CPLD Synthesis Design Guide

Getting Started with Synopsys for CPLDs

Designing with CPLDs

Compiling and Fitting a CPLD Design

Simulating your Design

Library Component Specifications

Attributes

Fitter Command and Option Summary

 Printed in U.S.A.

The Xilinx logo shown above is a registered trademark of Xilinx, Inc.

FPGA Architect, FPGA Foundry, NeoCAD, NeoCAD EPIC, NeoCAD PRISM, NeoROUTE, Timing Wizard, TRACE, XACT, XILINX, XC2064, XC3090, XC4005, XC5210, and XC-DS501 are registered trademarks of Xilinx, Inc.

The shadow X shown above is a trademark of Xilinx, Inc.

All XC-prefix product designations, A.K.A. Speed, Alliance Series, AllianceCORE, BITA, CLC, Configurable Logic Cell, CORE Generator, CoreGenerator, CoreLINX, Dual Block, EZTag, FastCLK, FastCONNECT, FastFLASH, FastMap, Foundation, HardWire, LCA, LogiBLOX, Logic Cell, LogiCORE, LogicProfessor, MicroVia, PLUSASM, PowerGuide, PowerMaze, QPro, RealPCI, RealPCI 64/66, SelectI/O, Select-RAM, Select-RAM+, Smartguide, Smart-IP, SmartSearch, Smartspec, SMARTSwitch, Spartan, TrueMap, UIM, VectorMaze, VersaBlock, VersaRing, Virtex, WebLINX, XABEL, XACT*step*, XACT*step* Advanced, XACT*step* Foundry, XACT-Floorplanner, XACT-Performance, XAM, XAPP, X-BLOX, X-BLOX plus, XChecker, XDM, XDS, XEPLD, Xilinx Foundation Series, XPP, XSI, and ZERO+ are trademarks of Xilinx, Inc. The Programmable Logic Company and The Programmable Gate Array Company are service marks of Xilinx, Inc.

All other trademarks are the property of their respective owners.

Xilinx, Inc. does not assume any liability arising out of the application or use of any product described or shown herein; nor does it convey any license under its patents, copyrights, or maskwork rights or any rights of others. Xilinx, Inc. reserves the right to make changes, at any time, in order to improve reliability, function or design and to supply the best product possible. Xilinx, Inc. will not assume responsibility for the use of any circuitry described herein other than circuitry entirely embodied in its products. Xilinx, Inc. devices and products are protected under one or more of the following U.S. Patents: 4,642,487; 4,695,740; 4,706,216; 4,713,557; 4,746,822; 4,750,155; 4,758,985; 4,820,937; 4,821,233; 4,835,418; 4,855,619; 4,855,669; 4,902,910; 4,940,909; 4,967,107; 5,012,135; 5,023,606; 5,028,821; 5,047,710; 5,068,603; 5,140,193; 5,148,390; 5,155,432; 5,166,858; 5,224,056; 5,243,238; 5,245,277; 5,267,187; 5,291,079; 5,295,090; 5,302,866; 5,319,252; 5,319,254; 5,321,704; 5,329,174; 5,329,181; 5,331,220; 5,331,226; 5,332,929; 5,337,255; 5,343,406; 5,349,248; 5,349,249; 5,349,250; 5,349,691; 5,357,153; 5,360,747; 5,361,229; 5,362,999; 5,365,125; 5,367,207; 5,386,154; 5,394,104; 5,399,924; 5,399,925; 5,410,189; 5,410,194; 5,414,377; 5,422,833; 5,426,378; 5,426,379; 5,430,687; 5,432,719; 5,448,181; 5,448,493; 5,450,021; 5,450,022; 5,453,706; 5,455,525; 5,466,117; 5,469,003; 5,475,253; 5,477,414; 5,481,206; 5,483,478; 5,486,707; 5,486,776; 5,488,316; 5,489,858; 5,489,866; 5,491,353; 5,495,196; 5,498,979; 5,498,989; 5,499,192; 5,500,608; 5,500,609; 5,502,000; 5,502,440; 5,504,439; 5,506,518; 5,506,523; 5,506,878; 5,513,124; 5,517,135; 5,521,835; 5,521,837; 5,523,963; 5,523,971; 5,524,097; 5,526,322; 5,528,169; 5,528,176; 5,530,378; 5,530,384; 5,546,018; 5,550,839; 5,550,843; 5,552,722; 5,553,001; 5,559,751; 5,561,367; 5,561,629; 5,561,631; 5,563,527; 5,563,528; 5,563,529; 5,563,827; 5,565,792; 5,566,123; 5,570,051; 5,574,634; 5,574,655; 5,578,946; 5,581,198; 5,581,199; 5,581,738; 5,583,450; 5,583,452; 5,592,105; 5,594,367; 5,598,424; 5,600,263; 5,600,264; 5,600,271; 5,600,597; 5,608,342; 5,610,536; 5,610,790; 5,610,829; 5,612,633; 5,617,021; 5,617,041; 5,617,327; 5,617,573; 5,623,387; 5,627,480; 5,629,637; 5,629,886; 5,631,577; 5,631,583; 5,635,851; 5,636,368; 5,640,106; 5,642,058; 5,646,545; 5,646,547; 5,646,564; 5,646,903; 5,648,732; 5,648,913; 5,650,672; 5,650,946; 5,652,904; 5,654,631; 5,656,950; 5,657,290; 5,659,484; 5,661,660; 5,661,685; 5,670,896; 5,670,897; 5,672,966; 5,673,198; 5,675,262; 5,675,270; 5,675,589; 5,677,638; 5,682,107; 5,689,133; 5,689,516; 5,691,907; 5,691,912; 5,694,047; 5,694,056; 5,724,276; 5,694,399; 5,696,454; 5,701,091; 5,701,441; 5,703,759; 5,705,932; 5,705,938; 5,708,597; 5,712,579; 5,715,197; 5,717,340; 5,719,506; 5,719,507; 5,724,276; 5,726,484; 5,726,584; 5,734,866; 5,734,868; 5,737,234; 5,737,235; 5,737,631; 5,742,178; 5,742,531; 5,744,974; 5,744,979; 5,744,995; 5,748,942; 5,748,979; 5,752,006; 5,752,035; 5,754,459; 5,758,192; 5,760,603; 5,760,604; 5,760,607; 5,761,483; 5,764,076; 5,764,534; 5,764,564; 5,768,179; 5,770,951; 5,773,993; 5,778,439; 5,781,756; 5,784,313; 5,784,577; 5,786,240; 5,787,007; 5,789,938; 5,790,479;

5,790,882; 5,795,068; 5,796,269; 5,798,656; 5,801,546; 5,801,547; 5,801,548; 5,811,985; 5,815,004; 5,815,016; 5,815,404; 5,815,405; 5,818,255; 5,818,730; 5,821,772; 5,821,774; 5,825,202; 5,825,662; 5,825,787; 5,828,230; 5,828,231; 5,828,236; 5,828,608; 5,831,448; 5,831,460; 5,831,845; 5,831,907; 5,835,402; 5,838,167; 5,838,901; 5,838,954; 5,841,296; 5,841,867; 5,844,422; 5,844,424; 5,844,829; 5,844,844; 5,847,577; 5,847,579; 5,847,580; 5,847,993; 5,852,323; Re. 34,363, Re. 34,444, and Re. 34,808. Other U.S. and foreign patents pending. Xilinx, Inc. does not represent that devices shown or products described herein are free from patent infringement or from any other third party right. Xilinx, Inc. assumes no obligation to correct any errors contained herein or to advise any user of this text of any correction if such be made. Xilinx, Inc. will not assume any liability for the accuracy or correctness of any engineering or software support or assistance provided to a user.

Xilinx products are not intended for use in life support appliances, devices, or systems. Use of a Xilinx product in such applications without the written consent of the appropriate Xilinx officer is prohibited.

Copyright 1991-1999 Xilinx, Inc. All Rights Reserved.

About This Manual

This manual has been created to provide guidance in use of Synthesis Design for XC9500, XC9500XL, and XC9500XV CPLDs in the workstation environment. Practical examples in this manual apply to the Synopsys compiler and simulator.

Additional Resources

For additional information, go to http://support.xilinx.com. Use the search function at the top of the support.xilinx.com page or click links that take you directly to online resources.

The following table provides information on tutorials and some of the resources you can access using the support.xilinx.com advanced Answers Search function.

Resource	Description/URL
Tutorial	Tutorials covering Xilinx design flows, from design entry to verification and debugging http://support.xilinx.com/support/techsup/tutorials/index.htm
Answers Database	Current listing of solution records for the Xilinx software tools Search this database using the search function at http://support.xilinx.com/support/searchtd.htm
Application Notes	Descriptions of device-specific design techniques and approaches http://www.support.xilinx.com/apps/appsweb.htm
Data Book	Pages from *The Programmable Logic Data Book*, which describe device-specific information on Xilinx device characteristics, including readback, boundary scan, configuration, length count, and debugging http://www.support.xilinx.com/partinfo/databook.htm

Resource	Description/URL
Xcell Journals	Quarterly journals for Xilinx programmable logic users http://www.support.xilinx.com/xcell/xcell.htm
Tech Tips	Latest news, design tips, and patch information on the Xilinx design environment http://www.support.xilinx.com/support/techsup/journals/index.htm

Manual Contents

This manual covers the following topics.

- "Getting Started with Synopsys for CPLDs" chapter. This chapter shows you how to prepare your setup files and verify your installation. It also provides a design walk-through as an overview of the basic steps for implementing Xilinx XC9000 CPLD designs using Synopsys.
- "Designing with CPLDs" chapter. This chapter discusses how to use design techniques, library cells and **cpld** command parameters to get the best performance from Xilinx XC9000 CPLDs.
- "Compiling and Fitting a CPLD Design" chapter. This chapter describes how to compile your design using the Synopsys Design Compiler shell (DC Shell).
- "Simulating your Design" chapter. The Xilinx CPLD Synopsys Interface supports both functional and timing simulation of VHDL designs using the VSS simulator. This chapter shows you how to prepare designs for simulation and how to use a test bench.
- "Library Component Specifications" appendix lists library components used in CPLD designs.
- "Attributes" appendix lists attributes used in CPLD designs.
- "Fitter Command and Option Summary" appendix lists fitter options entered from the Design Manager, and lists fitter options entered on the **cpld** command line.

Conventions

This manual uses the following typographical and online document conventions. An example illustrates each typographical convention.

Typographical

The following conventions are used for all documents.

- `Courier font` indicates messages, prompts, and program files that the system displays.

  `speed grade: -100`

- **`Courier bold`** indicates literal commands that you enter in a syntactical statement. However, braces "{ }" in Courier bold are not literal and square brackets "[ ]" in Courier bold are literal only in the case of bus specifications, such as bus [7:0].

  `rpt_del_net=`

  `Courier bold` also indicates commands that you select from a menu.

  `File` → **`Open`**

- *Italic font* denotes the following items.

  - Variables in a syntax statement for which you must supply values

    `edif2ngd` *design_name*

  - References to other manuals

    See the *Development System Reference Guide* for more information.

- Emphasis in text

  If a wire is drawn so that it overlaps the pin of a symbol, the two nets are *not* connected.

- Square brackets “[ ]” indicate an optional entry or parameter. However, in bus specifications, such as bus [7:0], they are required.

  `edif2ngd` [*option_name*] *design_name*

- Braces “{ }” enclose a list of items from which you must choose one or more.

  `lowpwr ={on|off}`

- A vertical bar “ | ” separates items in a list of choices.

  `lowpwr ={on|off}`

- A vertical ellipsis indicates repetitive material that has been omitted.

```
IOB #1: Name = QOUT'
IOB #2: Name = CLKIN'
.
.
.
```

- A horizontal ellipsis “. . .” indicates that an item can be repeated one or more times.

  `allow block` ***block_name loc1 loc2*** `. . .` ***locn;***

Online Document

The following conventions are used for online documents.

- Red-underlined text indicates an interbook link, which is a cross-reference to another book. Click the red-underlined text to open the specified cross-reference.
- Blue-underlined text indicates an intrabook link, which is a cross-reference within a book. Click the blue-underlined text to open the specified cross-reference.

Contents

Additional Resources v
Manual Contents vi
Typographical vii
Online Document viii

Chapter 1 Getting Started for Synopsys for CPLDs 1-1

Workstation Environment 1-1
Creating Synopsys Setup Files (Workstation) 1-2
The Design Compiler Setup File 1-2
The VSS Simulator Setup File 1-3
Analyzing the DesignWare and Simulation Libraries 1-3
Verifying Your Installation (Workstation) 1-4
Verifying Synopsys Software Access 1-4
Verifying Xilinx Software Access 1-4
Xilinx CPLD Design Flow 1-5
Design Example 1-7
Design Entry 1-11
Step1 - Create a Design Directory 1-11
Functional Simulation 1-12
Step 2 - Analyze Your Design 1-12
Step 3 - Analyze Your Test Bench 1-12
Step 4 - Invoke the Simulator 1-16
Step 5 - Select the Design 1-17
Step 6 - Trace Signals 1-17
Step 7 - Run the Simulation 1-17
Step 8 - Return to UNIX 1-17
Synthesizing Your Design (Compiling) 1-18
Step 9 - Enter the DC Shell Environment 1-18
Step 10 - Analyze Your Source Design 1-18
Step 11 - Elaborate Your Design 1-19
Step 12 - Synthesize Your Design 1-19

Step 13 - Specify Initial Register States 1-19
Step 14 - Specify Timing Constraints 1-20
Step 15 - Use a Global Clock Input Port 1-20
Step 16 - Place I/O Buffer Cells 1-20
Step 17 - Flatten the Design 1-20
Step 18 - Output the Netlist 1-20
Step 19 - Output Timing Constraints 1-21
Step 20 - Exit DC Shell 1-21
Step 21 -Translate Timing Constraints File 1-21
Fitting Your Design 1-21
Step 22 - Fit Your Design 1-22
Timing Simulation 1-22
Step 23 - Prepare A Timing Simulation File 1-22
Step 24 - Analyze Your Implemented Design 1-23
Step 25 - Analyze Your Test Bench 1-23
Step 26 - Invoke the VSS Simulator 1-23
Step 27 - Open the Waveform Viewer 1-23
Step 28 - Run the Simulation 1-24
Step 29 - Return to UNIX 1-24

Chapter 2 Designing with CPLDs 2-2

Target Device Selection 2-2
Selecting a Part from the Design Manager 2-2
Family 2-3
Device 2-3
Package 2-3
Speed Grade 2-3
Command Line Device Selection 2-4
Special I/O Ports 2-5
Clock Inputs 2-5
Output Enable Signals 2-7
Asynchronous Clear and Preset 2-8
Clock Enable 2-10
Controlling Register Initial State 2-11
Initial State Attribute 2-11
Controlling Power Consumption 2-12
Controlling Output Slew Rate 2-13
Controlling the Pinout 2-14
Pin Locking 2-14
Pin Assignment 2-15
Prohibiting the Use of Device Pins 2-17

Pin Assignment Precautions 2-17
Controlling Logic Optimization... 2-18
Multilevel Logic Optimization.. 2-18
Setting Multilevel Logic Optimization 2-19
Collapsing Product Term Limit 2-20
Preventing Collapsing of a Logic Node 2-21
Controlling Timing Paths... 2-22
Optimization for Speed 2-22
Timing Constraints.. 2-23
Clock Period 2-23
Point-to-Point Delays 2-24
Output Port Timing Constraint................................... 2-26
Input Port Timing Constraint 2-27
Disabling Timing Specifications................................ 2-28
Reducing Levels of Logic 2-29
XC9500 Local Feedback 2-30
Setting a Node to a Specific Function Block 2-30
Automatic Local Macrocell Feedback Optimization........ 2-30
XC9500 Local Pin Feedback 2-31

Chapter 3 Compiling and Fitting a CPLD Design 3-3

Compiling a Synopsys CPLD Design.................................. 3-1
Step 1 - Entering the dc_shell Environment 3-2
Step 2 - Analyzing the Design 3-2
Step 3 - Elaborating the Design 3-2
Step 4 - Compiling Your Design..................................... 3-3
Step 5 - Specifying Attributes... 3-3
Step 6 - Defining CPLD I/O Signals............................... 3-4
Step 7 - Flattening the Compiled Design 3-4
Step 8- Writing the Netlist 3-5
Step 9 - Writing Out Timing Constraints.......................... 3-5
Step 10 - Translate Timing Constraints File 3-5
Fitting Your Design 3-6
Using Design Manager Interface 3-6
Creating a New Project 3-6
To Implement a Design 3-7
Using Unix Command Line 3-10
CPLD Command Parameters 3-12
Compiling Behavioral Modules for Schematics.................... 3-14
Step 1 - Entering the dc_shell Environment 3-15
Step 2 - Analyzing the Module 3-15

Step 3 - Elaborating the Module 3-15
Step 4 - Compiling Your Module 3-16
Step 5 - Specifying Attributes... 3-16
Step 6 - Writing the Netlist 3-16

Chapter 4 Simulating Your Design 4-1

Recommended CPLD Simulation Strategy 4-1
Controlling the Initial States of Registers............................ 4-2
Simulating Power On Initialization.................................. 4-2
Preparing for Timing Simulation 4-2
Preparing for Functional Simulation 4-3
Creating a Test Bench File.. 4-4
Initializing Registers 4-4
Configuration Declaration... 4-4
Functional Simulation Using VSS 4-5
Design Implementation 4-7
Preparing the Timing Simulation Model 4-8
From Command Line 4-8
From Design Manager.. 4-9
Timing Simulation Using VSS 4-9

Appendix A Library Component Specifications A-1

AND2 — AND8 .... A-2
Inferencing A-2
Component Instantiation A-2
BUF... A-2
Inferencing A-2
Component Instantiation A-2
BUFE A-3
Inferencing A-3
Component Instantiation A-3
BUFG... A-3
Inferencing A-3
Component Instantiation A-3
BUFGSR... A-3
Inferencing A-3
Component Instantiation A-3
BUFGTS A-4
Inferencing A-4
Component Instantiation A-4
FDCE, FDCE_X A-4

Inferencing A-4
Component Instantiation A-4
FDCP A-4
Inferencing A-4
Component Instantiation A-5
FDPE, FDPE_X A-5
Inferencing A-5
Component Instantiation A-5
IBUF A-5
Inferencing A-5
Component Instantiation A-5
INV A-5
Inferencing A-5
Component Instantiation A-6
IOBUFE, IOBUFE_F, IOBUFE_S A-6
Inferencing A-6
Component Instantiation A-6
LD A-6
Inferencing A-6
Component Instantiation A-6
OBUF, OBUF_F, OBUF_S A-6
Inferencing A-7
Component Instantiation A-7
OBUFE, OBUFE_F, OBUFE_S A-7
Inferencing A-7
Component Instantiation A-7
OR2 — OR8 .... A-7
Inferencing A-7
Component Instantiation A-7
XOR2 — XOR8 .... A-8
Inferencing A-8
Component Instantiation A-8

Appendix B Attributes B-1

Instantiated Attributes B-1
KEEP B-1
Synopsys Attributes B-1
Global Input Ports B-2
Output Slew Rate B-2
Pin Assignment B-3
UCF/NCF File B-4

Function Block and Macrocell Assignment B-4
UCF/NCF File B-5
Register Initial State B-5
UCF/NCF File B-5
Macrocell Power Mode B-5
UCF/NCF File B-5
Timing Constraints B-6
create_clock B-6
UCF/NCF File B-6
set_max_delay.. B-6
UCF/NCF File B-7
set_output_delay B-7
UCF/NCF File B-8
set_input_delay B-8
UCF/NCF File B-9

Appendix C Fitter Command and Option Summary C-1

Design Manager... C-1
Invoking the Fitter.. C-1
Fitter Options C-2
CPLD Command.. C-4
Invoking the Fitter.. C-4
Fitter Options C-5

Chapter 1

Getting Started with Synopsys for CPLDs

This chapter shows you how to prepare your Synopsys setup files and verify your installation. It also provides a design walk-through as an overview of the basic steps for implementing Xilinx XC9000/XL/XV CPLD designs using Synopsys. This chapter contains the following sections:

- "Workstation Environment" section
- "Creating Synopsys Setup Files (Workstation)" section
- "Analyzing the DesignWare and Simulation Libraries" section
- "Verifying Your Installation (Workstation)" section
- "Xilinx CPLD Design Flow" section
- "Design Example" section

The remaining chapters in this manual provide additional detailed information on each step.

