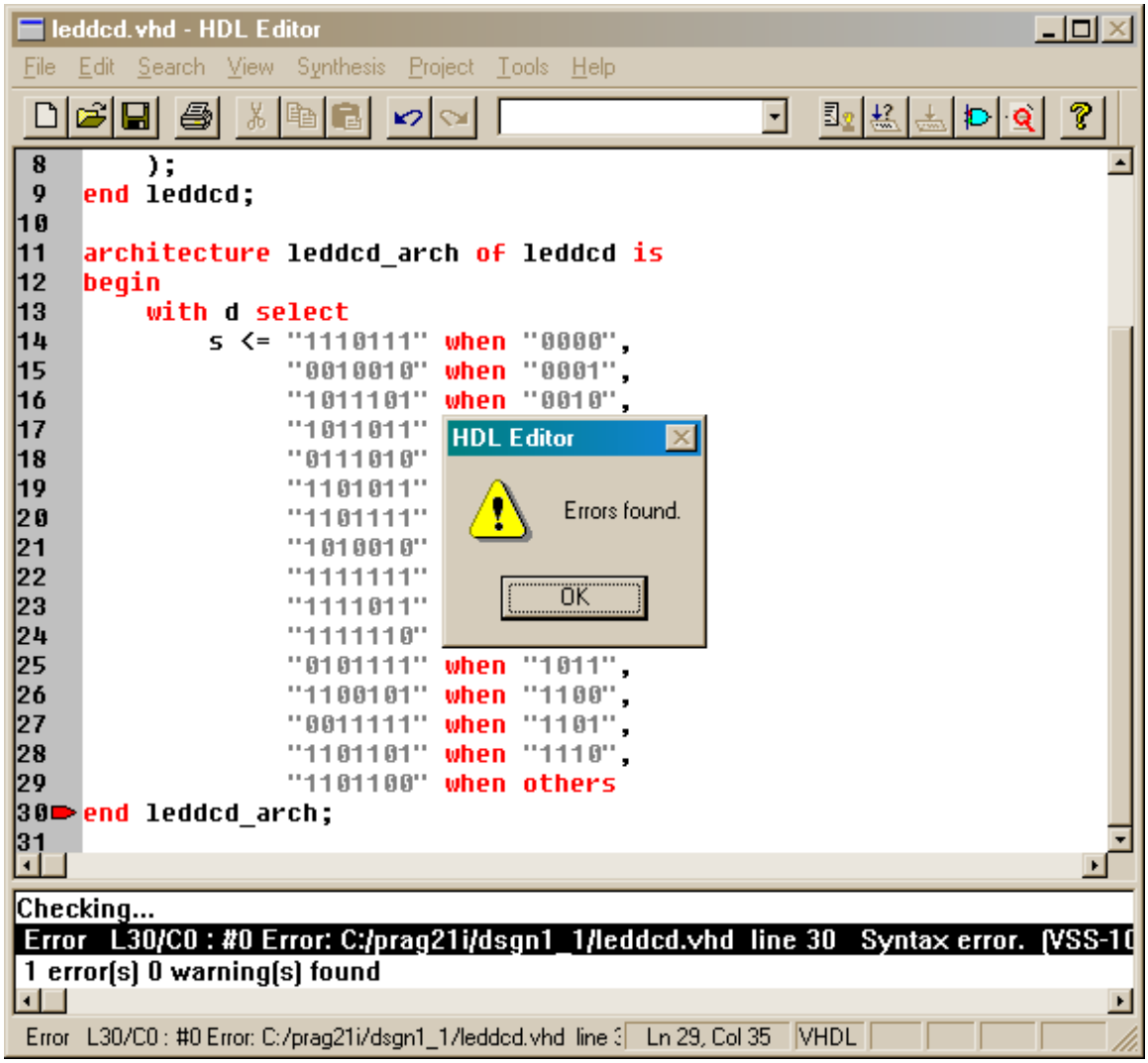


After entering the VHDL code, it's a good idea to see if there are any obvious errors. Do this by selecting the Synthesis→Check Syntax menu item.

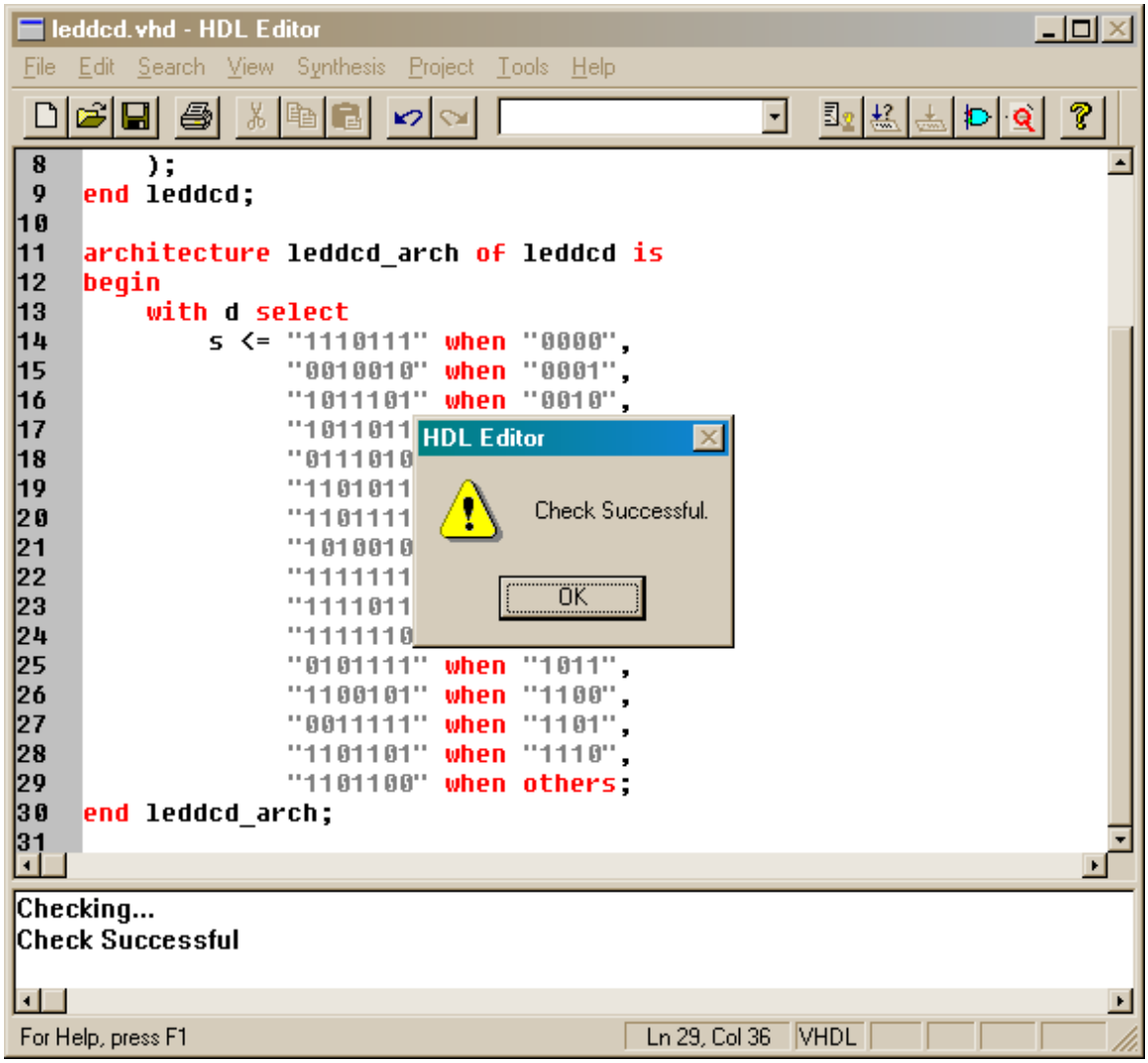


The syntax checker will parse the VHDL code and will return an error message in the bottom pane of the **HDL Editor** window informing you that there is an error on line 30. Click on the OK button to remove the pop-up error window.

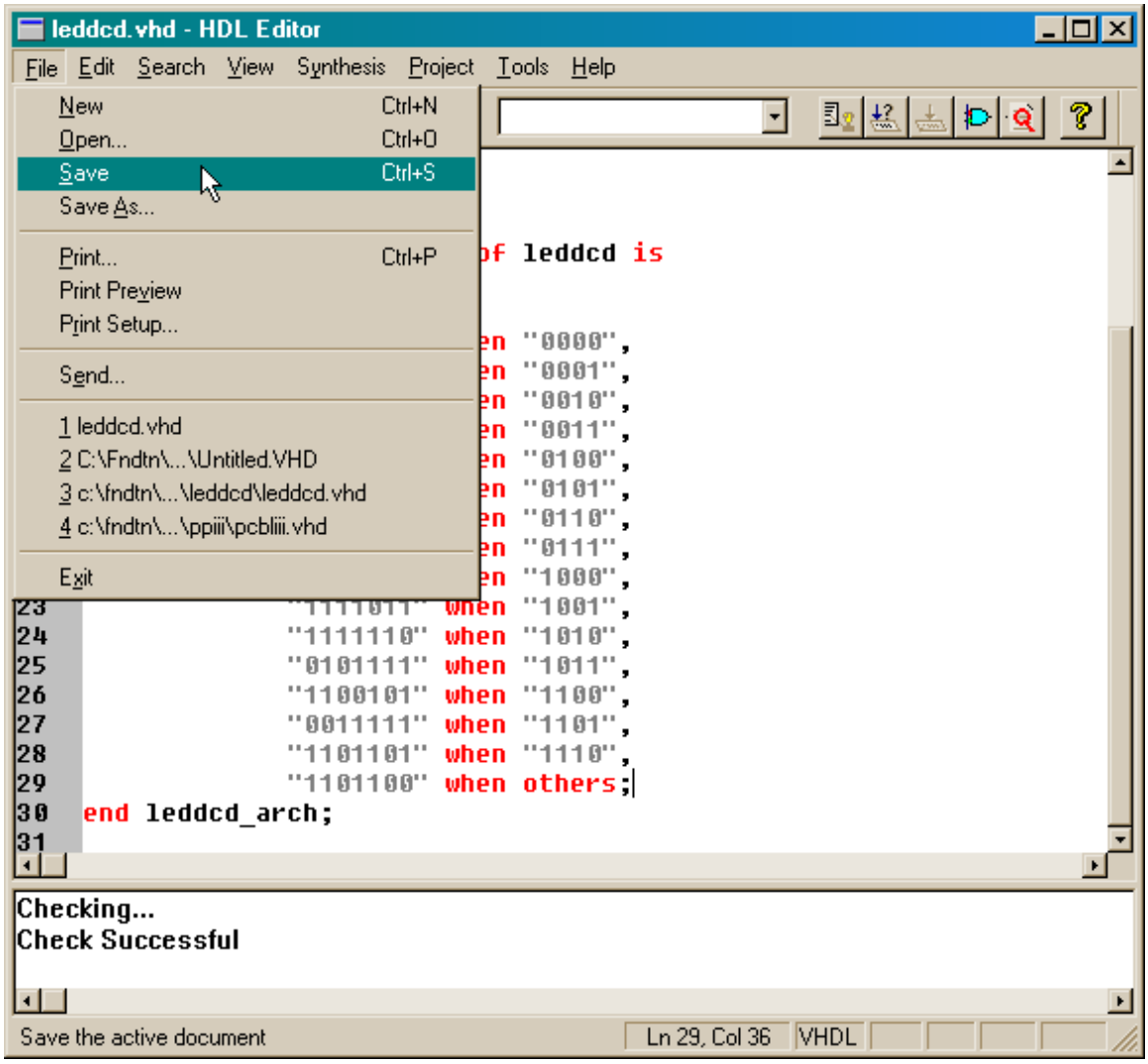
Line 30 was generated by the HDL Design Wizard, so it is unlikely that it has a syntax error. The actual error is at the end of line 29 where I have left off the terminating semicolon.



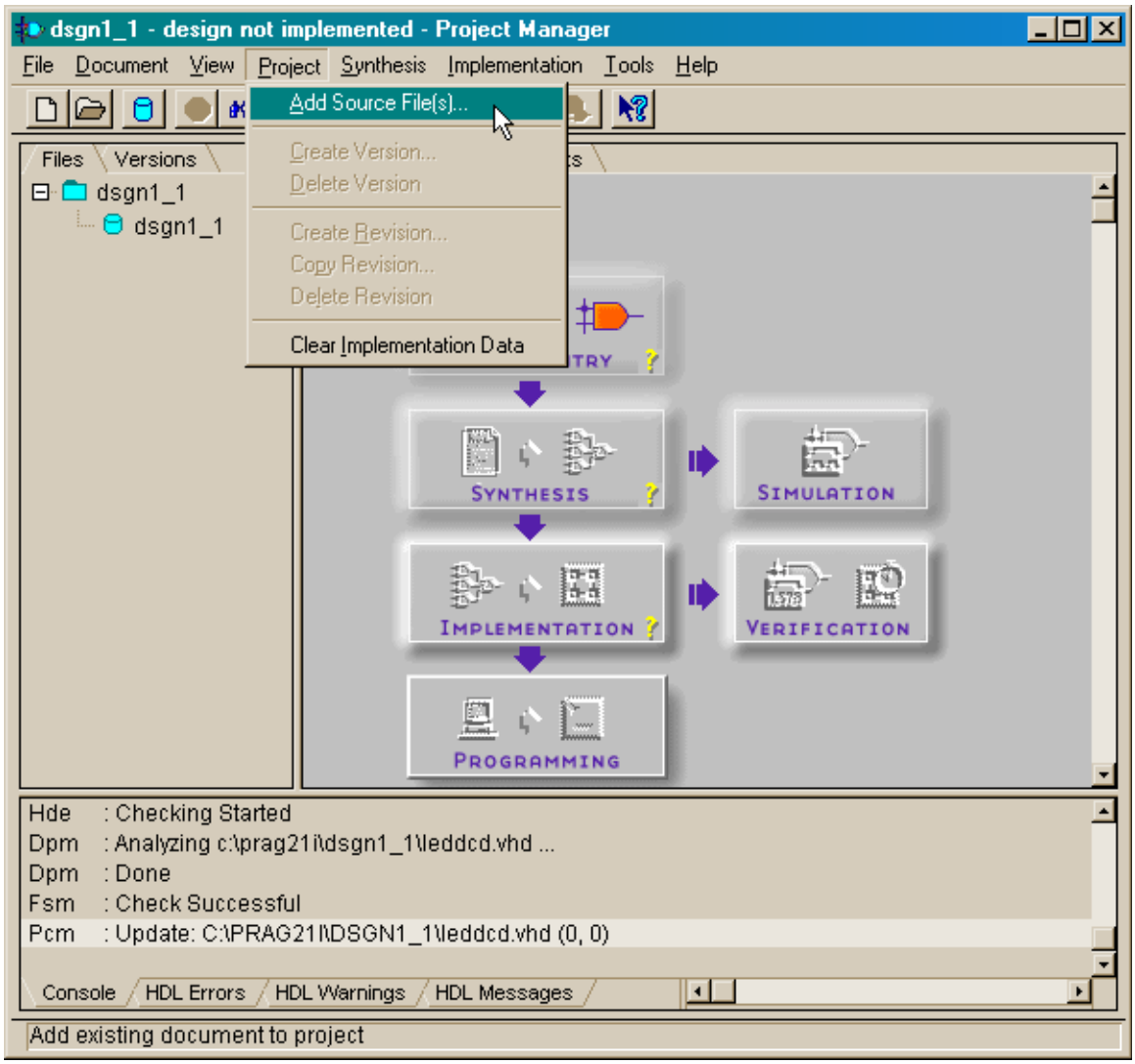
Once the semicolon is appended to line 29, the syntax checker is run again and the VHDL is judged to be free of syntax errors. Click on the OK button to remove the pop-up window.



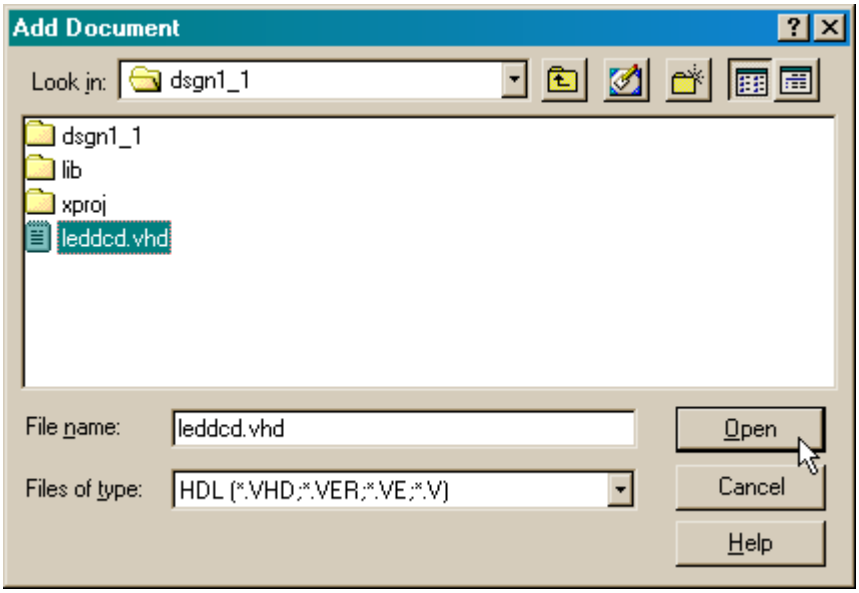
Now that the code has passed the syntax checker, save it in the leddcd.vhd file using the File→Save menu item. You can then exit from the editor.



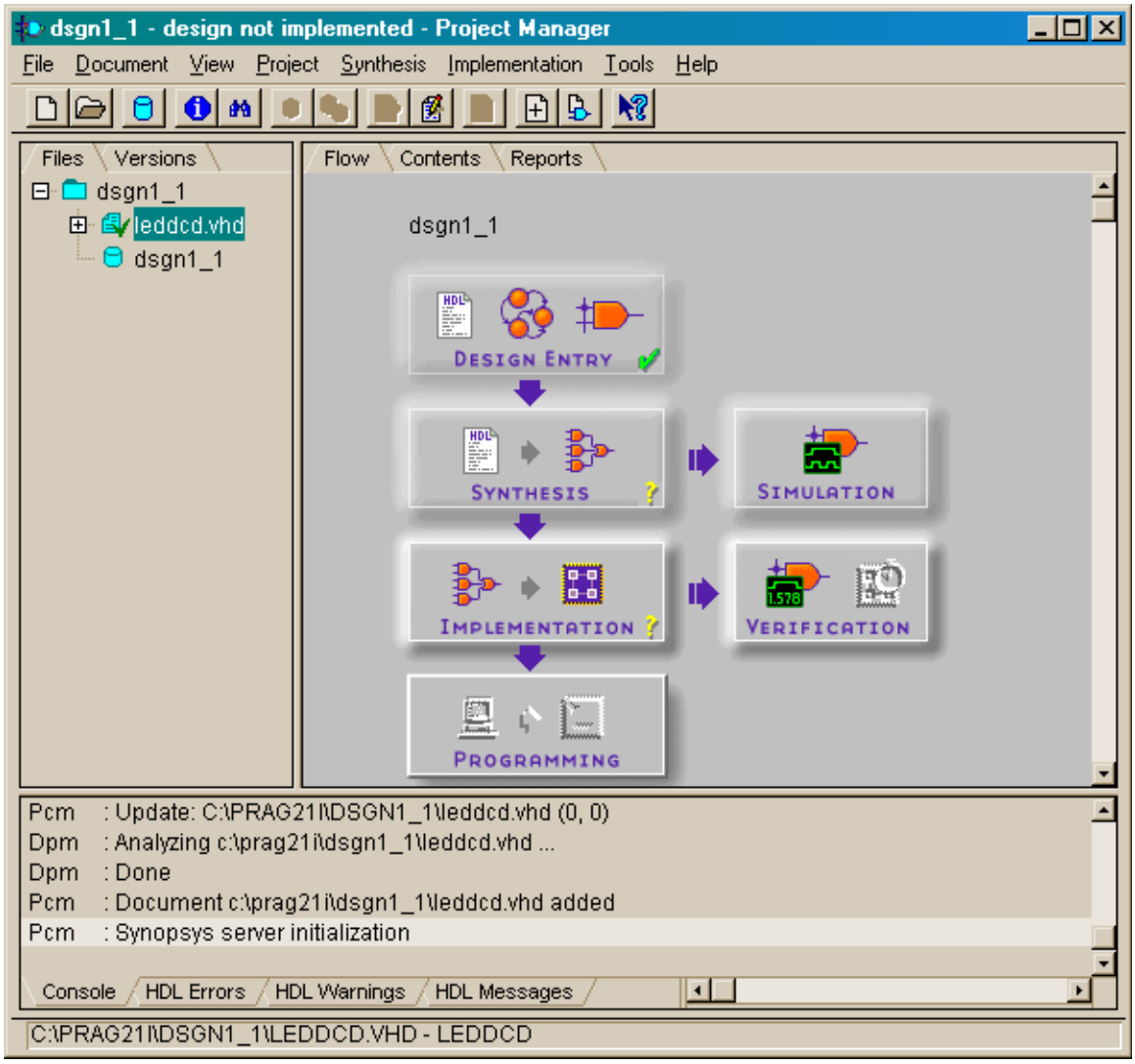
We have a VHDL file that describes the LED decoder, but it isn't a part of our poject yet. We must add the file to our project using the Project→Add Source files... menu item.



The leddcd.vhd file will be seen in the **Add Document** window that appears. Highlight this file and then click on Open to add the file to the project.

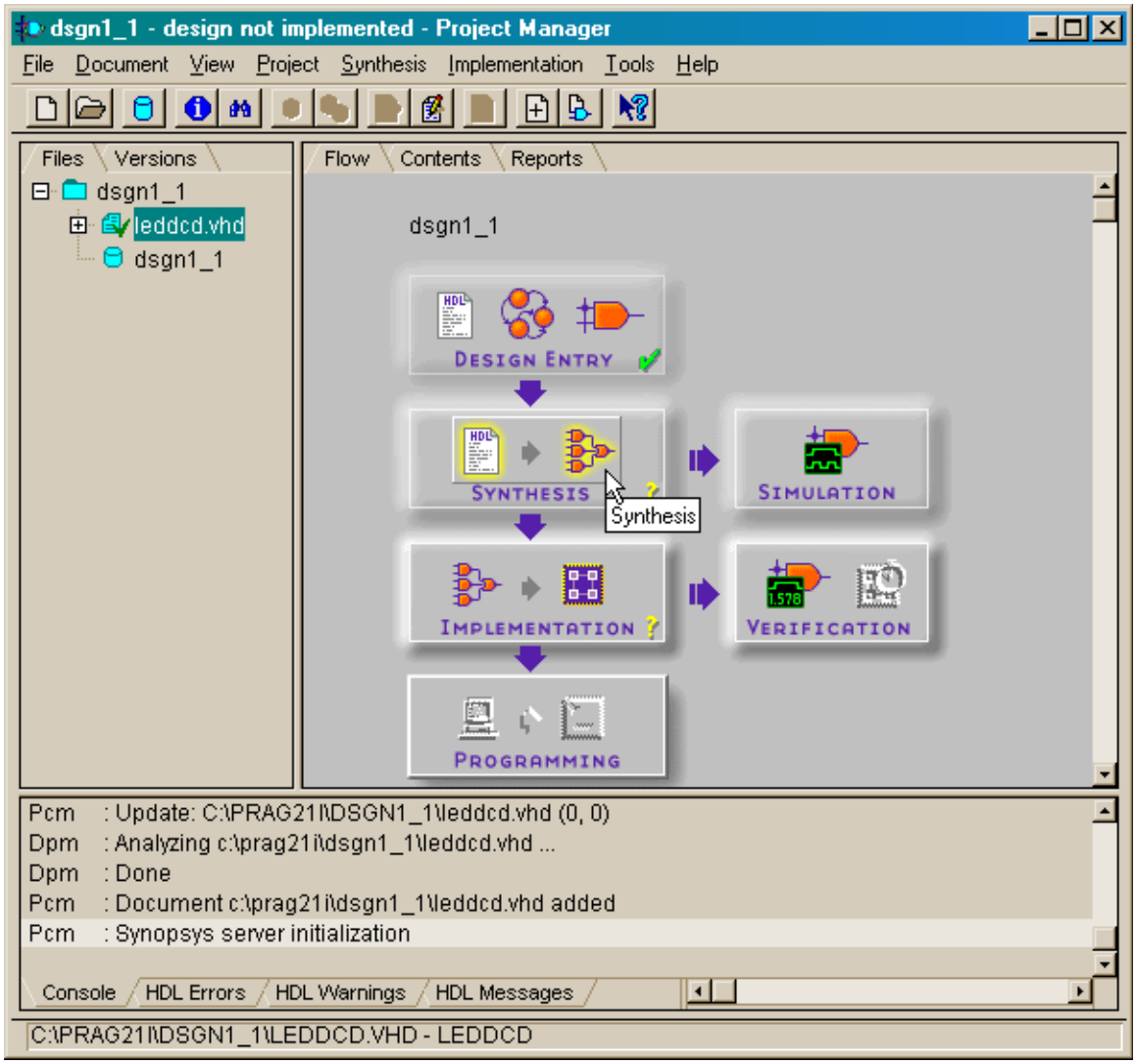


The left-hand pane of the **Project Manager** window now shows the leddcd.vhd file that we added. The green checkmark by the file name indicates that the VHDL in the file is syntactically correct. The green checkmark in the Design Entry box in the right-hand pane denotes that this phase of the design flow is completed.

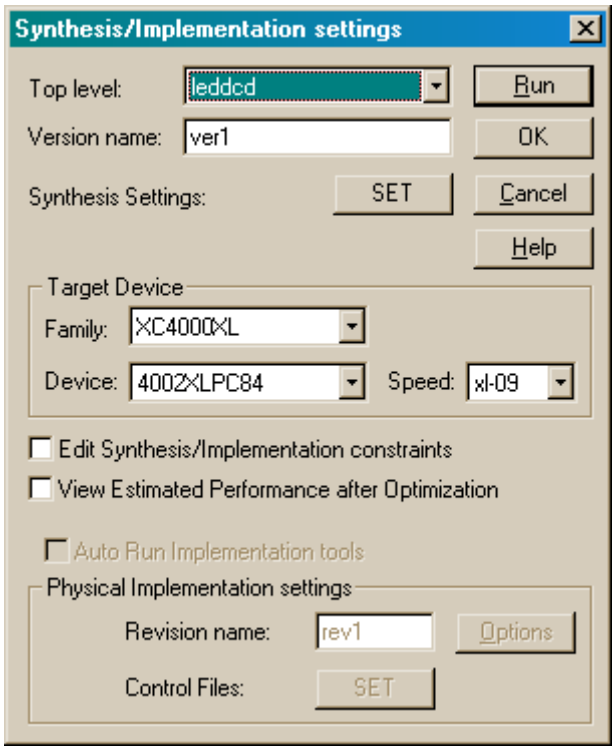


Synthesizing a Netlist

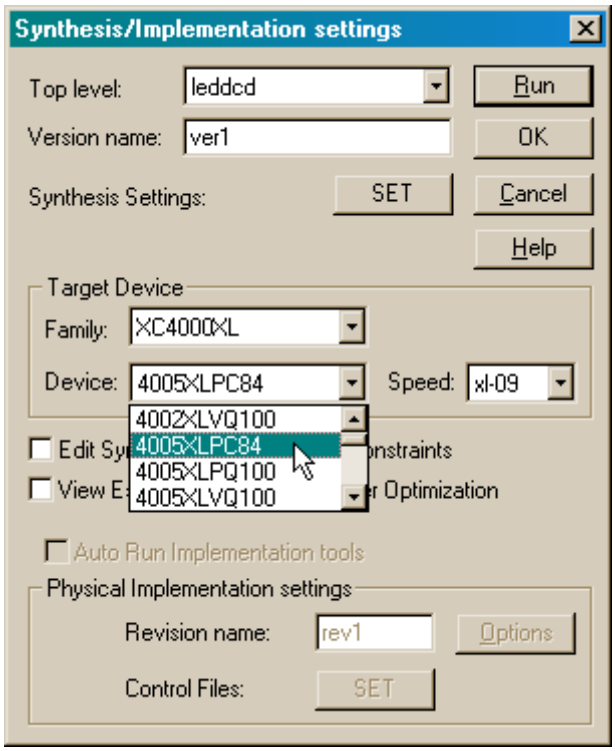
Now that we have the VHDL description of the LED decoder completed, we can run the synthesizer on it to produce a netlist for the circuit. You can initiate this phase by clicking on the Synthesis box in the **Design Flow** pane of the **Project Manager** window.



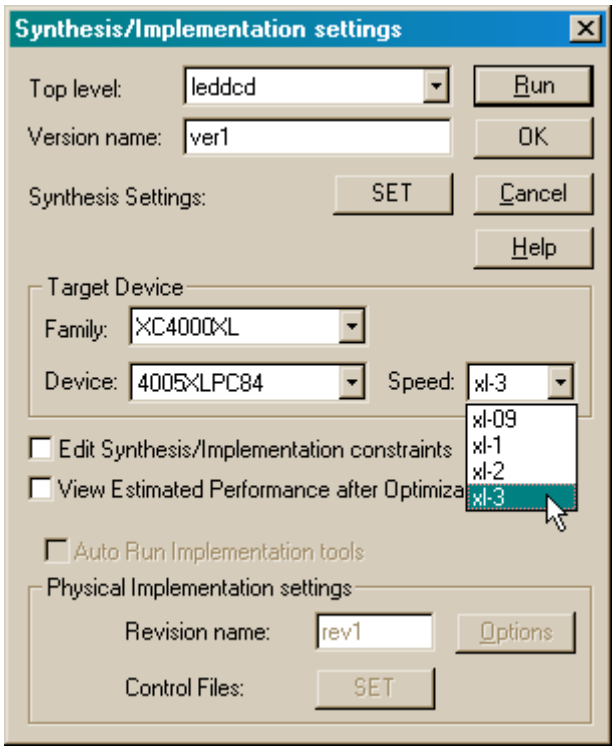
We set the parameters that control the synthesis process in the **Synthesis/Implementation settings** window that appears. Of most importance is to select the target device for the LED decoder circuit.



