Xilinx ISE 10 Tutorial

A Tutorial on Using the Xilinx ISE Software to Create FPGA Designs for the XESS XSA Boards

Release date: 6/2/2008
# Table of Contents

What This Is and *Is Not* ................................................................. 1

FPGA Programming ........................................................................ 3

Installing WebPACK ........................................................................ 5

  Getting WebPACK ....................................................................... 5

  Installing WebPACK .................................................................... 7

Getting XSTOOLs ........................................................................... 8

  Installing XSTOOLs ................................................................. 8

Getting the Design Examples .......................................................... 8

Our First Design ............................................................................. 9

  An LED Decoder ......................................................................... 9

Starting WebPACK Project Navigator ................................................. 11

Describing Your Design With VHDL .................................................. 16

Checking the VHDL Syntax .............................................................. 21

Fixing VHDL Errors ....................................................................... 22

Synthesizing the Logic circuitry for Your Design ............................... 25

Implementing the Logic Circuitry in the FPGA ..................................... 25

Checking the Implementation .......................................................... 28

Assigning Pins with Constraints ...................................................... 29

Viewing the Chip ........................................................................... 35

Generating the Bitstream ............................................................... 42

Downloading the Bitstream ............................................................ 47

Testing the Circuit ................................................................. 50

Hierarchical Design ....................................................................... 51

  A Displayable Counter ............................................................. 51

Starting a New Design ................................................................. 52
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adding a Counter</td>
<td>59</td>
</tr>
<tr>
<td>Tying Them Together</td>
<td>65</td>
</tr>
<tr>
<td>Constraining the Design</td>
<td>84</td>
</tr>
<tr>
<td>Synthesizing and Implementing the Design</td>
<td>88</td>
</tr>
<tr>
<td>Checking the Implementation</td>
<td>89</td>
</tr>
<tr>
<td>Checking the Timing</td>
<td>91</td>
</tr>
<tr>
<td>Generating the Bitstream</td>
<td>91</td>
</tr>
<tr>
<td>Downloading the Bitstream</td>
<td>97</td>
</tr>
<tr>
<td>Testing the Circuit</td>
<td>103</td>
</tr>
<tr>
<td>State Machine Design</td>
<td>104</td>
</tr>
<tr>
<td>Finite State Machines</td>
<td>104</td>
</tr>
<tr>
<td>Starting the Combination Lock Project</td>
<td>106</td>
</tr>
<tr>
<td>Creating the Keyboard Interface Module</td>
<td>106</td>
</tr>
<tr>
<td>Creating the Lock&amp;Key Module</td>
<td>112</td>
</tr>
<tr>
<td>Creating the Top-Level Module</td>
<td>120</td>
</tr>
<tr>
<td>Constraining the Design</td>
<td>122</td>
</tr>
<tr>
<td>Implementing the Design and Generating the Bitstream</td>
<td>122</td>
</tr>
<tr>
<td>Testing the Combination Lock</td>
<td>123</td>
</tr>
<tr>
<td>Going Further</td>
<td>125</td>
</tr>
</tbody>
</table>
What This Is and Is Not

There are numerous requests on newsgroups 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 a free version of their ISE software on the web (they call it WebPACK) so that anyone can download a set of tools for CPLD and FPGA-based logic designs. And XESS Corp. has written this tutorial that attempts to give you a gentle introduction to using the ISE tools.

This tutorial shows the use of the ISE tools on three simple design examples: 1) an LED decoder, 2) a counter which displays its current value on a seven-segment LED and 3) a reprogrammable combination lock. Along the way, you will see:

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

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

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

- It will not show you how to use the variety of peripheral devices available on the XSA Boards.

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

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

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

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

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

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

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

That's really all there is to it. XILINX ISE provides the HDL and schematic editors, logic synthesizer, fitter, and bitstream generator software. The XSTOOLS from XESS provide utilities for downloading the bitstream into the FPGA on the XSA Board.
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
       "4101101";
end leddcd_arch;
Installing ISE

Getting ISE

You can download the free ISE WebPACK software from this location: http://www.xilinx.com/ise/logic_design_prod/webpack.htm. Click on the link to download the software as shown below.
You will have to create an account and choose a user ID and password before you are allowed to enter the download section of the site. Once you do that, you will get to a page where you can select the software that you want to download. In this case, check the ISE WebPACK 10.1 box and click the Next button.
Now you are finally at the page where you can download the ISE WebPACK software. Expand the Download Files Individually option, and then click on the Download link. This will download the entire install file for ISE WebPACK. With a size of 2.25 GB, it will take at least an hour to download, even with a 5 Mb/s link.

**Installing ISE**

After the ISE WebPACK software download is complete, unpack the WebPACK_SFD.tar file and double-click the setup.exe file to start the installer. During the installation, you will have to enter the registration ID that you will receive in an email from Xilinx. Throughout the rest of the installation, accept the default settings for everything and you shouldn’t have any problems. The total installation will take at least an hour.
Getting XSTOOLs

If you are going to download your FPGA bitstreams into an XESS FPGA Board, then you will need to get the XSTOOLS software from [http://www.xess.com/ho07000.html](http://www.xess.com/ho07000.html). Just download the latest XSTOOLs setup file.

Installing XSTOOLs

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

Getting the Design Examples

You can download the project files for the design examples shown in this tutorial from [http://www.xess.com/projects/ise-10.zip](http://www.xess.com/projects/ise-10.zip).
Our First Design

An LED Decoder

The first FPGA design you 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:

<table>
<thead>
<tr>
<th>Four-bit Input</th>
<th>Hex Digit</th>
<th>LED Display</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>1011</td>
<td>B</td>
<td>b</td>
</tr>
<tr>
<td>1100</td>
<td>C</td>
<td>c</td>
</tr>
<tr>
<td>1101</td>
<td>D</td>
<td>d</td>
</tr>
<tr>
<td>1110</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>1111</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
A high-level diagram of the LED decoder looks like this:
Starting ISE Project Navigator

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

1. A **source pane** that shows the organization of the source files that make up your design. There are four tabs so you can view the functional modules, source files, different snapshots (or versions) of your project, or the HDL libraries for your project.

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

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

4. An **editor pane** where you can enter HDL code, schematics, state diagrams, etc.

To start your design, create a new project by selecting the File ➔ New Project item from the menu bar. This brings up the **New Project Wizard** window where you can enter the name of your project, the location of your project files, and the style in which you will describe your design at the top level. I have given my project the descriptive name of **design1** and will place the files
in the C:\TEMP\fpga_designs\design1 folder. (You may choose differently.) I am going to describe the LED decoder using VHDL, so I have set the top-level type (there will only be one level) to HDL (which would also be used if the design was done with Verilog). Click Next to continue creating your project.

![New Project Wizard - Create New Project](image)

Now you need to tell ISE what FPGA you are going to use for your design. The device family, family member, package and speed grade for the FPGA on each model of XSA Board are shown below.

<table>
<thead>
<tr>
<th>XSA Board</th>
<th>Device Family</th>
<th>Device</th>
<th>Package</th>
<th>Speed Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSA-50</td>
<td>Spartan2</td>
<td>XC2S50</td>
<td>TQ144</td>
<td>-5</td>
</tr>
<tr>
<td>XSA-100</td>
<td>Spartan2</td>
<td>XC2S100</td>
<td>TQ144</td>
<td>-5</td>
</tr>
<tr>
<td>XSA-200</td>
<td>Spartan2</td>
<td>XC2S200</td>
<td>FG256</td>
<td>-5</td>
</tr>
<tr>
<td>XSA-3S1000</td>
<td>Spartan3</td>
<td>XC3S1000</td>
<td>FT256</td>
<td>-4</td>
</tr>
</tbody>
</table>
For this tutorial, I will target my design to the XSA-3S1000 Board so I have set the Value field of the Family, Device, Package and Speed properties as shown below. (Set these fields to whatever values are appropriate for your particular board using the table shown above.) The other fields can be left at their default values, so you can just click on the Next button to continue creating the project.
Click the Next button in the following two windows for creating or adding source files. (You will create the VHDL source code for the LED decoder at a later step.)
The final screen shows the pertinent information for the new project. Click on the Finish button to complete the creation of the project.

![New Project Wizard - Project Summary](image)

Now the Sources pane contains two items:

1. A project object called design1.
2. A chip object called xc3s1000-4ft256.
Describing Your Design With VHDL

Once all the project set-up is complete, you can begin to actually design your LED decoder circuit. Start by adding a VHDL file to the `design1` project. Right-click on the `xc3s1000-4ft256` object in the Sources pane and select New Source … from the pop-up menu as shown below.

A window appears where you must select the type of source file you want to add. Since you are describing the LED decoder with VHDL, highlight the VHDL Module item. Then type the name of the module, `leddcd`, into the File name field and click on Next.
The **Define Module** window now appears where you can declare the inputs and outputs to the LED decoder circuit. In the first row, click in the Port Name field and type in `d` (the name of the inputs to the LED decoder). The `d` input bus has a width of four, so check the Bus box and type 3 in the MSB field while leaving 0 in the LSB field. Perform the same operations to create the seven-bit wide `s` output bus that drives the LEDs.
Click on Next in the **Define Module** window to get a summary of the information you just typed in:

![New Source Wizard - Summary](image)

- **Add to Project**: Yes
- **Source Directory**: C:TEMP\fpga\designs\design1
- **Source Type**: VHDL Module
- **Source Name**: ledcdc.vhd
- **Entity name**: ledcdc
- **Architecture name**: Behavioral
- **Port Definitions**:
  - d Bus: 3:0 in
  - s Bus: 6:0 out
After clicking on Finish, the editor pane displays a design summary and a VHDL skeleton for the LED decoder. (You can also see the leddcd.vhd file has been added to the Sources pane.) Click on the leddcd tab at the bottom of the editor pane and then scroll to the bottom of the VHDL skeleton. Lines 20-23 create links to the IEEE library and packages that contain various useful definitions for describing a design. The LED decoder inputs and outputs are declared in the VHDL entity on lines 30-33. You will describe the logic operations of the decoder in the architecture section between lines 37 and 40.
The completed VHDL file for the LED decoder is shown below. The architecture section contains a single statement which assigns a particular seven-bit pattern to the output bus for any given four-bit input on the bus (lines 39-54).