The design walk-through assumes that you have installed and configured the Xilinx software and libraries. For installation instructions, see the Release Document.

Workstation Environment

Before running any software, you must make sure that your workstation environment is set up properly. The following are required to process Xilinx designs using the Synopsys interface:

1. Set the Xilinx environment variable to the Xilinx installation directory:

   setenv XILINX *xilinx_path*

2. Set the Synopsys environment variable to the Synopsys software directory:

   setenv SYNOPSYS *synopsys_path*

3. Add the following Xilinx executable directory to your path (in addition to all executable directories required by Synopsys software):

   **set path=(**

   $XILINX/bin/*platform* ****

   $path)

   where *platform* is **sol** (for Solaris) or **hp** (for Hewlett-Packard).‘

Note: In UNIX and DC Shell commands shown in this book, where text is too long to print on one line, the back-slash (\\) character at the end of a line is used to indicate a continuation line. In actual usage, continuation line breaks are optional and may occur at any legal point in the command line.

Creating Synopsys Setup Files (Workstation)

After you have installed the Xilinx software you must configure the Synopsys Design Compiler and VSS simulator setup files to access the XC9000 libraries. This section shows you how to configure the setup files and verify that your setup is working properly.

The setup files are typically located in each design directory where Xilinx CPLD designs are processed.

Note: You will find a sample setup file in $XILINX/synopsys/examples/template.synopsys_dc.setup_9k. You can copy this file to your design directory and change the file name to .synopsys_dc.setup.

The Design Compiler Setup File

For XC9000 designs, your Design Compiler setup file (`.synopsys_dc.setup`) must contain the following lines:

**search path = { . **

*Xilinx_path***/synopsys/libraries/syn **
*Synopsys_path***/libraries/syn}**

```
link_library = {xc9000.db}
target_library = {xc9000.db}
symbol_library = {xc9000.sdb}
compile_fix_multiple_port_nets = true
bus_naming_style = "%s<%d>"
bus_dimension_seperator_style = "><"
bus_inference_style = "%s<%d>"
edifout_netlist_only = true

edifout_write_properties_list= {INIT LOC  PWR_MODE}
edifout_power_and_ground_representation = cell
edifout_no_array = true

edifout_ground_name=GND

edifout_ground_pin_name=GROUND

edifout_power_name=VCC

edifout_power_pin_name=VCC
```

Where *Xilinx_path* is the actual directory path where your Xilinx software is installed, and *Synopsys_path* is the actual path where your Synopsys software is installed.

Note: You cannot use UNIX environment variables directly in the `.synopsys_dc.setup` file, but you may use the dc_shell variables, as shown in the template files.

The VSS Simulator Setup File

For XC9000 designs, your VSS Simulator setup file, `.synopsys_vss.setup`, must contain the following lines:

```
SIMPRIM: $XILINX/synopsys/libraries/sim/lib/simprims
TIMEBASE = NS
TIME_RES_FACTOR = 0.1
```

Note: You may use either the `$XILINX` environment variable or the actual path specification in the .synopsys_vss.setup file.

Analyzing the DesignWare and Simulation Libraries

The Xilinx Synopsys Interface (XSI) provides simulation libraries supporting VSS. If you use VSS, you need to analyze the VHDL simulation models after you install the Xilinx Synopsys Interface and before you simulate your first Xilinx design. You must repeat these steps each time you install an update to your Synopsys software.

To analyze the VSS model files for Xilinx simulation, change your current directory to the simulation library source directory for the Xilinx SIMPRIMS library and run the analyze.csh script as follows:

```
cd $XILINX/synopsys/libraries/sim/src/simprims
./analyze.csh
```

The previous command analyzes the encrypted models for the SIMPRIMS library and places the output files into the $XILINX/ synopsys/libraries/sim/lib/simprims directory.

Note: The `analyze.csh` script attempts to create optimized models by using your system's C-compiler. If the `vhdlan` commands in the script encounter any difficulty accessing a C-compiler on your system, they will proceed by creating non-optimized models. The script may produce numerous warnings about this, but should complete successfully. The non-optimized models will produce the same results, but simulation run-time is typically longer.

Verifying Your Installation (Workstation)

Before attempting to compile and fit a design, it is a good idea to verify that you have access to the installed software. A simple verification process is described below.

Verifying Synopsys Software Access

To verify that your system is correctly configured to access the Synopsys software, enter the following UNIX commands:

```
which dc_shell
which vhdlan  (if you are using the VSS simulator)
```

If you get a negative response for either command, (such as "no vhdlan in ...") this means that either the Synopsys software is not installed properly or that your system path is not set properly to include the Synopsys software directories. Refer to the Synopsys documentation for installation instructions.

Verifying Xilinx Software Access

To verify that your system is correctly configured to access the Xilinx-supplied software, enter the following UNIX commands:

1. **which cpld**

If **cpld** cannot be found, the Xilinx software is not installed or is not in your path.

2. **echo $XILINX**

   This variable should also point to the Xilinx directory found in Step 1.

As a final verification that your Xilinx Synopsys interface is ready to use, we have provided a complete design example for you to run, which is described later in this chapter. To quickly verify Synopsys compilation, copy the sample design as described in step 1 and run scan.script as described in step 9.

Xilinx CPLD Design Flow

"Basic CPLD Design Flow" figure shows the basic design flow for creating CPLD designs. Each step is described in the following design example.

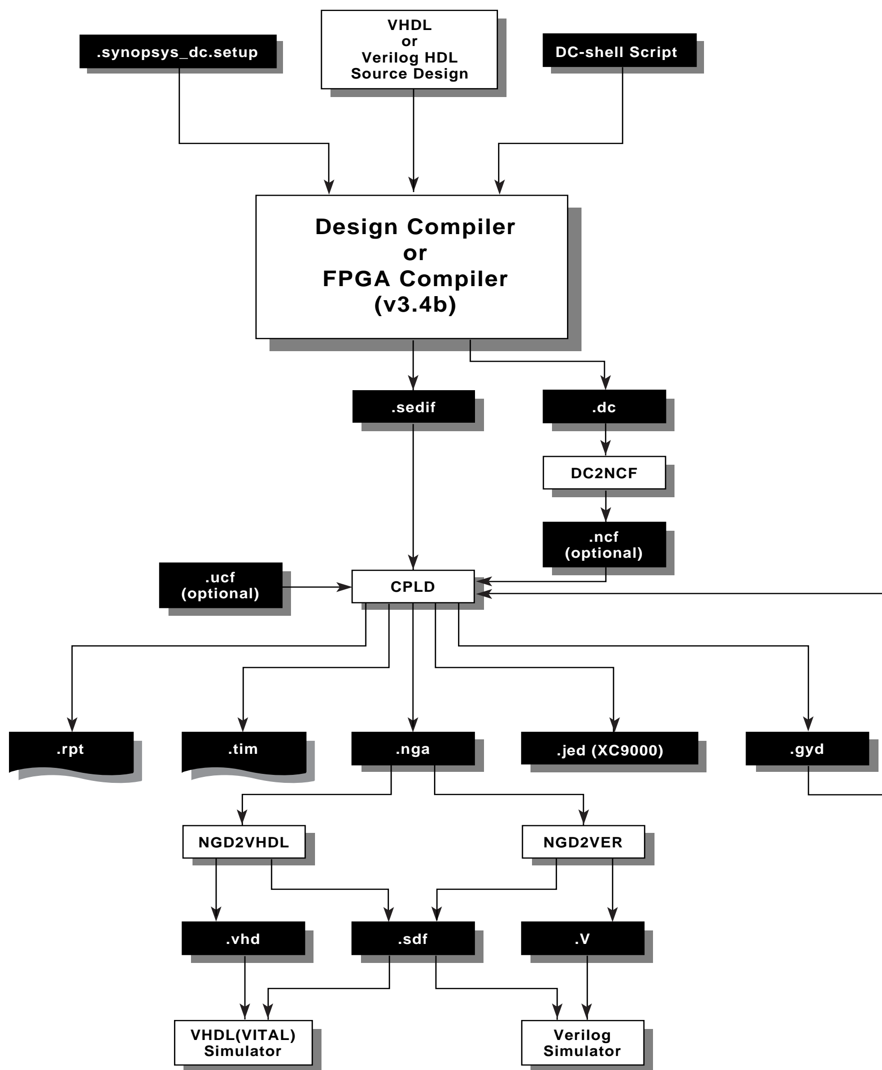

Figure 1-1 Basic CPLD Design Flow

Design Example

The following design example is used to demonstrate the basic CPLD design flow. This design implements a counter with variable start and stop values which are loaded into registers from a data input bus. When the START input is asserted, the start value is loaded into the counter and the counter outputs are enabled. The counter outputs increment on each clock cycle until the counter value matches the stop value. The counter outputs are disabled on the next clock cycle. The design can be implemented in a Xilinx XC95108-7PC84 device.

To help you understand the design, an equivalent schematic is shown in the "Schematic Representation - SCAN Design" figure.

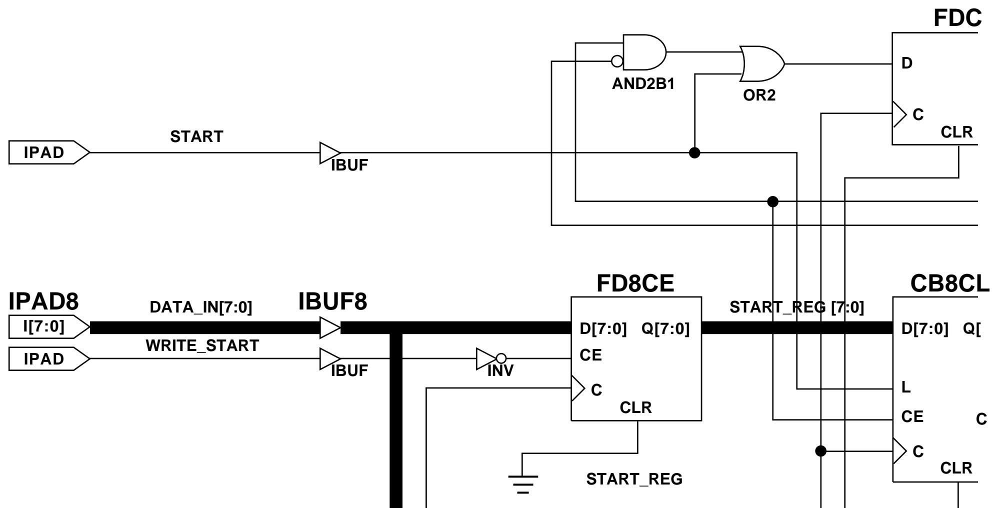

Figure 1-2 Schematic Representation - SCAN Design

Both a VHDL and a Verilog HDL version of this design are provided with the software as an example.

The VHDL source file (scan.vhd) for the scan example design is shown below.

```
----------------------------------------------------------------
```

```
--  Xilinx CPLD Synopsys VHDL Tutorial Design
--  File:      scan.vhd
--  Target Device: XC9536-5PC44
--  Author:     Xilinx Corporation
--                    Copyright (C) Xilinx Corporation
1994
--          All rights reserved
--   Requirements:   Xilinx XACT Version M1

--                       (Alliance core and Synopsys
interface)
--                   Synopsys: Design Compiler or FPGA
Compiler
--          v3.4b or later
-----------------------------------------------------
--------------
-- Standard library configuration --
Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity scan is
    port (CLOCK, CLEAR, START, WRITE_START, WRITE_END
: in std_logic;
          DATA_IN : in std_logic_vector (7 downto 0);
          C_OUT : out std_logic_vector (7 downto 0);
          DONE : out std_logic);
end scan;

architecture behavioral of scan is

signal START_REG : std_logic_vector (7 downto 0);
signal END_REG : std_logic_vector (7 downto 0) :=
"11111111";
```

```
signal COUNT : std_logic_vector (7 downto 0) :=
"00000000";
signal OE_REG, DONE_REG : std_logic := '0';
-- Initial states used by behavioral simulation only.

begin
--  Starting value register.
    process (CLOCK)
        begin
        if (CLOCK'event and CLOCK='1') then
            if (WRITE_START = '0') then
                START_REG <= DATA_IN;
            end if;
        end if;
    end process;

--  Ending value register with asynchronous preload.
    process (CLEAR, CLOCK)
        begin
        if (CLEAR = '1') then
            END_REG <= "11111111";
        elsif (CLOCK'event and CLOCK='1') then
            if (WRITE_END = '0') then
                END_REG <= DATA_IN;
            end if;
        end if;
    end process;

--  DONE flag and OE-control registers with
asynchronous clear.
    process (CLEAR, CLOCK)
        begin
```

```
            if (CLEAR = '1') then
                DONE_REG <= '0';
                OE_REG <= '0';
            elsif (CLOCK'event and CLOCK='1') then

--              Registered equality comparator:
                if (COUNT = END_REG) then
                    DONE_REG <= '1';
                else
                    DONE_REG <= '0';
                end if;

--              OE-control register:
                if (START = '1') then
                    OE_REG <= '1';
                elsif (DONE_REG = '1') then
                    OE_REG <= '0';
                end if;
            end if;
        end process;

--  Counter with asynchronous clear and parallel load.
    process (CLEAR, CLOCK)
        begin
        if (CLEAR = '1') then
            COUNT <= "00000000";
        elsif (CLOCK'event and CLOCK='1') then
            if (START = '1') then
                COUNT <= START_REG;
            elsif (OE_REG = '1') then
                COUNT <= COUNT + 1;
```

```
                end if;
            end if;
        end process;

    --  Three-state counter outputs.
        C_OUT <= COUNT when (OE_REG = '1')
            else "ZZZZZZZZ";
        DONE <= DONE_REG;
    end behavioral;
```

Design Entry

Typically you will enter your design in Synopsys VHDL/HDL form by using a text editor. However, all required source, setup, and test bench files for this design example have already been entered for you. The VHDL design files are contained in the $XILINX/ synopsys/tutorial/cpld/vhdl directory, and the Verilog HDL design files are in $XILINX/synopsys/tutorial/cpld/verilog.

Step1 - Create a Design Directory

Create a local copy of the scan tutorial directory as follows:

- Change your current working directory to a local, writable location in which you will place the scan working directory.
- Copy the entire VHDL or Verilog directory tree from the installed tutorial area into your current directory, for example:

  cp -r $XILINX/synopsys/tutorial/cpld/vhdl .

  or

  cp -r $XILINX/synopsys/tutorial/cpld/verilog .

- Change your current directory to the new working directory as follows:

  cd vhdl

  or

  cd verilog

If you need more information on design entry see the Synopsys Design Compiler manuals.

Functional Simulation

Functional simulation verifies the logic of your design. This will save you time by catching logic errors early in the development cycle. This section describes the simulation flow for the VHDL System Simulator (VSS). If you are not using VSS, skip this section and continue with step 9.

You must analyze your source design file and test bench before simulation.

Step 2 - Analyze Your Design

Analyze the scan design by entering the following Synopsys command on the UNIX command line:

```
vhdlan scan.vhd
```

You will see the analyzer version number and a copyright notice. If the analysis works properly you will be returned to the UNIX prompt with no error messages displayed.

Step 3 - Analyze Your Test Bench

For this example a test bench is provided (`scan_tb.vhd`). At the end of this file, a configuration named `CFG_SCAN_TB` is declared.

Analyze the test bench for scan by entering the following Synopsys command on the UNIX command line:

```
vhdlan scan_tb.vhd
```

Again, you will see the analyzer version number and a copyright notice. If the analysis works properly you will be returned to the UNIX prompt with no error messages displayed.