We plan to test this design on an XS40-005XL Board that contains a XILINX XC4005XL FPGA chip in an 84-pin PLCC package. The XC4000XL FPGA family is already set in the Family field of the Target Device area of the window. So we only need to select the particular device in the family (XC4005XLPC84 in this case) from the drop-down list of the Device field.

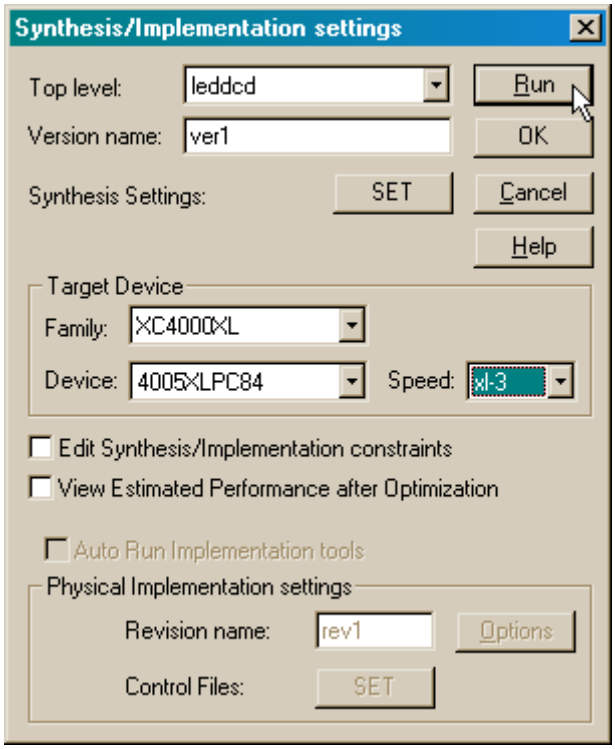


We can also specify the speed grade of the device using the Speed field. The device in the XS40 Board is usually the slowest model, so select the –3 speed grade as shown below.

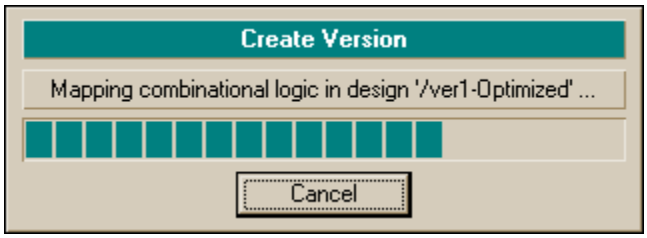


There are many other synthesis options we could adjust by clicking on the SET button and selecting various options in the dialog windows that would appear. There is no need to do this for this simple example, but we will explore these options in more detail in following chapters.

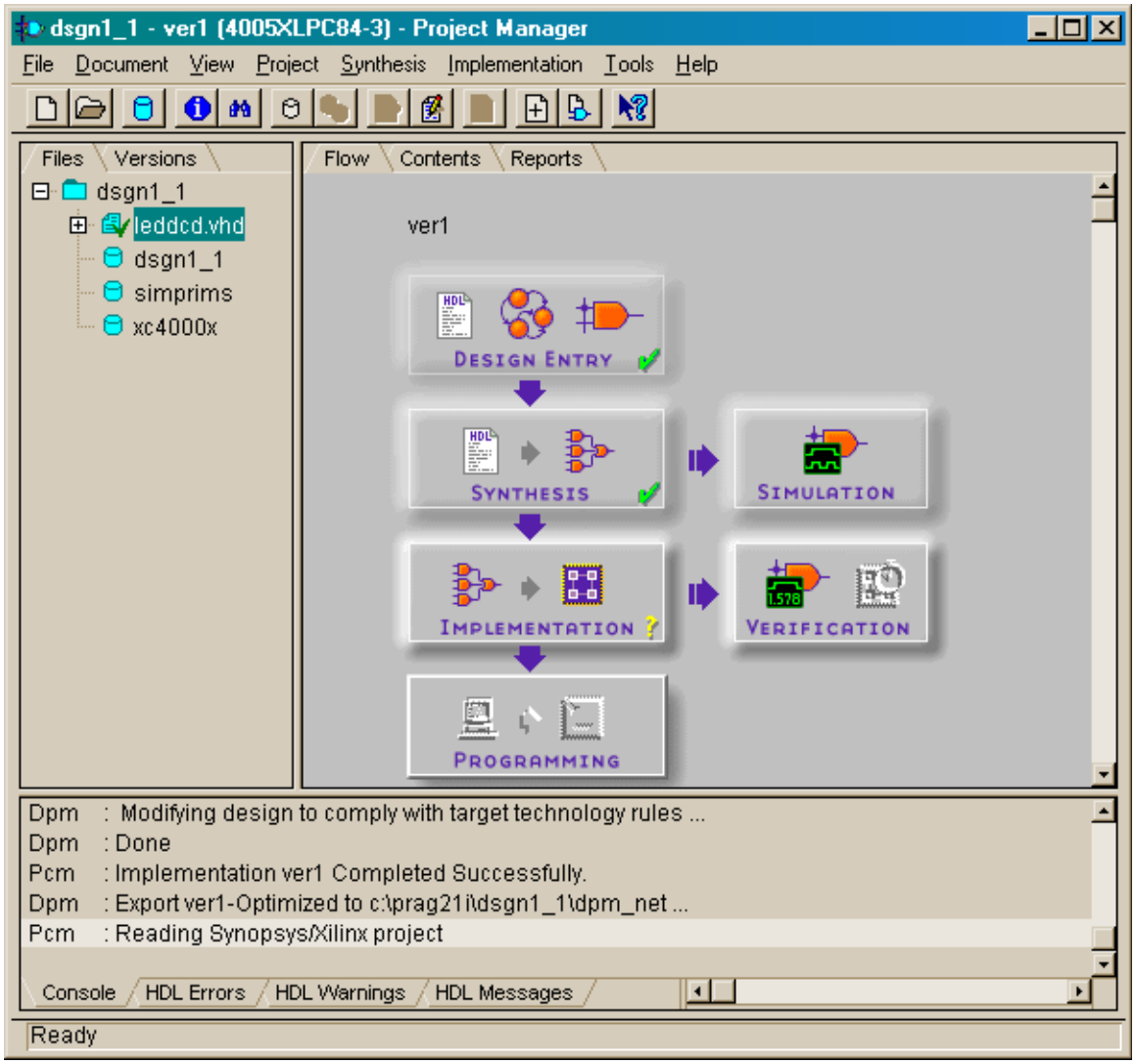
Click on the Run button to start the synthesizer.



A window with a progress bar will appear and show the various phases of the synthesis procedure. For this simple combinational circuit, the synthesis is completed in less than ten seconds on a fast PC.

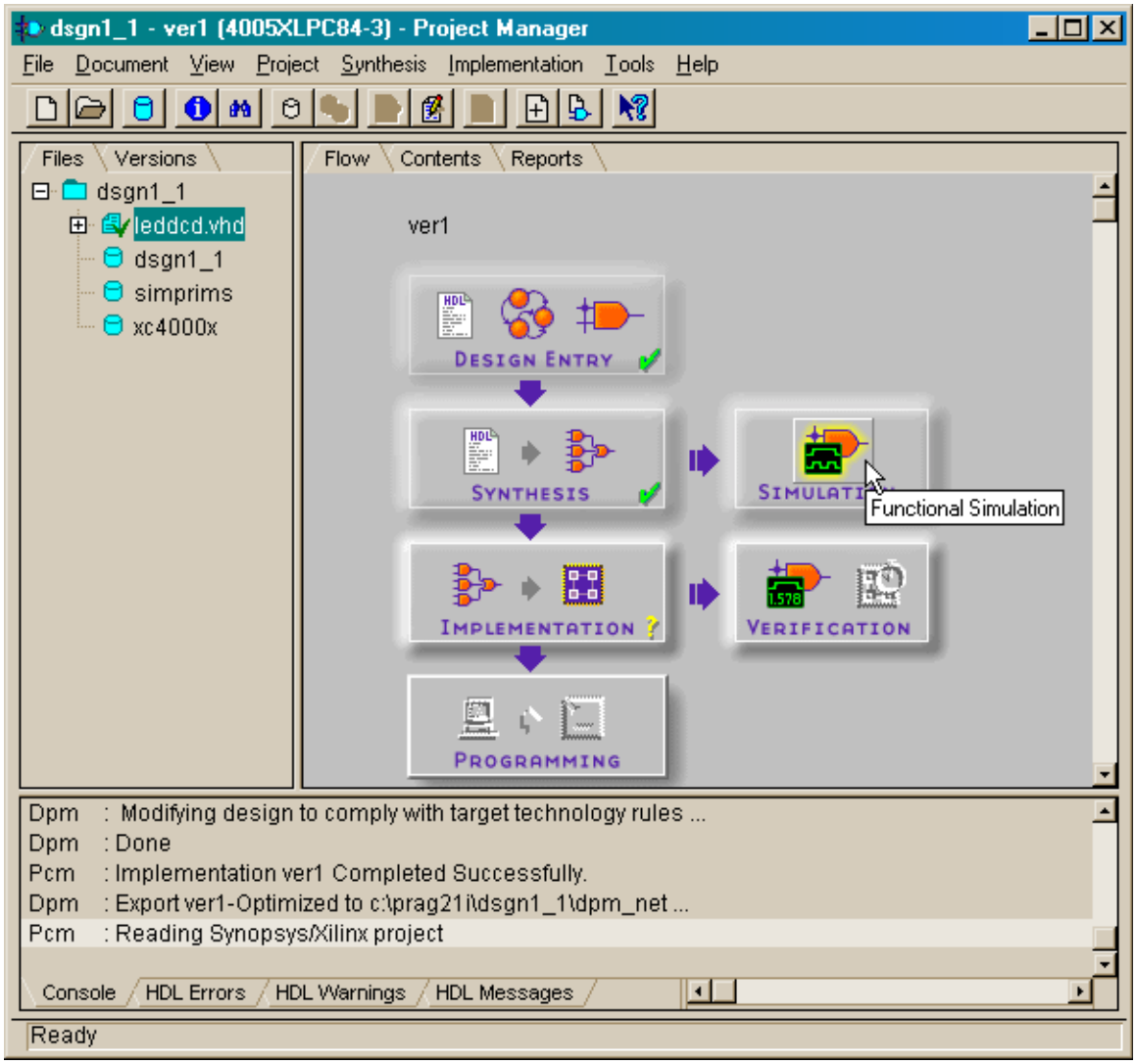


The synthesizer shouldn't encounter any problems generating the netlist for the LED decoder circuit. Upon successful completion of the synthesis, you will see a green checkmark in the Synthesis box of the **Design Flow** pane in the **Project Manager** window. There are also two libraries that have been added to the **Project Hierarchy** pane: xc4000x and simprims. The xc4000x library contains circuit elements that we can add to logic circuits that are targeted to XC4000 FPGAs. The simprims library contains simulation primitives that are used when the Foundation simulator program is simulating a logic circuit targeted at any XILINX FPGA or CPLD. Right now we don't have to be concerned with what's in these libraries.

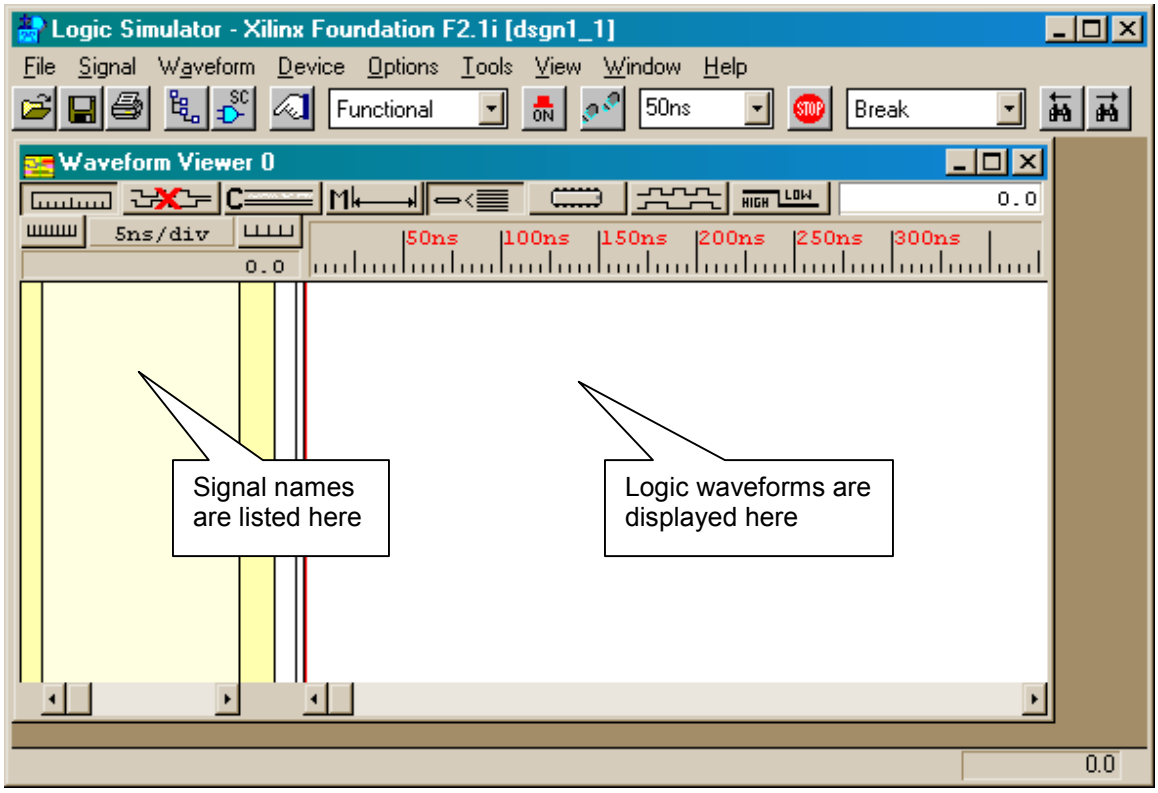


Running a Simulation

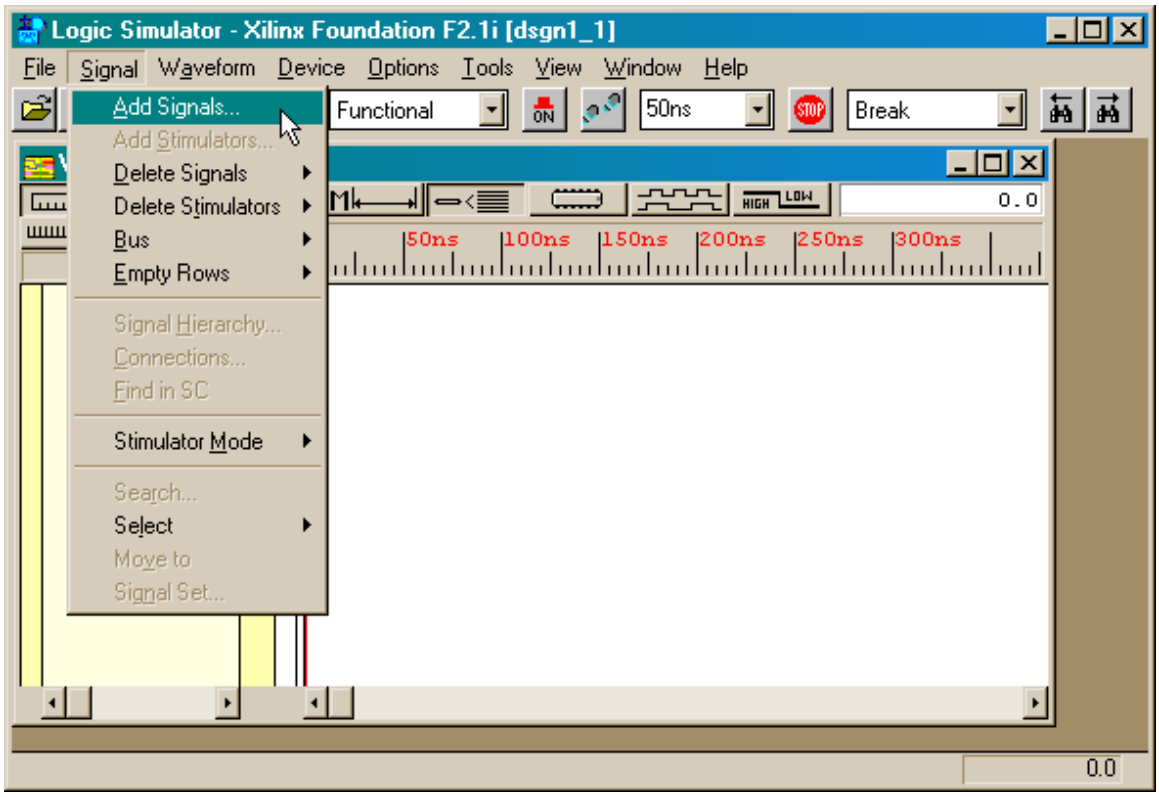
Now that we have a netlist, we can use the simulator in Foundation to check the operation of the LED decoder. Click on the Simulation box in the **Design Flow** pane to activate the functional simulator.



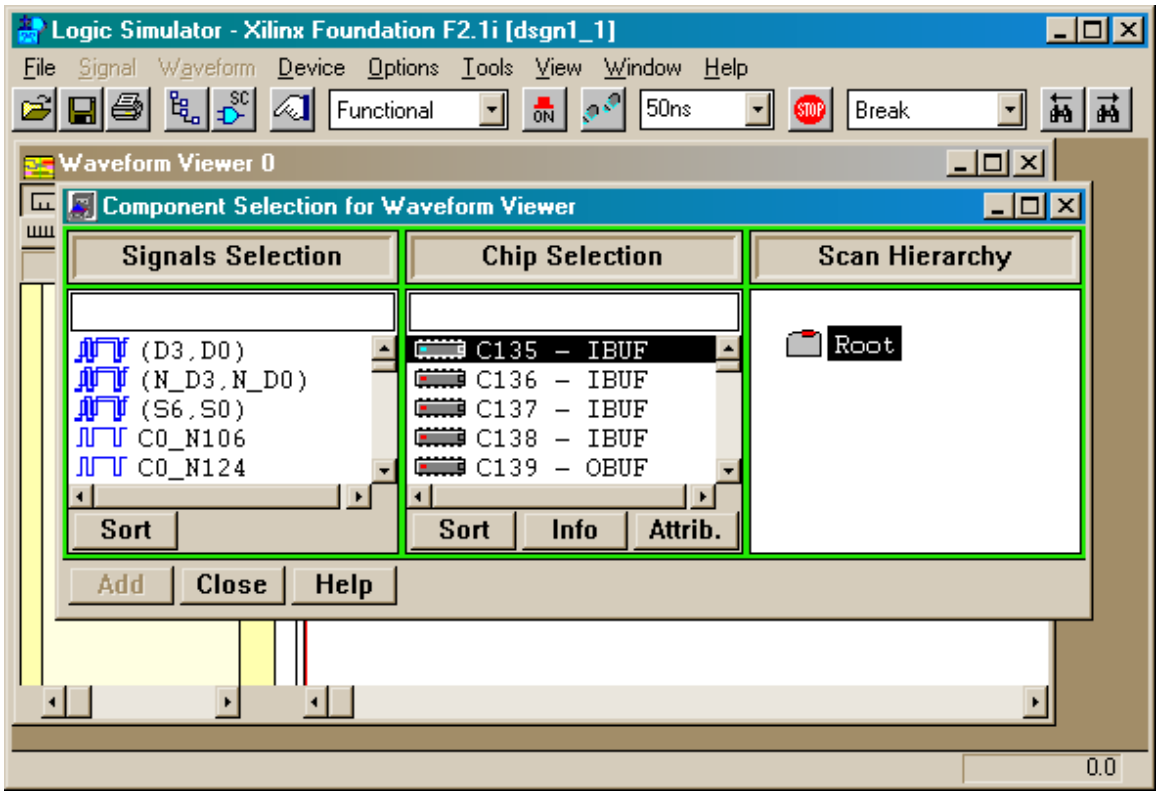
The **Logic Simulator** window that appears contains a single **Waveform Viewer 0** subwindow for viewing simulated waveforms of signals in the LED decoder circuit.



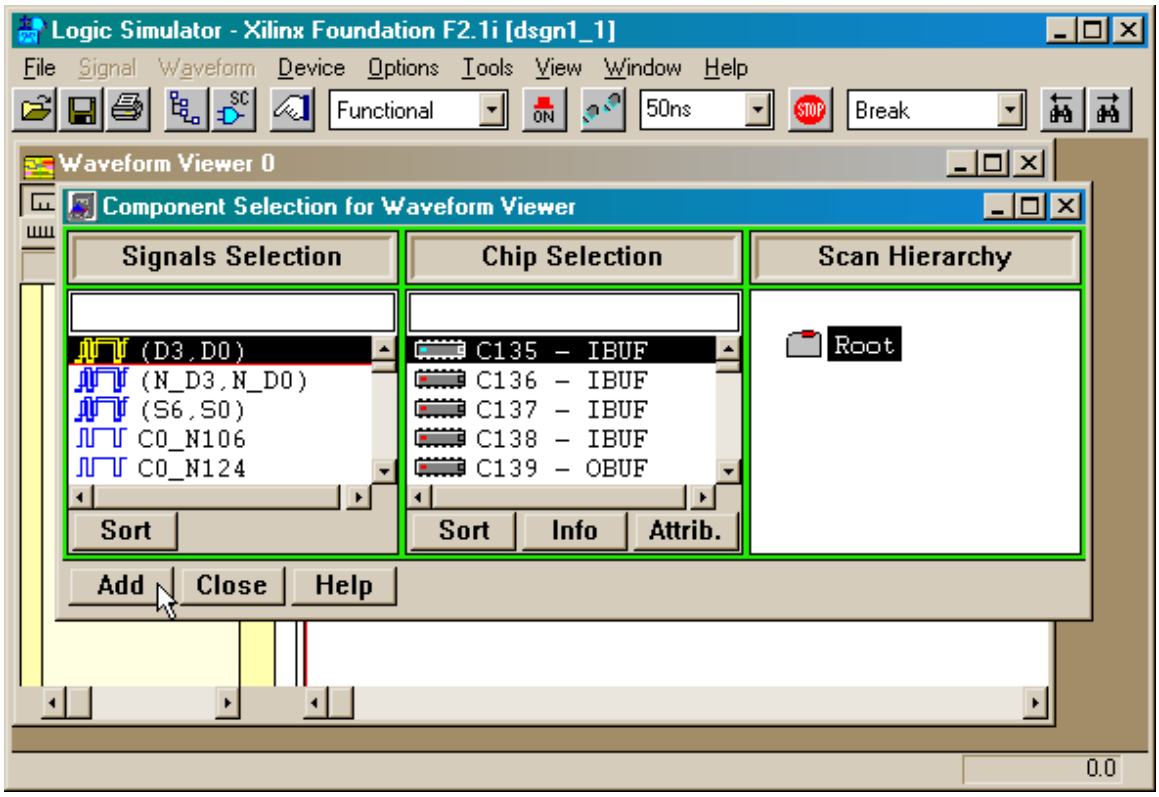
Now all we have to do is find a way to inject signals into the inputs of the LED decoder netlist and then observe the response of the outputs. We start by selecting the Signal→Add Signals... menu item.



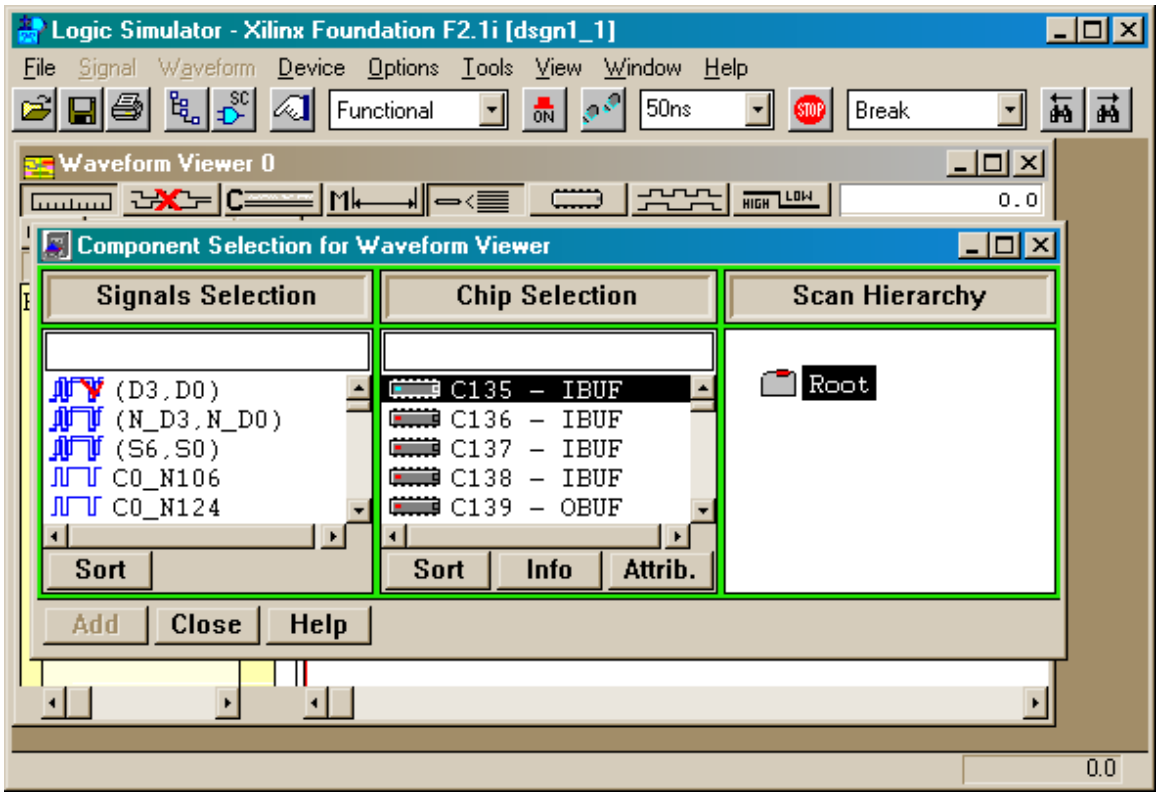
The **Component Selection for Waveform Viewer** window appears with three panes. For our simple LED decoder, we are only interested in the left-hand **Signals Selection** pane. All the input and output ports for the LED decoder circuit are listed in this pane along with all the internal signals that connect the gates in the synthesized netlist.



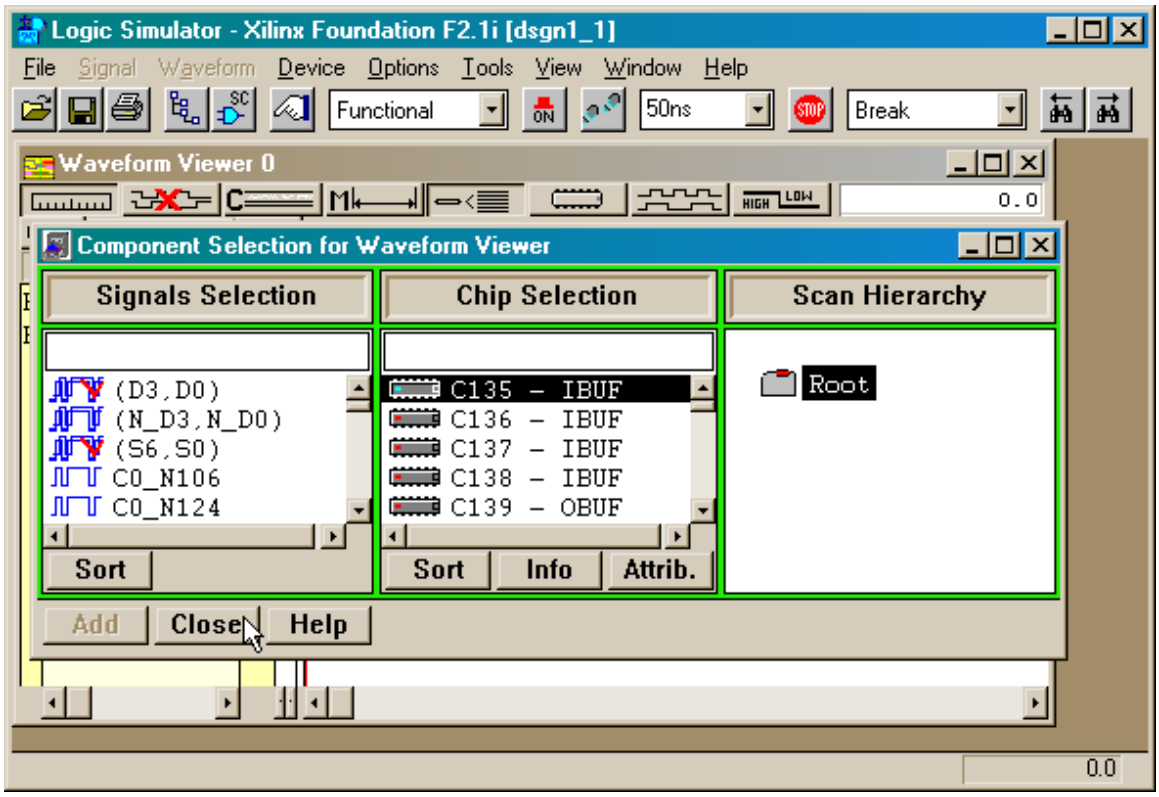
For this example we are only concerned with the inputs and outputs of the LED decoder. Click on the top entry in the **Signals Selection** pane to highlight the set of four inputs to the LED decoder (**D0**, **D1**, **D2**, and **D3**). Then click on the Add button.