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

--- Uncomment the following library declaration if instantiating
--- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

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

architecture Behavioral of ledccd is
begin

  s <= "1110111" when d="0000" else
   "0010010" when d="0001" else
   "1011001" when d="0010" else
   "1011011" when d="0011" else
   "0111010" when d="0100" else
   "1110111" when d="0110" else
   "1010010" when d="0111" else
   "1111111" when d="1000" else
   "1111011" when d="1001" else
   "1111101" when d="1010" else
   "0101111" when d="1011" else
   "0001101" when d="1100" else
   "0011111" when d="1101" else
   "1101011" when d="1110" else
   "1101100"

end Behavioral;
```

Once the VHDL source is entered, click on the button to save it in the `ledccd.vhd` file.
Checking the VHDL Syntax

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

The syntax checking tool grinds away and then displays the result in the process pane. In this 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

You can find the location of the error by clicking on the Errors tab at the bottom of the transcript pane. In this case, the error is located on line 44 and you can manually scroll there. You can also click on the error message in the log pane to go directly to the erroneous source. (This is most useful in more complicated projects consisting of multiple source files.) You can also get a more detailed explanation of the error by clicking on the ERROR hyperlink at the beginning of the message.
On line 44, you can see that the ‘e’ was left off the end of the `else` keyword. After correcting this error and saving the file, double-click the on Check Syntax in the Process pane to re-check the VHDL code. The syntax checker now finds another error on line 56 of the VHDL code.
Examining line 56, you can see it is just the end statement for the architecture section. The actual error occurred on line 54. The VHDL syntax checker was expecting to find a ‘;’ but it is missing. After adding the semicolon and saving the file, the Check Syntax process runs without errors and displays a ✓.
Synthesizing the Logic circuitry for Your Design

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

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

Implementing the Logic Circuitry in the FPGA

You now have a synthesized logic circuit for the LED decoder, but you need to translate, map and place & route it into the logic resources of the FPGA in order to actually use it. Start this
process by highlighting the `ledccd` object in the Sources pane and then double-click on the Implement Design process.
You can watch the progress of the implementation process in the console tab of the transcript pane. For a simple design this, the implementation is completed in less than 30 seconds (on a 2.0 GHz Athlon 64 X2 PC with 3 Gbytes of RAM). A successful implementation is indicated by the checkmark next to the Implement Design process. You can expand the Implement Design process to see the subprocesses within it. The Translate process converts the netlist output by the synthesizer into a Xilinx-specific format and annotates it with any design constraints you may specify (more on that later). The Map process decomposes the netlist and rearranges it so it fits nicely into the circuitry elements contained in the specified FPGA device. Then the Place & Route process assigns the mapped elements to specific locations in the FPGA and sets the switches to route the logic signals between them. If the Implement Design process had failed, a red X would appear next to the subprocess where the error occurred. You may also see a yellow triangle that indicates a successful completion but some warnings were issued or not all the subprocesses were enabled.
Checking the Implementation

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

The device utilization of the LED decoder circuit can be found near the top of the place & route report (or in the Design Summary tab). The circuit only uses four of the 7680 available slices in the XC3S1000 FPGA. Each slice contains two CLBs and each CLB can compute the logic function for one LED segment output.

Device utilization summary:

- Number of External IOBs: 11 out of 173 (6%)
- Number of LOCed External IOBs: 0 out of 11 (0%)
- Number of Slices: 4 out of 7680 (1%)
- Number of SLICEMs: 0 out of 3840 (0%)
The pad report shows what pins the LED decoder inputs and outputs use to enter and exit the FPGA. (The pad report was edited to remove unused pins and fields so it would fit into this document.)

<table>
<thead>
<tr>
<th>Pin Number</th>
<th>Signal Name</th>
<th>Pin Usage</th>
<th>Direction</th>
<th>IO Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>A7</td>
<td>s&lt;4&gt;</td>
<td>IOB</td>
<td>OUTPUT</td>
<td>LVCMOS25</td>
</tr>
<tr>
<td>A8</td>
<td>s&lt;0&gt;</td>
<td>IOB</td>
<td>OUTPUT</td>
<td>LVCMOS25</td>
</tr>
<tr>
<td>B6</td>
<td>s&lt;6&gt;</td>
<td>IOB</td>
<td>OUTPUT</td>
<td>LVCMOS25</td>
</tr>
<tr>
<td>B7</td>
<td>d&lt;0&gt;</td>
<td>IOB</td>
<td>INPUT</td>
<td>LVCMOS25</td>
</tr>
<tr>
<td>B8</td>
<td>s&lt;1&gt;</td>
<td>IOB</td>
<td>OUTPUT</td>
<td>LVCMOS25</td>
</tr>
<tr>
<td>C6</td>
<td>d&lt;3&gt;</td>
<td>IOB</td>
<td>INPUT</td>
<td>LVCMOS25</td>
</tr>
<tr>
<td>C7</td>
<td>d&lt;2&gt;</td>
<td>IOB</td>
<td>INPUT</td>
<td>LVCMOS25</td>
</tr>
<tr>
<td>C8</td>
<td>s&lt;3&gt;</td>
<td>IOB</td>
<td>OUTPUT</td>
<td>LVCMOS25</td>
</tr>
<tr>
<td>D7</td>
<td>s&lt;5&gt;</td>
<td>IOB</td>
<td>OUTPUT</td>
<td>LVCMOS25</td>
</tr>
<tr>
<td>D8</td>
<td>d&lt;1&gt;</td>
<td>IOB</td>
<td>INPUT</td>
<td>LVCMOS25</td>
</tr>
<tr>
<td>E7</td>
<td>s&lt;3&gt;</td>
<td>IOB</td>
<td>OUTPUT</td>
<td>LVCMOS25</td>
</tr>
</tbody>
</table>

### Assigning Pins with Constraints

The problem now is that the inputs and outputs for the LED decoder were assigned to pins picked by the implementation process, but these are not the pins you actually want to use on the FPGA. You want the inputs assigned to pins on the FPGA that you can force high and low so as to test the LED decoder operation for each possible input pattern. In addition, the outputs should be attached to a seven-segment LED to make it easy to verify the correct operation of the design.

The GXSPORT utility lets you set the levels on the eight data outputs of the PC parallel port. The parallel port data pins attach to a group of eight specific pins on the FPGA of each model of XSA Board. You should assign the LED decoder inputs to four of these so that you can control the inputs using GXSPORT. The four pins I selected from the group of eight on each XSA Board are shown below:

<table>
<thead>
<tr>
<th>LED Decoder Input</th>
<th>XSA-50</th>
<th>XSA-100</th>
<th>XSA-200</th>
<th>XSA-3S1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>d0</td>
<td>P50</td>
<td>P50</td>
<td>E13</td>
<td>N14</td>
</tr>
<tr>
<td>d1</td>
<td>P48</td>
<td>P48</td>
<td>C16</td>
<td>P15</td>
</tr>
<tr>
<td>d2</td>
<td>P42</td>
<td>P42</td>
<td>E14</td>
<td>R16</td>
</tr>
<tr>
<td>d3</td>
<td>P47</td>
<td>P47</td>
<td>D16</td>
<td>P14</td>
</tr>
</tbody>
</table>
Likewise, each XSA Board has a seven-segment LED attached to the following pins of the FPGA:

<table>
<thead>
<tr>
<th>LED Decoder Output</th>
<th>XSA-50</th>
<th>XSA-100</th>
<th>XSA-200</th>
<th>XSA-3S1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>P67</td>
<td>P67</td>
<td>N14</td>
<td>M6</td>
</tr>
<tr>
<td>s1</td>
<td>P39</td>
<td>P39</td>
<td>D14</td>
<td>M11</td>
</tr>
<tr>
<td>s2</td>
<td>P62</td>
<td>P62</td>
<td>N16</td>
<td>N6</td>
</tr>
<tr>
<td>s3</td>
<td>P60</td>
<td>P60</td>
<td>M16</td>
<td>R7</td>
</tr>
<tr>
<td>s4</td>
<td>P46</td>
<td>P46</td>
<td>F15</td>
<td>P10</td>
</tr>
<tr>
<td>s5</td>
<td>P57</td>
<td>P57</td>
<td>J16</td>
<td>T7</td>
</tr>
<tr>
<td>s6</td>
<td>P49</td>
<td>P49</td>
<td>G16</td>
<td>R10</td>
</tr>
</tbody>
</table>

How do you direct the implementation process so it assigns the inputs and outputs to the pins you want to use? This is done by using constraints. In this case, you are constraining the implementation process so it assigns the inputs and outputs only to the pins shown in the previous tables. Start creating these constraints by right-clicking the leddcd object in the Sources pane and selecting New Source... from the pop-up menu.
Select Implementation Constraints File as the type of source file you want to add and type `leddc1d` in the File Name field. Then click on the Next button.

You will receive a feedback window that shows the name and type of the file you created and the file to which the constraints apply (`leddc1d.vhd`). Click on the Finish button to complete the addition of the `leddc1d.ucf` constraint file to this project.
Now highlight the ledccd object in the Sources pane and double-click the Floorplan IO – Pre-Synthesis item in the Process pane to begin adding pin assignment constraints to the design.
The Xilinx PACE window now appears. Click on the I/O Pins item in the Design Browser pane. A list of the current inputs and outputs for the LED decoder will appear in the Design Object List – I/O Pins pane. You can change your pin assignments here. You start by clicking in the Loc field for the \texttt{d<0>} input. Then just type in the pin assignment for this input: \texttt{N14}. Do this for all of the inputs and outputs using the pin assignments from the previous table.

After the pin assignments are entered, save the file. A dialog window will appear requesting that you select the delimiter for the I/O buses. Select \texttt{<>} since the XST synthesizer is being used for this project, and then click on OK. Then close the Xilinx PACE window.

Now you can re-implement your design by highlighting the \texttt{ledcd} object in the Sources pane and double-clicking on the Implement Design process. After the implementation process completes,
double-click on Pad Report to view the pin assignments. Now the pad report shows the following pin assignments:

<table>
<thead>
<tr>
<th>Pin Number</th>
<th>Signal Name</th>
<th>Pin Usage</th>
<th>Direction</th>
<th>IO Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>M6</td>
<td>s&lt;0&gt;</td>
<td>IOB</td>
<td>OUTPUT</td>
<td>LVCMOS25</td>
</tr>
<tr>
<td>M11</td>
<td>s&lt;1&gt;</td>
<td>IOB</td>
<td>OUTPUT</td>
<td>LVCMOS25</td>
</tr>
<tr>
<td>N6</td>
<td>s&lt;2&gt;</td>
<td>IOB</td>
<td>OUTPUT</td>
<td>LVCMOS25</td>
</tr>
<tr>
<td>N14</td>
<td>d&lt;0&gt;</td>
<td>IOB</td>
<td>INPUT</td>
<td>LVCMOS25</td>
</tr>
<tr>
<td>P10</td>
<td>s&lt;4&gt;</td>
<td>IOB</td>
<td>OUTPUT</td>
<td>LVCMOS25</td>
</tr>
<tr>
<td>P14</td>
<td>d&lt;3&gt;</td>
<td>IOB</td>
<td>INPUT</td>
<td>LVCMOS25</td>
</tr>
<tr>
<td>P15</td>
<td>d&lt;1&gt;</td>
<td>IOB</td>
<td>INPUT</td>
<td>LVCMOS25</td>
</tr>
<tr>
<td>R7</td>
<td>s&lt;3&gt;</td>
<td>IOB</td>
<td>OUTPUT</td>
<td>LVCMOS25</td>
</tr>
<tr>
<td>R10</td>
<td>s&lt;6&gt;</td>
<td>IOB</td>
<td>OUTPUT</td>
<td>LVCMOS25</td>
</tr>
<tr>
<td>R16</td>
<td>d&lt;2&gt;</td>
<td>IOB</td>
<td>INPUT</td>
<td>LVCMOS25</td>
</tr>
<tr>
<td>T7</td>
<td>s&lt;5&gt;</td>
<td>IOB</td>
<td>OUTPUT</td>
<td>LVCMOS25</td>
</tr>
</tbody>
</table>

The reported pin assignments match the assignments made in the Xilinx Pace window, so it appears the I/O have been constrained to the appropriate pins.
Viewing the Chip

After the implementation process completes, you can get a graphical depiction of how the logic circuitry and I/O are assigned to the FPGA CLBs and pins. Just highlight the `leddcd` object in the Sources pane and then double-click the View/Edit Placed Design (FloorPlanner) process.
The **FloorPlanner** window will appear containing four panes:

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

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

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

4. The **Editable Floorplan** pane allows you to change the pin assignments of the I/O signals by dragging them to different pin locations.
The CLBs used by the LED decoder circuit are highlighted in light-green and are clustered near the lower right-hand edge of the CLB array. To enlarge this region of the array, click on the button and then draw a rectangle around the highlighted CLBs in the Placement pane. The enlarged view of the CLBs used by the LED decoder appear as shown below.
You can enable the display of the connections between I/O pins and CLBs by selecting the Edit ➔ Preferences menu item and then checking the boxes in the Ratsnest tab of the Edit Preferences window as shown below.
Now clicking on a CLB will highlight the nets connecting the inputs and output to the CLB.