The test bench for the scan design example (**scan_tb.vhd**) is shown below:

```
-------------------------------------------------------------------

--   Xilinx CPLD Synopsys VHDL Tutorial Design Test
Bench

--  File:        scan_tb.vhd
```

```
--  Target Device: XC9536-5PC44
--  Author:      Xilinx Corporation
--                    Copyright (C) Xilinx Corporation
1994
--               All rights reserved
--  Requirements:  Xilinx XACT Version M1
--                      (Alliance core and Synopsys
interface)
--                  Synopsys: VSS simulator v3.4b-vital
or later
-----------------------------------------------------
--------------


library IEEE;
use IEEE.std_logic_1164.all;
use STD.Textio.all;


entity scan_tb is
end scan_tb;


architecture test of scan_tb is


   constant tCW : time := 25 ns;


   component scan
      port (CLOCK, CLEAR, START, WRITE_START,
WRITE_END: in std_logic;
            DATA_IN: in std_logic_vector (7 downto 0);
            C_OUT: out std_logic_vector (7 downto 0);
            DONE: out std_logic);
   end component;
```

```
   signal CLOCK, CLEAR, START, WRITE_START, WRITE_END:
std_logic;
    signal DATA_IN: std_logic_vector (7 downto 0);
    signal C_OUT: std_logic_vector (7 downto 0);
    signal DONE: std_logic;

begin

    UUT: scan
port map (CLOCK, CLEAR, START, WRITE_START, WRITE_END,
                  DATA_IN, C_OUT, DONE);

DRIVER: process
begin
CLOCK <= '0';
CLEAR <= '1';
START <= '0';
WRITE_START <= '1';
WRITE_END <= '1';
DATA_IN <= "00000000";
wait for 200 ns;

CLEAR <= '0';
wait for tCW;
CLOCK <= '1';
wait for tCW;

CLOCK <= '0';
DATA_IN <= "01111101";
WRITE_START <= '0';
wait for tCW;
CLOCK <= '1';
```

```
wait for tCW;

CLOCK <= '0';
DATA_IN <= "10000001";
WRITE_START <= '1';
WRITE_END <= '0';
wait for tCW;
CLOCK <= '1';
wait for tCW;

CLOCK <= '0';
DATA_IN <= "00000000";
WRITE_END <= '1';
START <= '1';
wait for tCW;
CLOCK <= '1';
wait for tCW;

for I in 1 to 6 loop
  CLOCK <= '0';
  START <= '0';
  wait for tCW;
  CLOCK <= '1';
  wait for tCW;
end loop;

CLOCK <= '0';
START <= '1';
wait for tCW;
CLOCK <= '1';
wait for tCW;
```

```
CLOCK <= '0';
START <= '0';
wait for tCW;
CLOCK <= '1';
wait for tCW;

CLOCK <= '0';
CLEAR <= '1';
wait for tCW;
CLOCK <= '1';
wait for tCW;

CLOCK <= '0';
CLEAR <= '0';
wait for tCW;
wait;

end process;
end test;
configuration CFG_SCAN_TB of scan_tb is
   for test
        end for;
end CFG_SCAN_TB;
```

Step 4 - Invoke the Simulator

Invoke the simulator by entering the following Synopsys command on the UNIX command line:

```
vhdldbx &
```

You will see the following window for selecting the analyzed configurations:

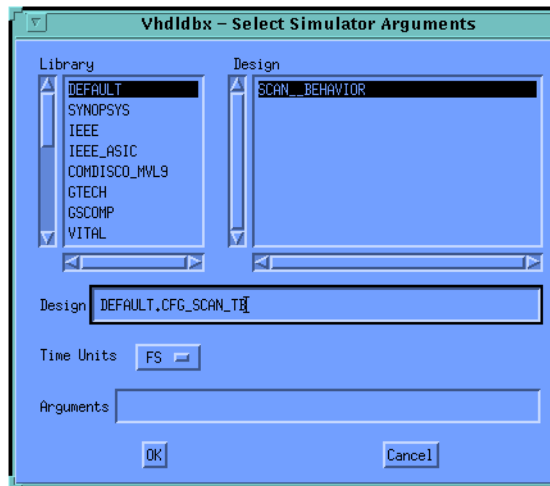

Figure 1-3 VHDLDBX Window

Step 5 - Select the Design

Select **CFG_SCAN_TB** from the menu. This brings up the Synopsys VHDL Debugger window.

Step 6 - Trace Signals

Click in the lower section of the Synopsys VHDL Debugger window and enter the following command:

```
trace *'signal
```

This command selects all signals at the test bench level for display and brings up the Dynamic Waveform Viewer (Waves).

Step 7 - Run the Simulation

Click the **RUN** button in the Debugger window to run the simulation waveform specified in the test bench. A trace display appears.

Step 8 - Return to UNIX

Return to the UNIX environment by selecting EXECUTE-QUIT from the VHDL Debugger menu.

If you need more information on functional simulation see the "Simulating Your Design" chapter.

Synthesizing Your Design (Compiling)

Synthesizing your design converts the VHDL or Verilog HDL source text into a netlist that is composed of logic primitives. The netlist is in a form that can be read by the Xilinx fitter.

Note: This design example demonstrates the compilation flow using dc_shell commands. You could instead use the Synopsys Design Analyzer GUI, but this is not shown in this book.

Step 9 - Enter the DC Shell Environment

Enter the Synopsys DC Shell environment by entering the following Synopsys command on the UNIX command line:

```
dc_shell
```

You will see the DC Shell license information and command-line prompt. Verify that the software version is v3.1 or newer.

Note: The commands required to compile the scan design example are shown in the following steps 10 through 16. These commands are also contained in compiler script files. The appropriate commands for either the Design Compiler orthe FPGA Compiler are contained in scan.script which you can run by entering the following Synopsys command:

```
include scan.script
```

If you choose to use these compiler scripts, go to step 17 when compilation is complete.

Unless otherwise specified, the commands in steps 10-16 are the same for both FPGA Compiler and Design Compiler. Unless otherwise specified, these commands are the same for either the VHDL or Verilog HDL version of the scan design.

Step 10 - Analyze Your Source Design

Read and analyze your VHDL source design file by entering the following Synopsys command:

```
analyze -format vhdl scan.vhd
```

For Verilog HDL source design, enter the following command:

```
analyze -format verilog scan.v
```

The warning messages you may see during this step are normal. The VHDL source file contains initial signal values that are used only for functional simulation and these values are ignored during synthesis. Actual register initial states are set using attributes as shown in step 13.

Step 11 - Elaborate Your Design

To build the design based on your analyzed VHDL file, entering the following Synopsys command:

```
elaborate scan
```

This command displays each register and 3-state buffer encountered in your design.

Step 12 - Synthesize Your Design

To synthesize an implementation of your design based on cells in the XC9000 technology library enter the following Synopsys command:

```
compile -map_effort low
```

The mapping effort is set to LOW to save compilation time because the synthesizer does not perform any speed or area optimization for CPLD designs; optimization is performed by the CPLD fitter.

Step 13 - Specify Initial Register States

In this design we want the counter, the OE_REG and DONE flip-flop to be initialized to zero. We also want the END_REG register to initialize to all ones to prevent the comparator from detecting a false DONE condition. The initial states of the remaining flip-flops are not critical for this design.

By default, all registers in an XC9500 or XC9500XL devices are initialized to zero when powered up, so we need not specify any dc_shell directives for the counter, OE_REG or DONE. To specify the all-ones initial state of the END_REG register, enter the following Synopsys command:

```
set_attribute END_REG* init S -type string
```

Note: The `init` attribute may be attached to either the flip-flop cells or their output nets in the design.

Step 14 - Specify Timing Constraints

We want to constrain clock period for this design to be no more than 12 ns. Enter the following Synopsys timing constraint:

```
create_clock CLOCK -period 12
```

Step 15 - Use a Global Clock Input Port

We want the CLOCK input for this design to be assigned to one of the global clock input pins (GCLK) of the device. Normally, the CPLD fitter would do this automatically whenever possible. To explicitly assign a global clock, enter the following Synopsys shell command:

```
set_pad_type -exact BUFG CLOCK
```

Step 16 - Place I/O Buffer Cells

To place I/O buffer cells on all top-level ports in the design, enter the following Synopsys commands:

```
set_port_is_pad "*"
insert_pads
```

Step 17 - Flatten the Design

Any hierarchy in the design must be flattened before attributes can be written into the netlist. Enter the following Synopsys shell command to flatten the design:

```
ungroup -all -flatten
```

Step 18 - Output the Netlist

The design database is now complete and ready to be output in netlist form.

Write an EDIF-formatted netlist by entering the following Synopsys command:

```
write -format edif -output scan.sedif
```

Step 19 - Output Timing Constraints

The timing constraint entered in step 14 is written into a separate DC_shell script file which is later read by the Xilinx implementation software. Enter the following Synopsys shell command to write the timing constraint file:

```
write_script > scan.dc
```

Step 20 - Exit DC Shell

Exit DC Shell by entering the following Synopsys command:

```
exit
```

You are returned to the UNIX prompt.

Step 21 -Translate Timing Constraints File

The DC_shell script file containing your timing constraints written in Step 19 must be translated into a Xilinx constraint file, scan.ncf. At the Unix prompt, enter the following command to run the translator:

```
dc2ncf scan.dc
```

If you need more information on compiling your design, see the "Compiling Your Design" chapter.

The synthesizer creates a gate-level design with no physical device information; the physical layout of the device is done in the fitter step. No speed or area estimates are provided by the XC9000 synthesis library. Therefore do not attempt to create a timing report or perform estimated timing simulation at this time.

Fitting Your Design

The CPLD fitter translates your logical design file (scan.sedif) into a physical device layout, and performs all device-specific logic optimization.

Note: This design example demonstrates the fitter flow using **`cpld`** command-line entry. You could instead use the Xilinx Design Manager GUI to process your design. This is explained in chapters 2 and 3 of this guide.

Step 22 - Fit Your Design

To fit your design into an XC9000 device, enter the following on the UNIX command line:

```
cpld scan
```

To fit the design into an XC9500XL device, enter the following command:

```
cpld -p 9500xl scan
```

The fitter displays a series of progress messages and a resource summary that shows how well your design fits into the target device.

When the fitter is finished, and assuming there are no errors, you need only to examine the fitter report file, scan.rpt. If you wish, you can also examine the static timing report file, scan.tim.

If you need more information on fitting, see the "Fitting Your Design" chapter.

Timing Simulation

Timing simulation uses the actual device delays based on the physical layout of your design after fitting.

Step 23 - Prepare A Timing Simulation File

The **cpld** command produces a timing simulation database file (**scan.nga**) each time the design is successfully implemented. To translate the **.nga** file for simulation using VSS or other VITAL simulator, enter the following command at the UNIX prompt:

```
ngd2vhdl -w scan scan_time
```

The **ngd2vhdl** command produces a VITAL-compliant structural VHDL file (**scan_time.vhd**) and an SDF-formatted timing back-annotation file (**scan_time.sdf**).

To translate the **.nga** file for simulation using a Verilog simulator, enter the following command:

```
ngd2ver -w scan scan_time
```

The **ngd2ver** command produces a structural Verilog file (**scan_time.v**) and an SDF-formatted timing book annotation file (**scan_time.sdf**).

Note: The remainder of this tutorial describes the procedure for performing timing simulation using the VSS simulator. If you are not using the VSS simulator, skip the remainder of this tutorial.

Step 24 - Analyze Your Implemented Design

Analyze the implemented design produced by the Xilinx **ngd2vhdl** command by entering the following Synopsys command on the UNIX command line:

```
vhdlan scan_time.vhd
```

You will see the analyzer version number and a copyright notice. If the analysis works properly you will be returned to the UNIX prompt with no error messages displayed.

Step 25 - Analyze Your Test Bench

Analyze the simulation test bench by entering the following Synopsys command on the UNIX command line:

```
vhdlan scan_tb.vhd
```

Again, you will see the analyzer version number and a copyright notice. If the analysis works properly you will be returned to the UNIX prompt with no error messages displayed.

Step 26 - Invoke the VSS Simulator

Invoke the Synopsys VSS simulator by entering the following Synopsys command on the UNIX command line:

```
vhdldbx  -sdf_top \
   /SCAN_TB/UUT -sdf scan_time.sdf CFG_SCAN_TB &
```

This will open the simulator window. The **-sdf** parameter specifies the timing back-annotation file produced by **ngd2vhdl**. The **-sdf_top** parameter specifies the level in the test bench hierarchy at which the back annotation information will be applied, which is the "UUT" instance of the scan test bench.

Step 27 - Open the Waveform Viewer

Use the **TRACE** command to specify the same signals used during functional simulation in step 6. Enter the following command on the VHDL Debugger command line:

```
trace *'signal
```

This opens the Dynamic Waveform Viewer window.

Step 28 - Run the Simulation

Run the simulation by clicking the RUN button in the lower section of the Synopsys VHDL Debugger window.

This will run the timing simulation test bench and display the simulation trace of your design.

Step 29 - Return to UNIX

Return to the UNIX environment by selecting **EXECUTE-QUIT** from the simulator menu.

If you need more information on timing simulation, see the "Simulating Your Design" chapter.

Chapter 2

Designing with CPLDs

This chapter discusses how to control various device features and get the best performance from Xilinx XC9500, XC9500XL, and XC9500XV CPLDs. You can control aspects of design implementation at these points:

- in your VHDL or Verilog HDL source design
- using **dc_shell** commands or Design Analyzer commands
- using FPGA Express implementation options
- using **cpld** command-line parameters or the Xilinx Design Manager templates

For more information on synthesis library cells, see the "Library Component Specifications" appendix. For more information on attributes, see the "Attributes" appendix. For more information on CPLD commands, see chapter 3, *Compiling and Fitting your Designs.*

This chapter describes how you can control the aspects of design implementation. It contains the following sections:

- "Target Device Selection" section
- "Special I/O Ports" section
- "Controlling Register Initial State" section
- "Controlling Power Consumption" section
- "Controlling Output Slew Rate" section
- "Controlling the Pinout" section
- "Controlling Logic Optimization" section
- "Controlling Timing Paths" section
- "XC9500 Local Feedback" section

Target Device Selection

You must always select a target family, such as XC9500 or XC9500XL. By default, the fitter will automatically select a device within that family for you, choosing, in general, the smallest part that will satisfy the needs and constraints of your design. Otherwise, you can select a specific device, package, speed, or any valid combination.

Selecting a Part from the Design Manager

You can select the target device family and, optionally, a specific device, package and speed from the Design Manager. To bring up the Design Implementation Dialog Box select:

Design → Implement

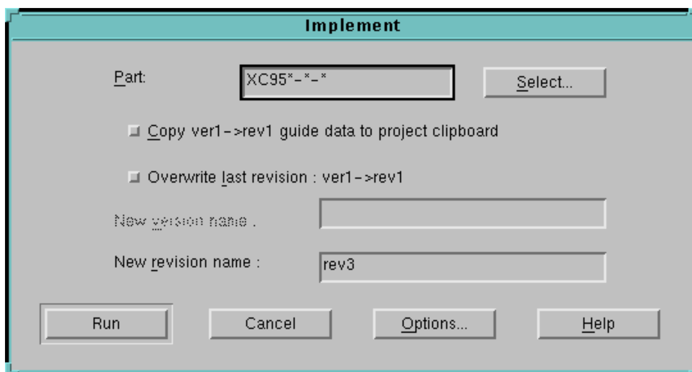

Figure 2-1 Design Implementation Dialog Box

Click once on the Select button to get the Part Selector Dialog Box.

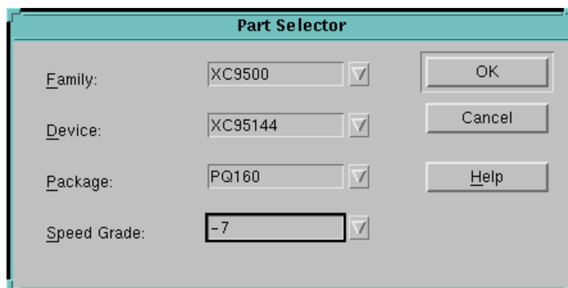

Figure 2-2 Part Selector

The part selector allows you to select from CPLD families, devices, packages and speed grades.

Family

This option allows you to select from a list of Xilinx families. Click once on the down arrow adjacent to this option box to display available families. If you select the XC9500 family, the device, package and speed boxes will all be set to **All**. Once you have selected a family you can select a device from within that family, or you can leave any or all of the remaining boxes set to **All**.

Device

This option allows you to select a specific CPLD device from the family selected on line one. Click once on the down arrow adjacent to this option box to list the devices for the **Family** selected above. Once you have selected a device, only packages and speed grades available for that device will appear in the **Package** and **Speed Grade** option lines below.

If you are only interested in selecting the fastest device available, leave this option on **All** and select the **Speed Grade** only.

Package

This line allows you to select from available packages. If you have already selected a device, the option line will only display packages that are available for that device.

Speed Grade

This option allows you to select from available speed grades. If you have already select a device from the **Device** option box, only the speed grades available for that device will be displayed when you click on the down arrow. However, if speed is your primary consideration, leave the **Device** option box on **All** and select your **Speed Grade** only. If you select a speed grade and then try to select a device which does not support that speed grade, the **Speed Grade** option box will revert to **All**.

Command Line Device Selection

You can optionally specify a target device on the **cpld** command line when you run the fitter. The format of the part-type parameter on the **cpld** command line is:

cpld -p *part_type design_name*

where

-p *part_type* — specifies the target device type or set of devices from which to choose (default is automatic device selection from the XC9500 family); where *part_type* can be:

9500 = any XC9500 family device (auto selection)

9500xl = any XC9500XL family device (auto selection)

9500xv = any XC9500XV family device (auto selection)

"*95ddd[xl][-ss][-pppp]*" — where *95ddd* is the device code (such as 95108), *ss* is the speed grade, *pppp* is the package code (such as PQ160), and an asterisk (*) can be used as a wildcard string (quotes required around *part_type* when asterisk is used).

You may specify either a unique device code, a range of eligible devices or an entire CPLD family from which the fitter will automatically choose. The fitter will, in general, automatically select the smallest device and package that will fit the design, and the fastest speed grade of the resulting device.

To specify a range of devices, you can use an asterisk (*) as a wildcard character. You may also specify an enumerated list of devices, separated by commas. If you use the asterisk character or an enumerated list in the **cpld** command, you must enclose the parameter string in quotes. If you use an asterisk in the part code field, your string must begin with a "9" (for XC9500). If you want to select from the XC9500XL family, the letters "XL" must appear in the part string; if you want to select from the XC9500XV family, the letters "XV" must appear in the part string; otherwise the fitter selects only from the XC9500 family. If you use a comma-separated list, all devices must be from the same CPLD family (either all XC9500, all XC9500XL, or all XC9500XV, but not mixed). For example, the following are valid part-type parameter specifications:

```
cpld -p 95108-10-PC84 design1
cpld -p "95108-*-PC84" design1
cpld -p "95*XL-*-PX84" design1
cpld -p "95108-10-PC84,95108-7-PQ*" design1
```

If you leave the speed grade unspecified, the fitter will always choose the fastest available speed grade for the selected device.

Refer to the Release Document for a list of CPLD device codes supported by the current version of the fitter software.