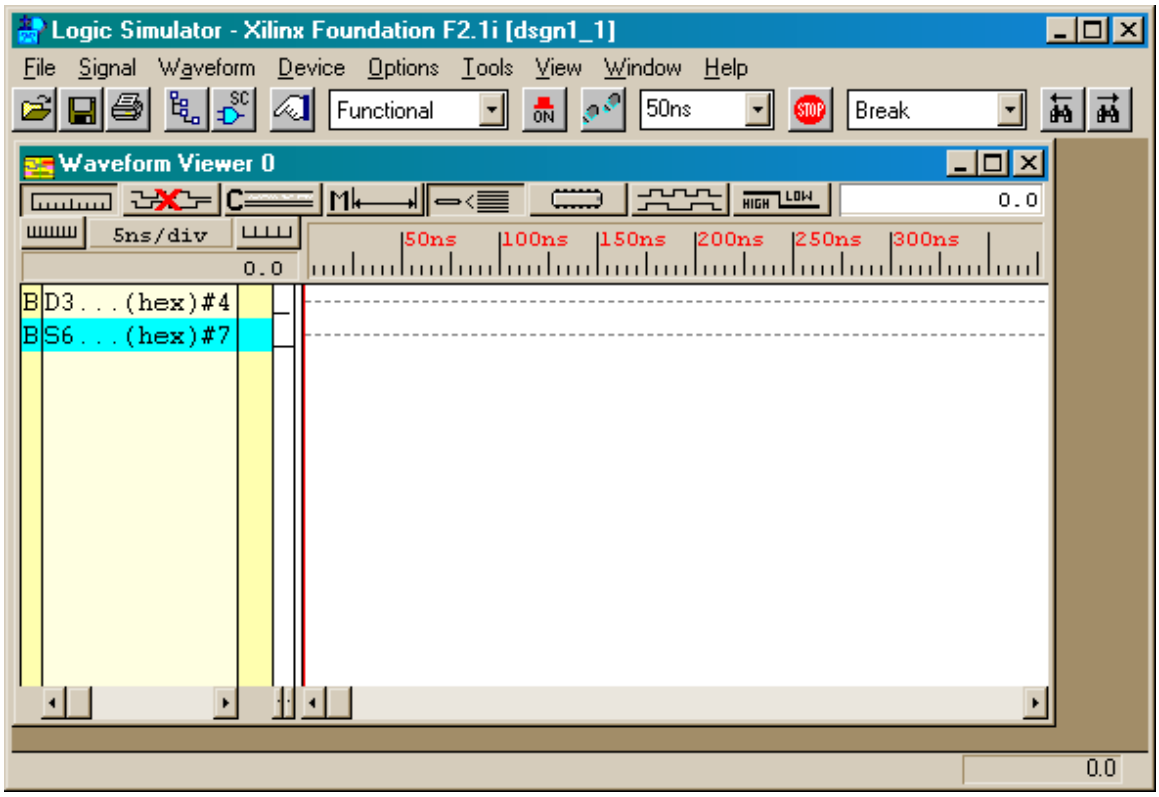
At this point a red checkmark should appear next to the input ports. This indicates that the four inputs to the LED decoder have been added to the **Waveform Viewer 0** window.



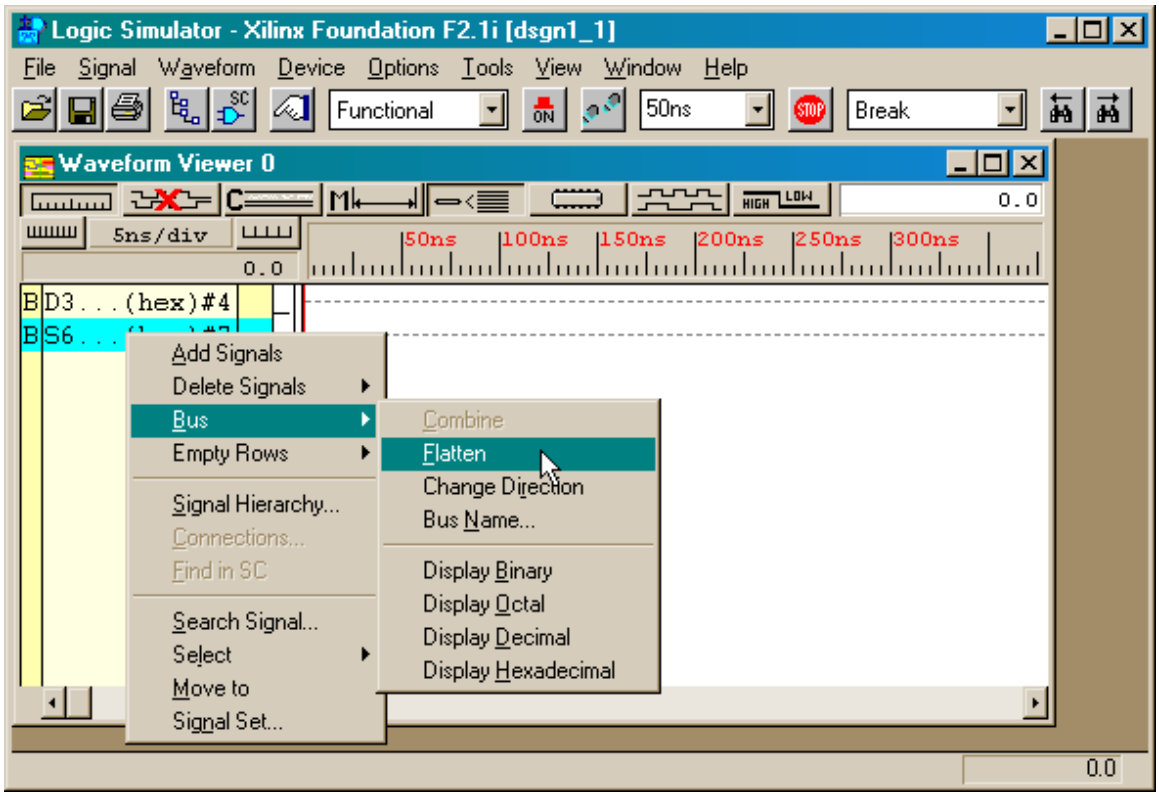
Repeat the previous operation with the seven LED decoder outputs (**S0**, **S1**, **S2**, **S3**, **S4**, **S5**, and **S6**). Then click on the Close button to remove the **Component Selection** window.



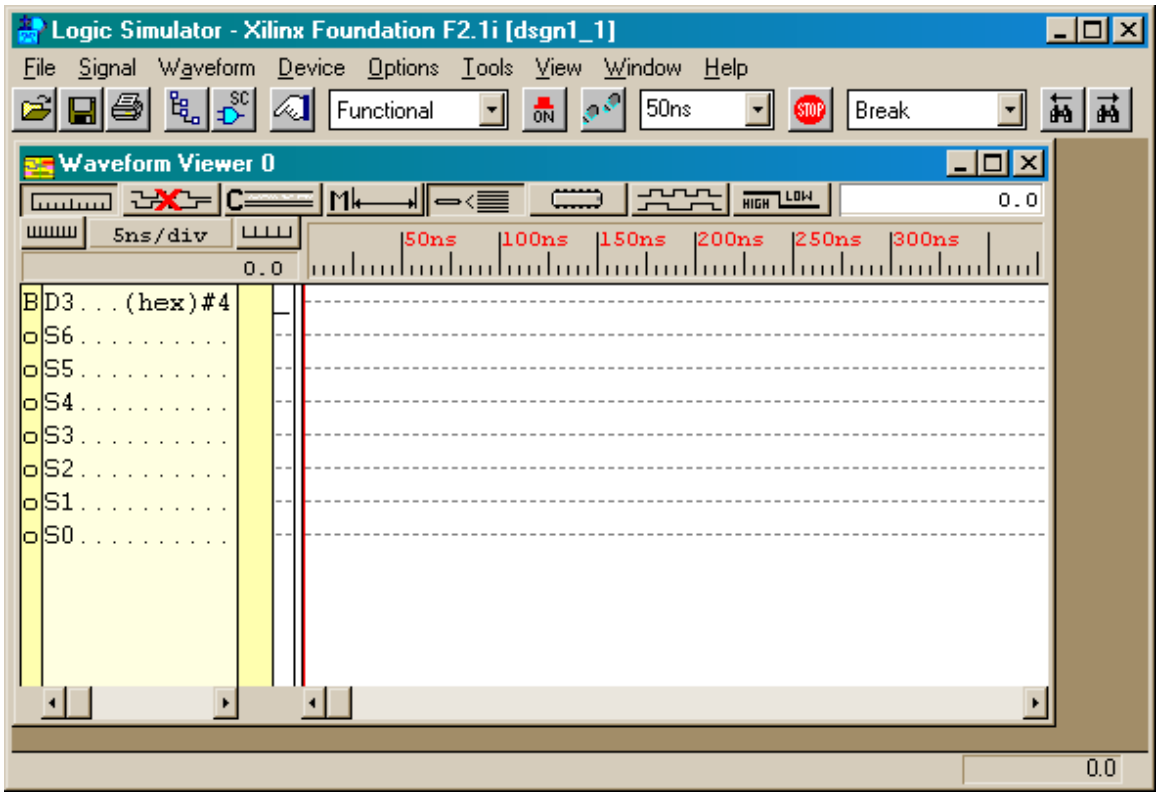
Now you will see the input and output ports have been added to the **Waveform Viewer** window. The four input ports and seven output ports are grouped into buses as indicated by the B in the left-most column next to their signal names. Therefore, the logic levels on the four inputs and seven outputs will be grouped and displayed as hexadecimal values in the logic waveform pane of the **Waveform Viewer** window.



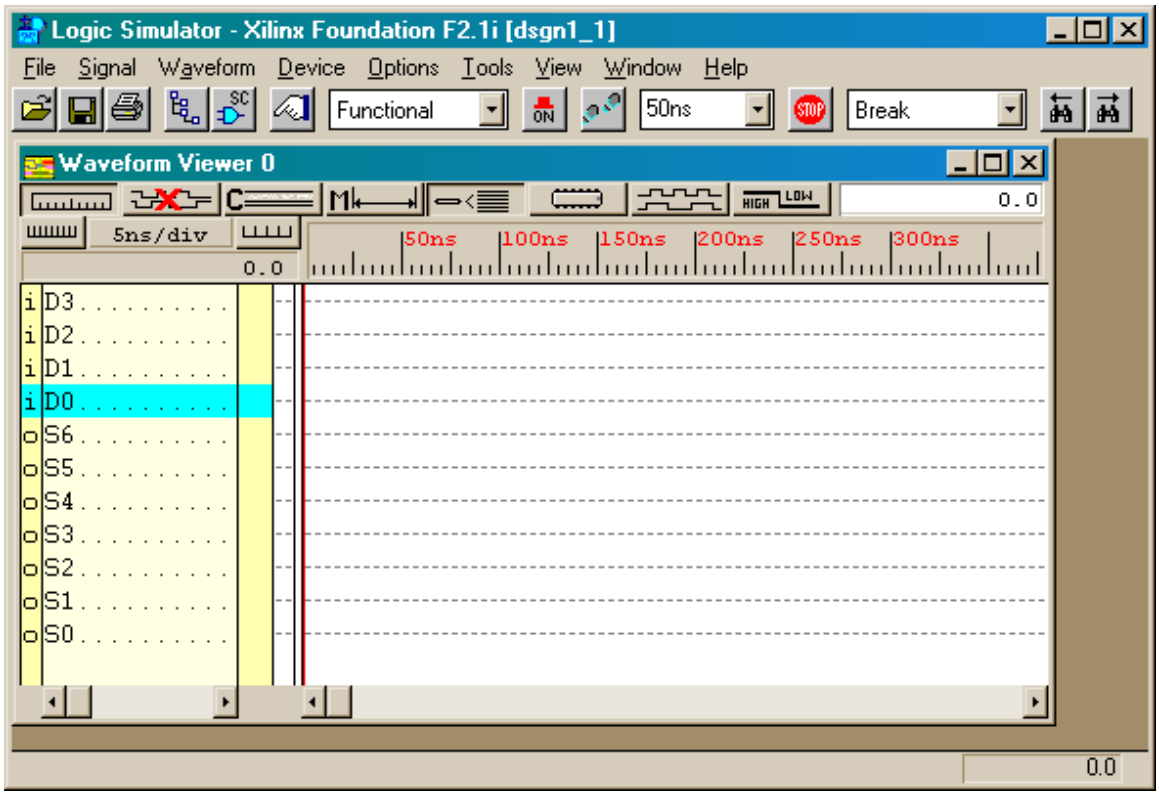
For this example it is more convenient to flatten the buses so we can observe the logic level on each individual input and output port. To do this, right-click on the output bus and select the Bus→Flatten menu item in the cascading pop-up menues.



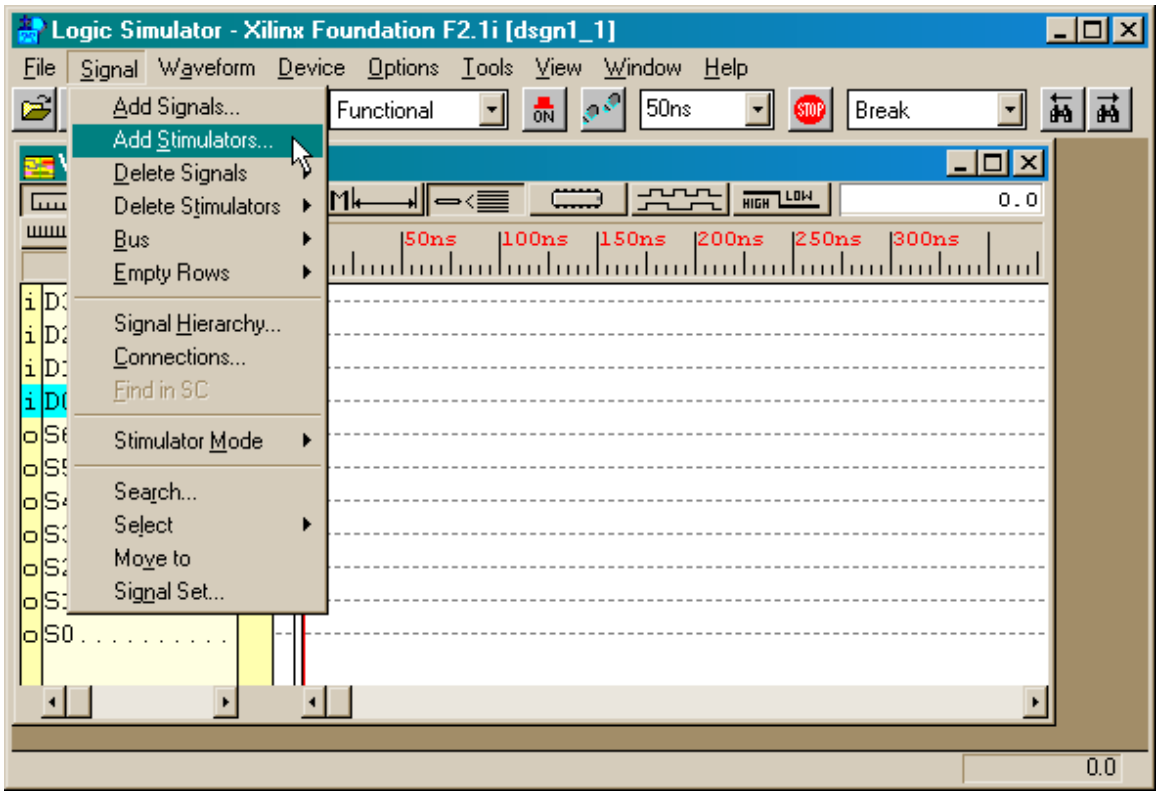
The individual signals in the output bus are now displayed in the **Waveform Viewer** window.



Next we repeat the bus flattening operation for the input bus. After doing this, we can see that the bus indicator in the left-most column of the **Waveform Viewer** window has been replaced with an i or o depending upon whether the flattened signal is an input or output, respectively.

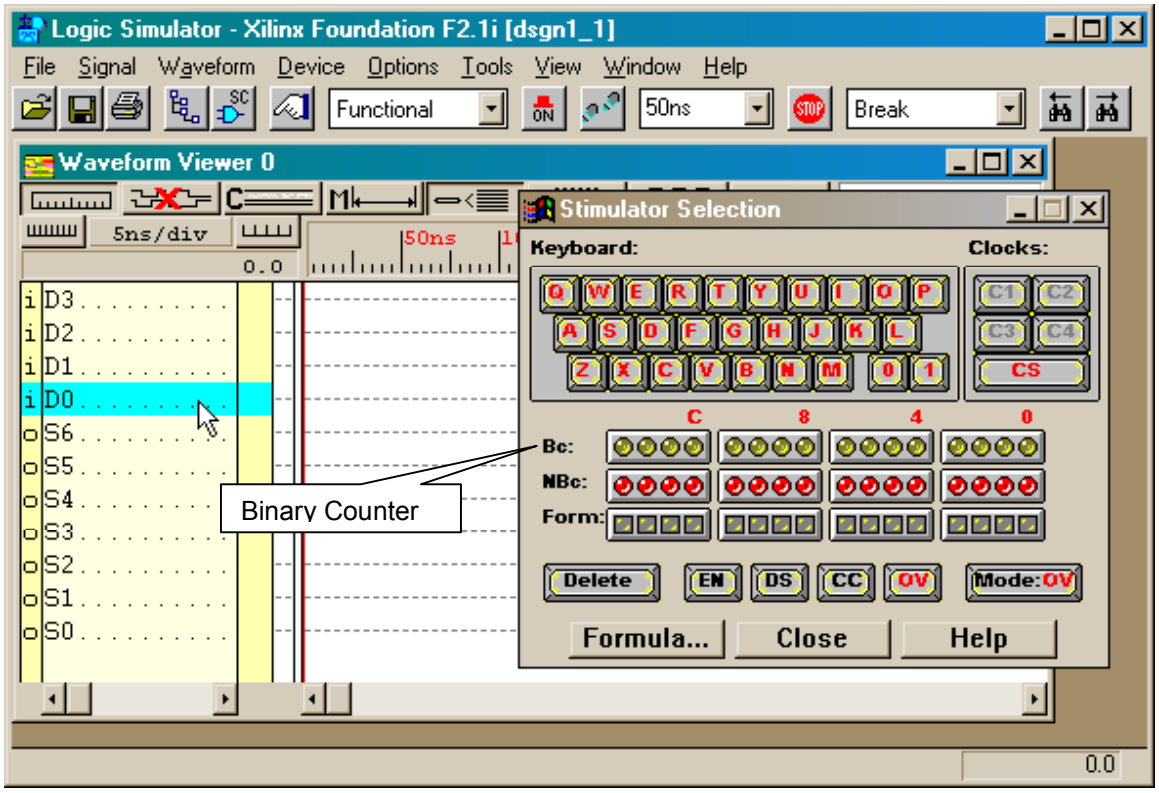


Now that we have all the inputs and outputs displayed in the proper format, we need a way to stimulate the input ports with binary logic levels so we can observe the response on the output ports. Select the Signal→Add Stimulators... menu item to begin adding stimulators to the input ports.

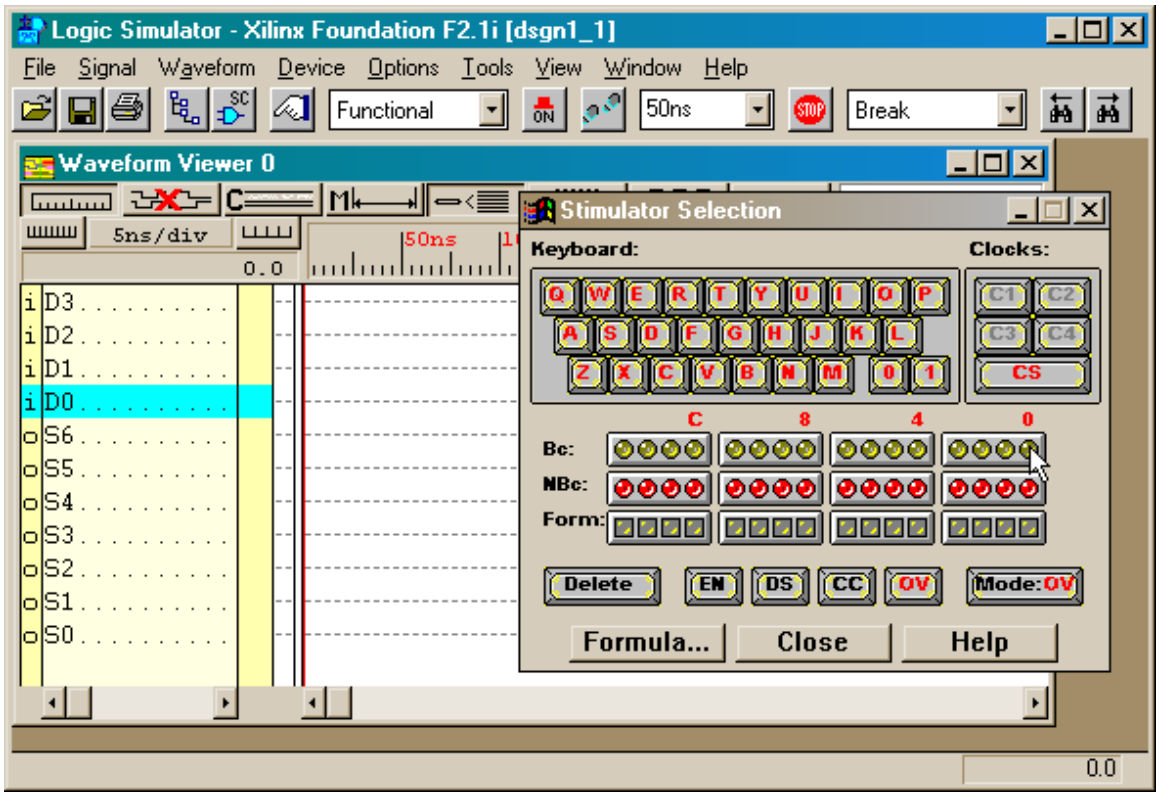


Now the **Stimulator Selection** window appears and it has all the buttons and controls you would ever want to see. But we will only use a few for this example. In the middle of the window is a binary counter labeled Bc. This counter will increment once for each time step that the simulator executes during the simulation of the LED decoder. If we attach the four LED decoder inputs to the lower four bits of this counter, then we can force all possible input combinations into the decoder circuit over a span of sixteen simulation cycles.

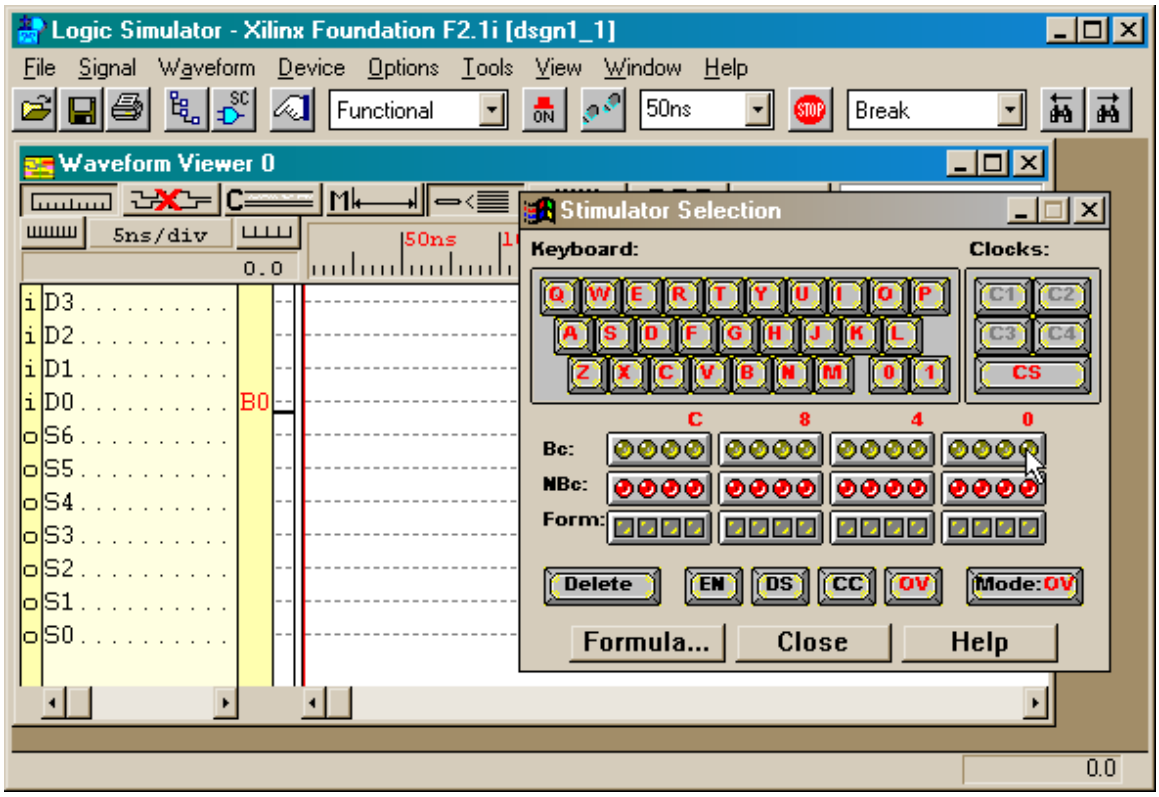
To begin attaching inputs to the binary counter bits, highlight the least-significant input port **D0**.



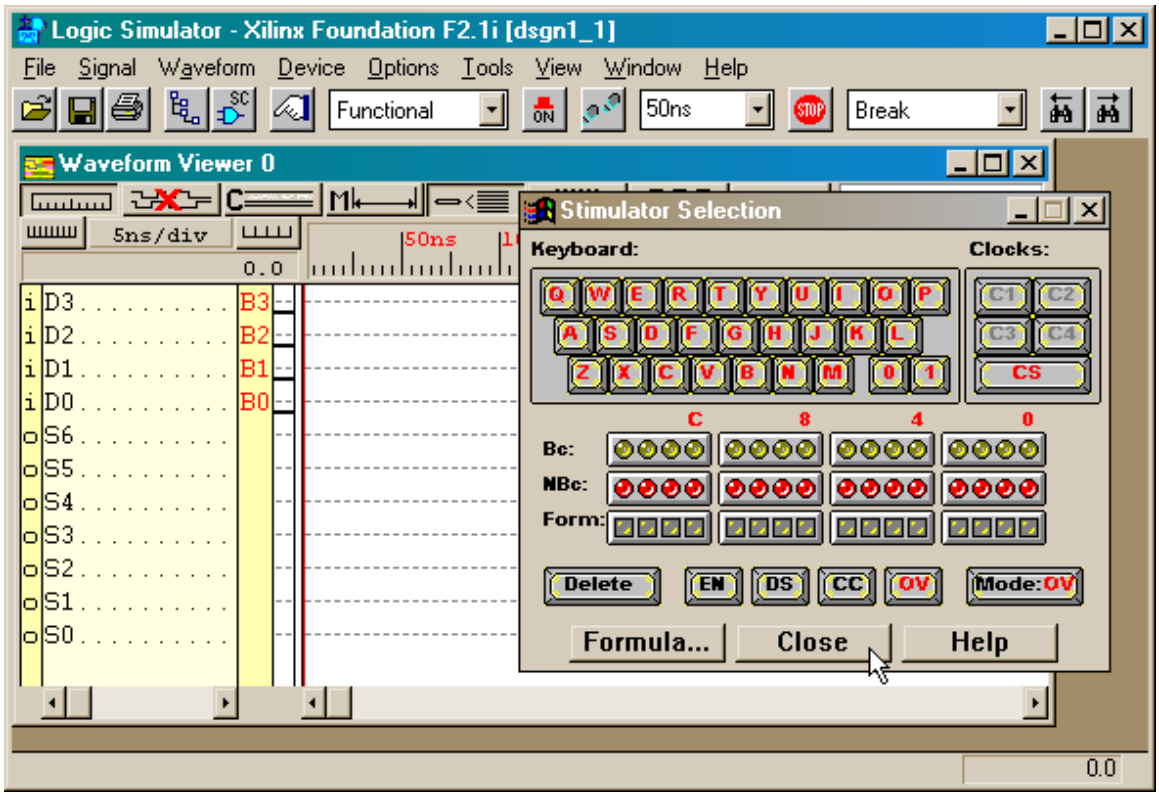
Once the **D0** input port is highlighted, click on the least-significant bit of the binary counter.



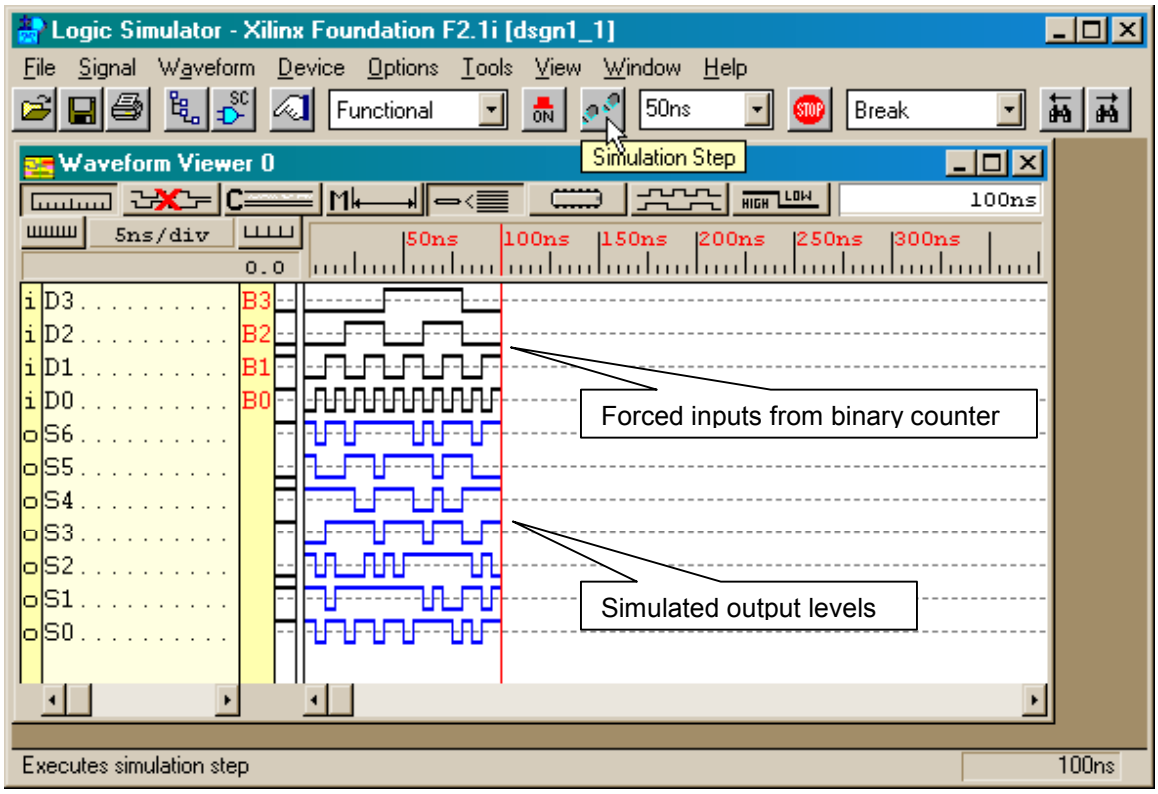
Now we see that B0 has appeared in the column to the right of the **D0** signal name, indicating that the least-significant input to the LED decoder has been connected to the least-significant bit of the stimulation counter.