In an analogous manner, you can click on an input pin to highlight which CLBs are dependent on that input. (For this design, each input affects every CLB.)
To view all the available FPGA resources (not just those that are used by the design), select Edit→Preferences and check all the boxes in the Resources tab as shown below.

Now the Placement pane shows all the LUTs, RAMs, buffers, etc. that are available in the FPGA.
Viewing the placement of circuit elements after the place & route process can give you insights into the resource usage of certain VHDL language constructs. In addition to viewing the placement of the design, the Floorplanner can be used to re-arrange and optimize the placement. This is akin to the software technique of hand-optimizing assembly code output by a compiler. You won’t do this here, but it is an option for designs which push at the limits of the capabilities of FPGAs.

**Generating the Bitstream**

Now that you have synthesized our design and mapped it to the FPGA with the correct pin assignments, you are ready to generate the bitstream that is used to program the actual chip.

When using the XSA-3S1000 Board, you need to set some options before generating the bitstream. (This is not needed if you are using the XSA-50, XSA-100 or XSA-200 Board, but it wouldn’t hurt, either.) Right-click on the Generate Programming File process and select Properties… from the pop-up menu.

Now select the Configuration Options tab in the Process Properties window and set all the Pull Up and Pull Down values to Float to disable the internal resistors of the FPGA. (At the minimum, the Unused IOB Pins value must be set to Float, but it is best to set them all to Float.) If they are not
disabled, the strong internal pull-up and pull-down resistors in the Spartan3 FPGA will overpower the external resistors on the XSA Board.
After setting all the values as shown below, click on OK.
Once you have set the bitstream generation options, highlight the `ledpdf` object in the Sources pane and double-click on the Generate Programming File process.
Within a few seconds, a green checkmark will appear next to the Generate Programming File process and a file detailing the bitstream generation process will be created. A bitstream file named leddcd.bit can now be found in the design1 folder.
Downloading the Bitstream

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

(You can also communicate with the XSA Board through a USB port by attaching an XSUSB interface as shown below. Downloading the bitstream is done in the same way regardless of the choice of interface.)
The XSA Boards are programmed using the \texttt{gxsload} utility. Double click the \texttt{gxsload} icon to bring up the \texttt{gxsload} window:

![gxsload window](image)

Then click in the Board Type field and select XSA-3S1000 from the drop-down menu since this is the board you are going to load with the bitstream.

![gxsload window with XSA-3S1000 selected](image)
Then open a window that shows the contents of the folder where you stored the LED decoder design (C:\TEMP\fpga_designs\design1 in my case). You just drag-and-drop the leddcd.bit file from the design1 window into the FPGA/CPLD pane of the gxsload window. Then you click on the Load button to initiate the programming of the FPGA. Downloading the leddcd.bit file to the XSA Board takes only a few seconds.
Testing the Circuit

Once the FPGA on the XSA Board is programmed, you can begin testing the LED decoder. The eight data pins of the PC parallel port connect to the FPGA through the downloading cable. You have assigned the inputs of the LED decoder to pins, which are connected to the parallel port data pins. The gxsport utility lets you control the logic values on these pins. By placing different bit patterns on the pins, you can observe the outputs of the LED decoder through the seven-segment LED on the XSA Board.

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

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

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

A Displayable Counter

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

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

You can start a new project using the File ➔ New Project... menu item. Name the project `design2` and store it in the same folder as the previous design: `C:\TEMP\fpga_designs`. Then click on the Next button.

![New Project Wizard - Create New Project dialog box](image)

- **Project name:** `design2`
- **Project location:** `C:\TEMP\fpga_designs\design2`

- **Top-level source type:** `HDL`
You will target the same FPGA on the XSA Board, so the other properties in the New Project window retain the same values you set in the previous project.

![New Project Wizard - Device Properties](image)

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product Category</td>
<td>All</td>
</tr>
<tr>
<td>Family</td>
<td>Spartan3</td>
</tr>
<tr>
<td>Device</td>
<td>XC3S1000</td>
</tr>
<tr>
<td>Package</td>
<td>FT256</td>
</tr>
<tr>
<td>Speed</td>
<td>4</td>
</tr>
<tr>
<td>Top-Level Source Type</td>
<td>HDL</td>
</tr>
<tr>
<td>Synthesis Tool</td>
<td>XST (VHDL/Verilog)</td>
</tr>
<tr>
<td>Simulator</td>
<td>ISE Simulator (VHDL/Verilog)</td>
</tr>
<tr>
<td>Preferred Language</td>
<td>VHDL</td>
</tr>
<tr>
<td>Enable Enhanced Design Summary</td>
<td>✔️</td>
</tr>
<tr>
<td>Enable Message Filtering</td>
<td>❌</td>
</tr>
<tr>
<td>Display incremental Messages</td>
<td>❌</td>
</tr>
</tbody>
</table>
You won’t add any new source files at the moment, so just click on the Next button on the Create New Source window. This brings up the Add Source window shown below that lets you add existing source files to your new project. It will save time if you re-use the LED decoder from the previous design, so click on the Add Source... button.
The Add Existing Sources window appears. Move to the C:TEMP\fpga_designs\design1 folder and highlight the ledccd.vhd file that contains the VHDL source code for the LED decoder.
The **New Project** window now shows that a copy of the leddcd.vhd file will be added to the project. The Copy to Project box is checked, so the leddcd.vhd file will be copied from the design1 folder to the design2 folder. Unchecking this box will cause the **design2** project to link directly to the leddcd.vhd file in the design1 folder, so any change in this file will affect both projects. For this example, I have chosen to make a separate copy so the box remains checked. This is the only existing file you need to add, so click on the Next button to move on to the next window.
The final screen shows the pertinent information for the new project. Click on the Finish button to complete the creation of the project.

An **Adding Source Files**... window will appear that lets you specify the intended use for the files you are adding. Files can be used during synthesis and implementation of an FPGA bitstream as we did in the previous design. They can also be used only during simulation, such as testbench files that specify stimulus patterns that are driven onto the inputs of a design. Or they can be for both synthesis/implementation and simulation (as most are) because they describe the function and/or structure of the design in both these domains. There are also files, such as text files containing design documentation that are added to a project but are not used in either domain. I selected All for the leddcd.vhd file even though it will only be used for synthesis and implementation in this example.
After you click on Finish in the **Adding Source Files...** window, the Sources pane contains only the single `ledcd.vhd` source file.
Adding a Counter

Now you have to add the counter to the design. There is no counter module yet, so you have to build one. Right-click on the xc3s1000-4ft256 object and select New Source... from the pop-up menu.
As in the previous example, you are prompted for the type of file to add to the project. This time, choose the IP (Coregen & Architecture Wizard) menu item. This will allow you to select a counter from a library of pre-built, configurable components. Then type `counter` into the File Name field and click on the Next button.

In the next window, expand the library tree until you find the binary counter component. Then click on Next.
A very uninteresting summary of your choice appears. Click Finish to move on to the screens that will let you configure the counter component for this design.
The initial configuration window specifies a sixteen-bit counter that increments its value upward by one on each clock cycle until it reaches 65,535 after which it rolls-over back to 0. You can change many of the features of this counter on the following screens. (Click on the View Data Sheet button to see all the details on how this counter can be configured.)

The only change that needs to be made for this design is to increase the counter width to 28 bits. Why build a 28-bit counter when only the upper four bits are used? The counter will be driven by a clock signal on the XSA Board that has a frequency of 50 MHz. The LED display would be changing much too quickly to see at this frequency. By connecting the LED decoder to the upper four bits of the 28-bit counter, the display will only change once in every \(2^{24}\) clock cycles. So the LED display will change every \(2^{24} / (50 \times 10^6) = 0.336\) seconds which is slow enough to read.

After changing the counter width, click on Next to continue configuring the counter.
None of the advanced configuration options for the counter are needed in this design, so click Next and Finish in the next two windows that appear.
After clicking Finish in the final configuration window, the **counter** module appears in the Sources pane. The counter is stored as a Xilinx Coregen object (.xco file suffix). If you need to change the operation of the counter, just double-click the counter module and make your changes in the Coregen wizard windows that appear.
Tying Them Together

You have the LED decoder and the counter, but now you need to tie them together to build the displayable counter. You will do this by connecting the counter to the LED decoder in a top-level schematic. Before you can do this, you have to create a schematic symbol for the LED decoder module from its VHDL source code. (The counter module already has a schematic symbol since it was created using Coregen.) To create the LED decoder schematic symbol, highlight the leddcd object in the Sources pane and then double-click the Create Schematic Symbol process. A message indicating the schematic symbol has been created will appear in the Console tab of the Transcript pane.
Now it is time to create the top-level schematic that will hold the counter and LED decoder symbols. Right-click on the xc3s1000-4ft256 object and select New Source… from the pop-up menu. Then highlight the Schematic entry in the **New Source** window and name the schematic `disp_cnt`. Then click on Next.

There is very little to do when initializing a schematic, so just click on the Finish button in the **Summary** window that appears.
The disp_cnt schematic object has now been added to the Sources pane. You can double-click it to begin creating the schematic, but a schematic editor window should open automatically once the file is created.
Click on the Symbols tab at the bottom of the Sources pane. The Symbols tab contains a list of categories for various logic circuit elements that can be used in a schematic. Below that is the list of circuit element symbols in the highlighted category. A symbol can be selected from this list and dropped into the drawing area to the right.
To start creating the top-level schematic, highlight the second entry in the category list. The c:/TEMP/fpga_designs/design2 category contains the schematic symbols for the design2 project’s counter and LED decoder modules. You can see the names of these modules in the symbol list.
Click on the counter entry in the Symbols list. Then move the mouse cursor into the drawing area and left-click to place an instance of the **counter** module into the schematic. Repeat this process with the **leddcd** module to arrive at the arrangement shown below. (For clarity, I used the Edit ➔ Preferences command to turn off the display of the grid in the schematic drawing area.)
Next, click on the button to begin adding wires to the schematic.
Left-click the mouse on the q(27:0) bus on the right-hand edge of the counter module. Then left-click on the d(3:0) bus on the left-hand edge of the ledcdc module. This creates a four-bit bus between the output of the counter and the input of the LED decoder.

However, the four-bit bus causes a problem. Click on the button to run a design-rule check. An error is reported in the Console tab:

```
Error:DesignEntry:20 - disp_cnt.sch: Pin 'q(27:0)' is connected to a bus of a different width.
```

You can highlight the place in the schematic where the error occurs by clicking on the disp_cnt.sch text in the error message. The error is caused by the mismatch between the 28-bit counter output and the four-bit bus. This problem is fixed as follows:

1. Click on the selection tool button: 
2. Right-click on the existing bus and select Delete from the pop-up menu to remove this bus.
3. Click on the wiring tool, \( \text{\includegraphics[width=0.05\textwidth]{wiring_icon.png}} \), click on the counter output, and draw a small bus stub outward. Terminate the bus with a double-click. This will create a 28-bit bus connected to the counter outputs.

4. Repeat the previous step to attach a four-bit bus to the LED decoder input.

5. Click on the selection tool, \( \text{\includegraphics[width=0.05\textwidth]{selection_icon.png}} \), and then hover the mouse pointer over the bus connected to the counter. The name of the bus will be shown. In my project, the bus is named \texttt{XLXN_2(27:0)}.

6. Right-click on the bus connected to the LED decoder and select Rename Selected Bus… in the pop-up menu.

7. Rename the LED decoder input bus from \texttt{XLXN_3(3:0)} to \texttt{XLXN_2(27:24)} as shown below and click on the Apply button followed by the OK button. This will connect the LED decoder inputs to the four most-significant bits of the counter output bus.

![Rename Bus dialog box](image-url)
At this point, the schematic should appear as shown below. The design-rule checker should no longer detect any problems.
Now highlight the IO category and select a single-bit output buffer (obuf) from the list of symbols. Attach the output buffer to the output of the LED decoder as shown below. Then run the design-rule checker.
The design-rule checker will detect a width-mismatch error in the above schematic since a single output buffer has been attached to the seven-bit LED decoder output bus. This error can be fixed by right-clicking on the OBUF symbol and selecting Object Properties from the pop-up menu. Then append \((6:0)\) to the instance name of the output buffer as shown below and click on Apply followed by OK. This will expand the single buffer into an array of seven buffers with each one connected to a bit of the LED decoder output bus.
Next, attach a short bus segment to the output of the OBUF. Then click on the button for adding I/O markers. Click on the Add an output marker button in the Options tab of the Processes pane and then click on the free end of the wire segment that you just added.
Clicking on the end of the wire creates a seven bit-wide set of output pins.
The output pins automatically assume the same name as the bus to which they are attached, but this name was automatically generated and doesn’t carry a lot of meaning. To change the name of the outputs (and the associated bus), right-click on the I/O marker and select Object Properties... from the pop-up menu.
The **Object Properties** window allows you to set the name and direction of the pins.

Double-click the existing bus name and replace it with **S(6:0)**. The direction of the bus pins is already set to Output so you can finish by clicking on the **OK** button.
The output pins now appear with their new name.
Once the outputs from the circuit are in place, you can connect a single input I/O marker to the clock input of the counter. (No input buffer is needed because the clock signal will enter the FPGA through a dedicated global clock input.) Right-click (or double-click) on the I/O marker and rename it to `clk`. After this, perform another schematic check to detect any errors and save the schematic using the File ➔ Save command.
Once you save the schematic for the top-level module, the hierarchy in the Sources pane gets updated. Now the **counter** and **ledcd** modules are shown as lower-level modules that are included within the top-level **disp_cnt** module.
Constraining the Design

Before synthesizing the displayable counter, you need to assign pins to the inputs and outputs. Start by right-clicking the disp_cnt object in the Sources pane and selecting New Source... from the pop-up menu.
Select Implementation Constraints File as the type of source file to add and type `disp_cnt` in the File Name field. Then click on the Next button.

You will receive a feedback window that shows the name and type of the file you created and the file to which it is associated. The pin assignments will be made for the top-level module in the design hierarchy, so the constraints file is associated with the `disp_cnt` module. Click on the Finish button to complete the addition of the `disp_cnt.ucf` file to this project.
Now highlight the disp_cnt.sch object in the Sources pane and double-click the Floorplan IO – Pre-Synthesis process to begin adding pin assignment constraints to the design.

The appropriate pin assignments for each model of XSA Board are shown below. The \texttt{clk} input is assigned to a dedicated clock input on each FPGA to which a 50 MHz clock signal is applied. The seven-segment LED pin assignments are the same as in the previous design.

<table>
<thead>
<tr>
<th>I/O Signal</th>
<th>XSA-50</th>
<th>XSA-100</th>
<th>XSA-200</th>
<th>XSA-3S1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{clk}</td>
<td>P88</td>
<td>P88</td>
<td>B8</td>
<td>P8</td>
</tr>
<tr>
<td>s0</td>
<td>P67</td>
<td>P67</td>
<td>N14</td>
<td>M6</td>
</tr>
<tr>
<td>s1</td>
<td>P39</td>
<td>P39</td>
<td>D14</td>
<td>M11</td>
</tr>
<tr>
<td>s2</td>
<td>P62</td>
<td>P62</td>
<td>N16</td>
<td>N6</td>
</tr>
<tr>
<td>s3</td>
<td>P60</td>
<td>P60</td>
<td>M16</td>
<td>R7</td>
</tr>
<tr>
<td>s4</td>
<td>P46</td>
<td>P46</td>
<td>F15</td>
<td>P10</td>
</tr>
<tr>
<td>s5</td>
<td>P57</td>
<td>P57</td>
<td>J16</td>
<td>T7</td>
</tr>
<tr>
<td>s6</td>
<td>P49</td>
<td>P49</td>
<td>G16</td>
<td>R10</td>
</tr>
</tbody>
</table>
In the **Design Object List – I/O Pins** pane of the Xilinx PACE window that appears, set the pin assignments for the clock input and LED segment drivers as shown below. Then save the pin assignments and close the PACE window.
Synthesizing and Implementing the Design

Now you can synthesize the logic circuit netlist and translate, map and place & route it into the FPGA by highlighting the top-level disp_cnt module in the Sources pane and double-clicking the Implement Design process. The software will automatically invoke the synthesizer and then pass the synthesized netlist to the implementation tools. There should be no problems synthesizing and implementing the VHDL, Coregen and schematic files in the design. (There will be a few warnings that you can view in the Warnings tab of the Transcript pane. These warnings are mostly due to the unconnected wires in the 28-bit bus, but this is not an error.)
Checking the Implementation

After the implementation process is done, you can check the logic utilization by clicking on the Design Summary tab (or by double-clicking on the Place & Route Report).

The displayable counter consumes 18 of the 7680 slices in the FPGA. Each slice contains two CLBs, so the displayable counter uses a maximum of 36 CLBs. The 28-bit counter requires at least 28 CLBs and the LED decoder requires 7 CLBs so this totals to 35 CLBs. So the device utilization statistics make sense.
As a precaution, you should also click on the Pinout Report in the Design Summary pane and check that the pin assignments for the clock input and LED decoder outputs match the assignments made with PACE. (You can make the comparison easier by clicking on the Signal Name column header to bring all the signal-pin assignments to the top.)
Checking the Timing

You have the displayable counter synthesized and implemented in the FPGA with the correct pin assignments. But how fast can the counter run? To find out, double-click on the Generate Post-Place & Route Static Timing process. This will determine the maximum delays between logic elements in the design taking into account logic and wiring delays for the routed circuit. (If a green checkmark is already visible, then the static timing analysis has already been done.)

After the static timing delays are calculated, click on the Static Timing Report item in the Design Summary pane to view the results of the analysis. From the information shown at the bottom of the timing report, the minimum clock period for this design is 4.820 ns which means the maximum clock frequency is 207.5 MHz. The clock frequency on the XSA Board is 50 MHz which is well below the maximum allowable frequency for this design.

Generating the Bitstream

Now that you have synthesized our design and mapped it to the FPGA with the correct pin assignments, you are ready to generate the bitstream that is used to program the actual chip.
In this example, rather than use the gxsload utility you will employ the downloading utilities built into ISE. The iMPACT programming tool downloads the bitstream through the JTAG interface of the FPGA, so you need to adjust the way the bitstream is generated to account for this. Right click on the Generate Programming File process and select the Properties... entry from the popup menu.
Select the Startup Options tab of the **Process Properties** window. Change the FPGA Start-Up Clock property to JTAG Clock so the FPGA will react to the clock pulses put out by the iMPACT tool during the final phase of the downloading process. If this option is not selected, the FPGA will not finish its configuration process and it will fail to operate after the downloading completes. Note that the startup clock is only used to complete the configuration process; it has no affect on the clock that is used to drive the actual circuit after the FPGA is configured.
Next, click on the Configuration Options tab and disable all the internal pull-up and pull-down resistors in the FPGA as we did in the previous design. Then click OK.
Now that the bitstream generation options are set, highlight the disp_cnt object in the Sources pane and double-click on the Generate Programming File process to create the bitstream file.