Special I/O Ports

Ordinarily, you need only to declare ports in your top-level entity to represent all the I/O pins on the CPLD device. The Synopsys set_port_is_pad and insert_pads commands automatically infer IBUF, OBUF, OBUFE and IOBUFE cells from XC9000 library to represent the I/O ports in the netlist.

The following sections describe special global control pins on CPLD devices that can be used for register clocking, tristate control and register set/reset, instead of ordinary IBUF inputs. Unless otherwise specified, the fitter automatically allocates these special global pins, if possible, when input ports in your design are used to perform these control functions.

Clock Inputs

To use a device input as a clock source, you can simply refer to a top-level input port as the clock condition in a process. For example:

```
entity xyz is
  port (CLOCK:in std_logic; ...
...
process (CLOCK)
  begin
  if (CLOCK'event and CLOCK='1') then
    ...
```

The fitter automatically uses one of the global clock pins (GCK for XC9000) whenever possible.

For XC9000 devices, a global clock input signal may perform negative-edge clocking. For example:

```
process (CLOCK)
  begin
  if (CLOCK'event and CLOCK='0') then
    ...
```

The same clock input may even be used both as both positive-edged and negative-edged to clock different processes on opposite edges of the clock signal. Global clock inputs may also be used as ordinary input signals to other logic elsewhere in the design.

If an input port signal passes through any logic function (other than an inverter) before it is used as a clock by any flip-flop, the input will not be routed to any flip-flops in the design using the global clock path. Instead, that clock signal will be routed through the logic array.

There are a limited number of global clock pins on each CPLD device (consult the device data sheet). If you need to explicitly control the use of global clock pins, you can specify the **set_pad_type** command in your **dc_shell** script with the parameter "**-exact BUFG**," and reference an input port of your design. For example:

```
set_pad_type -exact BUFG clock1
```

Note: The cell name BUFG must be upper case. The **set_pad_type** command must be executed before the **insert_pads** command.

The global clock pins provide much shorter clock-to-output delays than clocks routed through the logic array. Routing a clock through the logic array also uses up one extra p-term for each flip-flop.

If you want to prevent the fitter from automatically using the global clock pins, go to the Implementation Options template of the Design Manager and remove the check mark from **Use Global Clock(s)** as follows:

1. **Design** → **Implement**
2. Press the **Options** softkey.
3. Select **Edit Template**
4. Select the **Basic** tab.
5. Remove the check mark from **Use Global Clock(s)**.

Using command line you can also prevent the fitter from automatically using the global clock pins. To do this, specify the "-nogck" parameter on the **cpld** command line as follows:

```
cpld -nogck design_name
```

If **-nogck** is specified, input ports used as clocks will pass through the array. You can still use the **set_pad_type** command to explicitly specify global clock inputs.

If you use an internal signal as a clock, it will be routed to the flip-flops through the logic array.

Output Enable Signals

To use a device input to control tristate device outputs, you can simply refer to a top-level input port signal as a tristate condition in your design. For example:

```
entity xyz is
  port (ENABLE:in std_logic; ...
...
Q <= Q_VALUE when (ENABLE='1') else 'Z';
```

The fitter automatically uses one of the global tristate control pins (GTS for XC9000) whenever possible.

For XC9000 devices, a global tristate control input signal may perform an active-low output-enable. For example:

```
Q <= Q_VALUE when (ENABLE='0') else 'Z';
```

The same tristate control input may even be used both active-high and active-low to enable alternate groups of device outputs. Global tristate control inputs may also be used as ordinary input signals to other logic elsewhere in the design.

If an input port signal passes through any logic function (other than an inverter) before it is used as an output enable by any output port, the input will not be routed to device output drivers in the design using the global tristate control path. Instead, the output enable signal will be routed through the logic array.

There are a limited number of global tristate control pins on each CPLD device (consult the device data sheet). If you need to explicitly control the use of global tristate control pins, you can specify the set_pad_type command in your dc_shell script with the parameter “-exact BUFGTS”, and reference an input port of your design. For example:

```
set_pad_type -exact BUFGTS enable1
```

Note: The cell name BUFGTS must be upper case. The **`set_pad_type`** command must be executed before the **`insert_pads`** command.

The global tristate control pins provide much shorter input-to-output-enable delays than tristate controls routed through the logic array. Routing a tristate control signal through the logic array also uses up one extra p-term for each output.

If you want to prevent the fitter from automatically using the global tristate control pins, go to Implementation Options template of the Design Manager and remove the check mark from **`Use Global Output Enable`**.

1. **`Design`** → **`Implement`**
2. Press the **`Options`** softkey.
3. Select **`Edit Template`**
4. Select the **`Basic`** tab.
5. Remove the check mark from **`Use Global Output Enable(s)`**.

Using the command line, if you want to prevent the fitter from automatically using the global tristate control pins, specify the **`-nogts`** parameter on the **`cpld`** command line as follows:

`cpld -nogts` *design_name*

If **`-nogts`** is specified, input ports used for tristate control will pass through the array. You can still use the **`set_pad_type`** command to explicitly specify global tristate control inputs.

If you use an internal signal as an output enable, it will be routed to the outputs through the logic array.

Asynchronous Clear and Preset

To use a device input as an asynchronous clear or preset source, you can simply refer to a top-level input port as the set or reset condition in a clocked process. For example:

```
entity xyz is
  port (CLOCK, RESET : in std_logic; ...
...
process (CLOCK, RESET)
```

```
    begin
    if (RESET='1') then Q <= '0';
    elsif (CLOCK'event and CLOCK='1') then

      ...
```

For XC9000 designs, the fitter automatically uses the global set/reset pin (GSR) whenever possible. A global set/reset input signal may also perform active-low clear or preset. For example:

```
  process (CLOCK, PRESET)
    begin
    if (PRESET='0') then Q <= '1';
    elsif (CLOCK'event and CLOCK='1') then
      ...
```

A global set/reset inputs may also be used as an ordinary input signal to other logic elsewhere in the design.

If an input port signal passes through any logic function (other than an inverter) before it is used as an asynchronous clear or preset on any flip-flop, the input will not be routed to any flip-flops in the design using the global set/reset path. Instead, the clear/preset signal will be routed through the logic array. Routing a clear or preset through the logic array uses up one extra p-term for each flip-flop.

There is only one global set/reset pin on each XC9000 device. If you need to explicitly control the use of the global set/reset pin, you can specify the **set_pad_type** command in your **dc_shell** script with the parameter "**-exact BUFGSR**" and reference an input port of your design. For example:

```
set_pad_type -exact BUFGSR reset1
```

Note: The cell name BUFGSR must be upper case. The **set_pad_type** command must be executed before the **insert_pads** command.

If you use an internal signal as a set or reset, it will always be routed through the logic array.

Note: If a flip-flop has both a clear and preset condition and you assert both the clear and preset concurrently, its Q-output is unpredictable. This is because the fitter may arbitrarily invert the logic stored in a flip-flop to achieve better logic optimization. Individual clear and preset operations still produce the correct final logic state as dictated by the user design. Functional simulation does not accurately predict the ultimate behavior of the chip when clear and preset

are asserted concurrently. Timing simulation, however, is performed after logic optimization and behaves exactly as the chip will when programmed.

Clock Enable

If you express a synchronous clock-enable condition in a clocked process, and FDCE_X or FDPE_X primitive will be inferred from the XC9000 library. For example:

```
process (CLOCK)
  begin
  if (CLOCK'event and CLOCK='1') then
    if (CLOCK_EN='1') then Q<=D;
```

These FDCE_X or FDPE_X primitives are always expanded into an ordinary D-type flip-flop with its Q-feedback multiplexed into its D-input.

FDCE_X is an edge-triggered D-type flip-flop with asynchronous clear and clock enable. The synthesizer uses this component for all functions that require D-type registers with clock-enable, provided no asysnchronous preset condition is specified. FDCE_X is not intended to be instantiated into any design.

FDCE is also an edge-triggered D-type flip-flop primitive with clear and clock enable. But, FDCE is never inferred. Users must explicitly instantiate it or explicitly replace an FDCE_X cell that is inferred in the design. For XC9500XL and XC9500XV devices, the FDCE unconditionally uses the clock-enable product-term of the macrocell to implement the CE input. For XC9500 devices, FDCE is always expanded into a simple D-type flip-flop with its Q-feedback multiplexed into its D-input, just like the FDCE_X cell.

FDPE_X is an edge-triggered D-type flip-flop with asynchronous preset and clock enable. The synthesizer uses this component for all functions that require D-type registers with preset and clock-enable. FDPE_X is not intended to be instantiated into any design.

FDPE is also an edge-triggered D-type flip-flop primitive with preset and clock enable. But, FDPE is never inferred. Users must explicitly instantiate it or explicitly replace an FDPE_X cell that is inferred into the design. For XC9500XL and XC9500XV devices, the FDPE unconditionally uses the clock-enable product-term of the macrocell to

implement the CE input. For XC9500 devices, FDPE is always expanded into a simple D-type flip-flop with its Q-feedback multiplexed into its D-input, just like the FDPE_X cell..

If you use FDCE or FDPE cells and target an XC9500XL or XC9500XV device, you may find that the logic connected to the clock enable input in some designs may not get optimized into the same macrocell as the flip-flop. The XC9500XL or XC9500XV macrocell contains only a single product-term to implement clock enable input logic. The CPLD fitter does not attempt transform your clock enable input logic onto the D-input of the flip-flop if it cannot be completely implemented using the clock enable p-term. In general, only primary inputs (device input pins or macrocell feedbacks), their complements or positive-logic AND-gate functions of primary inputs or their complements can be completely implemented using the p-term. If you connect a more complex logic function to the clock enable input of an FDCE or FDPE cell and it does not get completely implemented on the clock enable p-term, your design may incur extra macrocell resources and combinational macrocell feedback delays.

Controlling Register Initial State

All registers in a CPLD device are initialized when the device is powered up. The initial state (preload value) of each register is programmable.

Registers in XC9000 macrocells have both asynchronous clear and asynchronous preset controls available. The initial power-on states of CPLD macrocell registers can be selected regardless of whether the register is asynchronously cleared of preset during operation.

Unless otherwise specified in your design, each register in an XC9000 device will initialize to the zero (reset) state at power-up.

Initial State Attribute

You can specify the preload states of selected register cells in your design by setting the initial state attribute in dc_shell as follows:

```
set_attribute register_cell init state -type string
```

where:

- *register_cell* is the name (or set of names) of register cell(s) in your design, or the names of register output nets.

- *state* is either R (reset to 0) or S (set to 1).

For example, to specify an initial state of "1" for the register named QOUT_reg, enter the following:

```
set_attribute QOUT_reg init S -type string
```

The initial state attribute is ignored if it is applied to any cell in your design that is not a flip-flop.

Controlling Power Consumption

The power consumption of each macrocell in an CPLD device is programmable. The standard (default) setting consumes more power and produces shorter propagation delay. The low-power setting reduces power consumption for less speed-critical paths.

By default, all macrocells in the design will operate in standard power mode. You can change the global power setting to use the low power mode throughout the design by selecting **Low** on the **Default Power Setting** in the **Implementation Options** template in the Design Manager:

1. **Design** → **Implement**
2. Press the **Options** softkey.
3. Select **Edit Template**
4. Select the **Basic** tab.
5. Select the **Low** option adjacent to **Macrocell Power Setting**.

When you run the fitter your design will be implemented with the low power setting.

To specify low power when using the Unix command line, use the "lowpwr" parameter with the **cpld** command as follows:

cpld -lowpwr *design_name*

You can also instruct the fitter to automatically reduce the macrocell power for paths that do not require standard power to meet timing constraints. In the Design Manager, check **Timing Driven**. On the Unix command line, specify the **-autopwrslew** parameter on the cpld command line:

cpld -autopwrslew *design_name*

Controlling Output Slew Rate

For XC9000 devices each output is programmable to operate either at full speed or with limited slew rate. Limiting the slew rate reduces output switching surges in the device. Slew rate control becomes important when your design uses a large number of outputs or you have transmission lines on your board which are sensitive to fast edge rates.

By default, the CPLD fitter will apply a fast slew rate to all outputs. If you want to limit the slew rate of a device output to decrease its switching speed, use the "set_pad_type" command in dc_shell. Enter the following in dc_shell:

```
set_pad_type -slewrate HIGH  port_list
```

where *port_list* is a list of output ports that are to operate with slow output slew rate.

If you need to explicitly set an output to use fast slew rate, enter the following in dc_shell:

```
set_pad_type -slewrate NONE  port_list
```

Note: The **`set_pad_type`** command must be executed before the **`insert_pads`** command in dc_shell.

By default, the fitter uses **`Fast`** for slew rate for all output drivers. To change the default to slow slew rate in the Design Manager:

1. **`Design`** → **`Implement`**
2. Press the **`Options`** softkey.
3. Select **`Edit Template`**
4. Select the **`Basic`** tab.
5. Place a check mark in **`Slow`** box adjacent to **`Default Output Slew Rate.`**

To specify default slow slew-rate from the UNIX command line, use the parameter **`-slowslew`** in the **`cpld`** command:

```
cpld -slowslew design_name
```

The fitter also has an option whereby it will automatically apply **`Slow`** slew rate to each output unless that would cause any propagation delay to that pin to fail to meet a timing specification.

To enable slew-rate optimization in the Design Manager, go to the Implementation Options template and check the **Fast** box on the **Default Output Slew Rate** line.

To enable automatic slew rate control from the Unix command line, use the parameter **-autoslewpwr** when executing the **cpld** command.

```
cpld -autoslewpwr design_name
```

Controlling the Pinout

When you first run the fitter before your pinout is committed, the software automatically selects pin locations for your I/O signals. Pin locations are selected which will give you the greatest flexibility to iterate your design without having to move any of the pins. Each time the fitter successfully implements your design, it creates a guide file (*design_name*.gyd), which contains all the resulting pinout information. After you commit your pinout, subsequent design iterations cause the guide file to be read by the fitter and your committed pinout will be preserved.

We strongly recommend that you allow the software to automatically generate your initial pinout. Attempting to select your own initial pin preferences reduces the ability of the fitter to implement your design successfully the first time. It further reduces the amount of logic changes you could make after locking your pinout.

Pin Locking

If you have successfully fit a design into an CPLD device and you build a prototype containing the device, you will probably want to "lock" the pinout.

1. In the Design Manager, select an existing design revision that was successfully run through the Fit step (typically, your most recent revision).
2. Select **Design** → **Lock Pins**. The pinout saved in the selected revision (stored in *design_name*.**gyd**) is translated into pin location (LOC) constraints and written into a user constraint file (*design_name*.**ucf**).
3. Select **View Lock Pins Report** in the dialog box to make sure no pin assignment conflicts were found.

4. When ready, run the fitter (**`Design`** → **`Implement`**). The previous pinout information will be read from the UCF file and used in the new design revision.

If you want to tell the fitter to directly read the guide file (*design_name*.**`gyd`**) on the Unix command line, you should specify the "pinlock" option with the **`cpld`** command as follows:

```
cpld -pinlock design_name
```

The **`-pinlock`** parameter tells the fitter to read and obey the pinout from the guide file that was saved the last time the fitter completed.

The fitter will not move any of the pins contained in the guide file, even if it prevents the design from successfully mapping.

Whenever you specify a guide file (pin locking), the fitter automatically uses the same device and package as previously used, unless you override it with a different specific device and/or package.

The pin locations stored in the guide file are specified based on the top-level port names in your design. If you change the name of any of your ports, the corresponding pin will no longer be constrained to the location stored in the guide file.

When you iterate your design while your pins are locked, you are free to delete existing ports and/or add new ports. The fitter will automatically select the best locations for any new ports you add, after placing all the existing ports constrained by the guide file.

Note: If you iterate your design and your pinout is not yet committed (you haven't built a prototype containing the device), you should not specify the pinlock option. Instead, allow the software to redefine the pinout of your modified design. This will continue to give you the greatest flexibility to iterate your design again after you commit your pinout.

Pin Assignment

You can assign explicit locations for pins in your design using the LOC attribute in dc_shell. Enter the following in dc_shell:

```
set_attribute port LOC pin_name -type string
```

where *port* is the name of the port being assigned.

For example, to place the "start" input port on pin 23:

```
set_attribute start LOC p23 -type string
```

For PC, PQ and VQ type packages, the *pin_name* takes the form "P*nn*" where *nn* is a number. For example, for the PC84 package, the valid range for *pin_name* is P1 through P84. For grid array type packages (PG and BG), the *pin_name* takes the form "*rc*", where *r* is the row letter and *c* is the column number.

You can also specify pin locations interactively using the Constraints Editor tool invoked from the Design Manager.

When your design contains LOC attributes, you should specify the target device type in the Design Manager **`Part Selector`** menu or using the **`cpld`** command's -p parameter (see Target Device Selection in this Chapter). The LOC attributes are typically not compatible when retargeting a design between different package types, device types or device families.

The LOC attributes are unconditional in that the software will not attempt to relocate a pin if it cannot achieve the specified assignment. If you specify a set of LOC attributes that the fitter cannot satisfy, the fitter will terminate with an error.

The LOC attributes override the pin assignments in the guide file if you specify the pinlock option. This allows you to make explicit changes to your committed pinout. If you override the guide file using LOC attributes, the software will issue a warning.

If your objective is to preserve a previously created pinout, we recommend you use the pinlock feature instead of creating a set of LOC attributes with the existing pin locations. The guide file saved from the previous design implementation contains additional information to help the fitter to successfully fit your modified design.

If you used LOC attributes when compiling your netlist but you want to temporarily allow the fitter automatically assign all I/O pins, place a check in the Ignore Design Assignments box in the Basic tab of the Implementation Options template:

1. **`Design`** → **`Implement`**
2. Press the **`Options`** softkey.
3. Select **`Edit Template`**
4. Select the **`Basic`** tab.

5. Remove the check mark next to **`Use Design Location Constraints`**.

To temporarily allow the fitter automatically assign all I/O pins when using the Unix command line, you can specify the **`-ignoreloc`** parameter on the **`cpld`** command:

`cpld -ignoreloc` *design_name*

The **`-ignoreloc`** option allows you to temporarily ignore all the LOC attributes in your netlist. This is useful if you want to test how your design fits a different target device without re-compiling your design.

Prohibiting the Use of Device Pins

`Prohibit I/O Locations` allows you to reserve device pins for later use, or simply prevent them from being used at all. For instance, if you anticipate design changes in the future and want to set traces on your printed circuit board now, you can use this feature to prevent the fitter from using pins associated with those traces. Then, when you decide to use the traces, you can use the LOC attribute to assign those pins to new input/output buffers you place in your design.