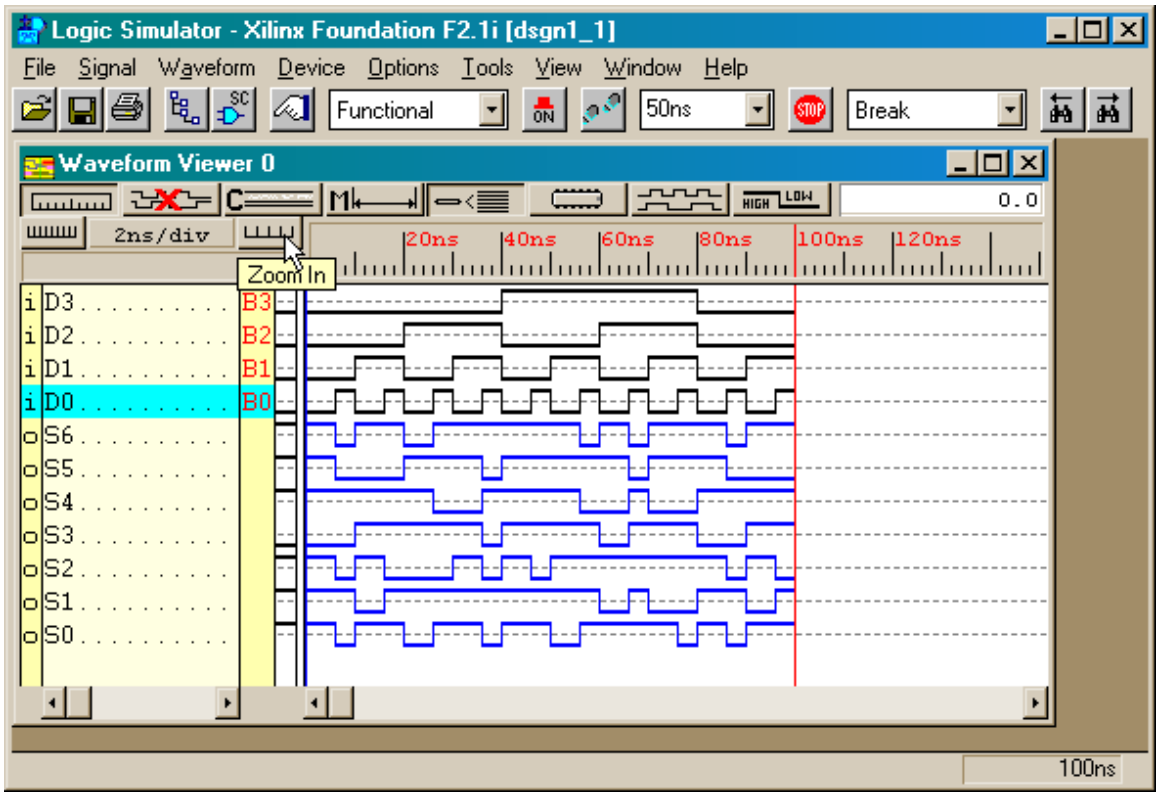
We can repeat this operation to connect the succeeding three inputs of the LED decoder to the next three bits of the counter. Then click the Close button.



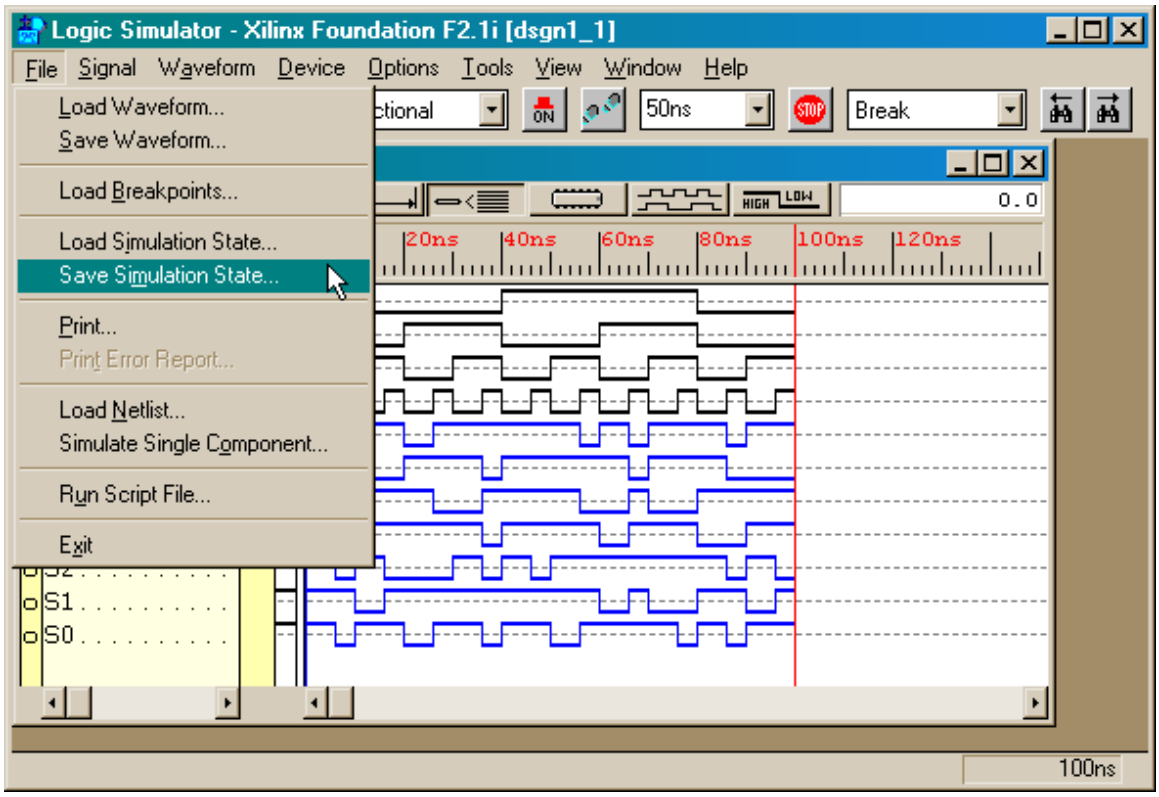
Now we are finally ready to test the LED decoder. To run the simulation, click on the Simulation Step button a few times. The right-hand pane of the **Waveform Viewer** window will display the binary counter values forced into the inputs and the resulting logic levels on the outputs of the LED decoder. The simulation covers a time interval of 50 ns.



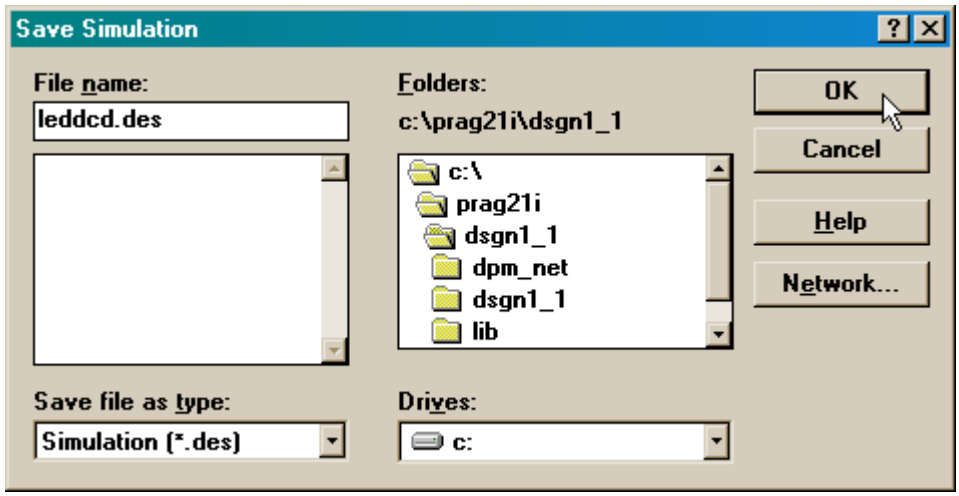
The waveforms may be a bit squeezed to view easily, so you can click the Zoom In button to expand the time scale. Now we can check the output levels for each input combination to see if they match the input-output pairs specified in the VHDL code. For example, when all the inputs are at logic level 0, then all the outputs are at a high logic level except for **S3**. This agrees with the behavior specified on line 14 of the leddcd.vhd file. You can check the response to the remaining fifteen input combinations in a similar fashion.



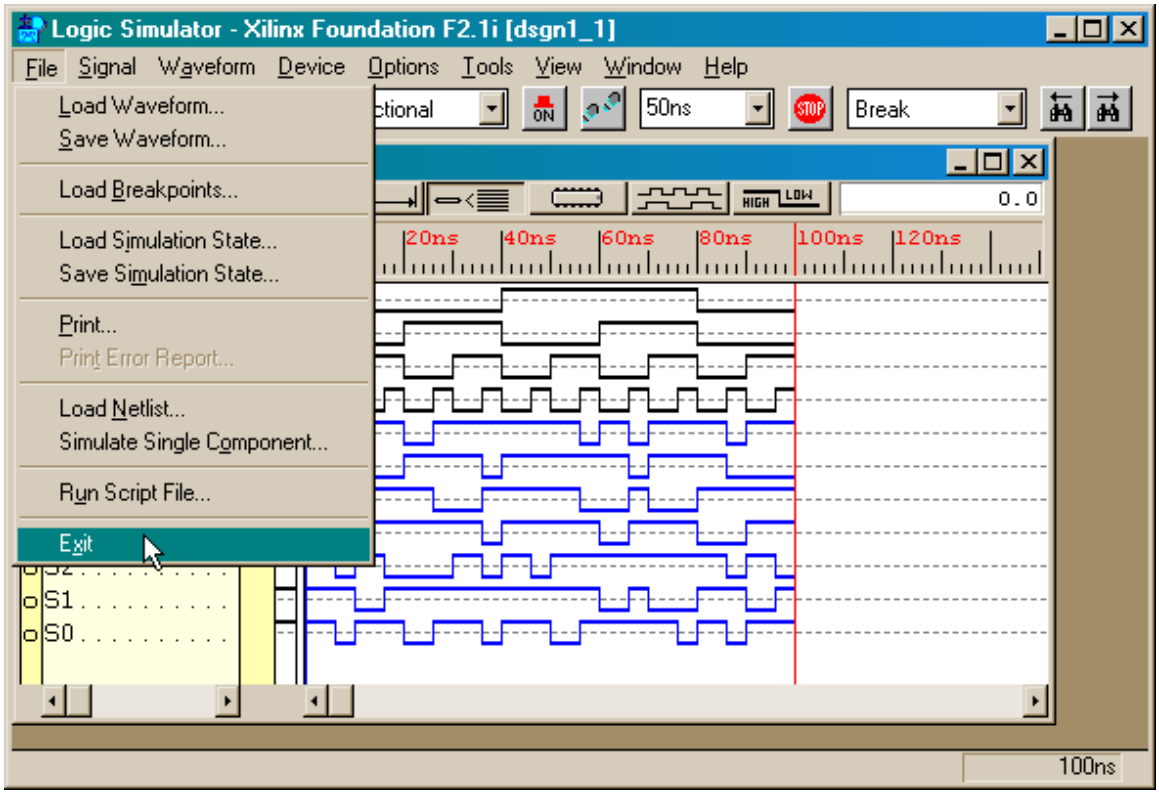
We can exit the simulator now that we have verified the operation of the LED decoder, but it is wise to save the state of the simulator in case we have to come back and do more tests later. That way we don't have to go through all the effort of adding the input and output signals and attaching stimulators again. Begin this task by selecting the File→Save Simulation State... menu item.



In the **Save Simulation** window that appears, type leddcd.des into the File name field and click on OK. The current state of the simulator will be saved into this file in the project folder. You can restore the simulator to this saved state by clicking File→Load simulation State... in the **Logic Simulator** window and selecting the leddcd.des file.



Finally we can select File→Exit to terminate this simulation session.



Implementing the Design

Now that the simulation has given us some confidence that our circuit is performing as an LED decoder, we want to transform the netlist into a bitstream that we can run on a real XC4005XL FPGA in an XS40 evaluation board. The FPGA has some of its pins connected to the parallel port of the PC attached to the XS40 board. We can use the PC to drive logic levels through the parallel port and onto the pins of the FPGA where they will serve as inputs to the LED decoder in the FPGA. The FPGA also has seven pins connected to a seven-segment LED digit. The outputs from the LED decoder should be connected to these pins so we can easily see if the circuit operates correctly. We will create a user-constraints file that will tell the Foundation implementation tools which LED decoder inputs or output should be assigned to a particular physical pin of the FPGA.

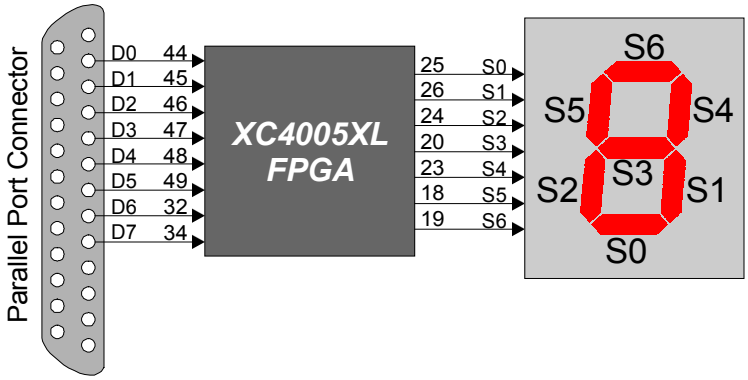
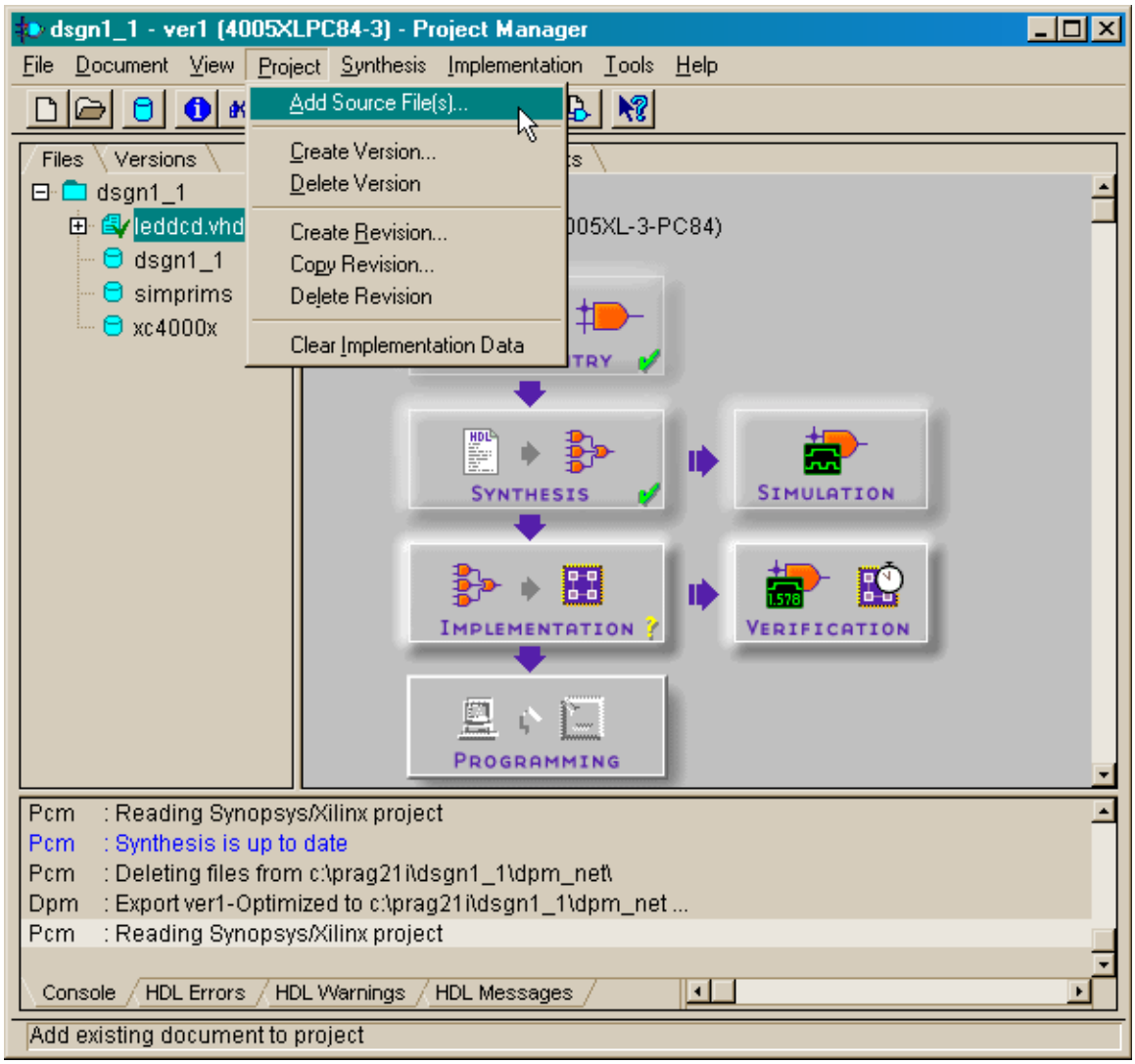
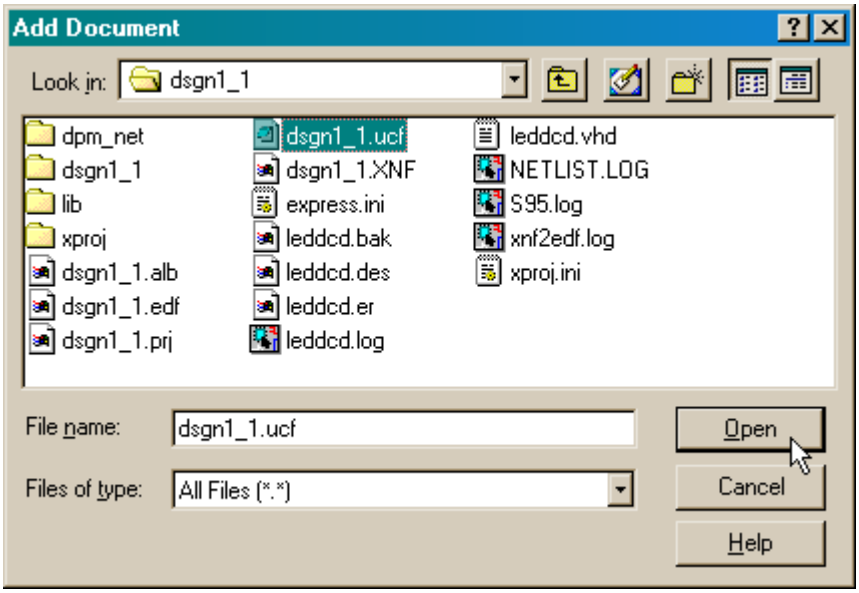


Figure 2: Connection of the parallel port and LED digit to the pins of the FPGA on the XS40 Board.

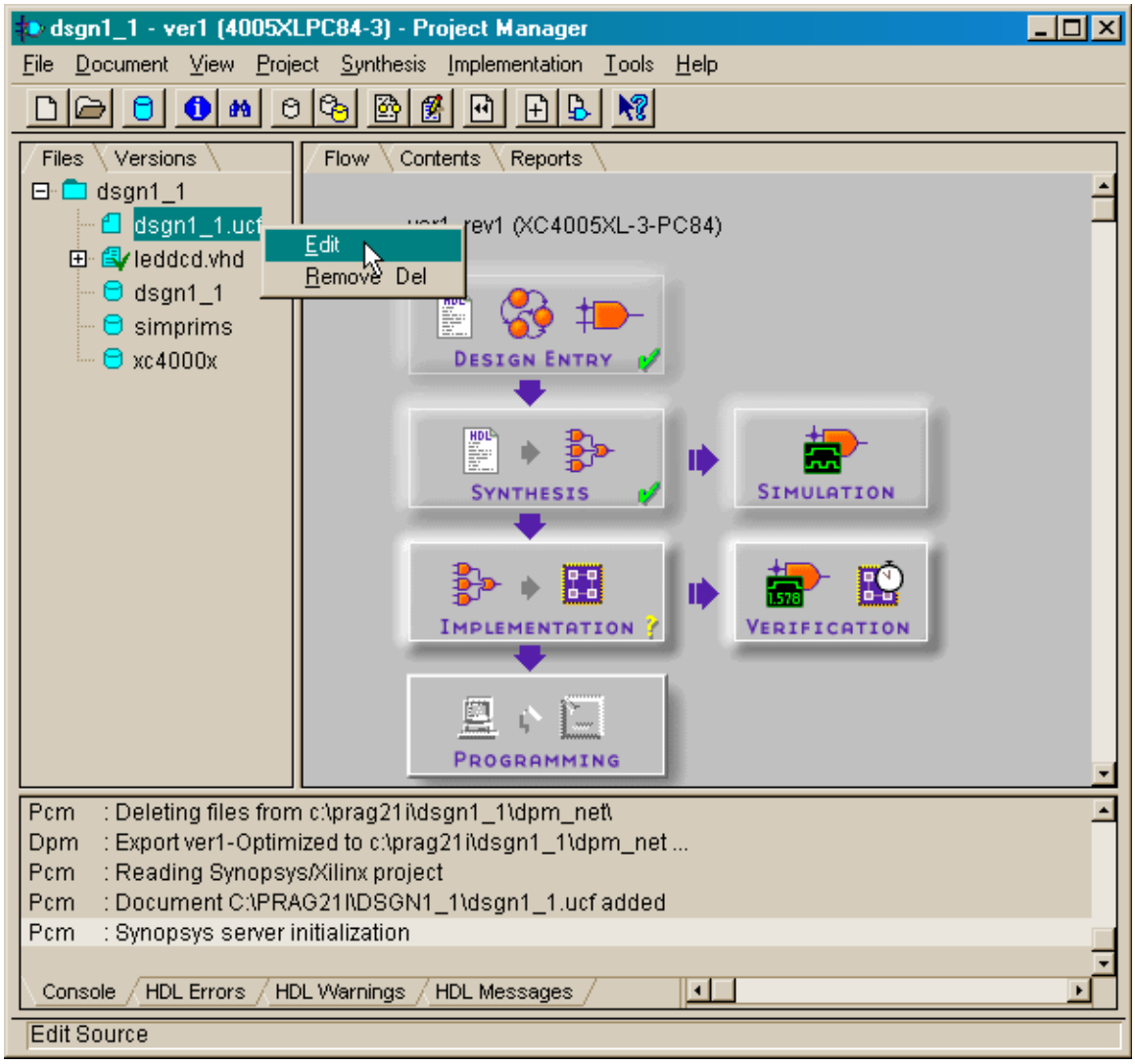
To begin creating the user constraints, select the Project→Add Source File(s)... menu item in the **Project Manager** window.



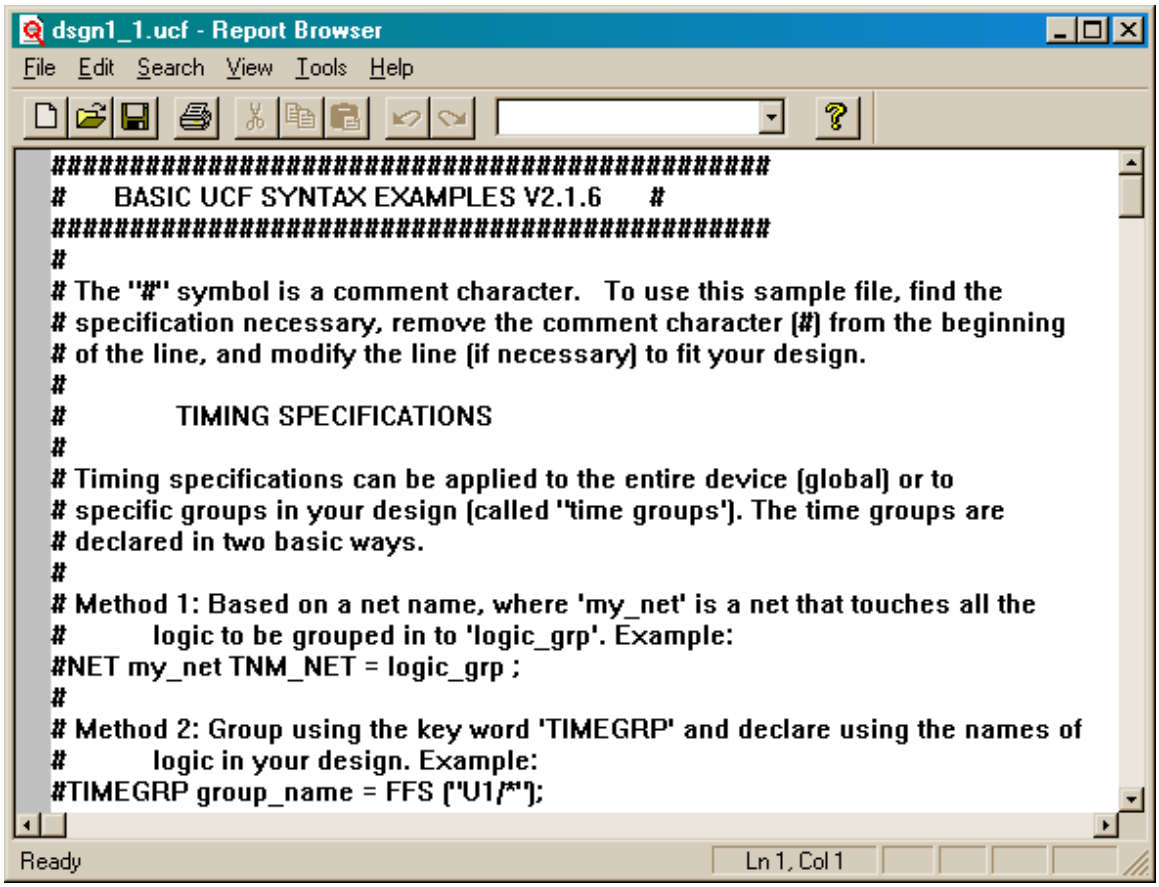
Foundation 2.1i automatically created a user-constraints file called dsgn1_1.ucf when we started this project. Select this file in the **Add Document** window and click on Open.



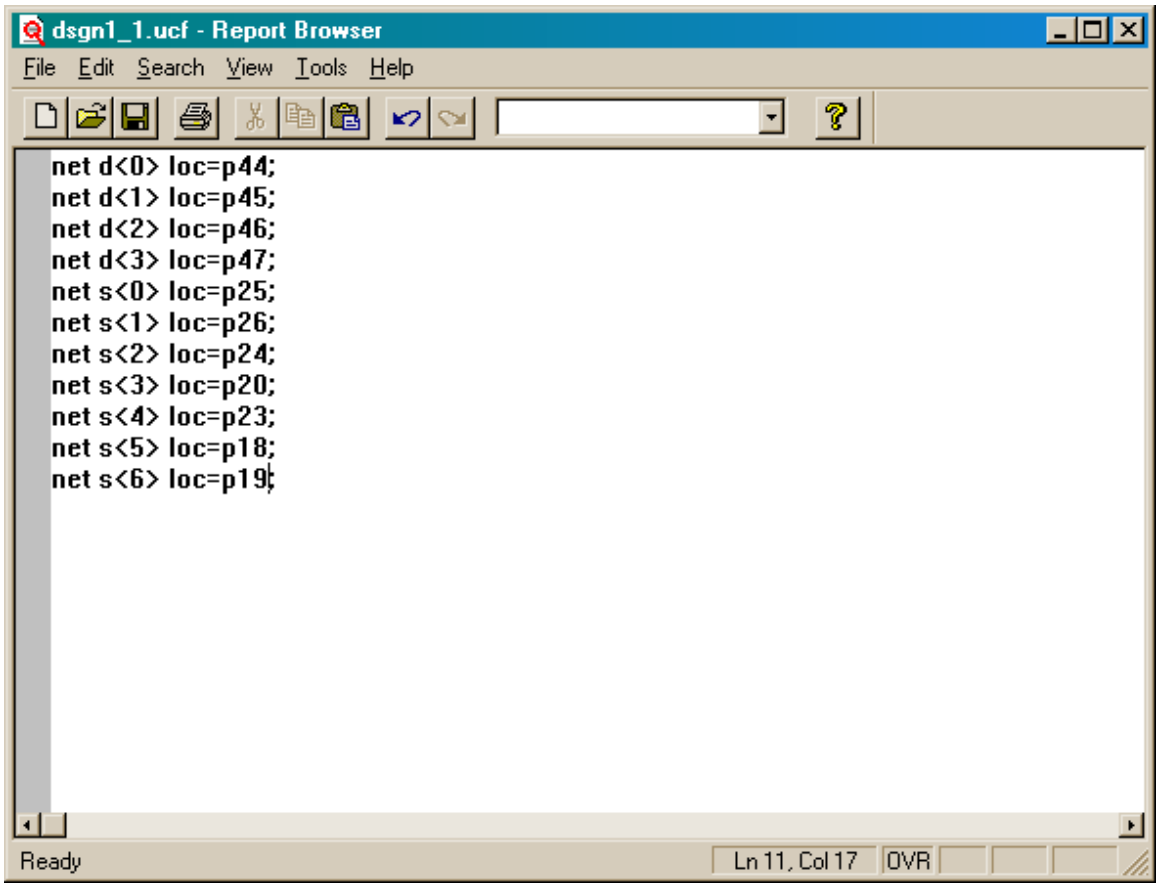
The dsgn1_1.ucf user-constraints file is now added to the project hierarchy. We will place our pin assignments for the LED decoder in this file. Right-click on the dsgn1_1.ucf label in the **Project Hierarchy** pane and select Edit in the pop-up menu.