Within a few seconds, a ✔ will appear next to the Generate Programming File process and a file detailing the bitstream generation process will be created. A bitstream file named disp_cnt.bit is placed in the design2 folder.
Downloading the Bitstream

Before downloading the disp_cnt.bit file, you must configure the interface CPLD on the XSA-3S1000 board so it will work with the iMPACT programming tool. Double click the icon and then drag & drop the p3jtag.svf file from the C:\Program Files\XSTOOLS\XSA\3S1000 folder into the FPGA/CPLD pane of the gxsload window. Then click on the Load button and the CPLD will be reprogrammed in less than a minute.

After the p3jtag.svf file is loaded into the XSA Board, move the shunt on jumper J9 from the xs to the xi position. The XSA Board is now setup so the FPGA can be configured through its JTAG boundary-scan pins with the iMPACT programming tool. Note that this process only needs to be done once because the CPLD on the XSA Board will retain its configuration even when power is removed from the board. (If you want to go back to using the gxsload programming utility, you must move the shunt on J9 back to the xs position and download the dwnldpar.svf file into the CPLD.)
Now double-click on the Manage Configuration Project (IMPACT) process.
The **iMPACT – Welcome to iMPACT** window now appears. Make sure the Configure Devices using Boundary-Scan (JTAG) radio button is selected. Boundary-scan mode allows the configuration of multiple FPGAs connected together in a chain. To accomplish this, the iMPACT software needs to know the types of the FPGAs in the chain. There is just a single FPGA on the XSA Board and you could easily describe this to iMPACT. But iMPACT can also probe the boundary-scan chain and automatically identify the types of the FPGAs. This is even easier, so select the Automatically connect to a cable and identify Boundary-Scan chain option and click on the Next button.
The **Assign New Configuration File** window now appears. You need to tell iMPACT what bitstream file to download into the FPGA. Go to the folder where your *design2* project is stored and highlight the `disp_cnt.bit` file. Then click on the **Open** button.

![Assign New Configuration File Window](image)

The **Device Programming Properties** appears next. You can check the Verify box if you want to make iMPACT readback the bitstream from the FPGA after the download to make sure it was sent correctly. Then click on the **OK** button.

![Device Programming Properties Window](image)
The iMPACT software will probe the boundary-scan chain and the main iMPACT window will appear showing a boundary-scan chain consisting of a single XC3S1000 FPGA.
Now right-click on the xc3s1000 icon and select the Program… item on the pop-up menu. This will initiate the download of the bitstream to the FPGA.

The progress of the bitstream download will be displayed. The download operation should take less than a minute.
After the download operation completes, you can check the status messages in the bottom pane of the iMPACT window to see if the FPGA was configured successfully.

**Testing the Circuit**

Once the FPGA on the XSA Board is programmed, the circuit will begin operating without any further action from you. The LED display should repeatedly count through the sequence 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, b, c, d, E, F with a complete cycle taking 5.4 seconds.
State Machine Design

Finite State Machines

A simple finite state machine (FSM) uses one or more flip-flops to store its internal state. The pattern of ones and zeroes on the flip-flop outputs are the current state. In a synchronous FSM, the current state is replaced with the next state on the rising edge of a clock signal. The next state is computed by a combinational logic circuit that accepts the current state and possibly some external signals as inputs. So a synchronous FSM is composed basically of a set of flip-flops fed by a combinatorial circuit that accepts feedback from the flip-flops on every clock cycle.

In this chapter we will build an FSM that acts like a combination lock. The requirements for this digital combination lock are:

1. The user enters a combination as a sequence of $n$ key presses on a keyboard.
2. The combination lock stores a particular combination as a sequence of $n$ key presses.
3. The combination lock will open if the user enters an $n$-key sequence that matches the combination. Otherwise, the lock stays locked.
4. The user must enter an entire $n$-key sequence before the lock either accepts or rejects the sequence.
5. Once the combination lock is unlocked, the user can relock it or enter a new combination as a sequence of key presses.
6. The lock will require the user to verify any new combination that is entered before it replaces the previous combination.

A hierarchical view of the combination lock and its lower-level modules is shown below. The combination lock consists of:

- **Keyboard interface**: This module accepts a serial data stream and clock signal from a standard PS/2 PC keyboard and converts it into a parallel scancode with an associated ready signal that indicates the presence of the scancode.

- **Lock&key mechanism**: This module accepts scancodes from the keyboard interface and determines whether or not the correct combination has been entered and manages the entry of new combinations.

The combination lock accepts the keyboard serial data and clock as inputs along with a main clock that synchronizes the operations of both modules. There is also a reset input to initialize
the entire FSM upon startup. The combination lock visually indicates its current status on a seven-segment LED.

Figure 1: Design hierarchy for a combination lock.
Starting the Combination Lock Project

We will begin the design of the combination lock by creating an HDL-based project for the Spartan3 FPGA as we did before. We will describe the lower-level keyboard interface and the lock&key modules using the HDL Editor, and then tie these modules together with a top-level schematic.

Creating the Keyboard Interface Module

A PS/2 keyboard connects to an XSA Board through two signals:

**psData**: This signal carries the serial data stream as each key is pressed and released. Each key is assigned an eight-bit scancode that is transmitted least-significant bit to most-significant bit with a preceding start bit and a terminating parity bit and stop bit.

**psClk**: The falling edge of this signal indicates when the psData signal is valid.
The keyboard interface will accept the serial data stream and will output the eight-bit scancode in parallel along with an rdy pulse that indicates a valid scancode is available. The rdy pulse will be generated when the psClk signal goes high and stays there. The timing of the psData, psClk, and rdy signals is shown in Figure 2.

![Figure 2: PS/2 keyboard waveforms.](image)

A single scancode is transmitted when a key is pressed. But two scancodes are transmitted when the key is released: an initial scancode of 11110000 to indicate the key release, and then the scancode for the key is sent again. The keyboard interface will be designed such that the rdy signal pulses only after the key has been released.

To begin the keyboard interface, add a VHDL module as shown below.

![New Source Wizard - Select Source Type](image)

The VHDL code for the keyboard interface is shown in Listing 1. The functions of the code for the scancodeReg module are as follows:

**Lines 36–41:** The module receives the psData and psClk inputs from the keyboard and outputs the eight-bit scancode and the rdy signals that were described above. A master clock
is also provided which synchronizes the operations of this module with the lock&key module. A reset signal initializes the module when it first powers up.

**Line 46:** This line declares a 10-bit shift-register that holds the scancode value as it arrives from the keyboard. The start bit, eight scancode bits, parity bit, and stop bit will enter the most-significant bit of the **sc_r** register and shift towards the least-significant bit. By the end of a scancode transmission the start bit will have shifted completely out of the register and be lost while the scancode will end up in the lower eight bits of **sc_r**. The stop and parity bits will be in the uppermost two bits.

**Lines 47–48:** These lines define a counter that is used to determine when the **psClk** signal is no longer pulsing. The timeout value (line 47) is determined by dividing the main clock frequency (defined in the GENERIC section on line 32) by the frequency of the **psClk** (line 33). If the main clock is 50 MHz and the keyboard clock is 10 KHz, then the timeout value is 5000 which means it will take 5000 pulses of the main clock to determine if the **psClk** signal is static. The timeout counter register is defined on line 48 as a natural that can hold a value as large as the timeout value. By defining the counter register in this way, we can change the frequency of the main clock or the keyboard clock and the timeout counter will be automatically resized by the synthesizer with exactly the number of bits needed to store the timeout value.

**Lines 56–66:** This process parallelizes the serial keyboard data. If the reset input is active, the scancode shift register is cleared to all zeroes. Otherwise, on falling edges of the keyboard clock the value on the keyboard data signal is placed into the most-significant bit of the shift register and the upper nine bits of the register are shifted one bit position downward. By the end of a scancode transmission, the start bit will have shifted completely out of the register and be lost while the scancode will end up in the lower eight bits of **sc_r**. The stop and parity bits will be in the uppermost two bits.

**Line 69:** The eight lower bits of the **sc_r** register are output as the scancode output of the module.

**Lines 73–94:** This process detects when the **psClk** signal has stopped pulsing and indicates that a scancode is available. The timeout counter and scancode ready flag are cleared when the module is reset. Then the counter is incremented as long as the **psClk** is at logic 1 and the counter has not reached its timeout value yet. The counter is reset to zero if **psClk** is ever low because that indicates the keyboard clock is still pulsing so the scancode cannot be complete. But if the counter ever reaches the value **TIMEOUT**-1, then the scancode ready flag is pulsed high for a single clock cycle.

**Lines 98–114:** This process checks the **scRdy_r** flag and looks for the scancode that matches the **KEYRELEASE** scancode defined on line 52. After seeing the key release scancode, this process looks for the next following scancode. Then it sets the flag that indicates the scancode received after the **KEYRELEASE** code is ready for output.

**Line 116:** The ready flag from the previous process is output from the module.

**Listing 1:** VHDL code for the keyboard interface.
-- Company: XESS Corp.