In the Constraints Editor, **`Prohibit I/O Locations`** prevents all selected I/O pins from being used by the design. This dialog can be entered using a dialog box provided in the **`Ports`** tab.

Pin Assignment Precautions

You can apply the LOC attribute to as many ports in your design as you like. However, each pin assignment further constrains the software making it more difficult for the fitter to automatically allocate logic and I/O resources for the remaining I/O signals in your design.

When you manually assign output and I/O pins, you force the software to place associated logic functions into specific macrocells and specific function blocks. If the associated logic does not exceed the available function block resources (macrocells, product terms, and FastCONNECT inputs), the logic is mapped into the macrocell and the design will route in the FastCONNECT.

It is usually best to allow the fitter to automatically assign most or all of the pins based on the most efficient placement of logic in the device. The fitter automatically establishes a pinout which best

allows for future design iterations without pin relocation. Any manual pin assignments you make in your design may not allow as much tolerance for changes in the logic associated with those pins, and in the logic physically mapped to nearby locations in the device.

If you are assigning pin locations to ports used as clocks, asynchronous set/reset, or output enable in your design, you should assign them to the GCK, GSR and GTS pins on the device if you want to take advantage of these global resources. The fitter will still automatically assign other clock, set/reset and output enable inputs to remaining GCK, GSR and GTS pins if available.

Controlling Logic Optimization

When you create combinational logic functions, the software attempts to collapse as much of the logic as possible into the smallest number of CPLD macrocells. Combinational logic optimization performed by the synthesis tool are generally not essential to the efficiency or performance of the resulting CPLD implementation. The CPLD fitter automatically performs all essential optimizations.

Any combinational logic function bounded between device I/O pins and flip-flops is subject to complete or partial collapsing. Collapsing the logic improves the speed of the logic path and can also reduce the amount of logic resources (macrocells, p-terms and FastCONNECT inputs) required to implement the function.

The process of collapsing logic into other logic functions is called "logic optimization".

Multilevel Logic Optimization

Multilevel Logic Optimization seeks to simplify the total number of logic expressions in a design, and then collapse the logic in order to meet user objectives such as density, speed and timespecs. This optimization targets CPLD architecture, making it possible to collapse to the macrocell limits, reduce levels of logic, and minimize the total number of pterms.

Multilevel Logic Optimization extracts combinational logic from your design. Combinational logic includes:

- register-to-register logic
- path-to-register logic

- register-to-path logic
- path-to-path logic

Multilevel Logic Optimization operates on combinational logic according to the following rules:

1. If timespecs are set, the program will optimize for speed to meet timespecs.
2. If timespecs are not set, the program will optimize either for speed or density, depending on the user setting of **`Timing Optimization`**.
   a) If **`Timing Optimization`** is turned on, the combinational logic will be mapped for speed.
   b) If **`Timing Optimization`** is turned off, the combinational logic will be mapped for density. The goal of optimization will then be to reduce the total number of pterms.
3. Logic marked with the attribute **`NOREDUCE`** will not be extracted or optimized.

Setting Multilevel Logic Optimization

Multilevel Logic Optimization can be set from the **`Advanced`** tab of the **`Implementation Options`** template of the Design Manager as follows:

1. **`Design`** → **`Implement`**
2. Press the **`Options`** softkey.
3. Select **`Edit Template`**
4. Select the **`Advanced`** tab.
5. Place a check in the **`Use Multilevel Logic Optimization`** box.

Multilevel Logic Optimization will operate when you run the fitter.

If you wish to disable multilevel logic optimization when running a design from the **`cpld`** command, use the option -**`nomlopt`**. If you do not specify this option, the fitter automatically uses multilevel logic optimization.

Collapsing Product Term Limit

When a larger combinational logic function consisting of several levels of AND-OR logic is completely collapsed (flattened), the number of product terms required to implement the function may grow considerably. By default, the fitter limits the number of p-terms used as a result of collapsing to 20 when using the **`Optimize Speed`** template, or 90 when using the **`Optimize Density`** template. If the collapsing of a logic level results in a logic function consisting of more p-terms than the limit (after Boolean reduction), then the collapsing of that logic level is not performed and the function will be implemented using two or more levels of AND-OR logic.

Note: The fitter will not exceed the collapsing p-term limit even if a timing constraint applied to a path cannot be met.

When the Timing Optimization option is off, as it is in the Optimize Density template, the fitter only performs collapsing on a node if the total number of p-terms used after collapsing would be less than the total number of p-terms used by the combined functions before collapsing.

The overall extent to which logic is collapsed throughout an XC9000 design can be controlled from the **`Advanced Optimization`** tab of the **`Implementation Options`** template.

1. **`Design`** → **`Implement`**
2. Press the **`Options`** softkey.
3. Select **`Edit Template`**
4. Select the **`Advanced`** tab.
5. Place a value in the **`Collapsing Pterm Limit`** box, or use the up and down arrows to raise or shrink the Pterm limit. The allowable range is between 2 and 90.

To change the Pterm limit from Unix, use the "-pterms" parameter on the **`cpld`** command line:

`cpld -pterms` *nn design_name*

where *nn* is the maximum allowable number of p-terms that can be used to implement a logic function after collapsing. The allowable range for the pterms parameter is between 2 and 90.

If you find that the path delay of a larger, multi-level logic function in an XC9000 design is not satisfactory, try increasing the pterms parameter to allow the larger functions to be flattened further. For example, you may try increasing the p-term limit to 35 when rerunning the fitter, as shown:

```
cpld -pterms 35 design1
```

The fitter report (*design_name*.rpt) indicates the number of p-terms used for each logic function. You should see these numbers increase as you raise the pterms limit, until the design is fully flattened. At the same time, you'll see the internal combinational nodes eliminated until none remain.

Preventing Collapsing of a Logic Node

Flattening typically increases the overall amount of p-term resources required to implement the design. Some designs which fit the target device initially may fail to fit if flattened too much. Other designs can be flattened completely and still fit. If you cannot increase the **cpld** pterms parameter enough to sufficiently flatten a critical path and still fit the target device, you may try applying logic optimization control at specific nodes in your design.

A special cell is provided in the XC9000 libraries, named KEEP, which is used to apply a logic optimization constraint to any signal passed through it. The KEEP cell has one input port named I and one output port named O (letter O). By instantiating a KEEP cell and passing through it a signal in the middle of a combinational logic function, you can prevent that signal from being collapsed.That is, you prevent the cell that drives the signal from being collapsed forward into any of its fanouts. The KEEP cell is instantiated as follows:

```
label: KEEP port map (O => outgoing_signal, I =>
incoming_signal);
```

In the following example, a KEEP cell is used to prevent the logic for an address decoder from being collapsed into the select input of a 16-bit multiplexer:

```
component KEEP port (O: out std_logic, I : in std_logic);
end component;
...
DECODE1 <= '1' when (ADDR_BUS = ADDR_1) else '0';
DATA_BUS <= A_BUS(0 to 15) when (DECODE1_NEW='1') else B_BUS;
U1: KEEP port map (O=>DECODE1_NEW,I=>DECODE1);
```

By preventing logic optimization, the fitter will not attempt to duplicate the logic of the address decoder in each bit of the multiplexer.

You can use KEEP to break logic chains in non-speed-critical paths and prevent those functions from using too many p-terms. If you set the pterms parameter too high and your design no longer fits, try using KEEP to reduce the size of selected non-critical paths.

Controlling Timing Paths

There are two mechanisms that can improve the timing of your design:

- Global Timing Optimization
- Timing Constraints

Optimization for Speed

By default, the fitter performs timing optimization on logic paths in your design. Timing optimization will automatically shorten your logic paths as much as it can. In general, timing optimization optimizes logic and allocates the fastest available resources for the longest paths in your design, assuming all paths are equally critical. In some cases, the fitter trades off density for a speed advantage.

These default fitter option settings that favor optimization for speed are included in a template named **`Optimize Speed`**. Timing Optimization will be set ON for all CPLD families, and **`Use Local Macrocell Feedback`** and **`Use Local I/O Pin Feedback`** will be set to on for XC9500 devices. If you want to change to a template containing fitter options that favor optimization for density, select the **`Optimize Density`** template:

1. **`Design`** → **`Implement`**
2. Press the **`Options`** softkey.
3. Adjacent to the **`Edit Template button,`** click once on the down arrow and select the **`Optimize Density`** implementation option. If you revert to optimizing for timing, set the template for **`default`** or **`Optimize Speed`**.

To turn off timing optimization from Unix, specify the "-notiming" parameter on the **`cpld`** command line as follows:

```
cpld -notiming design_name
```

Disabling timing optimization will optimize for density and may significantly reduce the processing time of the CPLD fitter.

Timing Constraints

The Synopsys Design Compiler (or FPGA Compiler) provides a set of timing constraint commands that you can use to specify the timing requirements of your Xilinx design. After compiling your design, the Xilinx software reads both your design netlist and your dc_shell timing constraint commands and performs timing optimization according to your specifications. You can enter timing constraints in the Design Analyzer, the dc_shell command line, or a dc_shell script file. The Synopsys Compiler does not use your timing constraints to optimize your logic or infer library cells during compilation of a CPLD design. All timing optimization is performed by the CPLD fitter after reading your dc_shell timing specifications.

The following path types can be controlled using timing constraints:

Pad-to-pad delay	Input port to an output port
Register setup time	Setup time of an input port to the data pin of a flip-flop, with respect to a clock
Register-to-register	Propagation delay from the output of a flip-flop to the data pin of the same or different flip-flop, including flip-flop setup requirements, measured from clock-edge to clock edge
Clock-to-output delay	Propagation delay from the clock of a flip-flop to an output port

This section lists the Synopsys commands that you can use to create timing specifications for your Xilinx CPLD designs.

Clock Period

You can use the **`dc_shell`** command **`create_clock`** to declare a clock input port and place a period timing specification on the speci-

fied clock net. The register-to-register delays between all flip-flops on the named clock will be constrained by the specified period.

The **create_clock** command creates a cycle time specification on the specified clock signal as follows:

```
create_clock clock_port -period delay
```

where *clock_port* is the name of the clock input port and *delay* is the clock cycle time in nanoseconds.

Note: The Synopsys **max_period** command is not supported by the Xilinx fitter; use the **create_clock** command instead.

Point-to-Point Delays

The **dc_shell** command **set_max_delay** specifies delay constraints for specific paths originating from input (or I/O) ports or flip-flop cells and terminating at output (or I/O) ports or flip-flop cells. The syntax of the **set_max_delay** command is:

```
set_max_delay delay -from source -to destination
```

For example, to specify the propagation delay from the CLEAR input port to the DONE output port:

```
set_max_delay 20 -from CLEAR -to DONE
```

The following table describes the various *source* and *destination* combinations you can use with the **set_max_delay** command.

Table 2-1 set_max_delay

Source	Destination	Affected Timing Path
input or I/O port (except clock)	output or I/O port	pad-to-pad propagation delay
input or I/O port (except clock)	register cell	register setup time from specified port(s) with respect to flip-flop's clock pin. **Note 1.**

Table 2-1 set_max_delay

Source	Destination	Affected Timing Path
register cell	register cell	register-to-register delay (cycle time), regardless of each register's clock source (overrides create_clock period constraint covering same registers
register cell	output or I/O port	register clock-to-output delay from flip-flop's clock pin to output pad. **Note 2**
clock input port	output or I/O port	register clock-to-output delay from the specified clock input to specified output port
clock input port	register cell	not used for CPLD designs

Note 1. To specify input setup time with respect to a clock pad, refer to the Input Port Timing Constraint section below.

Note 2. To specify clock-to-output delays beginning at the clock input pad, either use the set_max_delay command specifying the clock port as the source, or refer to the Output Port Timing Constraint section below.

If you use the **`set_max_delay`** command to specify a register setup time constraint on an input port to a named register, then the delay you specify in the command must be larger than the actual setup time requirement you want to have between the data and clock pads of the device. The amount you will need to add to your desired pin-to-pin setup time is the delay of the clock path from clock pad to the flip-flop.

To use the following command form to specify setup time:

```
set_max_delay delay -from input_port -to register
```

you should specify your delay value according to the following relationship:

```
delay = tSU + tGCK
```

where:

delay is the delay value specified in the **`set_max_delay`** command,

tSU is the desired setup time requirement on the data input pad with respect to the clock input pad, and

tGCK is the delay of the clock path from clock pad to the flip-flop's clock pin.

For XC9500 devices, the *tGCK* delay parameter is listed in the device data sheets.

Similarly, if you use the **set_max_delay** command to specify clock-to-output delay from a named register to an output port, the delay you specify in the command must be smaller than the actual pad-to-pad delay you want to have on the device. The amount you will need to deduct from your desired pin-to-pin delay is the delay of the clock path from clock pad to the flip-flop.

To use the following command form to specify clock-to-output delay:

```
set_max_delay delay -from register -to output_port
```

you should specify your delay value according to the following relationship:

```
delay = tCO - tGCK
```

where:

delay is the delay value specified in the **set_max_delay** command,

tCO is the desired clock-to-output delay between the clock input pad and the output pad, and

tGCK is the delay of the clock path from clock pad to the flip-flop's clock pin.

Output Port Timing Constraint

The **set_output_delay** command establishes clock-to-output delay specifications based on values specified in the **create_clock** or **set_max_delay** constraints or creates tighter constraints for named output ports as follows:

```
set_output_delay delay –clock clock output_port
```

When the named output ports are driven by registers covered by a **create_clock** period constraint, the **set_output_delay** constraint specifies how much time (*delay*) before the next clock edge the named outputs need to become stable.

In this case, the set_output_delay constraint specifies the delay path between the clock input pin of the CPLD device and the named output pin(s) according to the following relationship:

set_output_delay_value = *create_clock_period_value* – *cpld_clock_to_output_delay*

where *set_output_delay_value* is the *delay* value specified in the **set_output_delay** constraint, *create_clock_period_value* is the *period* value specified in a previous **create_clock** constraint, and *cpld_clock_to_output_delay* is the desired worst-case propagation delay between the clock input pin and output pin(s) of the CPLD.

For example, the following pair of commands sets the register-to-register delays between all flip-flops clocked by C1 to 20 ns and sets the clock-to-output delay from C1 to output Q2 to 6 ns:

```
create_clock C1 -period 20
set_output_delay 14 -clock C1 Q2
```

This command also changes the values of pad-to-pad or clock-to-output delay specifications created by the **set_max_delay** command, by making the constraints tighter by the amount specified by the *delay* value for the named outputs.

When using the **set_output_delay** constraint, the named clock must be explicitly declared as a global clock input port by using the **dc_shell** command

```
set_pad_type -exact BUFG clock
```

as described in the section Special I/O Ports earlier in this chapter.

Input Port Timing Constraint

The **set_input_delay** command establishes register setup time specifications based on **create_clock** or **set_max_delay** commands or creates tighter constraints on named input ports as follows:

set_input_delay *delay* **–clock** *clock input_port*

When the named input ports feed into registers covered by a **create_clock** period constraint, the **set_input_delay** constraint specifies how much time (*delay*) after the previous clock edge the named inputs are expected to become stable.

In this case, the set_input_delay constraint specifies the setup time requirements between data input pin(s) and the clock input pin of the CPLD device according to the following relationship:

set_input_delay_value = *create_clock_period_value* – *cpld_external_setup_time*

where *set_input_delay_value* is the *delay* value specified in the **set_input_delay** constraint, *create_clock_period_value* is the *period* value specified in a previous **create_clock** constraint, and *cpld_external_setup_time* is the desired worst case setup time between the data input pin(s) and the clock input pin of the CPLD.

For example, the following pair of commands sets the register-to-register delays between all flip-flops clocked by C1 to 20 ns and sets the setup time requirements on input D2 with respect to the C1 device input pin to 8 ns:

```
create_clock C1 -period 20
set_input_delay 12 -clock C1 D2
```

This command also changes the values of pad-to-pad delay or register setup time specifications created by the **set_max_delay** command, by making the constraints tighter by the amount specified by the *delay* value for the named inputs.

When using the **set_input_delay** constraint, the named clock must be explicitly declared as a global clock input port by using the **dc_shell** command

```
set_pad_type -exact BUFG clock
```

as described in the section Special I/O Ports earlier in this chapter.

Note: The **dc_shell** command **set_false_path** is not supported for CPLD designs at this time.

You can also enter timing constraints interactively using the Constraint Editor tool invoked from the Design Manager.

Disabling Timing Specifications

If you used timing constraints when compiling your design but want to run the fitter without using your timing specifications, you can temporarily ignore all timing specifications by removing the check from the **Use Timing Constraints** option in the **Implementation Options** template.

1. **Design** → **Implement**
2. Press the **Options** softkey.

3. Select **`Edit Template`**
4. Select the **`Basic`** tab.
5. Remove the check mark next to **`Use Timing Constraints`**.

If you want to do this on the Unix command line, use the **`-ignorets`** parameter with the **`CPLD`** command as follows:

`CPLD -ignorets` *design_name*

Reducing Levels of Logic

The XC9500 architecture, like most CPLD devices, is organized as a large, variable-sized combinational logic resource (the AND-array and XOR gate) followed by a register. If you place combinational logic before a register in your design, the fitter maps the logic and register into the same macrocell. The output of the register is then directly available at an output pin of the device. If, however, you place logic between the output of a register and the device output pin, a separate macrocell must used to perform the logic, decreasing both the speed and density of your design. The following example shows two functionally similar styles for designing a selectable divide-by-2 or divide-by-4 counter. The first design style is inefficient for CPLD architectures; the second example is more efficient.

```
-- Ineefficient style for CPLDs:
process (CLOCK)
  begin
  if (CLOCK’event and CLOCK=’1’) then
    DIV2 <= not DIV2;
    DIV4 <= DIV4 xor DIV2;
  end if;
end process;
DIV_OUT <= DIV2 when (SEL4=’0’) else DIV4;

-- More efficient style for CPLDs:
process (CLOCK)
  begin
  if (CLOCK’event and CLOCK=’1’) then
    DIV2 <= not DIV2;
    if (SEL4=’1’) then
        DIV_OUT <= (DIV_OUT xor DIV2);
     else
        DIV_OUT <= not DIV_OUT;
     end if;
```

```
    end if;
  end process;
```

XC9500 Local Feedback

XC9500 family devices (except XC9536) contain high-speed local feedback paths interconnecting the macrocells within each function block. Local feedback paths bypass the FastCONNECT array and provide shorter propagation delays between macrocells. Using local feedback requires that all logic sourcing and receiving local feedback signals be mapped to the same function block locations within the target device. When a timing constraint is applied to a path which would require local feedback routing in order to meet the specified constraint, the fitter will attempt to map the logic spanned by the timespec into the same function block and use local feedback routing.