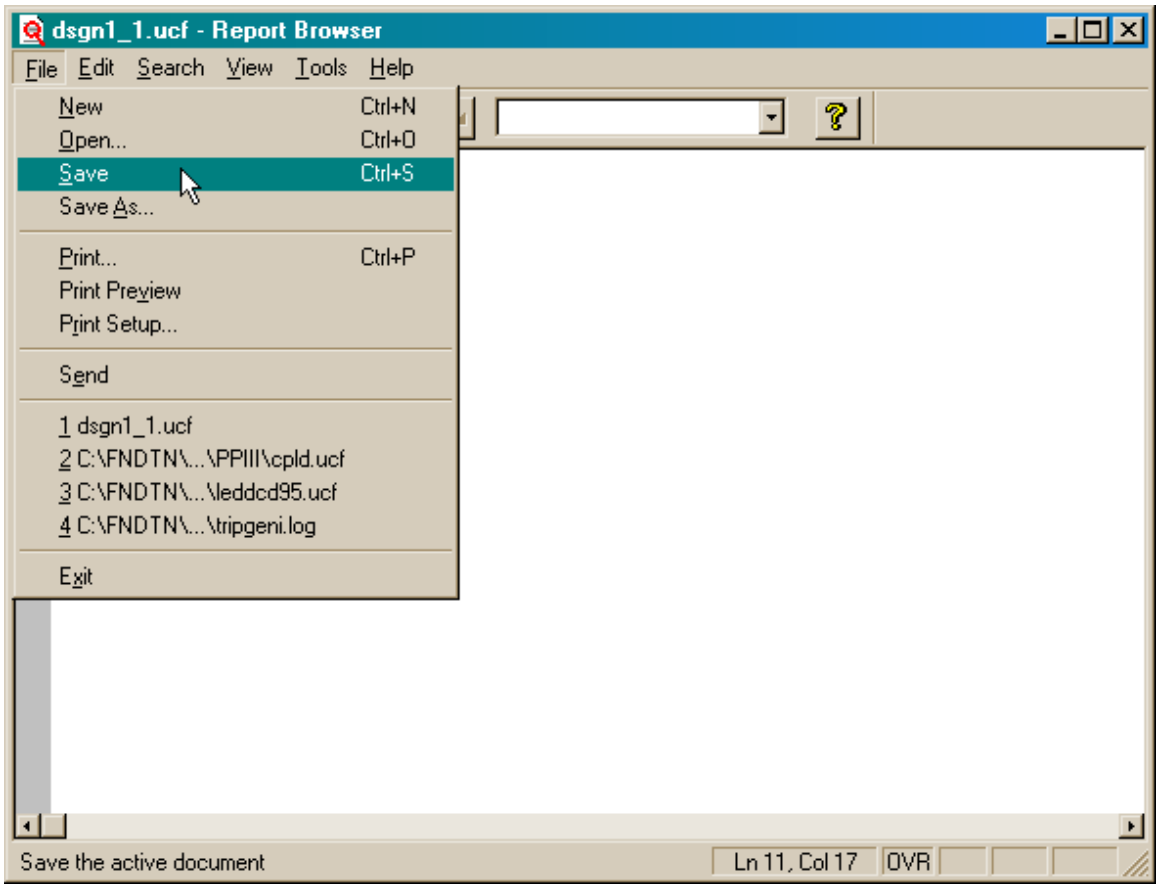
The design1_1.ucf file contents now appear in the **Report Browser** window. Foundation placed a large amount of information about writing constraints in the file when it was created. You can use Edit→Select All and Edit→Delete to remove all these comments from the file, or you can just add your own constraints to the file.



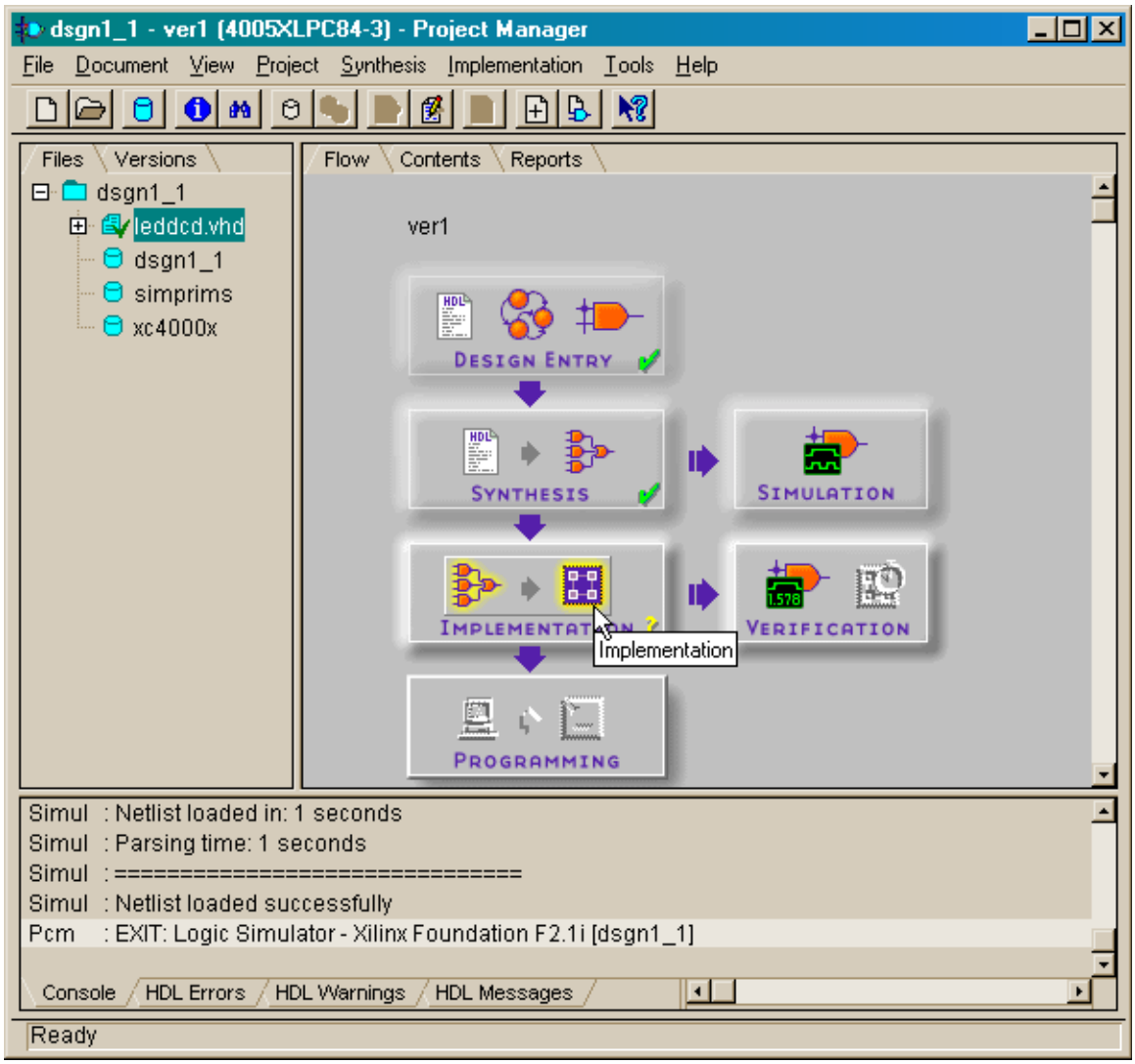
The window below shows the pin assignment constraints used when the LED decoder circuit is targeted to an XS40 Board. The **net** keyword is followed by the name of one of the inputs or outputs of the circuit. The Foundation synthesizer places **<>** around the index of each element in an input or output bus in the netlist, so signal **d0** (for example) must be referred to as **d<0>** in the user constraints file. The signal name is followed by a location constraint indicated by the **loc** keyword followed by the particular pin the signal is assigned to. For example, input signal **d0** is assigned to pin 44 on the 84-pin PLCC package of the XC4005XL FPGA on the XS40 Board.



After entering all the constraints, click on File→Save and File→Exit to save the new pin assignment constraints and close the file.

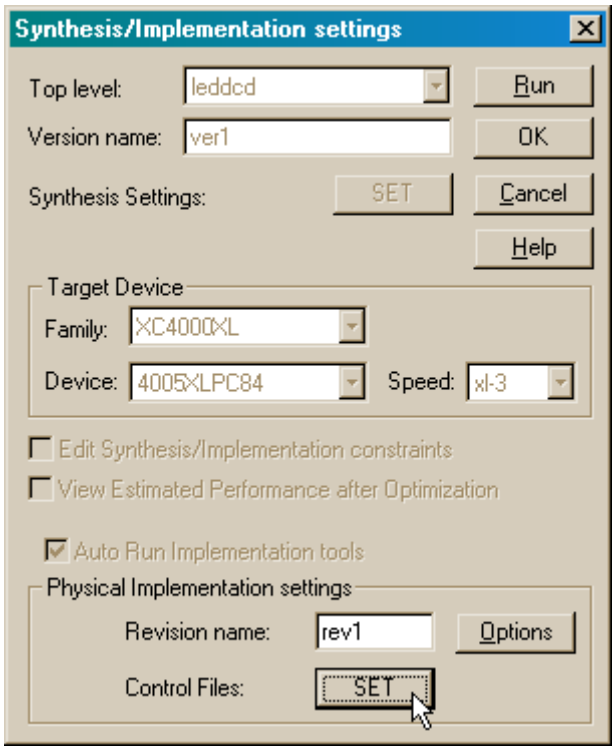


Now we can start the implementation tools by clicking on the Implementation block in the **Design Flow** pane.

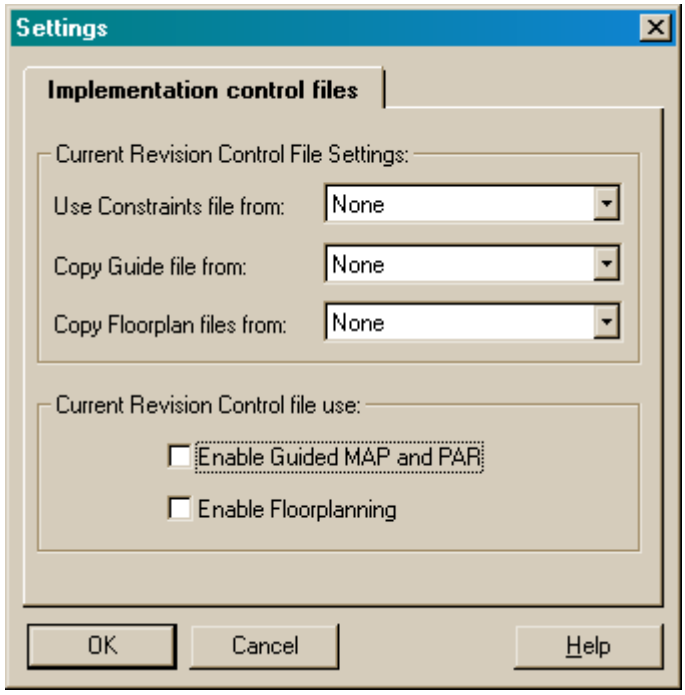


We are again presented with the **Synthesis/Implementation setting** window just as we were when we activated the synthesizer in a previous phase of the design flow. But now we can only set the options and control files that affect the implementation tools – all the synthesis settings and the target device selector are locked. This makes sense since the synthesizer generated a netlist based upon the characteristics of the XC4000 FPGA, so we cannot decide to implement that netlist on a different type of programmable device which may not have the same features.

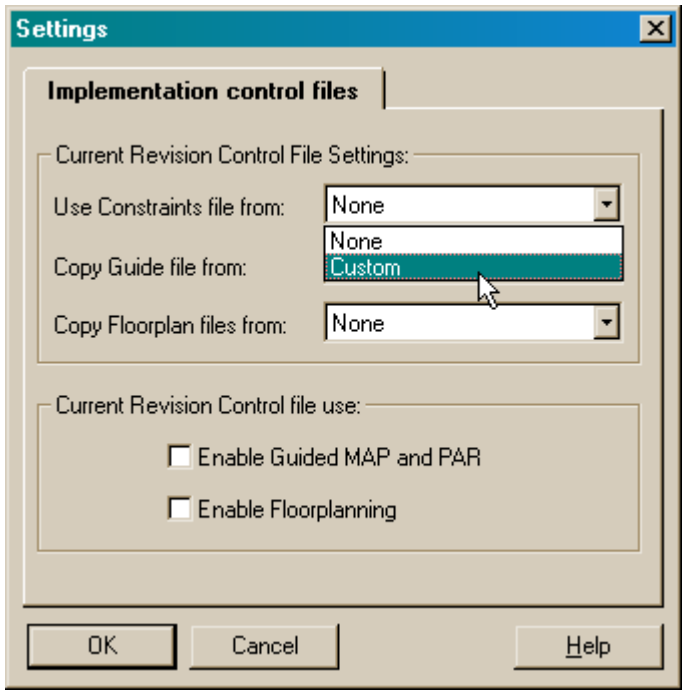
We need to inform the implementation tools about the user constraints file with the pin assignments for the LED decoder. Click on the SET button in the Physical Implementation settings area to do this.



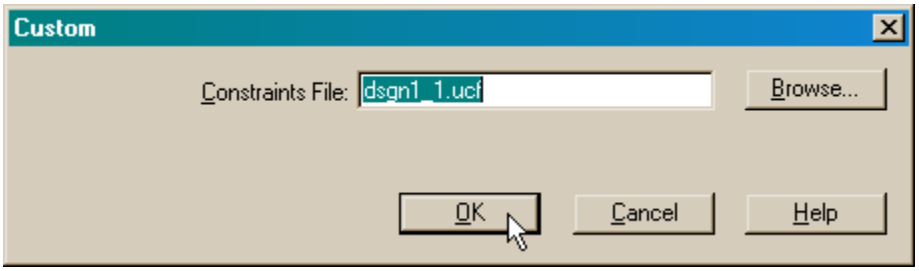
The **Settings** window lets us specify several types of control files. Guide files are used if we want the implementation tools to use what they learned in a previous implementation of this circuit to guide the current implementation process. Similarly, floorplanning files are created by the designer to give the implementation tools hints about where they should place the netlist components and wires within the FPGA or CPLD programmable arrays. Guide and floorplanning files can significantly reduce the time required to complete the implementation phase. We haven't run the implementation tools on the LED decoder yet, and we haven't created a floorplan for the circuit so neither of these options is of interest to us.



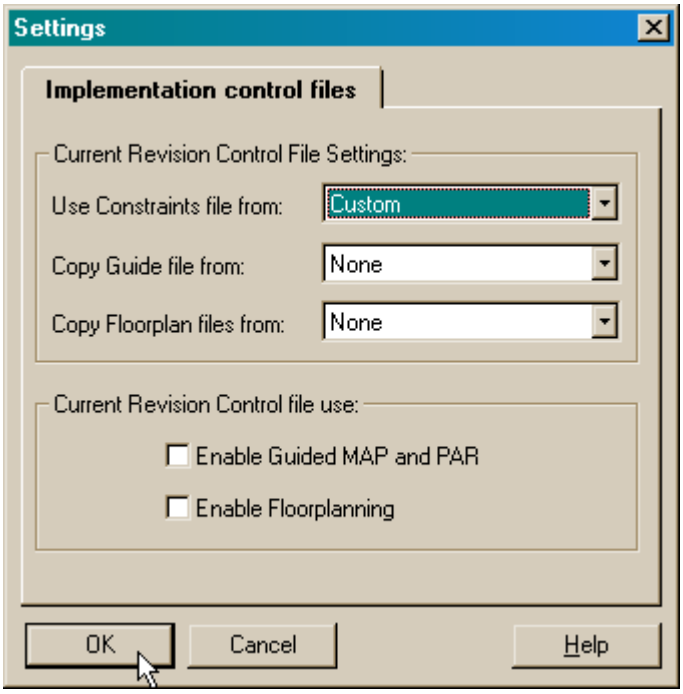
But we do have a user constraints file, so click on the Use Constraints file from: field and select Custom.



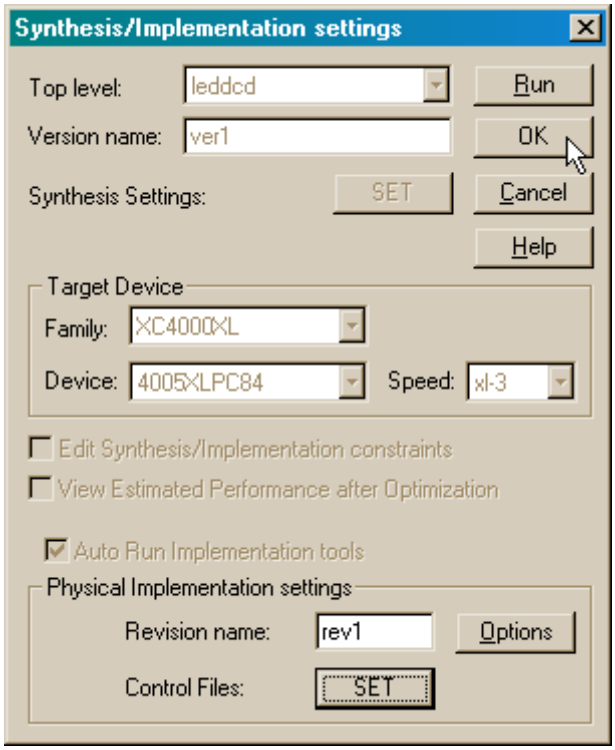
The **Custom** window will display the dsgn1_1.ucf file by default, so all we need to do is click on OK to enable the use of this control file.



Now click OK to finalize the settings for the implementation tools.



Then click on Run to start the implementation tools.



The **Flow Engine** window appears that depicts the progress through the five phases of the FPGA implementation process. Running appears below the currently active phase while Completed appears for each successfully completed step. Status and error messages generated during each implementation step scroll through the lower section of the window. The purpose of each phase is as follows:

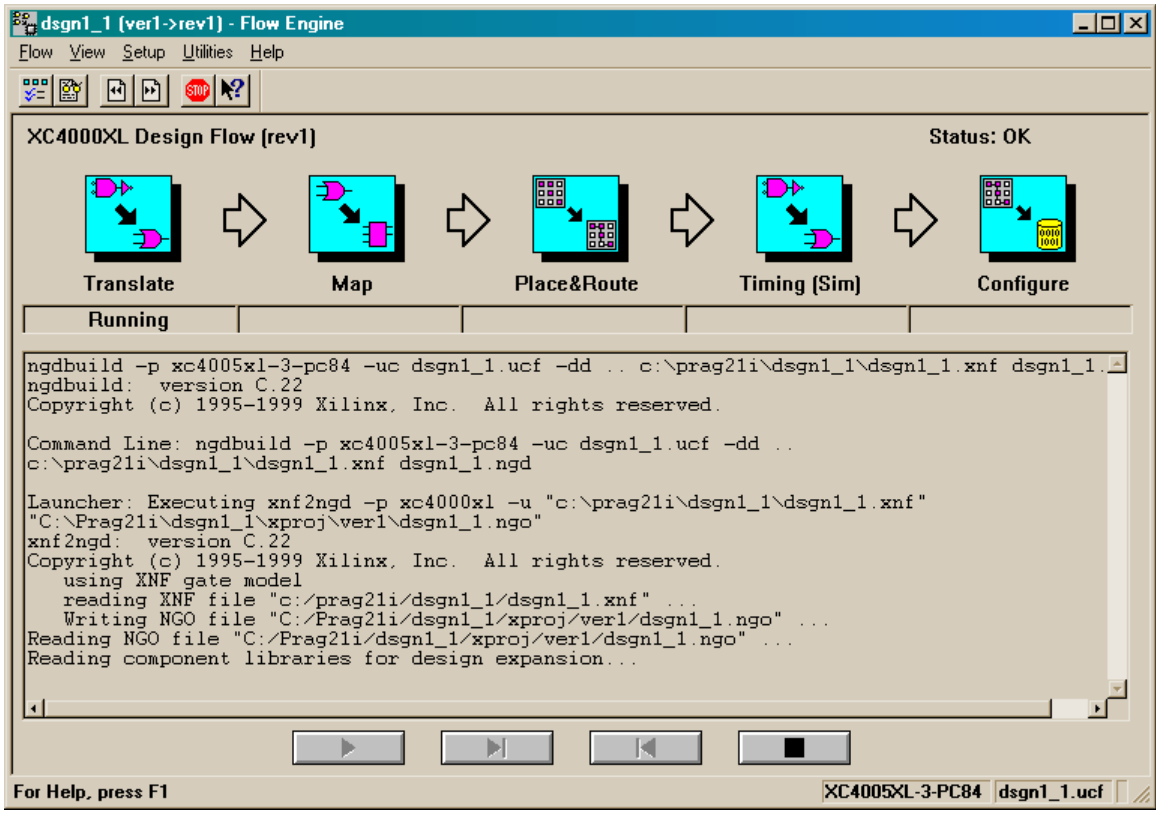
Translate: The synthesized netlist format and the user constraints are converted into an internal database format.

Map: The logic gates in the netlist are grouped to take advantage of the resources in the FPGA's CLBs.

Place&Route: The mapped logic gates are placed in specific locations within the FPGA's array of CLBs and the connections between the gates are routed through the FPGA's wiring resources.

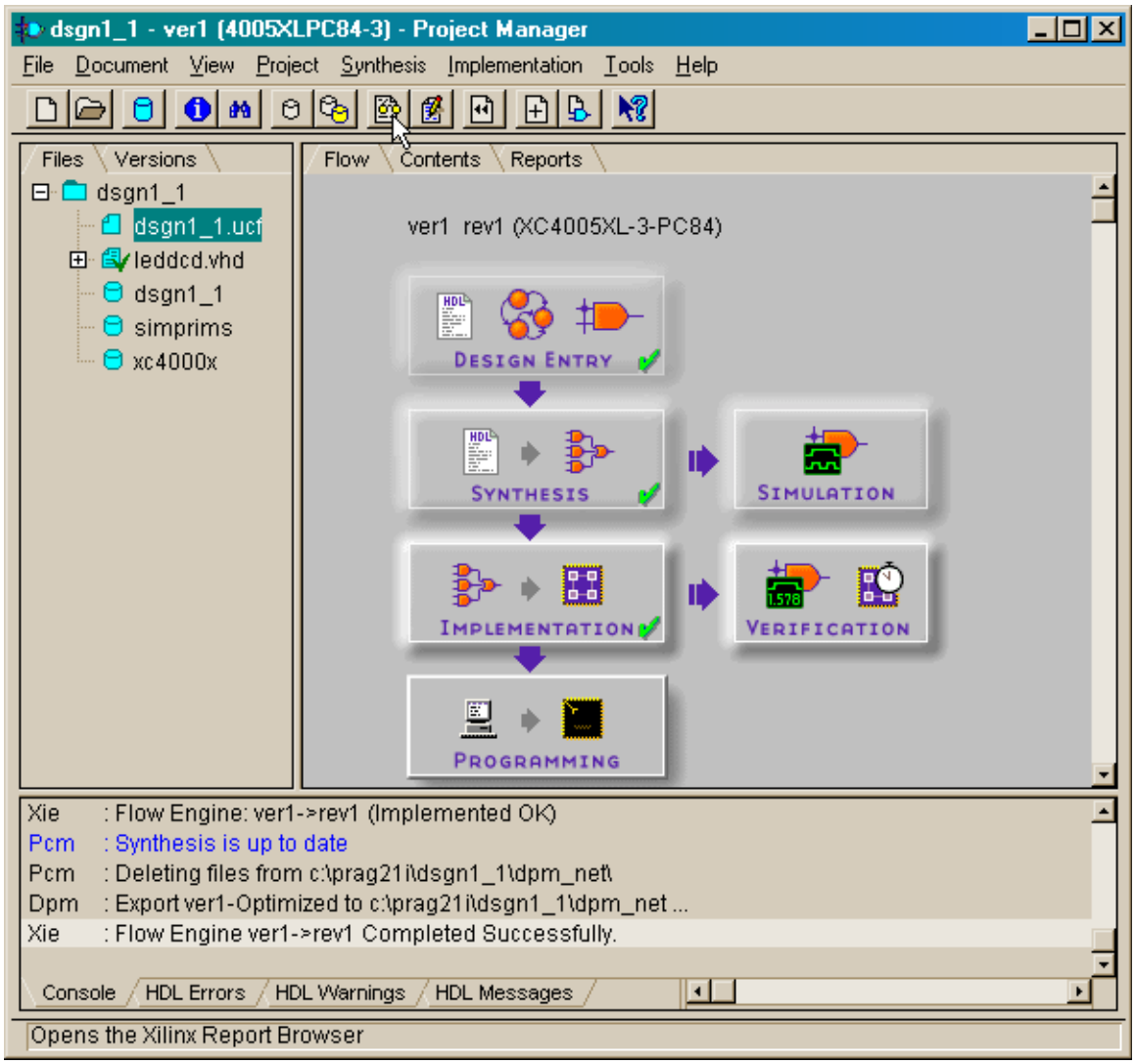
Timing (Sim): The propagation delays through the CLBs and routing are computed and stored for use during a timing simulation. (This phase is optional and can be turned off if you don't want to do timing simulations.)

Configure: The bitstream that will configure an FPGA with the placed-and-routed circuit is generated.

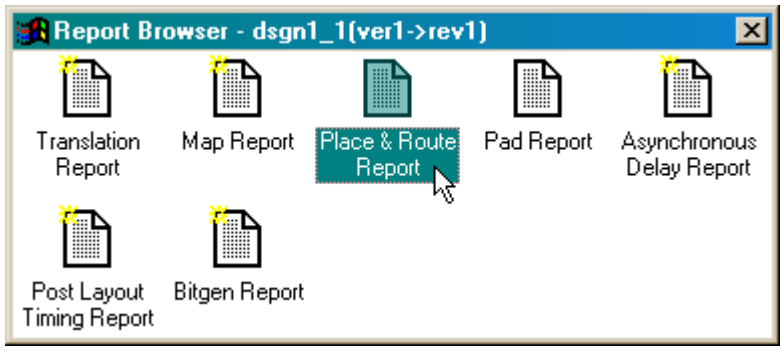


Once all five phases of the implementation process have completed, click on OK in the status pop-up window.

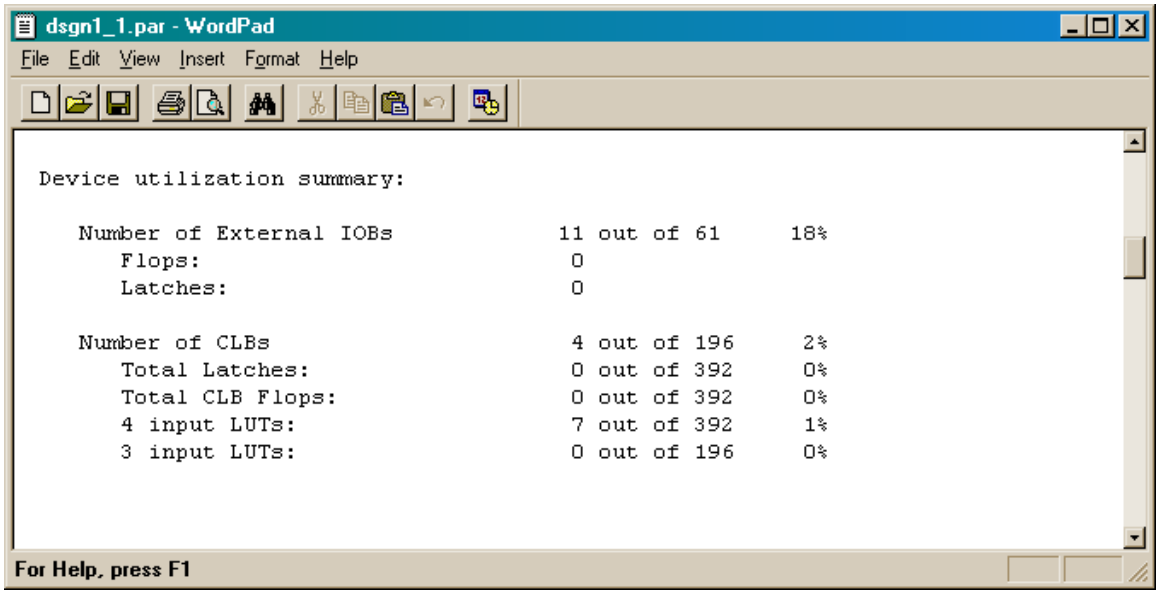
A successful implementation is indicated by the green checkmark in the Implementation box in the **Design Flow** pane. We can check how much of the FPGA is used by the LED decoder circuit by selecting the Implementation→View Report Files... menu item or clicking on the report browser toolbar button as shown below.



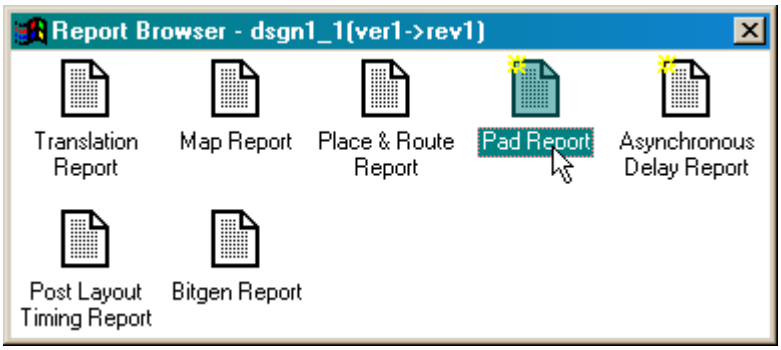
The **Report Browser** window lists the reports generated by each phase of the implementation process. We can double-click the Place & Route Report icon to get a view of the statistics on the usage of FPGA resources.