-- Engineer: Dave Vanden Bout
--
-- Create Date: 10:39:20 05/21/2006
-- Design Name: design3
-- Module Name: scancodereg - Behavioral
-- Project Name: design3
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity scancodereg is
  generic(
    CLKFREQ   : natural := 50_000;  -- main clock freq (KHz)
    PSCLKFREQ : natural := 10       -- keyboard clock freq (KHz)
  );
  port(
    clk       : in  std_logic;      -- main clock
    rst       : in  std_logic;      -- reset
    psClk     : in  std_logic;      -- keyboard clock
    psData    : in  std_logic;      -- keyboard data
    scancode  : out std_logic_vector(7 downto 0);  -- key scancode
    rdy       : out std_logic       -- true when scancode is ready
  );
end scancodereg;

architecture Behavioral of scancodereg is
  signal   sc_r       : std_logic_vector(9 downto 0);  -- scancode shift reg
  constant TIMEOUT    : natural := CLKFREQ/PSCLKFREQ;  -- psClk quiet timeout
  signal   cnt_r      : natural range 0 to TIMEOUT;    -- timeout counter
  signal   scRdy_r    : std_logic;  -- scan code is ready flag
  signal   rdy_r      : std_logic;  -- output scan code is ready flag
  signal   keyrel_r   : std_logic;  -- key has been released flag
  constant KEYRELEASE : std_logic_vector(7 downto 0) := "11110000";
begin
  -- this process gathers the keybrd scancode into the shift register
  process(psClk, rst)
  begin
    -- async. reset of scancode ready flag
    if rst = '1' then
      sc_r <= (others => '0');
    else
      -- accept keyboard data on falling edge of keyboard clock
      if falling_edge(psClk) then
        -- key data arrives LSB first so right-shift it into MSB of register
sc_r <= psData & sc_r(sc_r'high downto 1);
end if;
end process;

-- key scanocode is in the lower 8-bits of the shift register
scancode <= sc_r(scancode'range); -- output scanocode

-- this process detects the end of the scanocode by looking
-- for the absence of keyboard clock pulses
process(clk, rst)
begin
  if rst = '1' then
    cnt_r <= 0; -- clear the timeout counter
    scRdy_r <= '0'; -- clear the scanocode ready flag
  elsif rising_edge(clk) then
    scRdy_r <= '0'; -- by default, no key scanocode is ready for output
    if psClk = '0' then
      -- reset the timeout register whenever the keyboard clock pulses low
      cnt_r <= 0;
    elsif cnt_r /= TIMEOUT then
      -- increment the timeout counter if the keyboard clock is high
      -- and the counter hasn't reached the timeout value yet
      cnt_r <= cnt_r + 1;
      if cnt_r = TIMEOUT-1 then
        -- signal that a key scanocode is ready when the counter is
        -- equal to one less than the timeout value
        scRdy_r <= '1'; -- rdy signal pulses for one clock period
      end if;
    end if;
  end if;
end if;
end process;

-- this process detects when the keyboard key is released and
-- signals when the scanocode for the released key is ready
process(clk)
begin
  if rising_edge(clk) then
    rdy_r <= '0'; -- by default, no key scanocode is ready for output
    if scRdy_r = '1' then
      -- check the scanocode register when a code is ready
      if sc_r(7 downto 0) = KEYRELEASE then
        -- set flag if the key release prefix is detected
        keyrel_r <= '1';
      elsif keyrel_r = '1' then
        -- end up here on next scanocode received after key release prefix
        rdy_r <= '1'; -- released key scanocode is in the scanocode register
        keyrel_r <= '0'; -- reset the key release flag
      end if;
    end if;
  end if;
end if;
end process;

rdy <= rdy_r; -- signal that a key scanocode is ready
end Behavioral;
Once the VHDL code is entered and saved in the scancodereg.vhd file, you should check it for syntax errors.
Once any syntax errors are corrected, create a schematic symbol for the keyboard interface. You will use that later in the top-level schematic for this project.

Creating the Lock&Key Module

Now we can design the lock&key VHDL module. Add a new VHDL source module called lock and add the code shown in Listing 1.

The lock module follows the basic structure of almost all FSM’s I create. There is a combinational process (lines 106-247) that computes the FSM’s next state given the current state and inputs, and there is an update process (lines 251-277) that loads the computed values into the registers on the next clock edge.

I also follow a register naming convention that appendix _r to a signal or variable name if it stores the current state, and _x is appended to the name if it stores the computed value that will be loaded into the register on the next clock edge. This naming convention also allows me to visually check my VHDL for errors because I know that only state registers ending with _x
should be on the left-hand side of the assignment operator (<=) in the combinatorial process and registers with the _r register should only appear on the right-hand side because this process computes the next state values using the current state values as operands. The situation is reversed in the update process with the _r signals appearing on the left-hand side of the assignment and _x signals on the right.

The lock&key FSM has 5 states (declared on lines 47-51) with the operations in each state handled in a separate section of a case statement in the combinatorial process:

**enterComb (lines 127-150):** Upon entering this state, an $L$ is displayed on the LED to indicate the lock is locked. The value on the LED is incremented as each keypress is received from the user (indicated by the rdy signal going high), and the scancode received from the scancodereg module is compared against the next scancode of the active lock combination stored in the activeComb array. If any key fails to match the corresponding entry in the combination, then the match flag is cleared. Once the last key in the combination is checked (as indicated by the index register that steps through the activeComb array), the FSM transitions to the checkComb state.

**checkComb (lines 154-164):** The match flag is checked in this state. If the flag is set, then the combination was entered correctly and the FSM moves to the unlocked state. Otherwise, it returns to the enterComb state and waits for the user to enter another key sequence.

**unlocked (lines 167-180):** The LED displays a $U$ in this state to indicate the lock is unlocked. If the user presses a backspace key, the FSM transitions to the enterNewComb state so a new combination can be entered. (Thus, the user must know the current combination before he can enter a new combination.) Any other keypress relocks the lock and returns the FSM to the enterComb state.

**enterNewComb (lines 185-204):** The LED displays an $r$ upon entering this state to indicate the combination is being replaced. Each keypress entered in this state is stored in the newComb array and the value on the LED is incremented. After all the keys for a complete combination are entered, the FSM goes to the verifyNewComb state.

**verifyNewComb (lines 209-237):** In this state, the user must re-type the new combination so it can be compared against the contents of the newComb array. If any mismatch occurs, the FSM immediately goes to the enterComb state and relocks the lock without changing the combination. If the new combination is repeated correctly, the contents of the newComb array are transferred to the activeComb array and the lock is relocked with the new combination.

The reset of the FSM and the actual state transitions is handled by the update process on each rising edge of the master clock:

**Lines 255–265:** The FSM is synchronously reset if the reset input (rst) is high or if the power-on reset flag (rst_i) is set. The power-on reset flag is initially set on line 67 so the FSM is guaranteed to be reset whenever power is first applied to the FPGA. The FSM always resets to the enterComb state and the lock is locked. If a power-on reset has occurred, then the active combination is not defined so it must be loaded with a default combination (defined as the key sequence “123456789” on lines 61-63). A reset caused by the reset input, however, will not reload the active combination so it will remain at whatever key sequence was set by the user.
Lines 267-276: When the FSM is not being reset, then all the state registers are loaded with their next values here.

There are a few other points to note about the lock module:

- The FSM requires a single clock cycle duration for the scancode ready signal. If the rdy output from the keyboard interface module stayed high for more than a single clock cycle on each key press, this would cause a transition between multiple states of the FSM.

- The COMB_LENGTH generic parameter on line 32 can be set to any value between 1 and 9 to change the number of keys in the combination. The use of an array to store the combination (lines 56-58) allows the module to resize itself automatically if you decide to change the length of the combination.

- The VHDL synthesizer automatically generates the bit encodings for the FSM states on lines 47-51. If you want to explicitly code the states, you can place lines such as the following below the state register declaration on line 53:

  ```vhdl
  attribute ENUM_ENCODING: STRING;
  attribute ENUM_ENCODING of lockStateType:type is
  "00001 00010 00100 01000 10000";
  ```

Next, highlight the scancodereg module in the Sources pane (since this was the first module added to the project, it is the top-level module by default), and then right-click on the Synthesize process and select Properties from the pop-up menu:
Then select User in the FSM Encoding Algorithm field:

![Process Properties - HDL Options](image)

Now your explicit state encodings will be used. You can check this by looking in the synthesis report to see:

```
* Advanced HDL Synthesis *

Optimizing FSM <XLXI_2/lockState_r> on signal <lockState_r[1:5]> with one-hot encoding.

<table>
<thead>
<tr>
<th>State</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>entercomb</td>
<td>00001</td>
</tr>
<tr>
<td>checkcomb</td>
<td>00010</td>
</tr>
<tr>
<td>unlocked</td>
<td>00100</td>
</tr>
<tr>
<td>enternewcomb</td>
<td>01000</td>
</tr>
<tr>
<td>verifynewcomb</td>
<td>10000</td>
</tr>
</tbody>
</table>
```

Listing 2: VHDL code for the lock & key module.

```
-- Company:
-- Engineer:
-- Create Date: 09:36:57 06/03/2006
-- Design Name:
-- Module Name: lock - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
-- Dependencies:
-- Revision:
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity lock is
  Generic (  
    COMB_LENGTH : natural := 2  -- number of scancodes in a combination  
  );
  Port (     
    rst : in STD_LOGIC;   -- reset  
    clk : in  STD_LOGIC;  -- master clock  
    rdy : in  STD_LOGIC;  -- true when a new keystroke is available  
    sc  : in  STD_LOGIC_VECTOR (7 downto 0);  -- scancode for keystroke  
    led : out  STD_LOGIC_VECTOR (6 downto 0)  -- LED status indicators  
  );
end lock;

architecture Behavioral of lock is
  -- State definitions for the FSM.
  type lockStateType is (  
    enterComb,      -- user enters keystrokes which are checked against stored comb.  
    checkComb,      -- check the entered keystrokes to see if they match the comb.  
    unlocked,       -- user's keystrokes match combination, so unlock the lock  
    enterNewComb,   -- user enters a new combination  
    verifyNewComb   -- user re-types the new combination to verify it  
  );
  signal lockState_r, lockState_x: lockStateType;  -- current and next FSM state
  
  -- The key scancodes composing a combination are stored in an array.
  type combinationType is ARRAY (0 to COMB_LENGTH-1) of std_logic_vector(sc'range);
  signal activeComb_r, activeComb_x : combinationType;  -- active combination
  signal newComb_r, newComb_x : combinationType;  -- new combination
  
  -- The default combination is an array the size of the longest allowable
  -- combination filled with the scancodes for the keys "1","2","3",...,"9".
  type defaultCombinationType is ARRAY (0 to 8) of std_logic_vector(sc'range);
  constant defaultComb : defaultCombinationType :=
    (x"16",x"1e",x"26",x"25",x"2e",x"36",x"3d",x"3e",x"46");
  
  signal index_r, index_x : natural range COMB_LENGTH-1 downto 0; -- comb. array index
  signal match_r, match_x : std_logic; -- true when user keystrokes = active comb.
  signal rst_i            : std_logic := '1'; -- power-on reset flag
  
  -- Scancode for the key that initiates the entry of a new combination.
  constant NEW_COMB_KEY: std_logic_vector(sc'range) := "01100110"; -- backspace

  -- Function that displays a hex digit on the 7-segment LED.
  subtype ledOutputType is std_logic_vector(6 downto 0);
  function ledDecoder (digit: natural) return ledOutputType is
    begin
      case digit is
        when 0  => return "1110111";  -- 0
        when 1  => return "0010010";  -- 1
        when 2  => return "1101111";  -- 2
        when 3  => return "1011011";  -- 3
        when 4  => return "0111010";  -- 4
        when 5  => return "1101111";  -- 5
        when 6  => return "1011011";  -- 6
        when 7  => return "1001001";  -- 7
        when 8  => return "1111111";  -- 8
        when 9  => return "1111011";  -- 9
      end case;
when 10 => return "1111110";    -- A
when 11 => return "0101111";    -- b
when 12 => return "0001101";    -- c
when 13 => return "0011111";    -- d
when 14 => return "1101101";    -- E
when 15 => return "1101100";    -- F
when others => return "1101101";
end case;
end ledDecoder;

-- 7-segment LED output for various conditions
constant LED_LOCKED        : ledOutputType := "0100101"; -- "L" - locked
constant LED_UNLOCKED      : ledOutputType := "0110111"; -- "U" - unlocked
constant LED_ENTERNEWCOMB  : ledOutputType := "0001100"; -- "r" - replace comb
constant LED_VERIFYNEWCOMB : ledOutputType := "1100101"; -- "C" - check comb

begin

-- this process determines the next state given the current state and inputs
combinatorial: process(rdy,sc,lockState_r,index_r,match_r,activeComb_r,newComb_r)
begins

-- set default values for outputs and the next states of the registers
led         <= (others=>'0'); -- turn off all LEDs
match_x     <= match_r;       -- keep current value
index_x     <= index_r;       -- keep current value
lockState_x <= lockState_r;   -- keep in current state
for i in 0 to COMB_LENGTH-1 loop
-- keep combinations unchanged
  activeComb_x(i) <= activeComb_r(i);
  newComb_x(i)    <= newComb_r(i);
end loop;

-- determine the next state and the outputs
end case:

lockState_r is

-- FSM is in this state when the user is entering the combination
-- to unlock the lock. The index_r register points to the location
-- of the scancode in the combination that is being compared to the
-- current scancode entered by the user.
when enterComb =>
-- Show an "L" on the 7-segment LED before the user has pressed any keys.
-- After that, indicate the number of keypresses received from the user.
  if index_r = 0 then
    led <= LED_LOCKED; -- show "L" on LED to indicate lock is locked
  else
    led <= ledDecoder(index_r); -- index into comb. array = # of user keypresses
  end if;
-- Wait until a new key scancode arrives.
  if rdy = '1' then
    -- Clear the match flag if the current scancode fails to match the
    -- current scancode in the combination.
    if sc /= activeComb_r(index_r) then
      match_x <= '0';
    end if;
    -- Increment the index if the complete combination hasn't been entered yet.
    if index_r /= COMB_LENGTH-1 then
      index_x <= index_r + 1;
    else
      lockState_x <= checkComb;
    end if;
  end if;
-- FSM checks the match flag in this state to see if the user entered the
-- correct combination.
when checkComb =>
-- Reset the index and match flag.
  index_x <= 0;
  match_x <= '1';
-- If the match flag is still set, then all the user-entered scancodes
-- matched the active combination, so transition to the unlocked state.
if match_r = '1' then
    lockState_x <= unlocked; -- keys match combination, so unlock the lock
else
    lockState_x <= enterComb; -- otherwise, wait for another attempt
end if;

-- The lock is unlocked when the FSM is in this state.
when unlocked =>
    led <= LED_UNLOCKED; -- show "U" on LED to indicate lock is unlocked
-- Wait until a new key scanocode arrives.
    if rdy = '1' then
      -- After the active combination is entered, the user can hit the
      -- backspace key to move to the state that allow entry of a new comb.
      if sc = NEW_COMB_KEY then
        lockState_x <= enterNewComb;
      -- Otherwise, any other key will re-lock the lock and return to the state
      -- that checks for the active combination to be entered.
      else
        lockState_x <= enterComb;
      end if;
    end if;
end;

-- FSM is in this state when the user is entering a new combination
-- to replace the current active combination. The index_r register points to
-- the location where the next scancode will be stored in the new combination.
when enterNewComb =>
  -- Show an "r" on the 7-segment LED before the user has pressed any keys.
  -- After that, indicate the number of keypresses received from the user.
  if index_r = 0 then
    led <= LED_ENTERNEWCOMB; -- show "r" on LED to indicate comb. replacement
  else
    led <= ledDecoder(index_r); -- index into comb. array = # of user keypresses
  end if;
  -- Wait until a new key scanocode arrives.
  if rdy = '1' then
    newComb_x(index_r) <= sc; -- store current scancode into new combination array
    -- Increment the index if the new combination isn't complete yet.
    if index_r /= COMB_LENGTH-1 then
      index_x <= index_r + 1;
    else
      index_x <= 0;
      lockState_x <= verifyNewComb;
    end if;
  end if;
end;

-- FSM is in this state when the user is re-entering the new combination
-- to verify it against what was stored. The index_r register points to
-- the location of the next scancode that will be compared in the new comb.
when verifyNewComb =>
  -- Show a "C" on the 7-segment LED before the user has pressed any keys.
  -- After that, indicate the number of keypresses received from the user.
  if index_r = 0 then
    led <= LED_VERIFYNEWCOMB; -- show "C" on LED to indicate check of new comb.
  else
    led <= ledDecoder(index_r); -- index into comb. array = # of user keypresses
  end if;
  -- Wait until a new key scanocode arrives.
  if rdy = '1' then
    -- Abort the operation as soon as a scancode does not match the new comb.
    if sc /= newComb_r(index_r) then
      index_x <= 0; -- reset the index
      lockState_x <= enterComb; -- go wait for the user to enter the active comb.
    else
      -- Increment the index if the new comb. verification isn't complete yet.
      if index_r /= COMB_LENGTH-1 then
        index_x <= index_r + 1;
      -- Otherwise, the user has correctly re-entered the new comb. so overwrite
      -- the active comb. with the new comb. and return to the locked state.
      else
        lockState_x <= unlocked;
      end if;
    end if;
  end if;
end;
else
  for i in 0 to COMB_LENGTH-1 loop
    activeComb_x(i) <= newComb_r(i);
  end loop;
  index_x <= 0;
  lockState_x <= enterComb;
end if;
end if;
end if;
end if;

-- Something wrong has happened if the FSM ever gets into this state, so
-- reset into the locked state.
when others =>
  index_x <= 0;
  match_x <= '1';
  lockState_x <= enterComb;
end case;
end process;

-- This process just updates the various state registers with their next values
-- as computed by the previous process. It also handles the reset of the FSM.
update: process(rst,clk)
begin
  if rising_edge(clk) then
    -- Handle the reset input or the power-on reset.
    if rst='1' or rst_i='1' then
      rst_i   <= '0'; -- clear power-on reset
      index_r <= 0;
      match_r <= '1';
      lockState_r <= enterComb;
      -- On power-on, initialize the active combination from the default value.
      if rst_i = '1' then
        for i in 0 to COMB_LENGTH-1 loop
          activeComb_r(i) <= defaultComb(i);
        end loop;
      end if;
      -- Otherwise, update all registers with their next values.
      else
        index_r <= index_x;
        match_r <= match_x;
        lockState_r <= lockState_x;
        for i in 0 to COMB_LENGTH-1 loop
          activeComb_r(i) <= activeComb_x(i);
        end loop;
      end if;
    end if;
  else
    index_r <= index_x;
    match_r <= match_x;
    lockState_r <= lockState_x;
    for i in 0 to COMB_LENGTH-1 loop
      activeComb_r(i) <= activeComb_x(i);
    end loop;
  end if;
end process;
end Behavioral;

Once the VHDL for the lock module is entered, run a syntax check and generate a schematic symbol as you did for the scancodereg module.