Setting a Node to a Specific Function Block

If the fitter does not find a way to map logic into the same function block, you can explicitly map your logic to allow the fitter to use local feedback routing. To explicitly map a logic node to a specific function block, apply the LOC attribute to the signal (net) or cell as follows:

set_attribute *signal_name* **loc FB***nn* **-type string**

where *nn* is a legal function block number for the target device.

Automatic Local Macrocell Feedback Optimization

This option, when enabled, will use local feedback routing whenever a feedback node connects between macrocells that happen to get mapped to the same function block. This is done in addition to using local feedback to satisfy timespecs. The software will create clusters of equations and attempt to place them in the same function block. However, the software is allowed to break a cluster if it is impossible to place it in one function block.

Local Macrocell Feedback optimization can be selected from the **Advanced** tab of the **Implementation Options** template of the Design Manager as follows:

1. **Design → Implement**
2. Press the **Options** softkey.

3. Select **Edit Template**
4. Select the **Advanced** tab.
5. Place a check in the **Use Local Macrocell Feedback** box.

To specify local feedback using the **cpld** command, use the command option -**localfbk**

```
cpld -localfbk design_name
```

XC9500 Local Pin Feedback

This option enables the software to use local I/O pin feedback whenever possible in an XC9500 design (except XC9536). Pin feedback takes less time than the XC9500 FastCONNECT path. The software uses the pin feedback path instead of the FastCONNECT path for output pin signals that do not have tristate control or slow slew rate. By default, this option is off.

Local Pin Feedback optimization can be selected from the **Advanced** tab of the **Implementation Options** template of the Design Manager as follows:

1. **Design** → **Implement**
2. Press the **Options** softkey.
3. Select **Edit Template**
4. Select the **Advanced** tab.
5. Place a check in the **Use Local Pin Feedback** box.

To specify pin feedback from the **cpld** command use the command option -**pinfbk**.

```
cpld -pinfbk design_name
```

Chapter 3

Compiling and Fitting a CPLD Design

The Synopsys interface supports both VHDL and Verilog HDL design synthesis. Either the Synopsys FPGA Compiler or Design Compiler can be used to compile CPLD designs; there are no differences between the two compilers with regard to the supported features or implementation efficiency. In the following discussion, the term "compiler" refers to either FPGA Compiler or Design Compiler.

This chapter describes how to compile your design using the Synopsys Design Compiler shell (**dc_shell**). You can also use the Synopsys graphical user interface, Design Analyzer, to process your designs. This chapter also describes how to implement your design using both the Xilinx Design Manager and the **cpld** command-line. It contains the following sections:

- "Compiling a Synopsys CPLD Design" section
- "Fitting Your Design" section
- "Compiling Behavioral Modules for Schematics" section

Before compiling you will need to develop your VHDL or Verilog HDL source file (*design_name*.vhd or *design_name*.v). Usually it is a good idea to perform a functional simulation of your source design using VSS or some VHDL or Verilog compatible simulator before trying to synthesize it. See the "Simulating Your Design" chapter for information on functional simulation using VSS.

Compiling a Synopsys CPLD Design

This section describes the procedure for compiling a complete CPLD design based on VHDL or HDL. If you are preparing a VHDL/HDL-based module for inclusion in a schematic-based design, refer to the section "Compiling Behavioral Modules for Schematics" later in this chapter.

The Synopsys compiler synthesizes your source design and creates an EDIF 2.0.0 netlist file composed of logic primitives that is used by the Xilinx CPLD fitter to implement your design in a CPLD. All compiler commands are executed from within the Synopsys dc_shell environment.

Step 1 - Entering the dc_shell Environment

Enter the Synopsys dc_shell environment by entering the following Synopsys command on the UNIX command line:

```
dc_shell
```

You will see the dc_shell prompt.

Step 2 - Analyzing the Design

To interpret your design and verify that it is free of errors, enter the following Synopsys command for VHDL designs:

analyze -format vhdl *design_name*.**vhd**

or, for Verilog HDL designs:

analyze -format verilog *design_name*.**v**

For example, the command used inVHDL version of the scan example in the "Getting Started with Xilinx CPLDs" chapter:

```
analyze -format vhdl scan.vhd
```

If your source file contains initial signal values (which are used only for functional simulation) they will cause warnings that can be safely ignored; these initial signal values are not used during synthesis. Actual register initial states are set using attributes, as described in Chapter 2.

If the **analyze** command finds errors, you will need to make the necessary corrections to your source file and repeat the analyze command before continuing with synthesis.

Step 3 - Elaborating the Design

To derive a logical design, based on your VHDL/HDL description, enter the following Synopsys command:

elaborate *entity_name*

where *entity_name* is the name of your top-level entity in your design.

For example, the command used in the scan example in the "Getting Started with Xilinx CPLDs" chapter:

```
elaborate scan
```

During this step, the compiler displays information about all registers and 3-state buffers encountered in your design.

You are now ready to compile your design using the XC9000 target library.

Step 4 - Compiling Your Design

When you compile your design, the Synopsys synthesizer uses the components in the Xilinx XC9000 technology library to create an actual logic implementation of your design. The library used during compilation is defined by the dc_shell target_library variable, typically specified in your `.synopsys_dc.setup` file.

To synthesize your design based on target CPLD technology library, enter the following Synopsys command:

```
compile [-map_effort low]
```

The mapping effort parameter is optional. However, it is recommended that you set it to LOW to save compilation time. The synthesizer does not perform any speed or area optimization for CPLD designs; this optimization is performed after compilation by the CPLD fitter.

Step 5 - Specifying Attributes

Attributes are used to control the physical implementation of your design as described in Chapter 2. All attributes are optional. The attributes that you may want to set at this time are:

- Register initial states
- Global buffer assignment
- Pin assignments
- Output slew rate
- Timing constraints

For example, the attributes used in the scan example in the "Getting Started with Xilinx CPLDs" chapter:

```
set_attribute END_REG* init -type string S
```

See the "Attributes" appendix for complete details on all supported attributes.

Step 6 - Defining CPLD I/O Signals

Now you must define which signals are connected to the physical I/O pins of the CPLD.

Use the following command to identify all ports in your design for which the synthesizer needs to infer an I/O buffer:

set_port_is_pad *port_name*

Do not use this command for any ports for which you instantiated I/O buffer cells from the library.

Normally, you would automatically place I/O buffer cells on all top-level ports in the design. Enter the following Synopsys command:

```
set_port_is_pad "*"
```

For the ports that were specified by **set_port_is_pad**, the following command infers the appropriate I/O buffer cells into your design:

```
insert_pads
```

Note: If you want to control output slew rate or explicitly assign global buffers to input ports of your design, the **dc_shell set_pad_type** command must be invoked before the **insert_pads** command, as described in *Special I/O Ports*.

Step 7 - Flattening the Compiled Design

Before writing the netlist, you should flatten the hierarchy of your design. Any attribute or timing constraints attached to objects in any hierarchy levels below the top would be lost unless the design is flattened. Enter the following command to flatten your design:

```
ungroup -all -flatten
```

Step 8- Writing the Netlist

Write your synthesized design file in EDIF netlist format by entering the following Synopsys command:

```
write -format edif -hierarchy -output design_name.sedif
```

where:

- **-format edif** specifies the EDIF file format.
- **-hierarchy** specifies that all levels of the design hierarchy are to be written.
- **-output** *design_name*.**sedif** specifies your output file name, which should be the same as your source file name, with the extension **.sedif**.

For example, the command used in the scan example in the "Getting Started with Xilinx CPLDs" chapter:

```
write -format edif -hierarchy -output scan.sedif
```

Step 9 - Writing Out Timing Constraints

If you specified any timing constraints for your design (in dc_shell), you must write them to a dc_shell script file so they can be read by the Xilinx software. Enter the **write_script** command and redirect its output to a file as follows:

```
write_script > design_name.dc
```

Note: Your design must be flattened using the ungroup -all -flatten command before writing the script file.

This is the end of the required processing in dc_shell. Before exiting you may wish to save the design database in Synopsys db format by executing the **write** command. You can exit dc_shell by entering the following Synopsys command:

```
exit
```

Step 10 - Translate Timing Constraints File

If you specified any timing constraints for your design and wrote a **dc_shell** script file as in Step 9, you must translate the script file into a Xilinx constraint file (**.ncf**) that can be read by the fitter. Enter the **dc2ncf** command at the Unix prompt as follows:

```
dc2ncf design_name.dc
```

Note: None of the Synopsys timing or area analysis reports are useful at this time because the CPLD technology libraries do not contain timing or area estimation data. The Xilinx fitter provides a Static Timing Report which shows the calculated worst case timing for each logic path in your design.

You are now ready to begin the fitting process as described in the next section.

Fitting Your Design

Using Design Manager Interface

You can start the Design Manager from the command line by entering the following command:

```
dsgnmgr
```

Creating a New Project

After opening the Design Manager for the first time, you must create a new project for your design.

A project includes all design versions, implementation revisions, reports, and any other Xilinx data created while you work with a design. The Design Manager graphically displays information about these items in the project view. When you create a new project, you specify a design to open and a directory for the project. You can create as many projects as you want, but you can only work with one at a time.

The following procedure explains how to create a new project by importing a design.

1. Select **`File`** → **`New Project`** from the Design Manager menu.

   The New Project dialog box appears, as shown in the "New Project Dialog Box" figure.

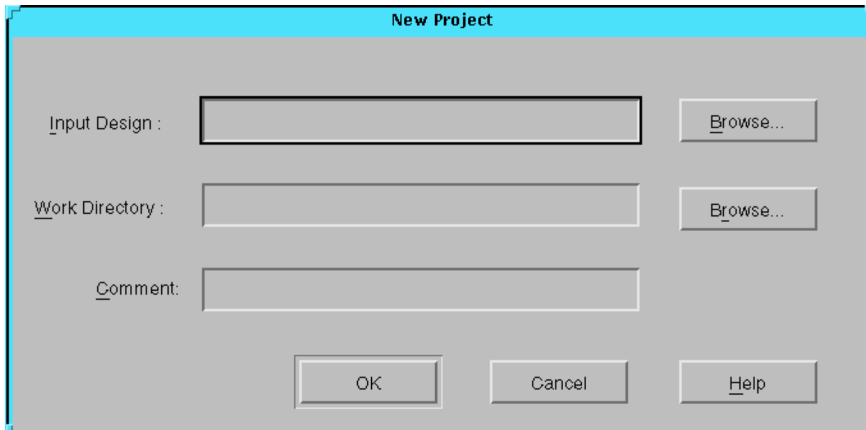

Figure 3-1 New Project Dialog Box

2. Specify a design file to open using one of the following methods.
   - In the Input Design field, type the name of a design file to open.
   - Click on the Input Design **`Browse`** button to the right of the Input Design box. The Open dialog box appears. Select an SEDIF file to open. Click on **`OK`**.

Note: The Design Manager automatically creates a subdirectory named xproj and appends it to the work directory. The Design Manager uses the xproj directory to store all the data files for the project. By default, the design directory is used as the work directory; however, you can change the default directory by typing a path in the Work Directory field or by using Browse to select a directory.

3. In the New Project dialog box, click **`OK`**.

   After your design has loaded, the Design Manager window appears, configured for the loaded design.

To Implement a Design

The following procedure describes how to implement a design automatically from the Design Manager.

1. Select **`Design`** → **`Implement`** from the Design Manager menu.

The Implement dialog box appears, as shown in the "Implement Dialog Box" figure. The options in this dialog box are described in the *Design Manager/Flow Engine Reference/User Guide.*

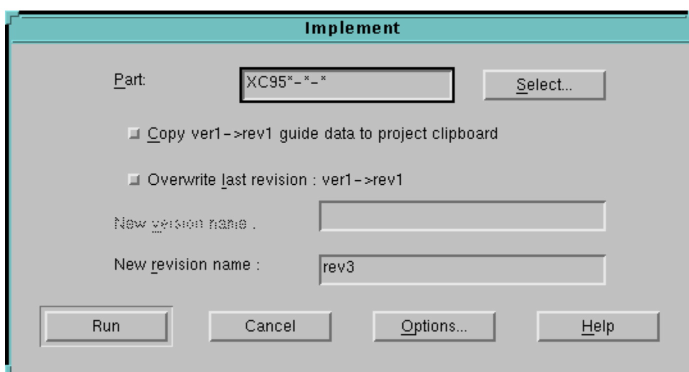

Figure 3-2 Implement Dialog Box

2. In the Implement dialog box, click on the `Select` button to the right of the Part text field.

   The Part Selector appears, as shown in the "Part Selector Dialog Box" figure.

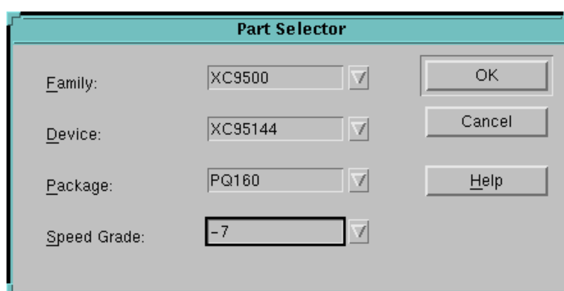

Figure 3-3 Part Selector Dialog Box

3. Select the family, device, package, and speed grade.

By default, the CPLD fitter automatically selects the device, package and speed for you. You can select any specific device, package or speed, or leave any of these boxes as "All."

For a specific explanation of each option, see the *Design Manager/ Flow Engine Reference/User Guide.*

4. Click on `OK` to set the part type.
5. To set design implementation options, click on the `Options` button in the Implement dialog box.

   The Options dialog box appears, as shown in the "Options Dialog Box" figure.

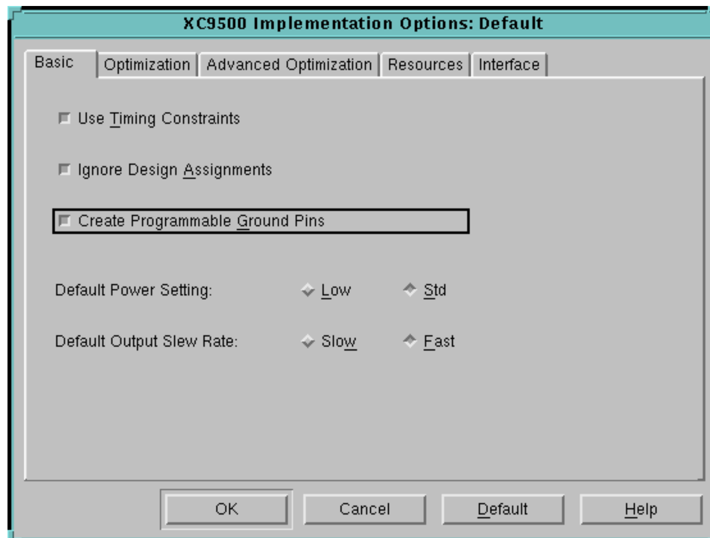

Figure 3-4 Options Dialog Box

6. Select desired options, such as templates to use and reports to generate.

   For a specific explanation of each option see the *Design Manager/ Flow Engine Reference/User Guide.* The "Implementation Options" chapter discusses all implementation options for the XC9500 and XC9500XL CPLD families, including Simulation Data Options.

7. Click on `OK` to set the options.

8. In the Implement dialog box, enter a version and revision name if you want to change the default names.

9. Click on the **Run** button to implement the design.

   The Flow Engine window appears. When processing is complete, the Flow Engine closes and the Implementation Status dialog box, shown in the "Implementation Status Dialog Box" figure, appears.

Figure 3-5 Implementation Status Dialog Box

10. In the Implementation Status dialog box, click on **Reports** to view the reports generated by the Flow Engine or click on **View Logfile** to view the implementation logfile.

Note: At this point you can also perform timing simulation and program the device. Timing simulation is described in the Interface User Guide for your system. Device programming is described in the *JTAG Programmer Guide.*

Using Unix Command Line

The **cpld** command is used to invoke the Xilinx CPLD fitter software. CPLD uses the logical design produced by the Synopsys compiler to create a physical layout for a target CPLD.

To invoke the fitter, enter the following Xilinx command on the UNIX command line:

cpld [*options*] *design_name*

Invoking the **cpld** command with no parameters produces a listing of all available command-line options.

The *design_name* is the name of the netlist file produced by the Synopsys compiler, without path qualifiers, and either with or without the extension, **.sedif**.

If *design_name* is specified without extension, the **cpld** command automatically searches for and reads the netlist with file extension .**sedif** as produced by Synopsys FPGA Compiler and Design Compiler. The **cpld** command also accepts files with extensions .**edif**, .**edn**, .**xnf**, .**sxnf** and .**pld**. If you happen to have any files with the same name as your design and with one of these other extensions, you should remove them from your design directory before running the **cpld** command to prevent the wrong file from being read inadvertently.

If you do not specify any optional parameters, the fitter automatically selects a device from the XC9500 family which fits your design (if possible).

The **cpld** command performs the following functions:

- Reads the netlist file (*design_name***.sedif**) produced by the Synopsys compiler.
- If you specified timing constraints for your design and saved them in a dc_shell script file (*design_name***.dc**), the **cpld** fitter automatically reads the **.dc** file, translating it first into a Xilinx netlist constraint file (*design_name***.ncf**).
- Minimizes and collapses the combinational logic of your design so that it requires the least number of macrocell and product term resources.
- Partitions and maps your design to fit within the architecture of the CPLD, optionally selecting the target device.
- Creates a device programming file (*design_name*.jed).
- Creates a fitter report (*design_name*.rpt) that shows you information such as the type and quantity of device resources used, and the resulting pinout.
- Creates a Static Timing Report (*design_name*.tim) that shows the calculated worst-case timing for all signal paths in your design.
- Creates a guide file (*design_name*.gyd) that is used to lock signal names to device pins, allowing you to keep the device pinouts during subsequent design iterations.

- Creates a timing simulation database file (*design_name*.**nga**) that can be translated into structural VHDL (for the Synopsys VSS or other VITAL-compliant simulator) or structural Verilog.