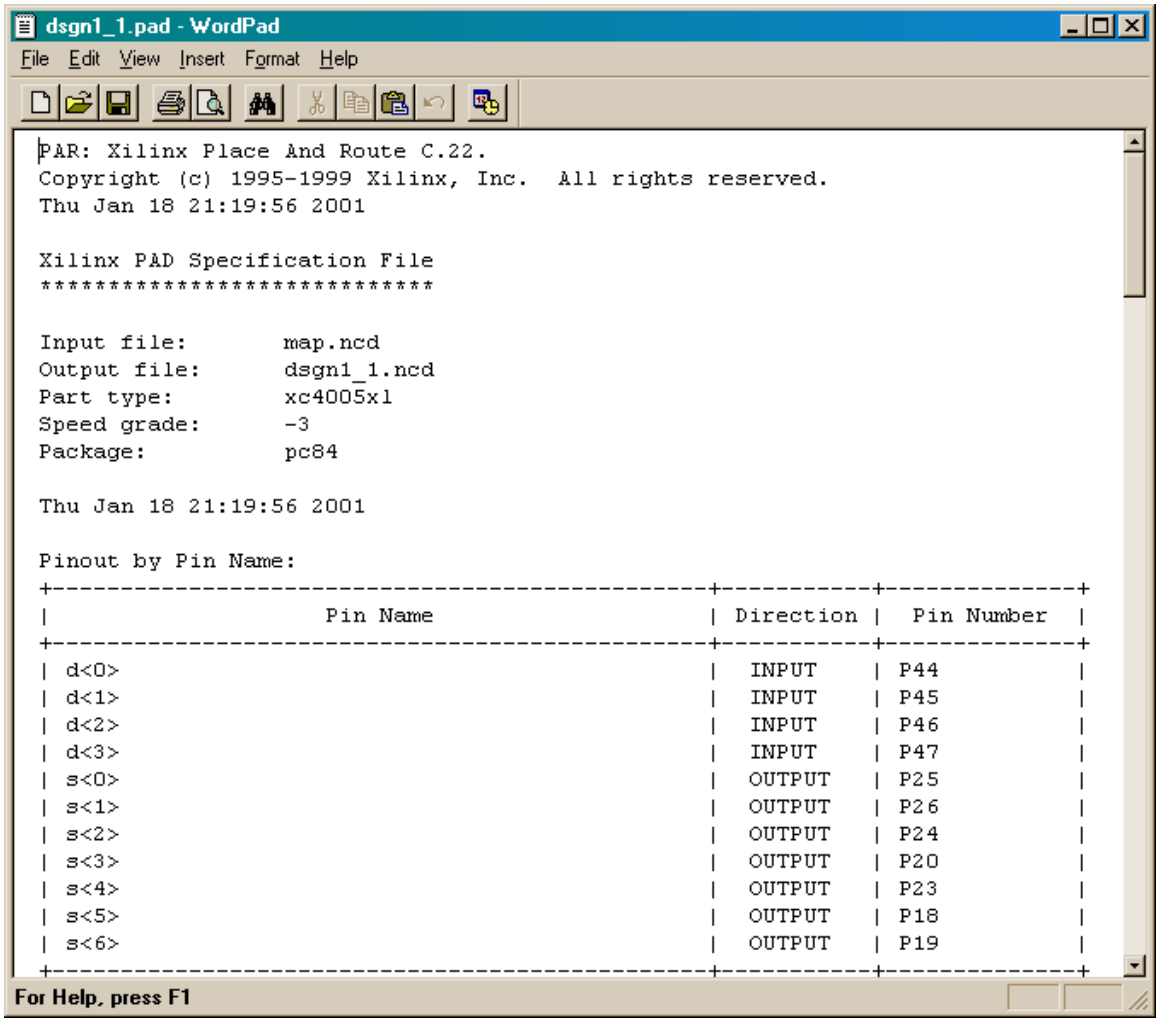
Scrolling downward through the place-and-route report brings us to the device utilization summary. The LED decoder has four inputs and seven outputs, so it uses eleven of the 61 input/output blocks (IOBs) of the XC4005XL FPGA. The design is completely combinational so its uses none of the FPGA's flip-flops and latches. It does use four of the 392 CLBs in the FPGA. Each CLB contains two four-input lookup tables (LUTs) and one three-input LUT that perform all the combinational logic duties. The LED decoder requires seven LUTs to generate its seven outputs, and these seven LUTs are packed into four CLBs. Overall, we see the LED decoder circuit doesn't use much of the FPGA at all.



We should also check to see if our pin assignment constraints were obeyed. Double-click the Pad Report icon to inspect the actual pin assignments.

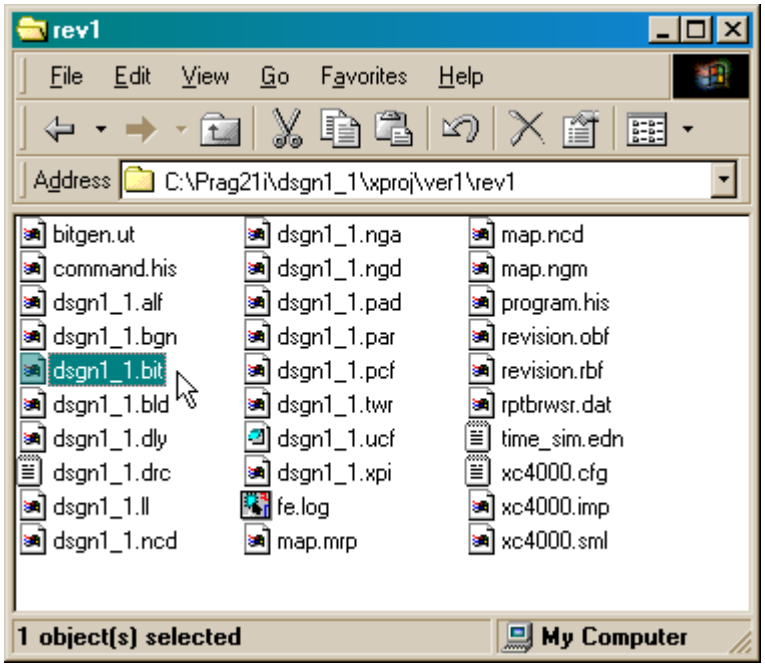


The input and output names are listed in the pad report file along with the pin number to which they were assigned during the implementation phase. We can see that the actual pin assignments match the assignments we placed in the dsgn1_1.ucf user constraints file.



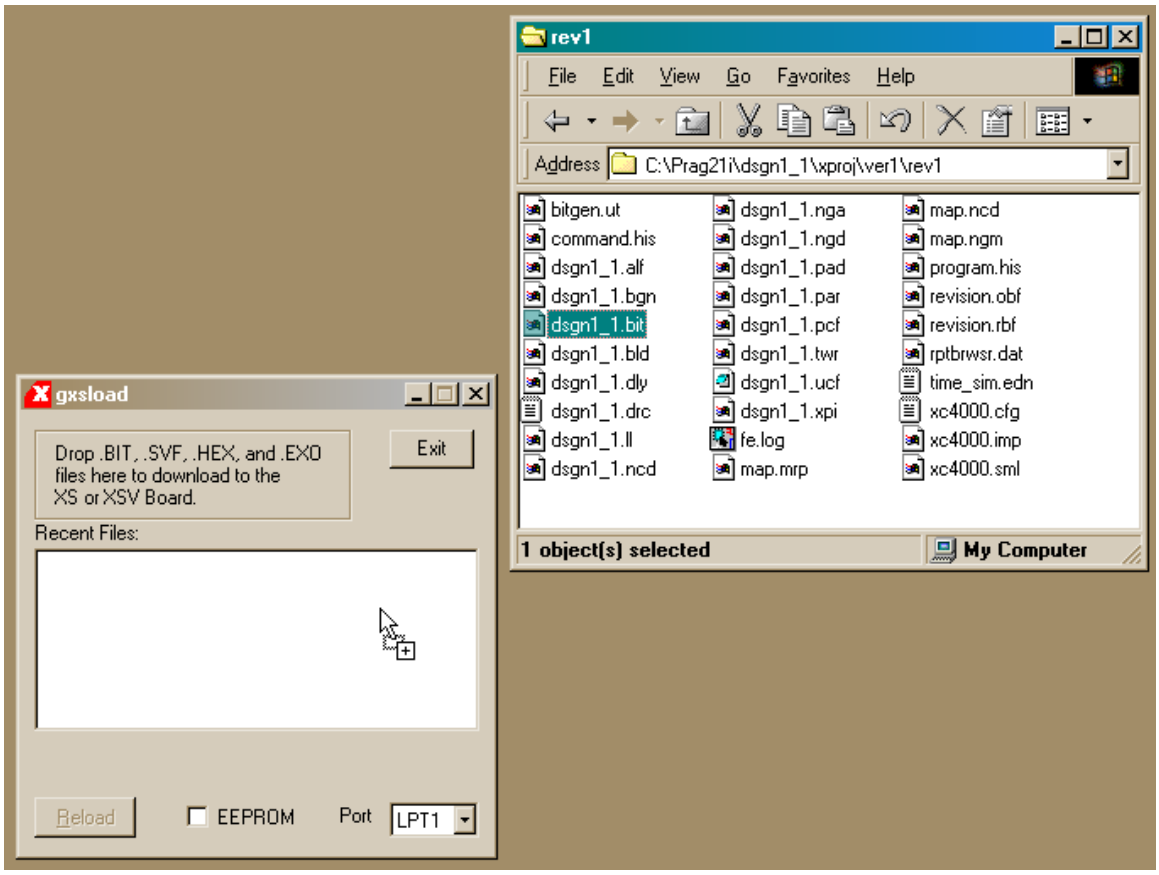
Downloading and Testing

We have successfully synthesized, simulated, and implemented the LED decoder, so now we can test it in an actual FPGA. First we should locate the actual LED decoder bitstream. It can be found in the C:\Prag21\dsgn1_1\xproj\ver1\rev1 folder in the dsgn1_1.bit file.

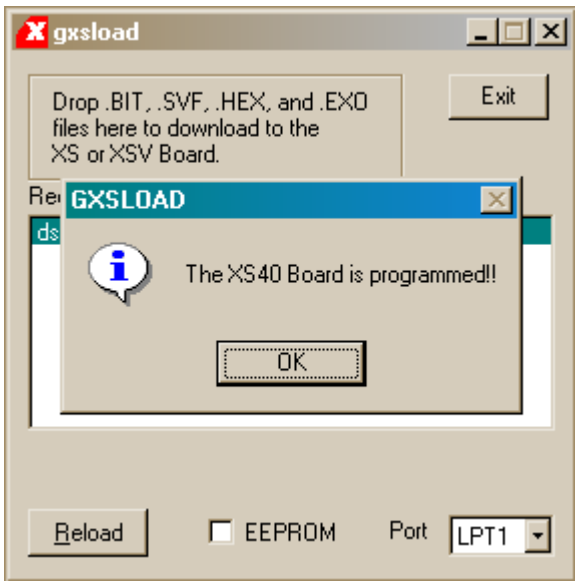


Now apply power to the XS40 Board, connect it to the parallel port of your PC, and double-click the GXSLOAD icon to start the process of loading the LED decoder bitstream into the XC4005XL FPGA.

Then just drag-and-drop the dsgn1_1.bit file from the **rev1** window into the **gxsload** window.

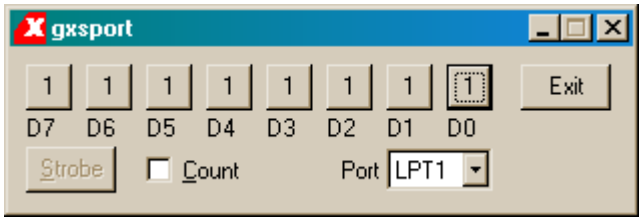


Once the bitstream file is dropped in the **gxsload** window, a progress bar will appear to show the percentage of the bitstream transmitted to the FPGA. The download should take less than ten seconds, after which a pop-up window will appear to indicate the XS40 FPGA Board is programmed. Click on OK to remove the pop-up window.

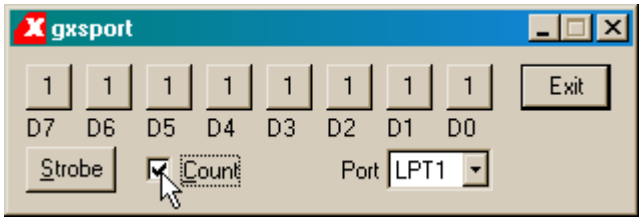


Now we need to drive the inputs of the LED decoder and observe its response on the LED digit of the XS40 Board. Double-click the GXSPORT icon to start up the utility that can drive the FPGA inputs through the PC parallel port.

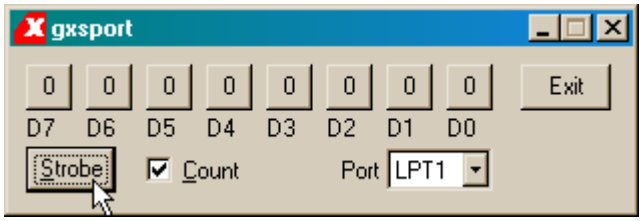
The **gxsport** window has eight buttons that correspond to the eight parallel port data pins. The current level on each data pin is reflected in the binary digit displayed within each button. Clicking a data button toggles the level, *but the new output is not driven onto the parallel port pin until the Strobe button is pressed*. We can drive test vectors into the LED decoder circuit by pressing buttons D0, D1, D2, and D3 to set the desired pattern and then clicking Strobe.



But there is a faster way than typing in all sixteen input vectors. Just check the Count box to make the eight-bit value represented by D7…D0 increment every time Strobe is pressed.

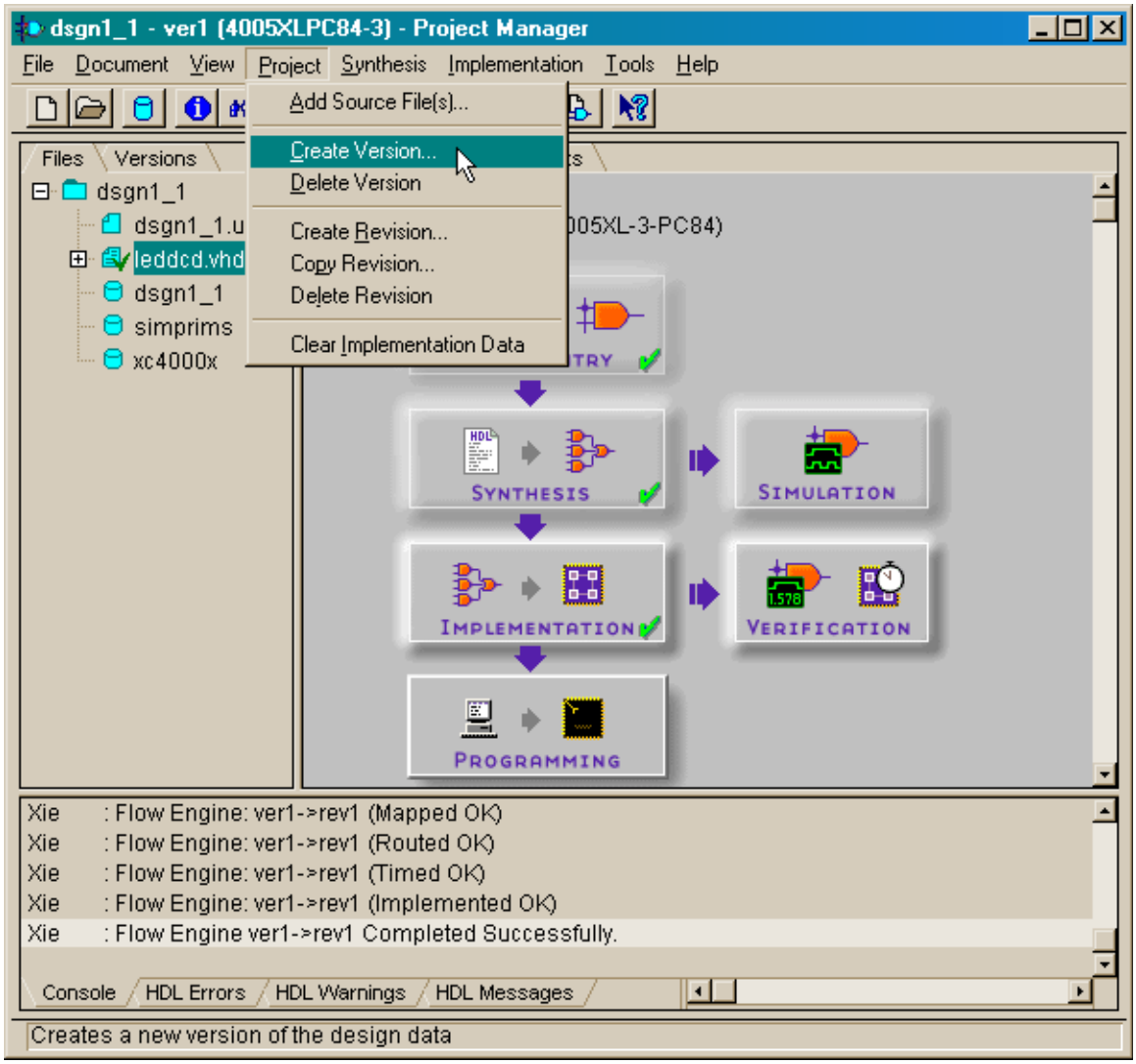


Then just click Strobe sixteen times. You should see the LED display the next hexadecimal digit after each click.

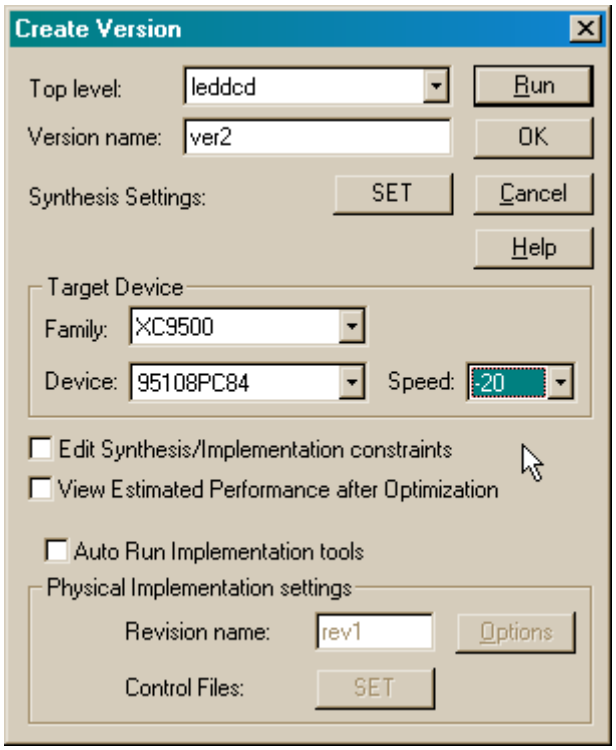


Retargeting the Circuit at an XC9500 CPLD

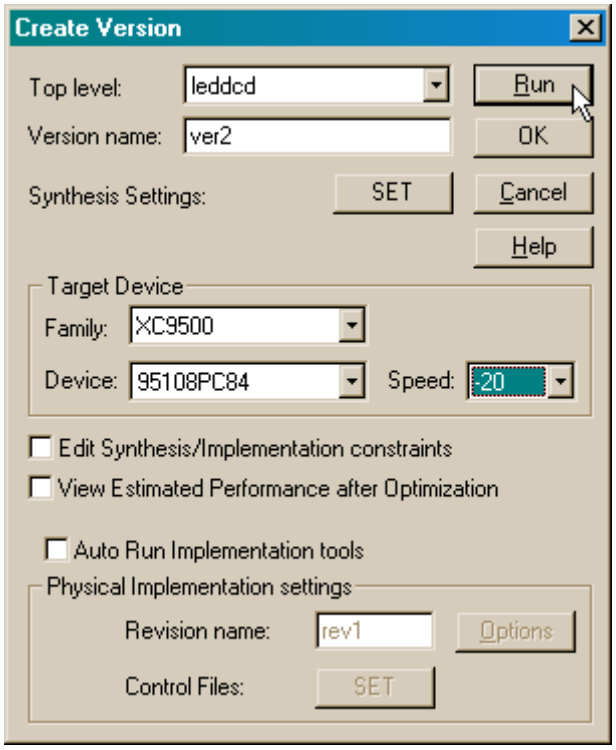
At this point we have synthesized, simulated, implemented, downloaded, and tested the LED decoder on an XC4005XL FPGA. Now we will retarget the LED decoder at an XC95108 CPLD device. We will create a new version of the project to hold the modified design. To start, select the Project→Create Version... menu item.



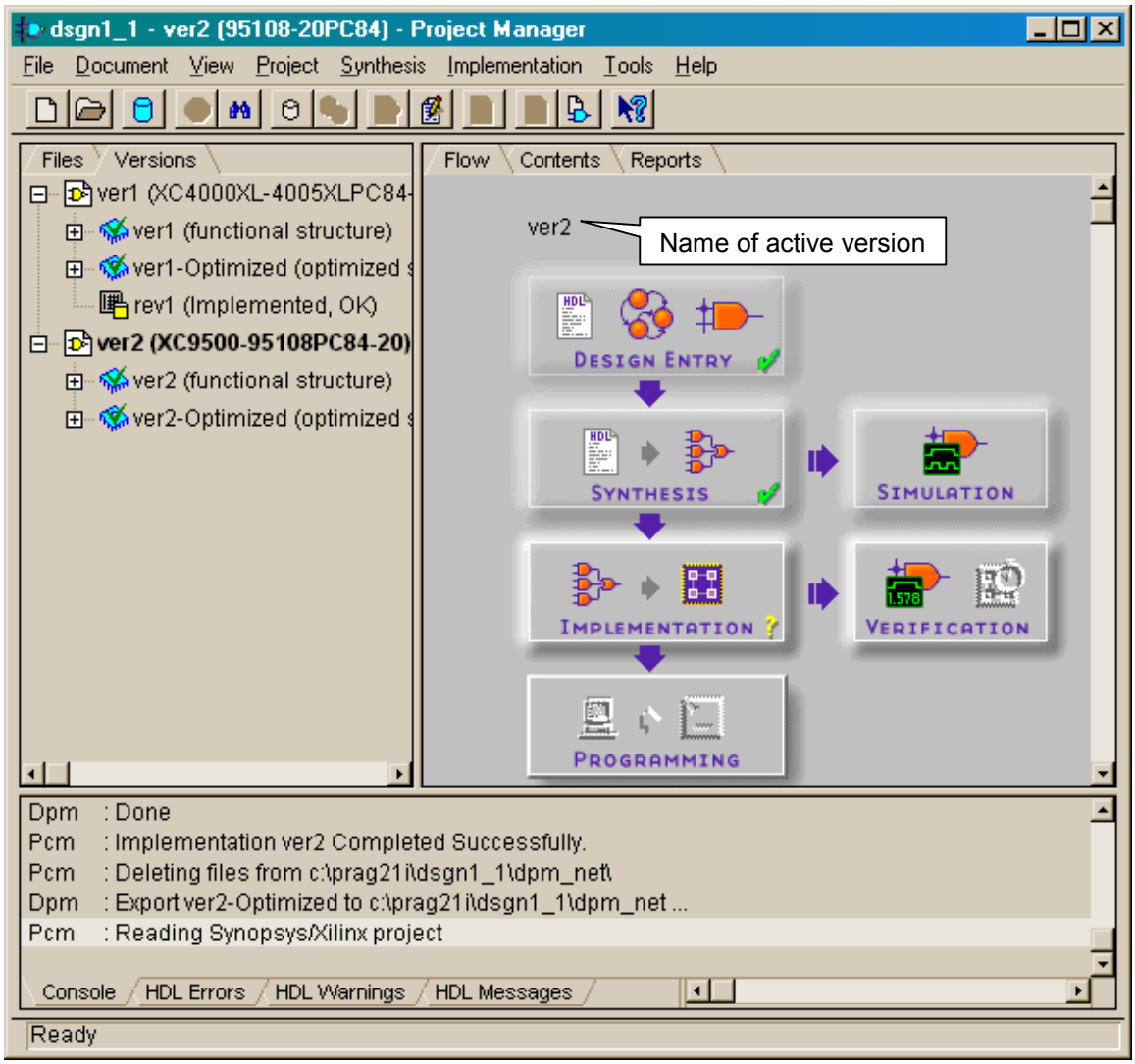
Note that the Version name field has automatically been set to ver2 In the **Create Version** window that appears. Set the fields in the Target Device area to the values needed for the XS95 Board as shown below.



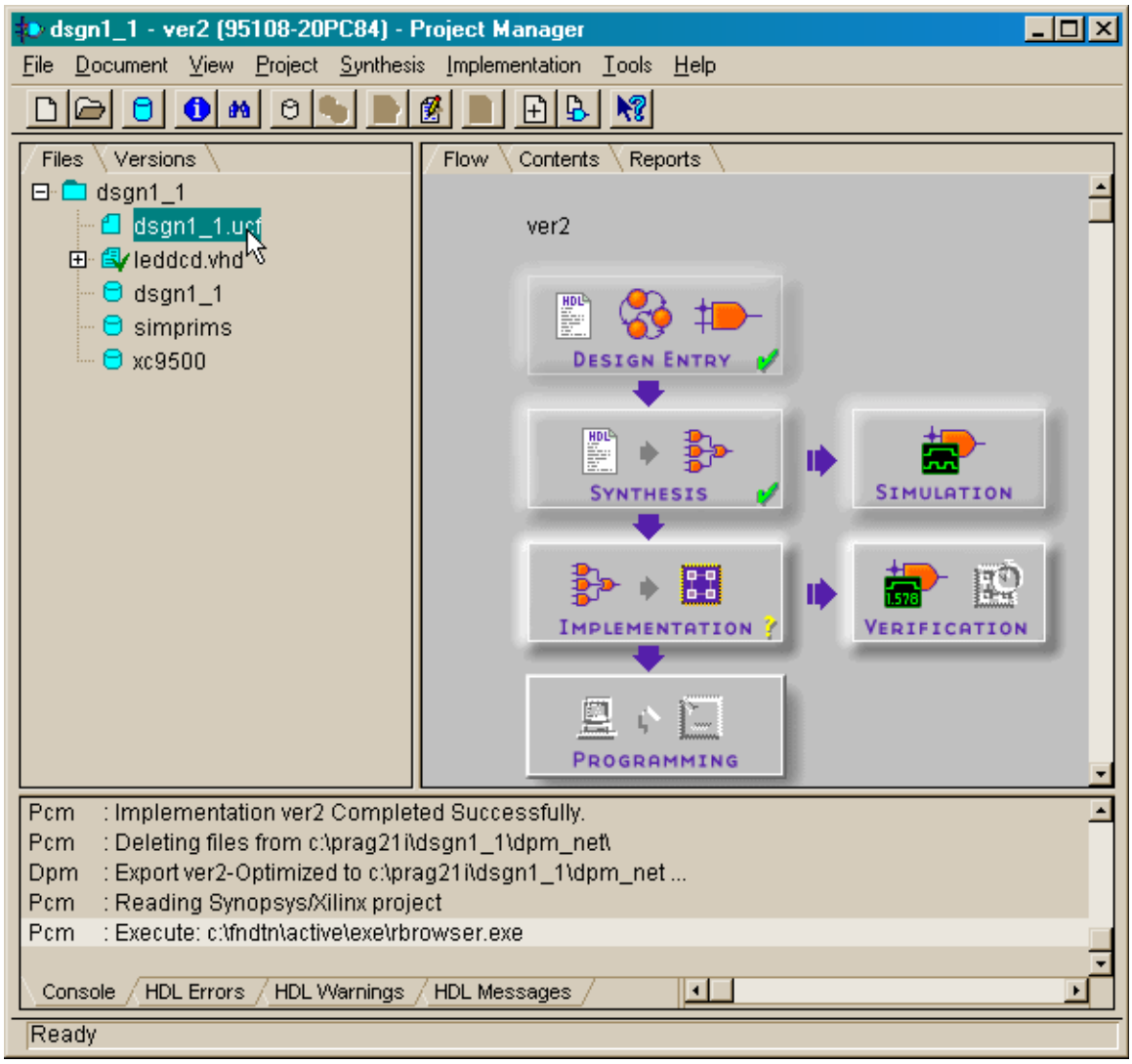
Next, click on the Run button to activate the synthesizer. The synthesizer will create a new netlist the takes into account the specific features of the XC9500 CPLD architecture.



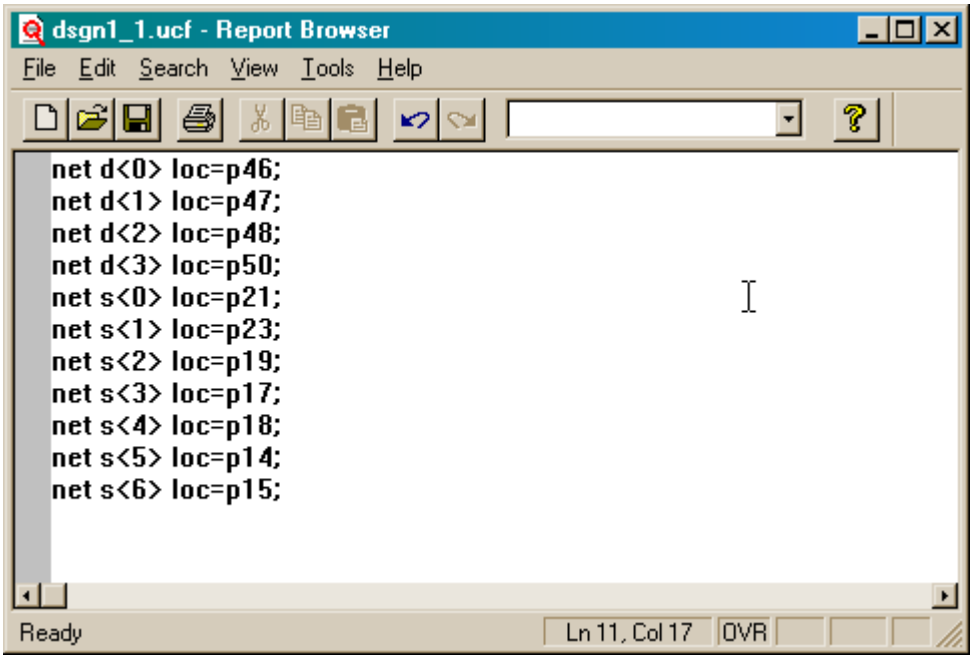
After the netlist synthesis is complete, the **Project Manager** window shows a green checkmark in the Synthesis block of the **Design Flow** pane. The version label in the **Design Flow** pane has also been changed to ver2. We can view more details about the different versions of the LED decoder project by clicking on the Versions tab in the **Project Hierarchy** pane. We see that ver1 is targeted to an XC4005XL FPGA and has been successfully synthesized (both functional and optimized netlists are present) and implemented. The ver2 version, however, is targeted at an XC95108 CPLD and has only been synthesized to this point. No implementation phase has been performed yet. (That accounts for the yellow question mark in the Implementation block of the **Design Flow** pane.) We can activate a version by clicking on its name in the Versions tab. The ver2 version is currently active, so we will leave it that way.