The VHDL description of the FSM can be useful for two reasons:

1. The editing area of the State Editor window gets very cluttered for complicated FSMs. You can use the State Editor to draw an initial, simplified version of your FSM and then add the rest of your description directly to the VHDL file. You cannot automatically back-annotate the additions to the VHDL file back into the State Editor, so the VHDL file must be used as the master design file for the FSM after you do this.

2. If you are unsure how to write FSM descriptions using VHDL, you can create simple FSMs in the State Editor and export them as VHDL to view the basic language constructs that are used.
Creating the Top-Level Module

The top-level module of the combination lock is built by connecting the keyboard interface and the lock&key modules together in a schematic. Right-click on the Sources pane and select New Source... from the pop-up menu. Then create a new schematic named combLock as shown below:

Enter the following components and connections into the schematic editor window:
Note the following:

1. The main clock (CLK) enters the FPGA on a dedicated clock pin (because that is the way it is connected on the XSA Board) so the input pad can connect directly to a general clock buffer (BUFG). Using the BUFG ensures that the clock signal reaches all the flip-flops in the design with minimal skew so they all change state at the same time.

2. The clock from the PS/2 keyboard (PSCLK) enters on a generic I/O pin so it must go through an input buffer (IBUF) before going through a BUFG.

3. The keyboard serial data signal (PSDATA) and the reset signal are standard, non-clock inputs so they just connect to IBUFs.

4. The reset signal (RST_N) is sourced by an active-low DIP-switch button on the XSA Board, so it passes through an inverter before going to the active-high reset inputs of the scancodereg and lock modules.

5. An array of seven output buffers (OBUF) as was created as in the design example of the previous chapter. The seven-bit led bus of the lock module connects to the inputs of the output buffers, and the buffer outputs connect to the output pads.

Once the components are connected to each other and the I/O, run a design-rule check to make sure there are no errors and then save the schematic. Then, make the schematic the top-level module of the design by right-clicking it in the Sources pane and selecting Set as Top Module from the pop-up menu.
Constraining the Design

Now you need to assign pins to the inputs and outputs, either by using PACE or by entering the pin assignments directly into the \texttt{combLock.ucf} constraints file. The appropriate pin assignments for each model of XSA Board are shown below. The \texttt{rst\_n} input is driven by DIP-switch SW1-1 on the XSA Board. The \texttt{clk} input is assigned to a dedicated clock input on each FPGA to which a 50 MHz clock signal is applied. The \texttt{psclk} and \texttt{psdata} inputs are attached to the equivalent pins on the PS/2 port. The seven-segment LED pin assignments are the same as in the previous designs.

<table>
<thead>
<tr>
<th>I/O Signal</th>
<th>XSA-50</th>
<th>XSA-100</th>
<th>XSA-200</th>
<th>XSA-3S1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>rst_n</td>
<td>P54</td>
<td>P54</td>
<td>P11</td>
<td>K4</td>
</tr>
<tr>
<td>clk</td>
<td>P88</td>
<td>P88</td>
<td>B8</td>
<td>P8</td>
</tr>
<tr>
<td>psclk</td>
<td>P94</td>
<td>P94</td>
<td>F4</td>
<td>B16</td>
</tr>
<tr>
<td>psdata</td>
<td>P93</td>
<td>P93</td>
<td>E1</td>
<td>E13</td>
</tr>
<tr>
<td>s0</td>
<td>P67</td>
<td>P67</td>
<td>N14</td>
<td>M6</td>
</tr>
<tr>
<td>s1</td>
<td>P39</td>
<td>P39</td>
<td>D14</td>
<td>M11</td>
</tr>
<tr>
<td>s2</td>
<td>P62</td>
<td>P62</td>
<td>N16</td>
<td>N6</td>
</tr>
<tr>
<td>s3</td>
<td>P60</td>
<td>P60</td>
<td>M16</td>
<td>R7</td>
</tr>
<tr>
<td>s4</td>
<td>P46</td>
<td>P46</td>
<td>F15</td>
<td>P10</td>
</tr>
<tr>
<td>s5</td>
<td>P57</td>
<td>P57</td>
<td>J16</td>
<td>T7</td>
</tr>
<tr>
<td>s6</td>
<td>P49</td>
<td>P49</td>
<td>G16</td>
<td>R10</td>
</tr>
</tbody>
</table>

Implementing the Design and Generating the Bitstream

Once you have specified the correct pin assignments, just double-click the \texttt{Generate Programming File} in the Process pane. Unfortunately, instead of a bitstream, you’ll probably get an error like this:

\texttt{ERROR:Place:1018 - A clock IOB/clock component pair have been found that are not placed at an optimal clock IOB/clock site pair. The clock component <XLXI_5> is placed at site <BUFGMUX5>. The IO component <PSCLK> is placed at site <B16>. This will not allow the use of the fast path between the IO and the Clock buffer. If this sub optimal condition is acceptable for this design, you may use the CLOCK_DEDICATED_ROUTE constraint in the \texttt{.ucf} file to demote this message to a WARNING and allow your design to continue. However, the use of this override is highly discouraged as it may lead to very poor timing results. It is recommended that this error condition be corrected in the design. A list of all the COMP\_PINs used in this clock placement rule is listed below. These examples can be used directly in the \texttt{.ucf} file to override this clock rule.}

\texttt{< NET "PSCLK" CLOCK_DEDICATED_ROUTE = FALSE; >}

The Xilinx software tends to balk at using general-purpose inputs as clock sources for registers, and that’s exactly what is being done with the clock from the PS/2 keyboard. The PS/2 clock has a very low frequency and it runs asynchronously with respect to the main logic clock, so it doesn’t really matter that the PS/2 clock routing is sub-optimal. You can get around this error by placing the constraint listed above (strip off the ‘<’ and ‘>’ delimiters) into the \texttt{combblock.ucf}
Then you can re-run the synthesis and implementation processes to get an FPGA bitstream in the `comblock.bit` file.

## Testing the Combination Lock

Attach a PS/2 keyboard to the six-pin mini-DIN socket at the bottom of the XSA Board. Then use GXSLOAD to download the bitstream to the XSA Board. Make sure DIP-switch SW1-1 is in the OFF position so the `rst_n` input is high. Now the combination lock should be ready to respond to key presses. A sequence of key presses and the results are shown below:

<table>
<thead>
<tr>
<th>Press key...</th>
<th>LED displays...</th>
<th>New State...</th>
<th>This means...</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>![LED display]</td>
<td>enterComb</td>
<td>The combination lock is locked and is waiting for the default combination to be entered from the keyboard.</td>
</tr>
<tr>
<td>1</td>
<td>![LED display]</td>
<td>enterComb</td>
<td>The first key of the default combination was entered.</td>
</tr>
<tr>
<td>2</td>
<td>![LED display]</td>
<td>checkComb → unlocked</td>
<td>The second key of the default combination was entered. The lock is now unlocked</td>
</tr>
<tr>
<td>backspace</td>
<td>![LED display]</td>
<td>enterNewComb</td>
<td>The backspace initiates the replacement of the current combination with a new one.</td>
</tr>
<tr>
<td>q</td>
<td>![LED display]</td>
<td>enterNewComb</td>
<td>The first key of the new was entered.</td>
</tr>
<tr>
<td>w</td>
<td>![LED display]</td>
<td>verifyNewComb</td>
<td>The second key of the new combination was entered. Now the combination must be re-typed to check it.</td>
</tr>
<tr>
<td>q</td>
<td>![LED display]</td>
<td>verifyNewComb</td>
<td>The first key of the combination was re-typed correctly.</td>
</tr>
<tr>
<td>w</td>
<td>![LED display]</td>
<td>enterComb</td>
<td>The second key of the combination was entered correctly and the lock is now waiting for the new combination to be entered.</td>
</tr>
<tr>
<td>1</td>
<td>![LED display]</td>
<td>enterComb</td>
<td>The first key of the previous combination was entered.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>enterComb</td>
<td>The second key of the previous combination was entered and the lock remains locked because this combination is no longer active.</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>-----------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>2</td>
<td>L</td>
<td>enterComb</td>
<td>The first key of the new combination was entered.</td>
</tr>
<tr>
<td>q</td>
<td>l</td>
<td>checkComb</td>
<td>The second key of the new combination was entered. The lock is now unlocked.</td>
</tr>
</tbody>
</table>
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:

- Select Help ➔ Software Manuals. You will be presented with an Adobe Acrobat document that lists all the manuals for the ISE software. This includes a 300-page set of guidelines on synthesis and simulation techniques for FPGA designs.

- Select Help ➔ Xilinx on the Web ➔ Xilinx Application Notes. This will take you to a large set of interesting designs that have been done using Xilinx FPGAs.


- Read the comp.arch.fpga newsgroup for helpful questions and answers about programmable logic design.