Whenever the **cpld** command is invoked, it copies any existing fitter report file (.rpt), timing report file (.tim), guide file (.gyd), and programming file (.jed) to a subdirectory named "backup.".

CPLD Command Parameters

The [options] field of the cpld command represents an optional list of one or more command-line parameters. The following are the cpld command-line parameters that apply to Synopsys design entry:

- **-autopwrslew** — reduces macrocell power mode after meeting timing specifications.
- **-autoslewpwr** — reduces slew rate and then power mode to meet timing specifications.
- **-detail** — produces a detailed path timing report (*design_name*.tmd) in addition to the default summary report (*design_name*.tim).
- **-grounds** — creates programmable ground pins on unused I/O ports.
- **-ignoreloc** — temporarily ignores all LOC attributes in the design, allowing the fitter to assign the locations of all I/O pins.
- **-ignorets** — temporarily ignores all timing specifications in the *design_name*.**ncf** or UCF file.
- **inputs** <***n***> — sets collapsing input limit per macrocell function (default is 36 input signals).
- **-localfbk** — uses local feedback to improve timing when possible (XC9500 only, except XC9536). Default is to use local feedback only when needed to meet timespecs.
- **-loweffort** — specifies low fitting effort.
- **-lowpwr** — set the default power mode to low for all macrocells in the design (default is standard power).
- **-nodt** — disables automatic transformation between D-type and T-type macrocell registers.

- **`-nogck`**— disables global clock optimization.
- **`-nogsr`** — disables global set/reset optimization.
- **`-nogts`** — disables global output-enable (GTS) optimization.
- **`-nomlopt`** — disables multi-level logic optimization.
- **`-nota`** — bypasses the timing analyzer so that no summary static timing report (*design_name*.`tim`) is generated.
- **`-notsim`** — prevents generation of timing simulation database file (*design_name*.**`nga`**).
- **`-notiming`** — inhibits the default global timing optimization performed by the fitter; only paths with timing specifications are optimized to improve timing.
- **`-nouim`** — disables formation of “wire-and” functions in the FastCONNECT structure of XC9500 devices.
- **`-noxor`** — disables factorization and transformation between pure sum-of-products logic and logic using macrocell XOR-gate.
- **`-p`** *part_type* — specifies the target CPLD device type or set of devices from which to choose (default is automatic device selection from the XC9500 family); where *part_type* can be:
  - 9500 = any XC9500 family device (auto selection)
  - 9500xl = any XC9500XL family device (auto selection)
  - “*95ddd[xl][-ss][-pppp]*” — where *95ddd* is the device code (such as 95108), *ss* is the speed grade, *pppp* is the package code (such as PQ160), and an asterisk (*) can be used as a wildcard string (quotes required around *part_type* when asterisk is used). You must include the device code suffix “xl” to select any devices from the XC9500XL family. For example, “95*xl-*-pc*” selects any XC9500XL device in a pc-type package.
- **`-pinfbk`** — uses pin feedback to improve timing when possible.
- **`-pinlock`** — uses the guide file (*design_name*.**`gyd`**) from the last successful invocation of the fitter to reproduce the same pin locations (default is automatic pin assignment).
- **`-pterms`** *nn* — sets the limit to nn for the number of product terms allowed as a result of collapsing (default=20).

Note: If you have complex combinational logic in your design, such as state machines, comparators, etc., you may need to specify the **-pterms** option with a limit higher than 20 to achieve higher performance results. Refer to the *Controlling Logic Optimization* section in Chapter 2 for information.

- **-s** *signature* — specifies the user signature string to be programmed into the device for identification purposes, where *signature* is a string of 1-4 alphanumeric characters (default is the design name truncated to 4 characters).
- **-slowslew** — sets the default output slew-rate to slow (default is normally fast).
- **-ucf** *file_name*— reads user constraints from *file_name*.**ucf** file (by default, the fitter reads *design_name*.ucf if it exists).
- **-xactfit** — Use this option only if you have a design implemented in XACT v6 and cannot get the same pinout using the current software. If this is not used, advanced fitting is used as default.

Compiling Behavioral Modules for Schematics

If you are developing a schematic-based design using some other schematic entry tool (such as Viewlogic or Mentor), you can include module symbols in your schematic that are functionally defined using Synopsys VHDL or Verilog HDL. These are called "behavioral modules".

This section describes how to prepare a synthesis-based behavioral module using Synopsys FPGA Compiler or Design Compiler. Behavioral modules are represented by custom symbols in the schematic design. In general, the names of the pins on your behavioral module symbol should match the names of the top-level entity ports in your Synopsys source file. Refer to the *Schematic Design Guide* for information on how to include the behavioral module symbol in your schematic design.

The procedure for compiling a behavioral module is similar to the procedure for compiling a complete CPLD design, as described earlier in this chapter. For behavioral modules, however, you do not specify device I/O pads; you would therefore omit the **set_port_is_pad** and **insert_pads** commands. Also, many of

the dc_shell attributes, such as LOC, are not applicable to behavioral modules.

Behavioral modules are compiled and written as EDIF netlists by performing the following steps:

Step 1 - Entering the dc_shell Environment

Enter the Synopsys dc_shell environment by entering the following Synopsys command on the UNIX command line:

```
dc_shell
```

You will see the dc_shell prompt.

Step 2 - Analyzing the Module

To interpret your synthesis module and verify that it is free of errors, enter the following Synopsys command for VHDL modules:

analyze -format vhdl *module_name***.vhd**

or, for Verilog HDL modules:

analyze -format verilog *module_name***.v**

If your source file contains initial signal values (which are used only for functional simulation) they will cause warnings that can be safely ignored; these initial signal values are not used during synthesis. Actual register initial states are set using attributes, as described in Chapter 2.

If the **analyze** command finds errors, you will need to make the necessary corrections to your source file and repeat the analyze command before continuing with synthesis.

Step 3 - Elaborating the Module

To derive a logical design, based on your VHDL/HDL description, enter the following Synopsys command:

elaborate *entity_name*

where *entity_name* is the name of your top-level entity in your module.

During this step, the compiler displays information about all registers and 3-state buffers encountered in your module.

You are now ready to compile your module using the XC9000 synthesis.

Step 4 - Compiling Your Module

When you compile your module, the Synopsys synthesizer uses the components in the Xilinx XC9000 technology library to create an actual implementation of your module. The library used during compilation is defined by the dc_shell *target_library* variable, typically specified in your `.synopsys_dc.setup` file.

To synthesize your module based on target CPLD technology library, enter the following Synopsys command:

```
compile [-map_effort low]
```

The mapping effort parameter is optional. However, it is recommended that you set it to LOW to save compilation time. The synthesizer does not perform any speed or area optimization for CPLD designs; this optimization is performed after compilation by the CPLD fitter.

Step 5 - Specifying Attributes

The only attribute that you may set for behavioral modules is:
Register initial states

For example:

```
set_attribute Q_REG init -type string S
```

If you set any attributes on design objects in any hierarchy levels below the top level, you must flatten the design so that your attributes get written to the EDIF netlist. Enter the following command to flatten your deign:

```
ungroup -all -flatten
```

Step 6 - Writing the Netlist

Write your synthesized module file in EDIF netlist format by entering the following Synopsys command:

```
write -format edif -hierarchy -output
module_name.sedif
```

where:

- **`-format edif`** specifies the EDIF file format.
- **`-hierarchy`** specifies that all levels of the module hierarchy are to be written.
- **`-output`** *module_name*`.sedif` specifies your output file name, which should be the same as your source file name, with the extension .sedif.

The EDIF file produced by the Compiler will be read when the fitter finds the behavioral module symbol in your schematic design.

Chapter 4

Simulating your Design

This software supports both functional and timing simulation of VHDL designs using the VSS simulator. This package also supports functional and timing simulation of Verilog designs. This chapter shows you how to prepare designs for simulation and how to use a test bench. It contains the folling sections:

- "Recommended CPLD Simulation Strategy" section
- "Controlling the Initial States of Registers" section
- "Creating a Test Bench File" section
- "Functional Simulation Using VSS" section

Recommended CPLD Simulation Strategy

Because of the flexibility of the simulation environment, there are many ways in which you can verify your design. The following steps, which are explained in subsequent sections, show you one recommended flow for CPLD simulation.

1. Specify the initial states of your registers. If you use attributes to control the initial states of the registers in your actual design implementation, you should also re-specify those initial states in your source VHDL design file for functional simulation.
2. Create a test bench file. By following the guidelines described in this chapter, the same test bench can be used for both functional and timing simulation without modification.
3. Perform functional simulation. This allows you to debug the logic in your source design before implementing a CPLD.
4. Implement the design in a CPLD. This provides the necessary physical resource information necessary for timing simulation.

5. Prepare the timing model. The **ngd2vhdl** command or Simulation Output options in the Design Manager prepare the VITAL (the **ngd2ver** command prepares a Verilog HDL timing model) timing model of your design for simulation.
6. Perform timing simulation. By re-using the functional simulation test bench file, you can easily compare results and prevent errors that can be caused by accidental differences between separate test bench files.

All of these preparation and simulation steps are demonstrated in the design example shown in the "Getting Started with Xilinx CPLDs" chapter.

Controlling the Initial States of Registers

This section shows you how to declare the initial states of registers in your design for simulation. If your design does not depend on the initial states of any registers, then you can skip this section and go to the next section, "Creating a Test Bench File".

The actual initial states of your registers are determined by the initial state attributes specified in DC Shell during compilation. By default, all registers initialize to zero in CPLD designs.

The timing simulation model produced by the Xilinx software reflects the actual register initial states that are implemented in the device.

Simulating Power On Initialization

All registers in Xilinx CPLDs are initialized when power is applied. You must perform the necessary steps to initialize the registers in your design at beginning of timing simulation for consistent simulation results.

The following sections show you how to set up your design to perform register initialization for both functional and timing simulation.

Preparing for Timing Simulation

When you generate your timing simulation model, **ngd2vhdl** or Simulation Output Options on the Design Manager automatically create a new signal in the model that you can stimulate in your test bench at the beginning of the simulation waveform to simulate

power-on. For CPLD designs an internal net named PRLD is normally created. If you want to make the XC9000 PRLD signal accessible to a VHDL test bench, you can specify the **ngd2vhdl -gp** option to create a port named PRLD.

When simulating, you must first pulse PRLD high, prior to exercising the logic, to get all the registers into their initial states. If you used the **ngd2vhdl -gp** option to create a PRLD port for your XC9000 design simulation, you must list PRLD in the port list of the CPLD in your test bench.

The `PRLD` signal is used for timing simulation only; it is not used for functional simulation and it cannot be used in your design. However, if you include it in your functional simulation test bench, that test bench can also be used later for timing simulation without modification.

If you include the `PRLD` signal in your test bench file for functional and timing simulation, you must also include PRLD in your port declarations in your source design file as follows:

```
port (... PRLD : in std_logic ...);
```

`PRLD` is not used anywhere else in your design. It will be ignored during synthesis; you will get warnings about the unconnected port during the **Compile** and **Insert_pads** operations. The Xilinx fitter software will also discard the unconnected port during implementation.

If the behavior of your CPLD design does not depend on the power-up state of any register, you do not need to use the **ndg2vhdl -gp** option and you do not need to pulse the PRLD net during simulation.

Preparing for Functional Simulation

Simulate register initialization by defining, in your VHDL source design file, the initial values for registered signals. Use signal declarations such as the following:

port *signal_name: port_direction signal_type := initial_value;*
signal *signal_name: signal_type := initial_value;*
variable *signal_name: signal_type := initial_value;*

For example:

```
port Nreg5: out std_logic := '0';
signal Qreg6: std_logic := '1';
variable Qreg: std_logic_vector := "00000001";
```

These initial values are used only for functional simulation; they are not used during synthesis and the compiler will give you a warning that these values are being ignored. Also, these initial values are not used by the Xilinx software for device implementation because the initial values from these declarations are not written into the netlist.

You are now ready to create a test bench file.

Creating a Test Bench File

This section shows you how to create a test bench file that can be used for both functional and timing simulation. The example test bench presented here consists of a VHDL file containing one instance of a CPLD design being tested and a procedure that applies simulation input waveforms to the CPLD.

Initializing Registers

If you are using the **`-gp`** option, prepare your test bench to pulse the appropriate initialization port.

In the test bench, include the `PRLD` input port in the CPLD component declaration and in its instance port map as shown in the following section. At the beginning of the simulation sequence, apply an active-high pulse to the `PRLD` port to initialize the registers during timing simulation. The pulse is ignored during functional simulation because the `PRLD` signal is not used anywhere in the source design. For functional simulation, all registers are initialized before the first simulation cycle (at time zero) by the initial values declared in your source design file.

If you are simulating a CPLD design using Verilog, pulse the internal PRLD net high at the beginning of simulation to initialize your registers.

Configuration Declaration

For any design or test bench you wish to simulate using VSS, you must declare a configuration which identifies the specific architecture you are applying to a design. When you invoke the VSS simulator,

you must select the name of a configuration that has been previously analyzed.

The following example shows a typical configuration declaration in a VHDL test bench file for a CPLD design for which the **ngd2vhdl -gp** option has been enabled. If the test bench is always used to simulate the design source file, the design does not need its own configuration declaration.

```
entity scan_tb is
end scan_tb;                                    --test bench has no ports--

architecture test of scan_tb is
     component scan
          port (CLOCK, CLEAR, ...       --same as in scan.vhd--
               PRLD : in std_logic);
     end component;
     signal CLOCK, CLEAR, ...PRLD;      --same as ports of scan.vhd--
begin
     UUT: scan port map (CLOCK, CLEAR, ... PRLD);--connect local signals
                                                     to ports--
     driver: process begin
          PRLD <= '1';CLEAR <='0';...   --assert initial values on all
                                           inp ports--
          wait for 25ns;                --wait, --
          PRLD <= '0';...               --release PRLD before applying
                                           other input transitions--
         wait;                       --after all inputs, suspend process--

end process;

end test;

configuration CFG_SCAN_TB of scan_tb is
   for test
        end for;
end CFG_SCAN_TB;
```

After you have created a test bench file, you are ready to begin using a VSS simulator (such as vhdldbx) for functional simulation.

Functional Simulation Using VSS

Functional simulation is used to debug your logic before fitting your design into a CPLD. The Xilinx CPLD Synopsys Interface fully supports functional simulation using the Synopsys VSS simulator, including all instantiated cells from the XC9000 library.

To prepare a test bench configuration for simulation, you must analyze each of the design and test bench source files in the proper bottom-up sequence.

The following procedure uses the stand-alone VHDL Analyzer (vhdlan) and the VHDL Debugger Simulator (vhdldbx).

1. Analyze your source CPLD design file. Enter the following UNIX command:

   vhdlan *design_name*.**vhd**

   For example:

   vhdlan scan.vhd

2. Analyze the test bench file. Enter the following UNIX command:

   vhdlan test_bench_name.vhd

   For example:

   vhdlan scan_tb.vhd

3. Invoke the Synopsys VSS Simulator. Enter the following UNIX command to invoke the VHDL debugger:

   vhdldbx

You are then prompted for a configuration name. Select the name of the configuration declared in the test_bench_name.vhd file. For example, for the scan design, select the following:

```
CFG_SCAN_TB
```

The vhdldbx selector window appears.

After you click OK, the vhdldbx user interface window appears.

To run your simulation, typically you first declare the signals you want to display in a trace window. For example, to display all signals appearing on the CPLD pins, you can enter the following vhdldbx command:

```
trace *'signal.
```

To run all the simulation vectors in your test bench, select the RUN command. A trace window will be displayed.

After functional simulation is successful, you are ready to implement your design and create the physical layout information required for timing simulation.

Design Implementation

After you have debugged your design using functional simulation, you can compile it using synthesis and implement it in a CPLD using the Xilinx fitter. Design implementation is a prerequisite for performing timing simulation.

You can use DC Shell or you can use the Synopsys graphic interface (Design Analyzer) to create the EDIF netlist file required by the Xilinx fitter. This gate-level netlist file consists of cells from the XC9000 library but does not contain timing information. The Xilinx fitter processes the netlist file and places the logical design into the physical architecture of a target CPLD.

After the design is implemented by the Xilinx fitter, the actual target device timing information is available for timing simulation.

The following steps show you an overview of the CPLD implementation procedure.

1. Analyze the source design file. This must be repeated in the synthesis environment (DC Shell); the results of vhdlan cannot be used for synthesis.
2. Compile the design, targeting the XC9000 library, and create a netlist.
3. Run the Xilinx fitter, using the `cpld` command or the Design Manager to process the netlist.

Usually, simulation is not repeated until after fitting when all actual timing results have been applied.

Examine the appropriate fitter report files to verify that the fitter completed successfully. You may wish to target a smaller device or add more functions to your design if there are remaining unused resources.

After design implementation, you are ready to prepare the timing model for timing simulation.

Preparing the Timing Simulation Model

From Command Line

The **ngd2vhdl** command translates the timing simulation database file (*design_name*.nga) produced by the **cpld** command into the required VHDL simulation output file(s).

If you prefer to create a PRLD input port and control it using your testbench, create your timing simulation model as follows:

ngd2vhdl *design_name* **-w -gp** *design_name*_**time**

If you prefer to use the automatic ROC cell to pulse the PRLD net, create your timing simulation model as follows:

ngd2vhdl *design_name* **-w** *design_name*_**time**

Invoking the **ngd2vhdl** command with no parameters produces a listing of all available command-line options.

The *design_name* is the name of the design as specified when running the **cpld** command, without path qualifiers and without extension.

The **ngd2vhdl** command produces a structural VHDL file (*design_name*_time.vhd) and an SDF-formatted timing back-annotation file (*design_name*_time.sdf), for use with the Synopsys VSS simulator or other VITAL-compatible simulator. A procedure for using the VSS simulator is described below.

Similarly, the **ngd2ver** command produces a structural Verilog HDL file (*design_name*_time.v) and an SDF-formatted timing back-annotation file (*design_name*_time.sdf).

The **-gp** option forces **ngd2vhdl** or **ngd2ver** to add a PRLD input port to the output VHDL or Verilog HDL file for CPLD designs. This allows the PRLD initialization signal to be stimulated as a top-level port by your test bench.

Note: When the fitter processes your design, some of your original nodes may be removed or replaced due to logic optimization. Such nodes cannot be viewed or stimulated during timing simulation. All of the device I/O port signals and register output signals are always maintained.