As compared to the XS40 Board, the CPLD on the XS95 Board has different pins connected to the parallel port and LED digit. Therefore, we need to change the pin assignment constraints in the dsgn1_1.ucf file. Click on the Files tab in the **Project Hierarchy** pane and then double-click the dsgn1_1.ucf file name to edit the user constraints file.



Edit the pin assignments for the XS95 Board as follows. Then save the file and exit.



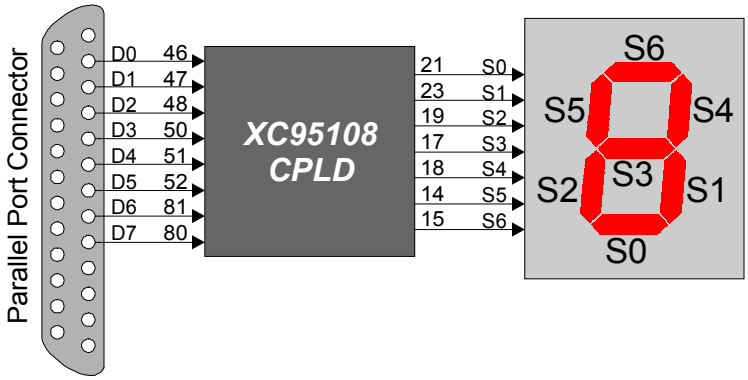
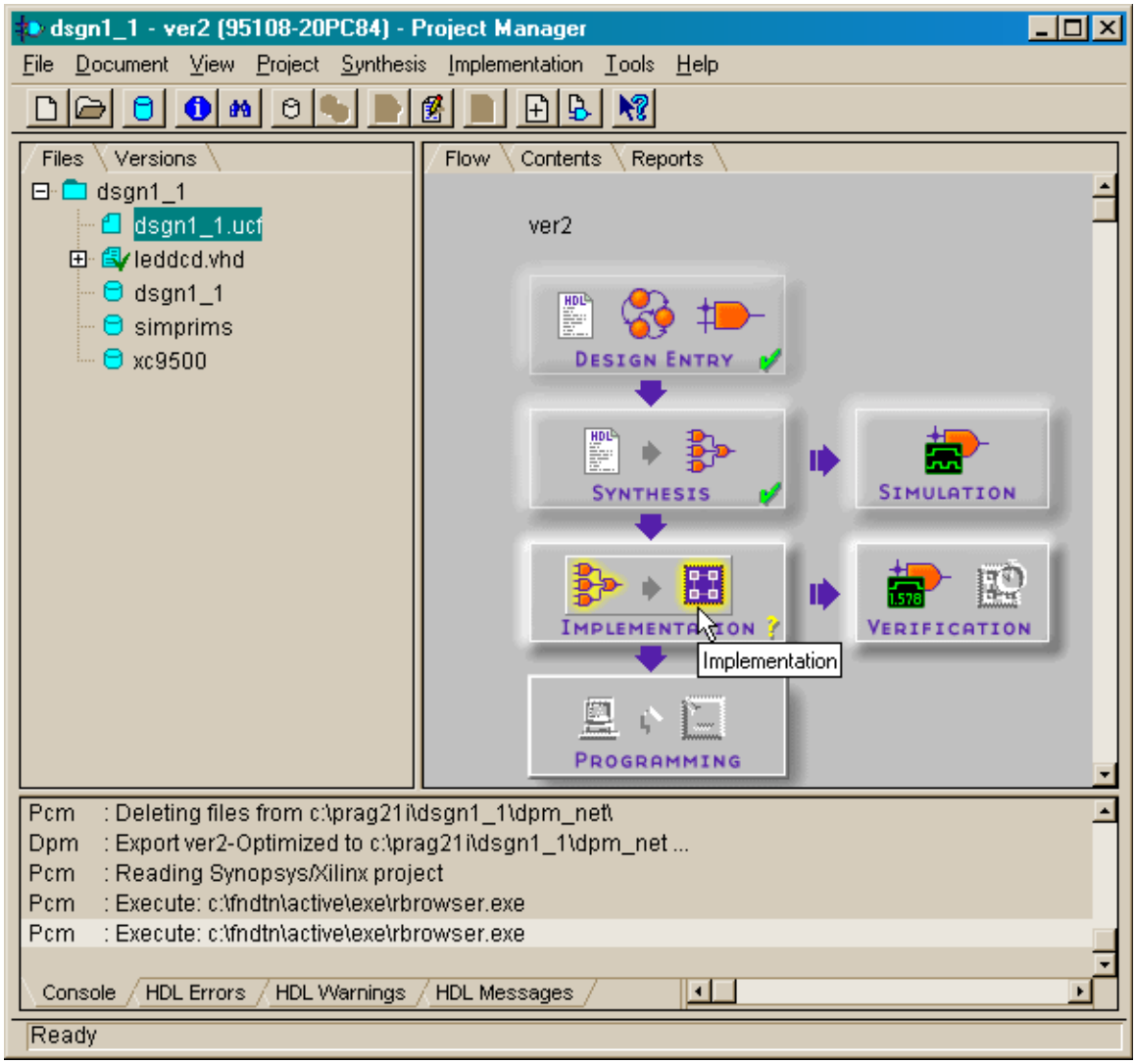
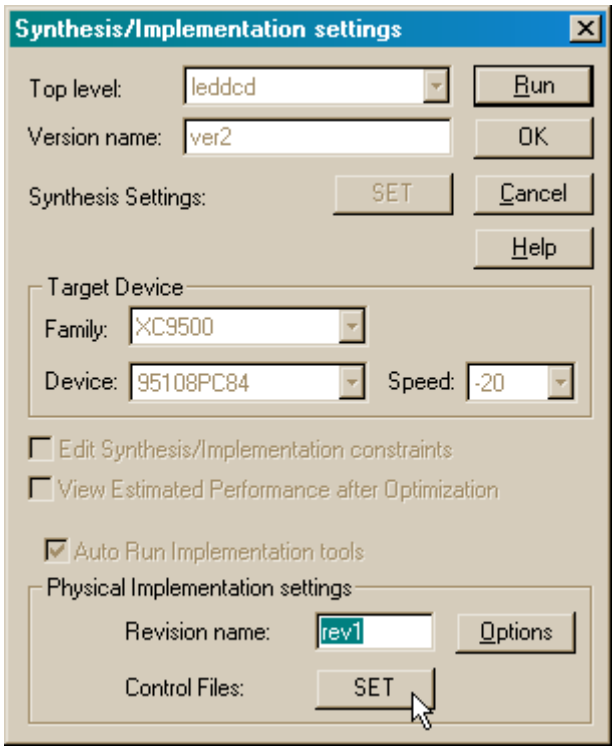


Figure 3: Connection of the parallel port and LED digit to the pins of the CPLD on the XS95 Board.

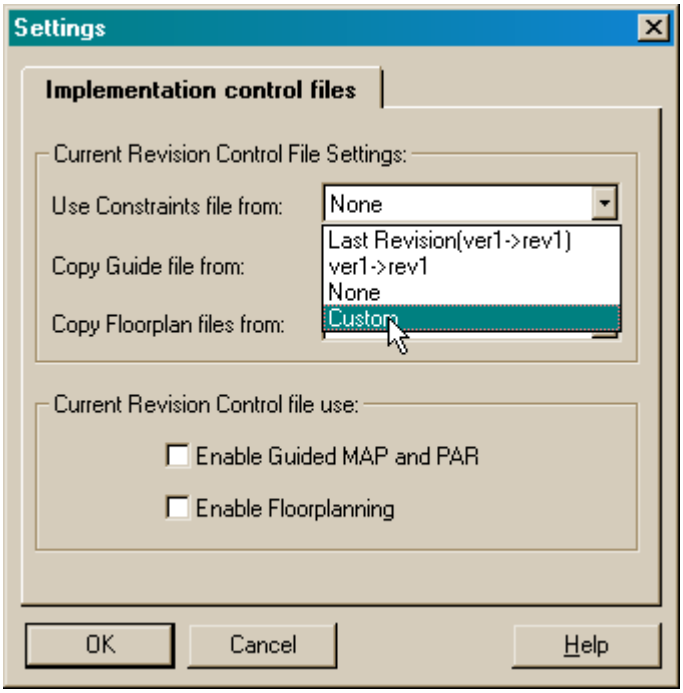
Now start the implementation tools that will map the netlist to the XC95108 CPLD.



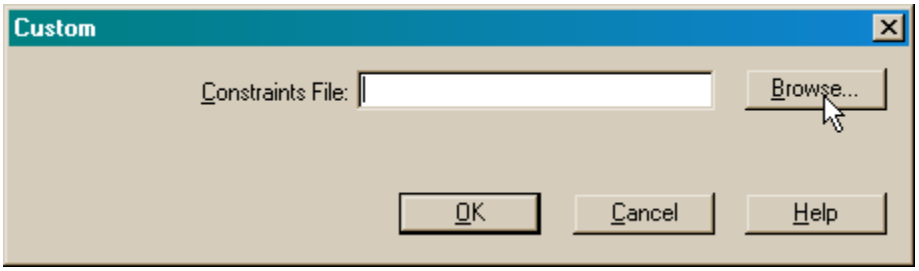
The **Synthesis/Implementation setting** window will appear. Press the SET button so we can tell the implementation tools where to find the updated user constraint file.



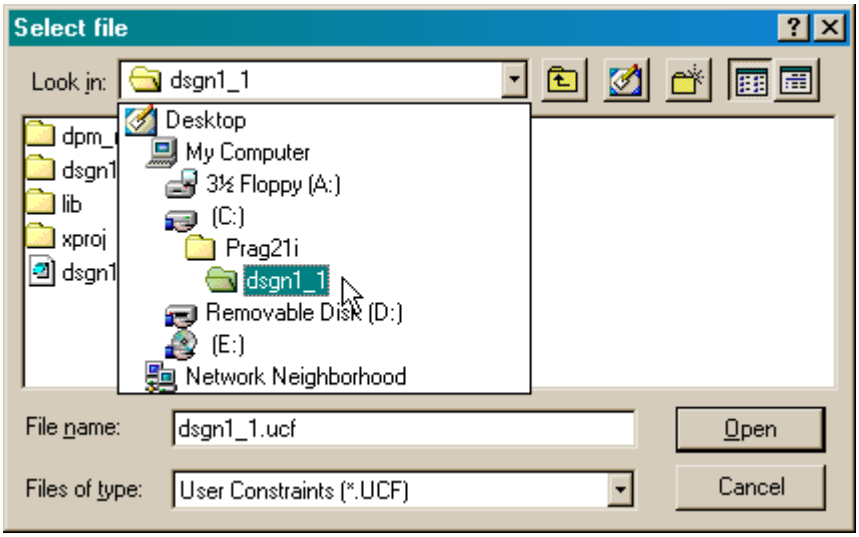
In the **Settings** window, the drop-down list for the Use Constraints file from: field contains a few more entries than it did last time. The first two entries refer to constraint files for the previous version targeted at the XC4005XL FPGA, so they are not suitable for this version. Select the Custom entry.



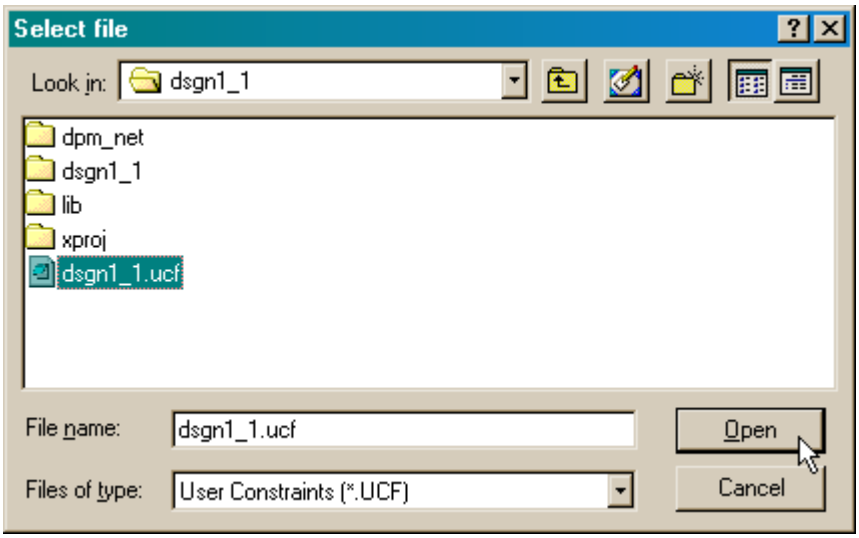
Click on the Browse button in the **Custom** window that appears.



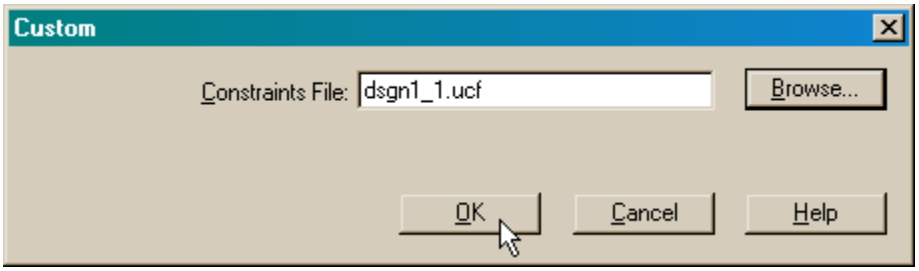
Then move to the top level of the project where the modified user constraints file is found.



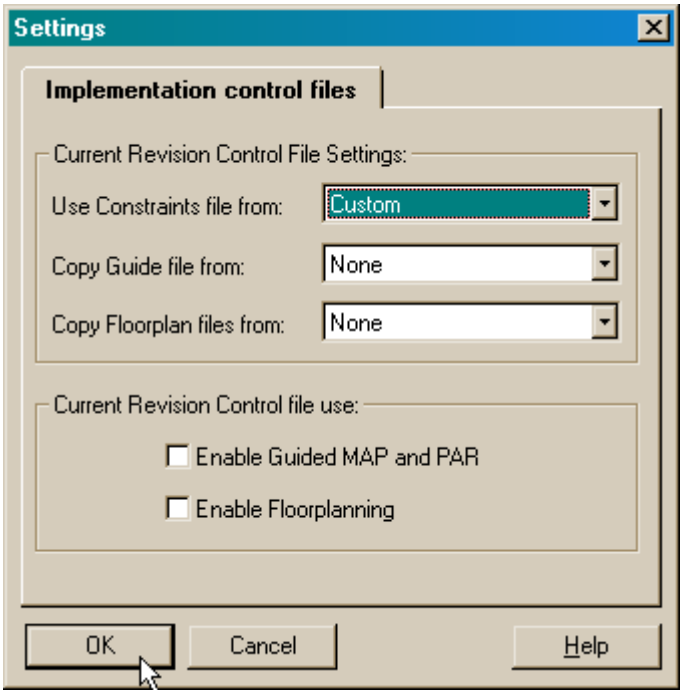
Highlight the user constraints file and click on Open.



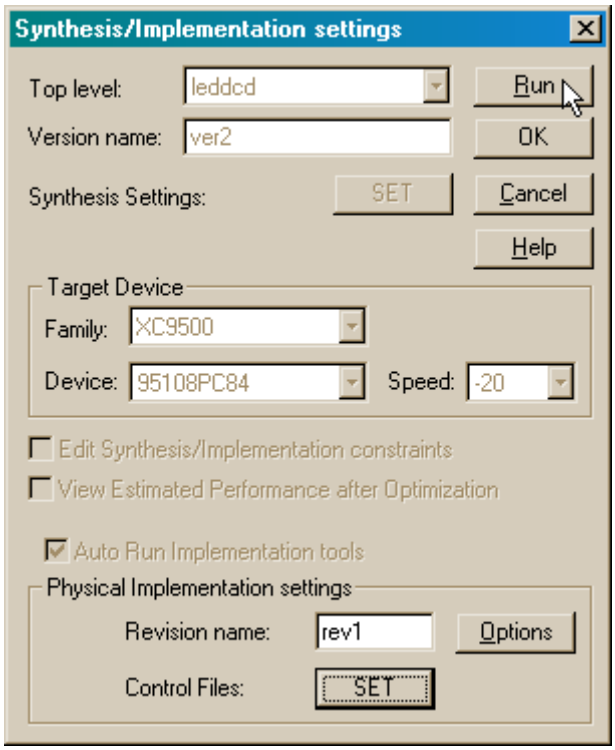
The **Custom** window now reflects the name of the user constraints file. Click on OK.



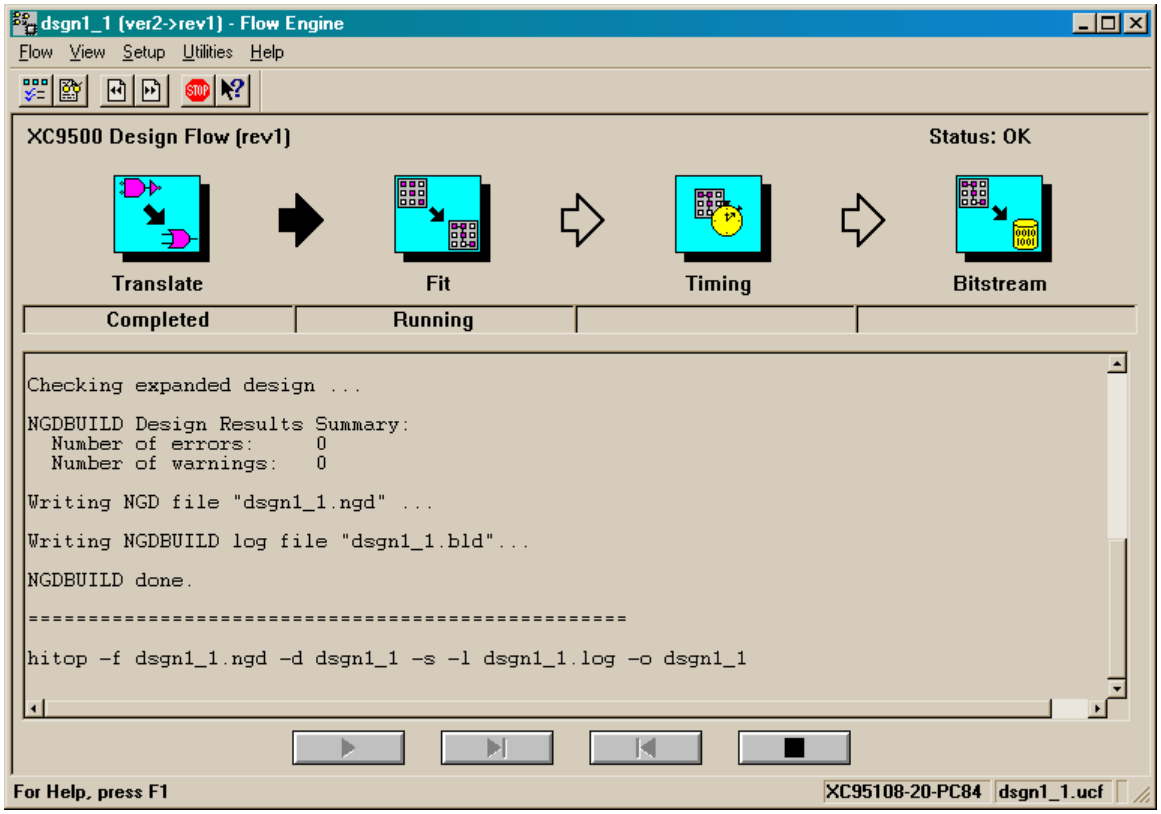
Then click on OK in the **Settings** window.



Now that the user constraints file for the XS95 Board has been specified, click on Run to start the implementation tools.

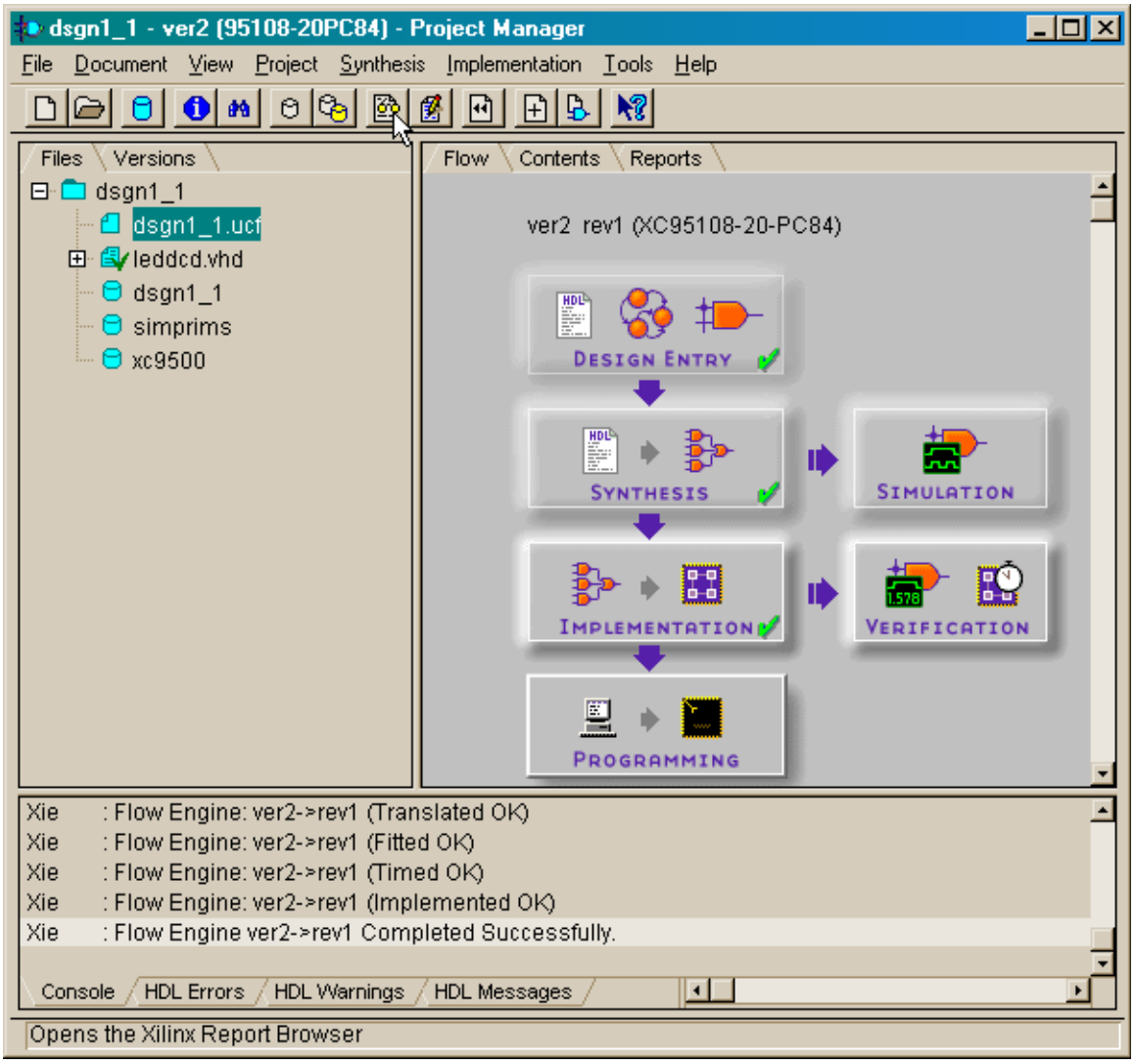


The **Flow Engine** window appears, but only four steps are required for a CPLD implementation versus the five steps used in an FPGA implementation. (The Fit step does the same things for a CPLD that the Map and Place&Route steps do for an FPGA.)

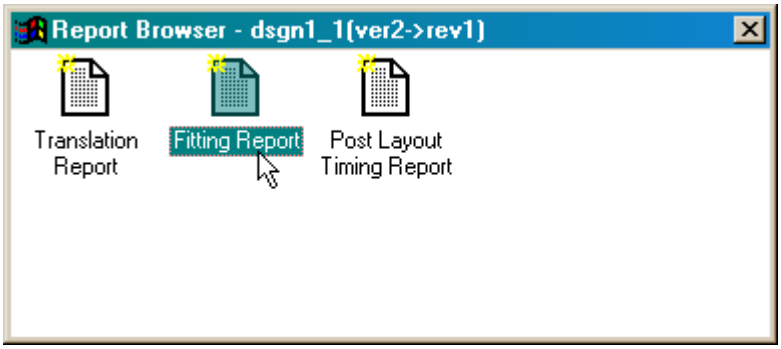


The implementation phase should complete without incident.

Now that the implementation is completed (note the green checkmark in the Implementation box), we can check the CPLD utilization and pin assignments with the report browser.

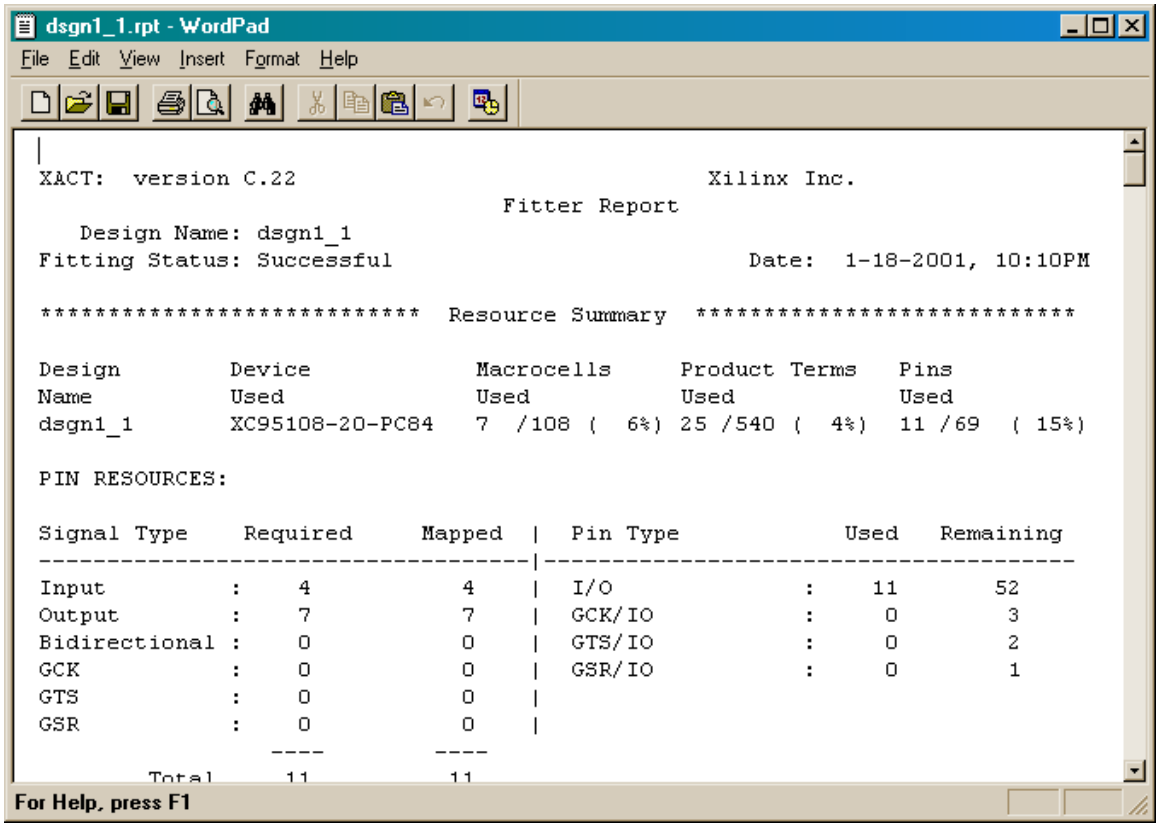


For CPLDs, the device utilization summary and the pin assignments are both recorded in the Fitting Report. Double-click the icon to view this file.

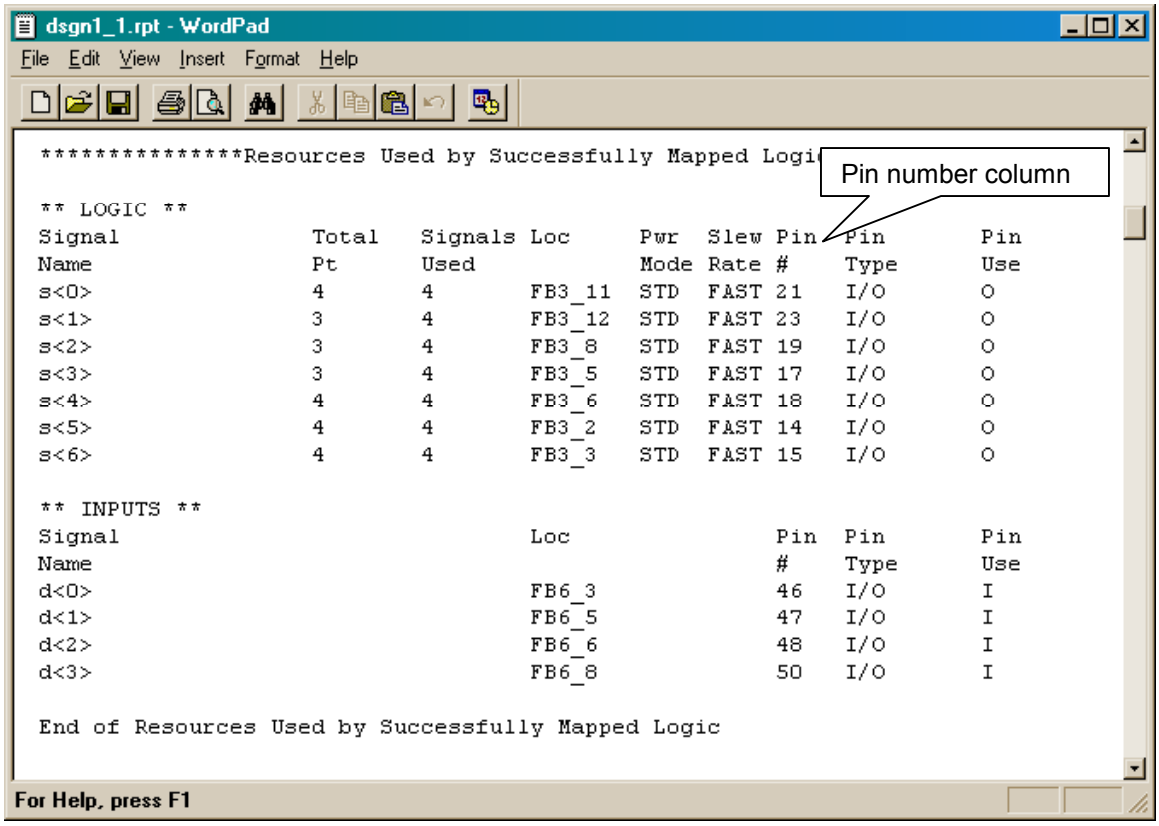


The summary of the CPLD resources that are used by the LED decoder is placed at the top of the file. The circuit uses eleven I/O pins of the 69 present on the CPLD. Seven

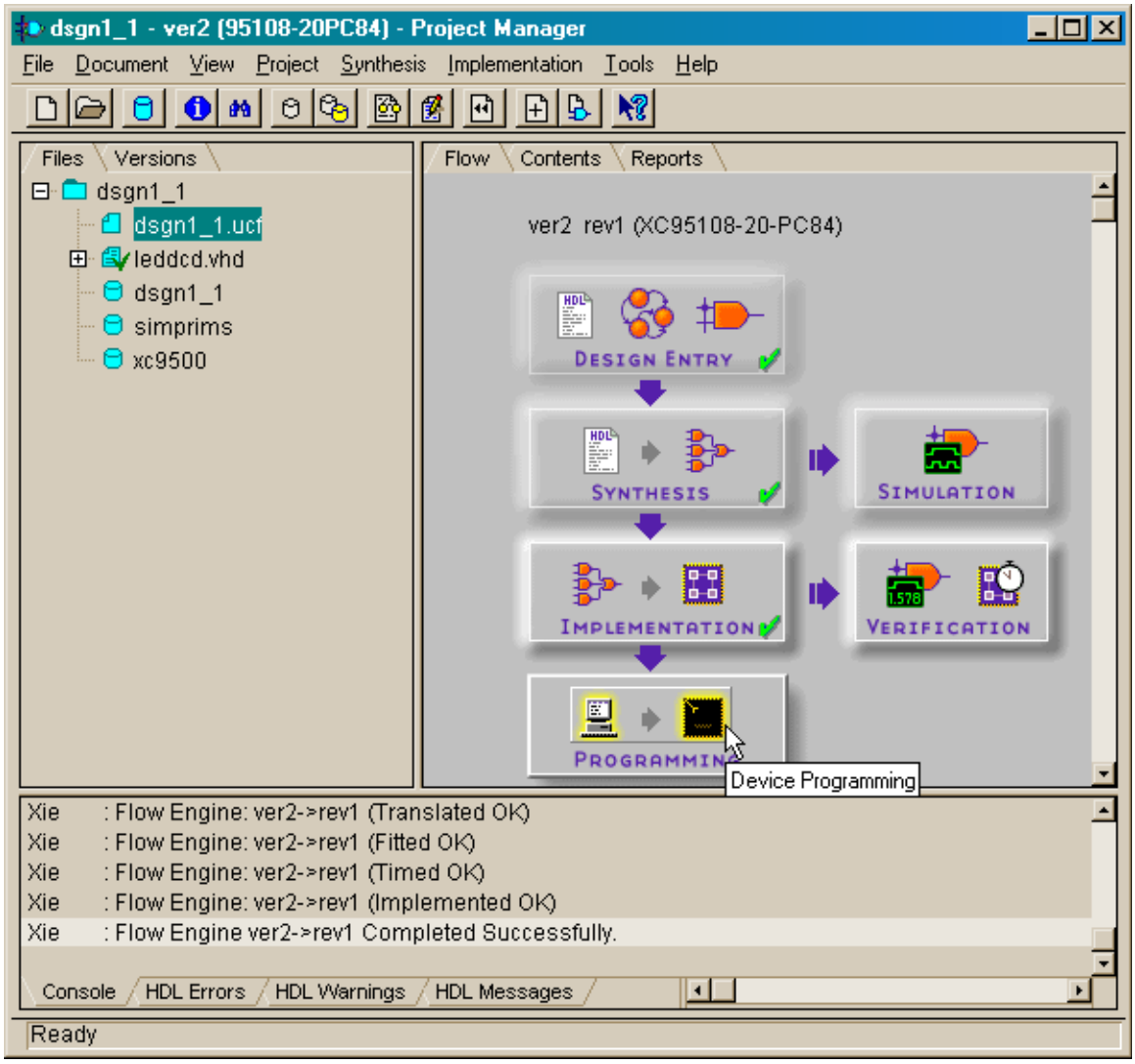
macrocells are used to hold the combinational logic for the LED decoder's outputs. As with the FPGA, the LED decoder does not consume much of the CPLD resources.



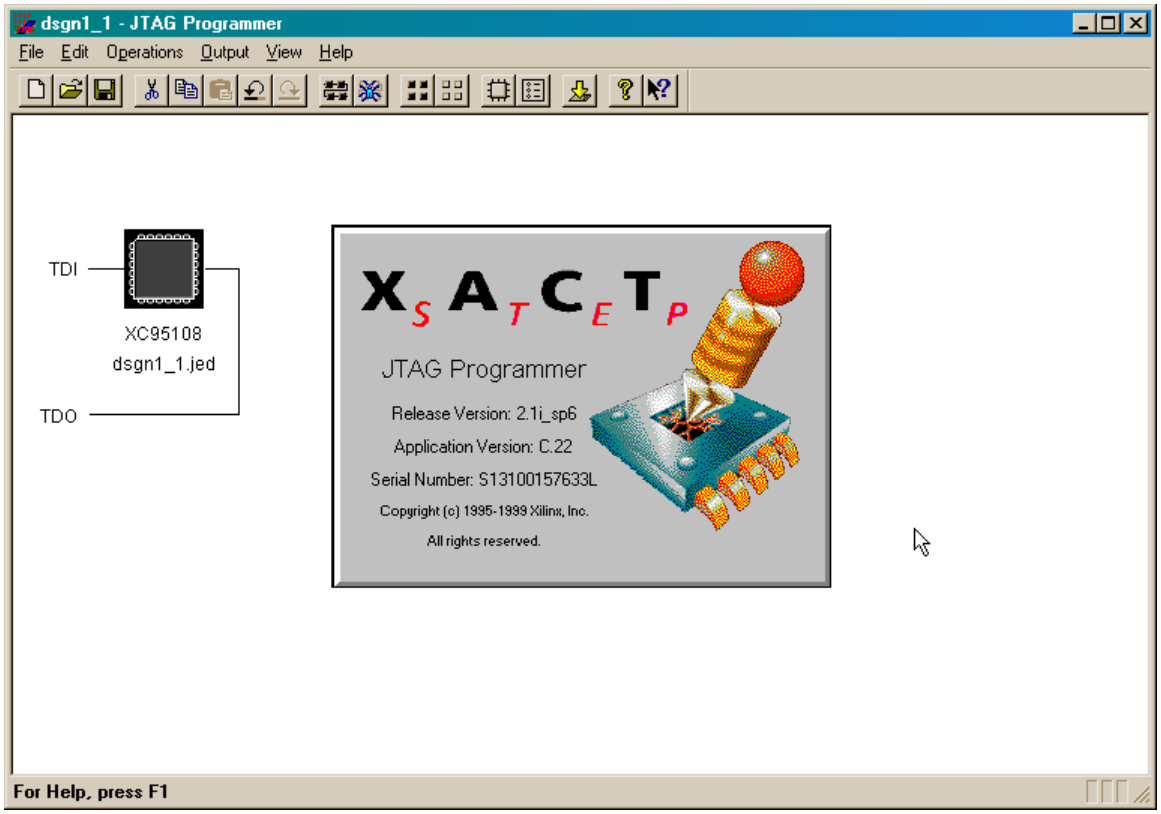
The pin assignments are found further down in the file. As expected, the implementation tools assigned the inputs and outputs to the same pins that we specified in the user constraints file.



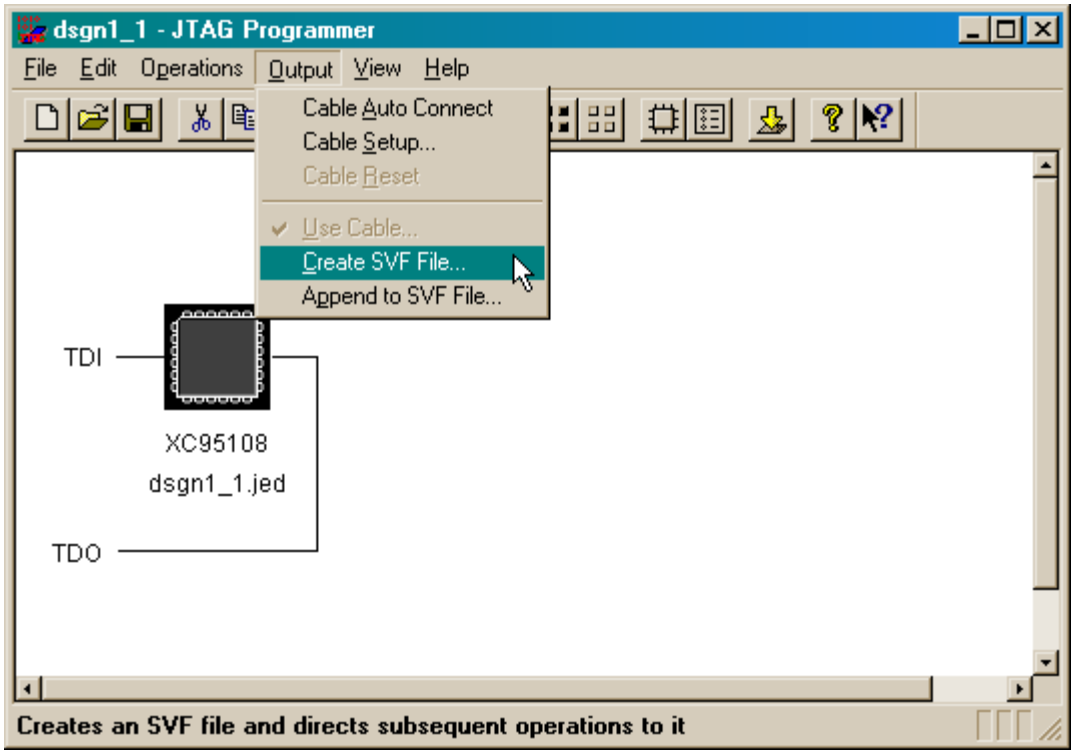
We need to do a little more processing on the bitstream output by the implementation tools before we can download them to an XC9500 CPLD. Click on the Programming box in the **Design Flow** pane to begin this step.



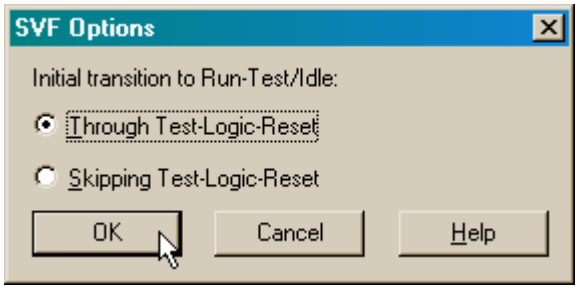
The **JTAG Programmer** window will appear displaying a single XC95108 device.



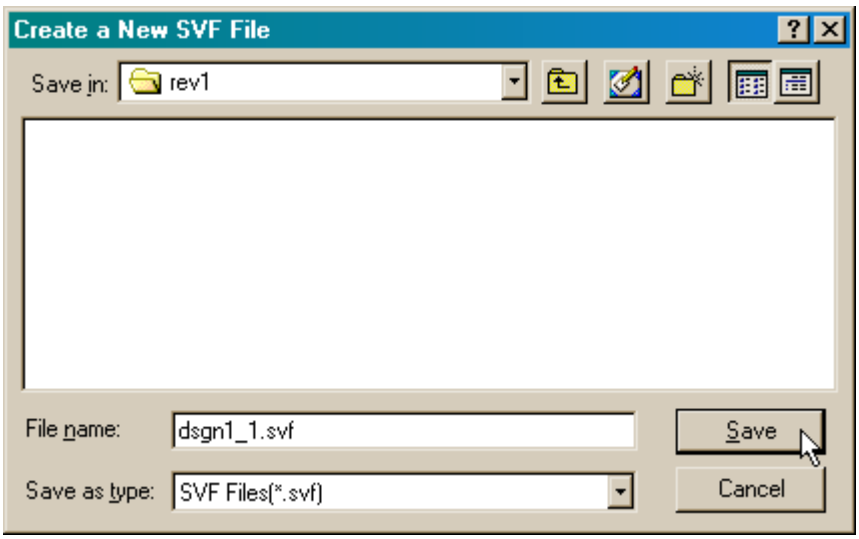
We need to translate the bitstream into the SVF format that is compatible with the GXSLOAD utility. Select Output→Create SVF File... to store the translated bitstream into a file.



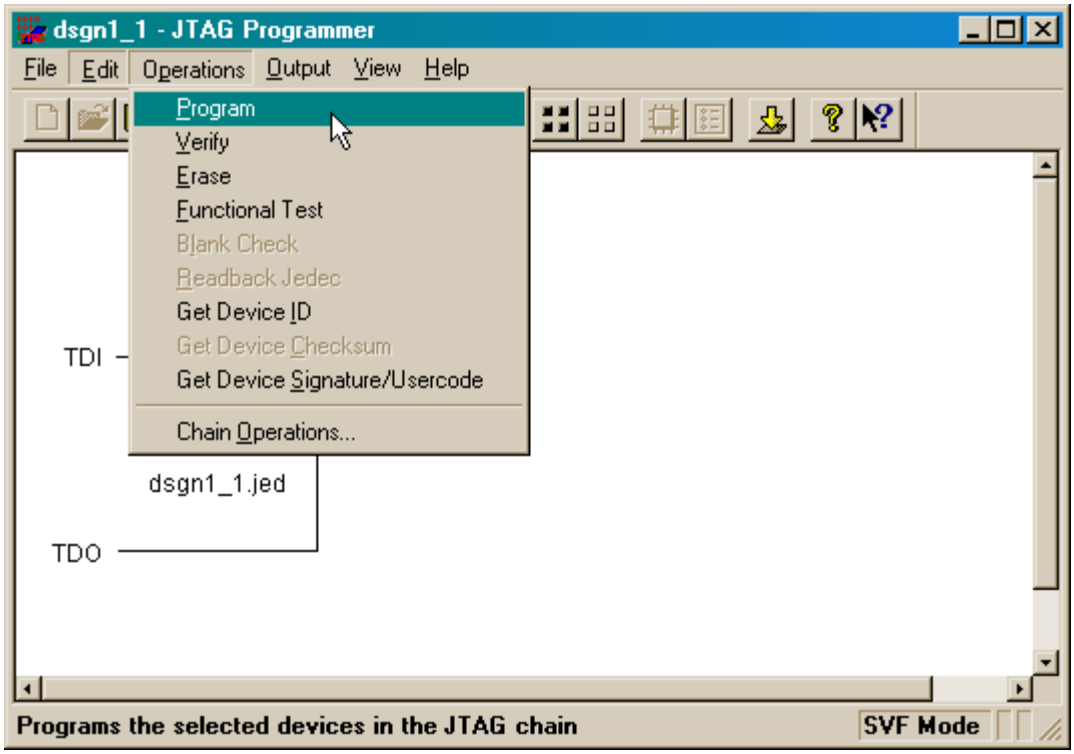
In the **SVF Options** window, select the Through Test-Logic-Reset radio button. This will insure that the downloading circuitry of the XC9500 CPLD is properly initialized before the bitstream enters the device. Then click on OK.



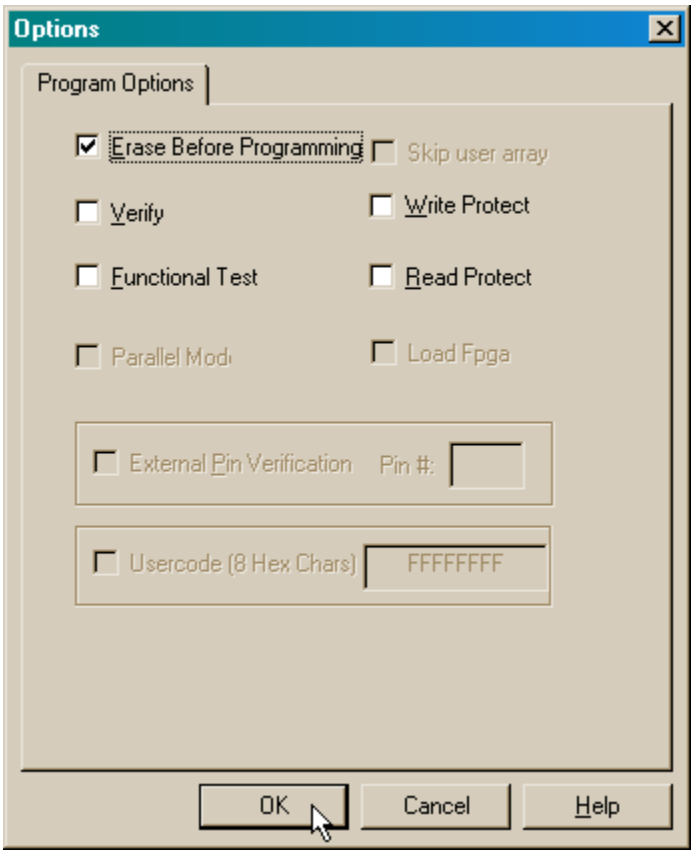
The **Create a New SVF File** window will appear next. By default, the SVF file will be stored in the folder for this verison and revision of the project in a file called dsgn1_1.svf. Just click on Save to accept these defaults.



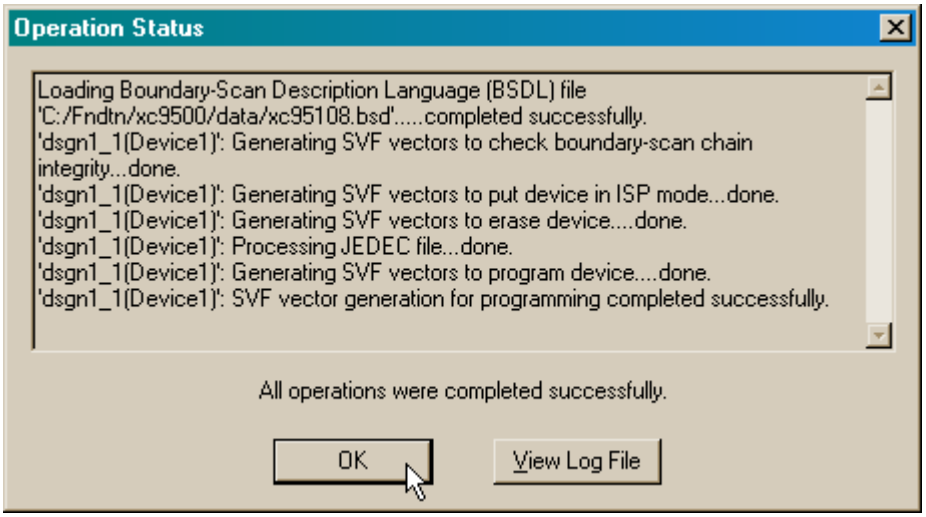
Once the SVF file specified, it is time to generate the translated bitstream. Since the bitstream is going to be programmed into the CPLD, select the Operations→Program item from the menu list.



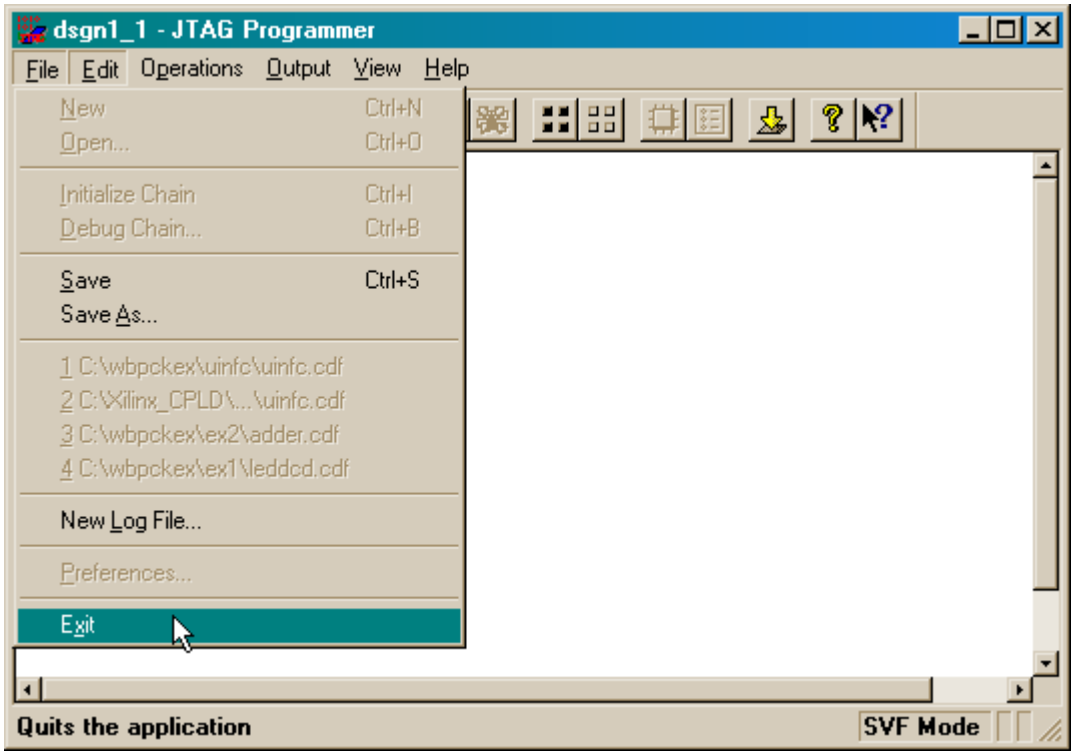
The **Options** window will appear with a set of actions we can add to the programming bitstream. The Erase Before Programming option should be checked to force the CPLD to erase its internal Flash storage before the new bitstream is loaded. (Failure to erase the storage can lead to errors if the CPLD was previously programmed with a different bitstream.) None of the other options are applicable to this example so they should be left unchecked. (*Definitely do not check the Write Protect box or you will be unable to load new bitstreams into the CPLD in the future.*) Click on OK after setting the options as shown below.



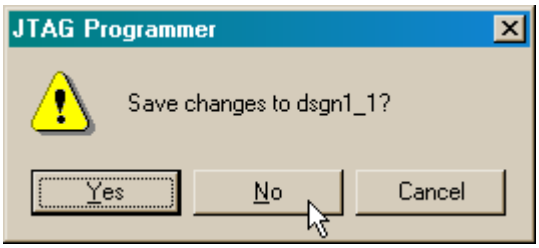
An **Operation Status** window appears to show you the progress as the SVF file is generated. Click on OK after all operations are complete.



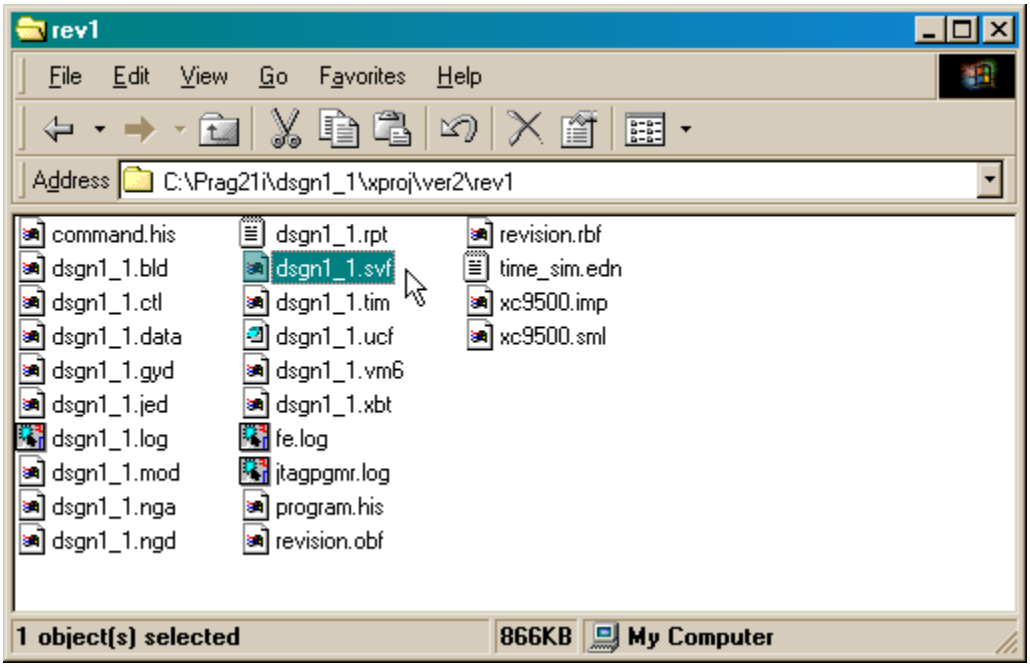
Now that the SVF file with the translated bitstream has been created, we can exit the **JTAG Programmer** window.



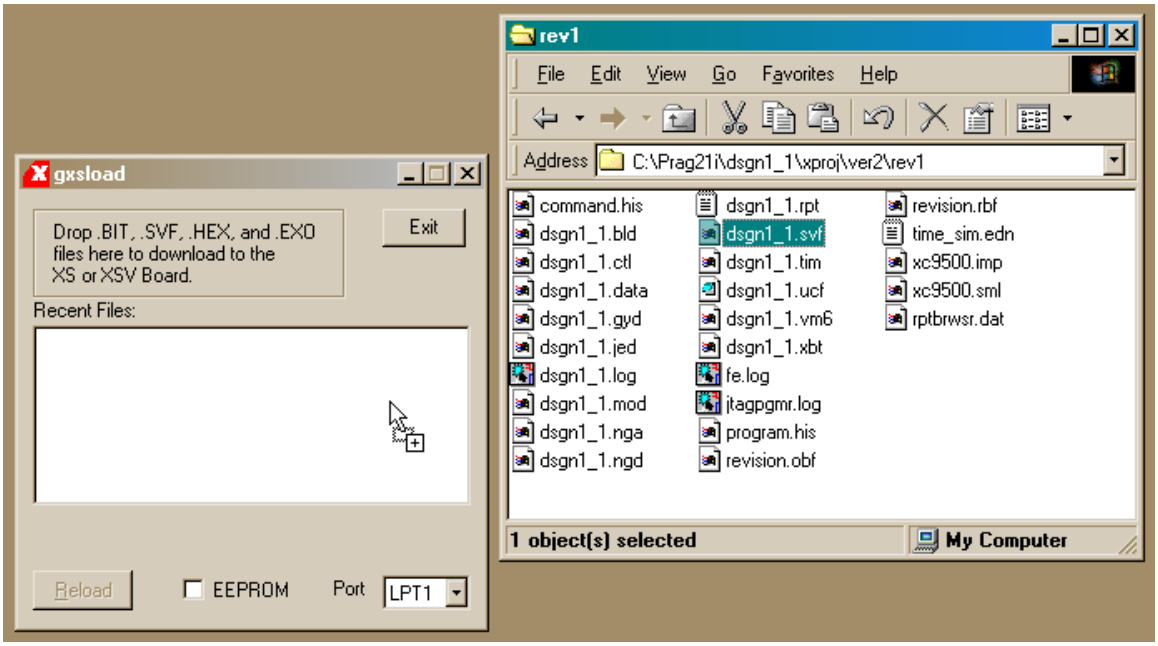
Upon exiting the **JTAG Programmer** window, you will be asked if you want to save any changes. This is rarely necessary so click on No. (This will not affect the SVF file that was already created.)



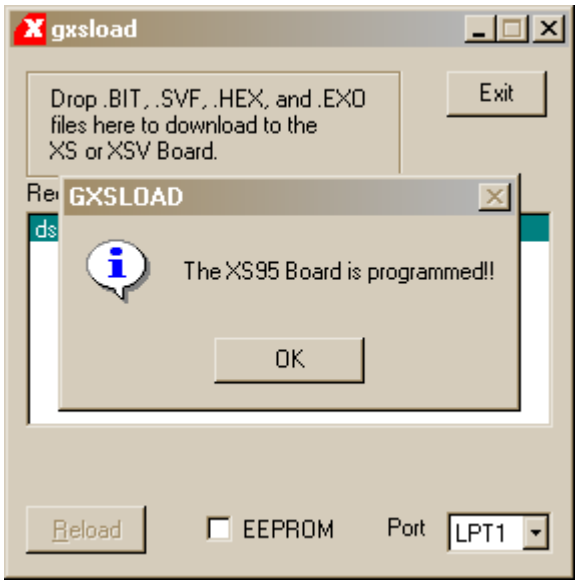
Now we have to go find the SVF file with the bitstream. It will be stored in the folder for reversion rev1 of version ver2 of the dsgn1_1 project as shown below.



With the XS95 Board connected to a power supply and the PC parallel port, just drag-and-drop the dsgn1_1.svf file over to the **gxsload** window.



It will take a minute or so for the bitstream to download into the XC95108 CPLD on the XS95 Board. (Most of this time is used to erase the nonvolatile Flash memory in the CPLD that stores the bitstream.)



Once the XS95 Board is programmed, we can test it in exactly the same way we tested the LED decoder circuit with the XS40 Board. The results should be the identical.