From Design Manager

1. Open the Options dialog box using one of the following methods.
   - If you haven't yet performed design implementation, from the Design Manager menu, select **`Design`** → **`Implement`**. Click on the **`Options`** button in the Implement dialog box.
   - If you have already completed implementation, select the revision, then go to the Flow Engine by selecting **`Tools`** → **`Flow Engine`**. From the Flow Engine menu, select **`Setup`** → **`Options`**.

   The **`Options`** dialog box appears.
2. In the Options dialog box, select the **`Produce Timing Simulation Data`** check box then select the **`Edit Template`** softkey.

   The **`Implementation Options`** template appears.
3. Select the **`Interface`** tab on the **`Implementation Options`** template.
4. Select **`VHDL`** or **`Verilog`** as the output format.
5. Click **`OK`**.

   When you implement the design, the Flow Engine produces timing simulation data files. Each time the data is produced, it is automatically exported to your design directory.

You can now use these files to simulate the design with a supported third party simulation tool.

Timing Simulation Using VSS

If you prepared your test bench as described earlier you can use the same test bench for timing simulation as used for functional simulation. By using the same test bench you can easily verify that the functionality of the device after mapping matches the functionality of your source design. You also eliminate any risk of errors from accidental differences between separate test bench files.

1. Analyze the timing simulation model produced by **`ngd2vhdl`**:

   `vhdlan` *design_name*_**`time.vhd`**

   For example:

```
vhdlan scan_time.vhd
```

2. Analyze the test bench file name as used for functional simulation. Enter the following UNIX command:

   vhdlan *test_bench_name*.**vhd**

   For example:

   ```
   vhdlan scan_tb.vhd
   ```

   The simulation data base now contains the test bench design which interfaces to the chip through your source design entity read in step 1 but it contains the timing model architecture read in step 2.

3. Invoke the Synopsys VSS Simulator. Enter the following UNIX command:

   ```
   vhdldbx
   ```

   You are then prompted for the configuration named in the *test_bench_name*.vhd file. For example, for the scan design, select the following:

   ```
   CFG_SCAN_TB
   ```

   Before clicking "OK" you must specify the timing backannotation file information in the Arguments box.

All back-annotated timing in the .sdf file is applied to various instances within the *design_name*_time.vhd file. However, if you are simulating with a test bench, you must specify (to the simulator) the CPLD design instance to which you want to apply the back-annotated timing. It can then find all the referenced instances.

If you are using **vhdldbx** you need to specify two parameters:

- The sdf_top instance in the test bench configuration to which the back-annotated timing is applied:

  -sdf_top *chip_instance_name*

  For example:

  ```
  -sdf_top /scan_tb/UUT
  ```

  All back-annotated timing parameters in the **.sdf** file are applied relative to the chip instance.

- The file name of the **.sdf** backannotation timing file:

-sdf *design_name*_**time.sdf**

For example:

```
-sdf scan_time.sdf
```

You can specify these parameters either in the dialog box which appears after invoking **vhdldbx**, or on the UNIX command line as you invoke **vhdldbx**.

The command line invocation format is:

vhdldbx -sdf_top *chip_instance_name* **-sdf **
*design_name*_**time.sdf** *configuration_name*

For the scan design example, you should enter the following:

```
vhdldbx -sdf_top /scan_tb/UUT \

-sdf scan_time.sdf CFG_SCAN_TB
```

Note: If you use the **-tb** option of the **ngd2vhdl** command to create a testbench template file (**.tvhd**), all the instance names in the **.sdf** timing back-annotation file will be prefixed with "UUT/". In this case, you would omit the instance name "/UUT" from your **vhdldbx -sdf_top** parameter. For example, if you prepared the scan design using the command:

```
ngd2vhdl -w -tb scan scan_time
```

then you would invoke the VSS simulator using the command:

```
vhdldbx -sdf_top /scan_tb -sdf scan_time.sdf \
CFG_SCAN_TB
```

Now you can run the same simulation vectors for timing simulation as you ran for functional simulation. However, in timing simulation, the registers are set to their initial states in response to the active-high pulse on PRLD.

Appendix A

Library Component Specifications

This appendix describes each of the components (cells) in the Xilinx XC00 synthesis library.

Component Name	Component Description	Inferable
AND2-AND8	AND Gates	X
BUF	Buffer	
BUFE	Tristate buffer (not available in XC9500XL or XC9500XV designs)	X
BUFGSR	Global set/reset input buffer	
BUFGTS	Global tristate control input buffer (uses clock-enable p-term in XC9500XL and XC9500XV)	
BUFG	Global clock (FastCLK) input buffer	
FDCE	D-Type Flip-Flop with Clear and Clock Enable	
FDCE_X	D-Type Flip-Flop with Clear and Clock Enable	X
FDCP	D-Type Flip-Flop with Asynchronous Clear and Preset	X
FDPE	D-Type Flip-Flop with Preset and Clock Enable (uses clock-enable p-term in XC9500XL and XC9500XV)	
FDPE_X	D-Type Flip-Flop with Preset and Clock Enable	X
IBUF	Input Buffer	X
INV	Inverter	X
IOBUFE	Bi-Directional I/O Buffer	X
IOBUFE_F	Bidirectional I/O Buffer--fast slew rate	X
IOBUFE_S	Bidirectional I/O Buffer--slow slew rate	X

Component Name	Component Description	Inferable
LD	D-Type Latch	X
OBUF	Output Buffer	X
OBUF_F	Output Buffer--fast slew rate	X
OBUF_S	Output Buffer--slow slew rate	X
OBUFE	Tristate Output Buffer	X
OBUFE_F	Tristate Output Buffer--fast slew rate	X
OBUFE_S	Tristate Output Buffer--slow slew rate	X
OR2-OR8	OR Gates	X
XOR2-XOR8	XOR Gates	X

AND2 — AND8

AND2 through AND8 are AND gates with 2 to 8 inputs.

Inferencing

The synthesizer uses these components when creating functions that require AND gates.

Component Instantiation

```
U1: AND2 port map (O=>out,I1=>in2,I0=>in1);
```

BUF

BUF is a non-inverting buffer.

Inferencing

The synthesizer does not use this component by inference.

Component Instantiation

```
U1: BUF port map (O=>out_port, I=>in_port);
```

BUFE

BUFE is a non-inverting tristate buffer, with active-high enable. BUFE must not appear in XC9500XL or XC9500XV designs.

Inferencing

The synthesizer uses these components when creating functions that require tristate buffers that drive internal signals.

Component Instantiation

```
U1: BUFE port map (O=>ts_out, I=>inp, E=>enable);
```

BUFG

BUFG is an input buffer used to drive the Global clock signal (GCK).

BUFG signals may be used for active-high or active-low (inverted) clocking, and for any other logic functions in the design.

Inferencing

The synthesizer does not use this component by inference.

Component Instantiation

```
U1: BUFG port map (O=>global_clk, I=>in_port);
```

BUFGSR

BUFGSR is an input buffer used to drive the Global set/reset signal. BUFGSR signals can drive the CLR or PRE input of any flip-flop components, and any other logic functions in the design.

Inferencing

The synthesizer does not use this component by inference.

Component Instantiation

```
U1: BUFGSR port map (O=>global_sr, I=>in_port);
```

BUFGTS

BUFGTS is a an input buffer used to drive the global tristate control signal (GTS). BUFGTS may be used either active-high or active-low (inverted) to drive the E input of OBUFE and IOBUFE type components, and any other logic functions in the design.

Inferencing

The synthesizer does not use this component by inference.

Component Instantiation

```
U1: BUFGTS port map (O=>global_oe, I=>in_port);
```

FDCE, FDCE_X

FDCE and FDCE_X are edge-triggered D-type flip-flops with clear and clock enable.

Inferencing

The synthesizer uses the FDCE_X component for all flip-flop functions requiring clock-enable, but not requiring asynchronous preset. The synthesizer does not use the FDCE component by inference.

Component Instantiation

```
U1: FDCE port map (Q=>out, D=>data, C=>clock,
CLR=>async_clr, CE=>clk_enable);
```

FDCP

FDCP is an edge-triggered D-type flip-flop with preset and clear.

Inferencing

The synthesizer uses this component for all functions that require D-type registers, but not clock-enable.

Component Instantiation

```
U1: FDCP port map (Q=>out, D=>data, C=>clock,
CLR=>async_clr, PRE=>async_set);
```

FDPE, FDPE_X

FDPE and FDPE_X are edge-triggered D-type flip-flops with preset and enable.

Inferencing

The synthesizer uses the FDPE_X component for all flip-flop functions requiring clock-enable and asynchronous preset. The synthesizer does not use the FDPE component by inference.

Component Instantiation

```
U1: FDPE port map (Q=>out, D=>data, C=>clock,
PRE=>async_preset, CE=>clk_enable);
```

IBUF

IBUF is an input buffer.

Inferencing

The synthesizer uses these components to receive inputs from device pins.

Component Instantiation

```
U1: IBUF port map (O=>received_signal,
     I=>in_port);
```

INV

INV is an inverter.

Inferencing

The synthesizer uses this component for signal inversion.

Component Instantiation

```
U1: INV port map (O=>not_in1, I=>in1);
```

IOBUFE, IOBUFE_F, IOBUFE_S

IOBUFE is a non-inverting tristate I/O buffer with active-high enable. Output slew rate is controlled by CPLD fitter options (default is fast).

IOBUFE_F is an I/O buffer with fast output slew rate.

IOBUFE_S is an I/O buffer with slow output slew rate.

Inferencing

The synthesizer uses these components to transfer signals to and from bidirectional device I/O pins.

Component Instantiation

```
U1: IOBUFE port map (O=>received_signal,
     IO=>inout_port, I=>driving_signal,
     E=>output_enable);
```

LD

LD is a D-type latch.

Inferencing

The synthesizer uses LD for all transparent latches. This component can be used by inference.

Component Instantiation

```
U1: LD port map (Q=>out, D=>data,
     G=>latch_enable);
```

OBUF, OBUF_F, OBUF_S

OBUF is an output buffer. Output slew rate is controlled by CPLD fitter options (default is fast).

OBUF_F is an output buffer with fast output slew rate.

OBUF_S is an output buffer with slow output slew rate.

Inferencing

The synthesizer uses this component when creating external outputs to device pins.

Component Instantiation

```
U1: OBUF port map (O=>out_port,
     I=>driving_signal);
```

OBUFE, OBUFE_F, OBUFE_S

OBUFE is a tristate output buffer with active-high enable. Output slew rate is controlled by CPLD fitter options (default is fast).

OBUFE_F is a tristate output buffer with fast output slew rate.

OBUFE_S is a tristate output buffer with slow output slew rate.

Inferencing

The synthesizer uses this component when creating tristate external outputs which connect to device pins.

Component Instantiation

```
U1: OBUFE port map (O=>out_port,
     I=>driving_signal, E=enable);
```

OR2 — OR8

OR2 through OR8 are OR gates with 2 to 8 inputs.

Inferencing

The synthesizer uses these components when creating functions that require OR gates.

Component Instantiation

```
U1: OR2 port map (O=>out, I1=>in2, I0=>in1);
```

XOR2 — XOR8

XOR2 through XOR8 are XOR gates with 2 to 8 inputs.

Inferencing

The synthesizer uses these components when creating functions that require XOR gates.

Component Instantiation

```
U1: XOR2 port map (O=>out, I1=>in2, I0=>in1);
```

Appendix B

Attributes

Attributes are used to control how the software uses the architecture specific features of CPLDs. See the device data sheets for more information about these device features.

This appendix contains the following sections:

- "Instantiated Attributes" section
- "Synopsys Attributes" section
- "Timing Constraints" section

Instantiated Attributes

Instantiated attributes are applied by instantiating the following components in your design and connecting them to the affected signal.

KEEP

This attribute inhibits the software from optimizing the logic that drives the signal passing through the KEEP cell.

To specify that a signal is to remain as a macrocell output, use:

```
U1: KEEP port map (O=>outgoing_signal,
I=>incoming_signal);
```

Synopsys Attributes

The following attributes can be specified in the Synopsys dc_shell for Xilinx CPLD designs. See the Synopsys Design Compiler manual for more information on using the **set_attribute** and **set_pad_type** commands.

Global Input Ports

The dc_shell **set_pad_type** command with parameter **-exact** explicitly controls allocation of global input buffers. The format is:

set_pad_type -exact *buffer_type port_name*

where *buffer_type* is one of BUFG, BUFGTS, or BUFGSR.

The dc_shell **set_pad_type** command with parameter **-exact BUFG** is used to explicitly control the use of global clock pins, and reference an input port of your design. For example:

```
set_pad_type -exact BUFG clock1
```

If you need to explicitly control the use of global 3-state control pins, you can specify the **set_pad_type** command in your dc_shell script with the parameter **-exact BUFGTS**, and reference an input port of your design. For example:

```
set_pad_type -exact BUFGTS enable1
```

If you need to explicitly control the use of the global set/reset pin, you can specify the **set_pad_type** command in your dc_shell script with the parameter **-exact BUFGSR** and reference an input port of your design. For example:

```
set_pad_type -exact BUFGSR reset1
```

Note: The cell names **BUFG**, **BUFGTS**, and **BUFGSR** must be upper case.

Note: The **set_pad_type** command must be invoked before the **insert_pads** command.

As an alternative, you can use a UCF file to apply the BUFG property to an ordinary input port after synthesis. The UCF file syntax is:

NET *port_name* **BUFG=***buffer_type***;**

where *buffer_type* is one of **CLK**(for BUFG), **OE**(for BUFGTS), or **SR**(for BUFGSR).

Output Slew Rate

The **set_pad_type -slewrate** command controls the output buffer slew rate.

The format is:

set_pad_type -slewrate *slew_value port_list*

where *slew_value* is either **HIGH** (for slow slew rate) or **NONE** (for fast slew rate), and *port_list* is either one or more output port names or the keyword **all_outputs**().

- **HIGH** - Slows the output signal transition time and thus reduces internal switching noise, and edge rates.
- **NONE** - Maintains fast output signal transition time (default).

By default, the slew rate of all the output buffers (OBUF, OBUFE and IOBUFE) is controlled by the fitter options (default fast). However, in order to reduce possible noise problems, it is recommended that you use the fast transition default only for those output signals that require maximum speed.

To set all outputs to slow slew rate, use the following command:

```
set_pad_type -slewrate HIGH all_outputs ()
```

After you have globally changed all outputs to the **HIGH** option (for slow signal transition) you can set any individual output for fast signal transition by using the following command:

set_pad_type -slewrate NONE *port_name*

Note: The **set_pad_type** command must be invoked before the **insert_pads** command.

As an alternative, you can use a UCF file to apply the **FAST** or **SLOW** property to an ordinary output port where no slewrate was set using the **set_pad_type** command during synthesis. The UCF file syntax is:

NET *port_name* **FAST**;

or

NET *port_name* **SLOW**;

Pin Assignment

The **LOC** attribute is used to specify the pins on which to place output signals.

The format is:

set_attribute *port_name* **LOC** *pin_number* **-type string**

where:

- *port_name* = The name of the top-level design port.

The format of *pin_number* is:

- `P`*nn* for PC and PQ and VQ packages, where *nn* is the pin number.
- *rc* for PG and BG packages, where *rc* are the row letter and column number.

For example, for PC and PQ packages:

```
set_attribute RDY LOC p23 -type string
```

For example, for PG and BG packages:

```
set_attribute RDY LOC K13 -type string
```

Note: The pin assignment attribute overrides previously saved pinouts when running **`cpld`** with the **`-pinlock`** option.

UCF/NCF File

```
NET port_name LOC=pin_number;
for instance,
NET ABC LOC=P12;
```

Function Block and Macrocell Assignment

The LOC attribute can also be used to specify the function block or macrocell number into which a node is to be mapped. Function block assignment may be useful to take advantage of the high-speed local feedback paths of XC9500 devices.The syntax is:

```
set_attribute node_name loc FBnn -type string
```

or

```
set_attribute node_name loc FBnn_mm -type string
```

where *node_name* is the name of the signal net appearing at the output of a CPLD macrocell (or the cell driving that signal), *nn* is a legal function block number for the target device, and *mm* is a legal macrocell number within the function block.

UCF/NCF File

```
NET node_name LOC=FBnn[mm];
```

Register Initial State

The **init** attribute is used to specify the initial (power up) state of registers in your design.

The syntax is:

```
set_attribute register_name init state -type string
```

where:

- *register_name* is the name of a register cell in your design or the name of its output net
- *state* is either S (set) or R (reset)

For example:

```
set_attribute "QOUT<2>_reg" init S -type string
```

UCF/NCF File

```
NET net_name INIT=state;
```

or

```
INST inst_name INIT=state;
```

Macrocell Power Mode

The **pwr_mode** attribute can be used to select the power consumption (standard or low-power) of specific macrocells in the design. The syntax is:

```
set_attribute node_name pwr_mode mode -type string
```

where *node_name* is the name of a signal (net) appearing at the output of a CPLD macrocell (or the cell that drives the signal), and *mode* is either **std** (standard) or **low**.

Note: If the logic driving the named signal is collapsed by the fitter, the **pwr_mode** attribute will be ignored.

UCF/NCF File

```
NET node_name PWR_MODE= mode;
```

Timing Constraints

The following **dc_shell** timing constraints are available:

- **create_clock**
- **set_max_delay**
- **set_output_delay**
- **set_input_delay**

create_clock

You can use the following command to declare a clock input port and place a timing specification on the specified clock net. The register-to-register delays between all flip-flops on the named clock will be constrained by the specified period.

The **create_clock** command creates a cycle time specification on the specified clock signal as follows:

create_clock *clock_port* **-period** *delay*

where *clock_port* is the name of the clock input port and *delay* is the clock cycle time in nanoseconds.

Note: The Synopsys **max_period** command is not supported by the Xilinx fitter; use the **create_clock** command instead.

UCF/NCF File

NET *clock_port* **TNM=***clock_port*;

TIMESPEC TS*id***=PERIOD:***clock_port:delay*;

set_max_delay

The **set_max_delay** command specifies delay constraints for specific paths originating from input (or I/O) ports or flip-flop cells and terminating at output (or I/O) ports or flip-flop cells. The syntax of the **set_max_delay** command is:

set_max_delay *delay* **-from** *source* **-to** *destination*

For example, to specify the propagation delay from the CLEAR input port to the DONE output port:

```
set_max_delay 20 -from CLEAR -to DONE
```

Table 4-1 set_max_delay

Source	Destination	Affected Timing Path
input or I/O port (except clock)	output or I/O port	pad-to-pad propagation delay
input or I/O port (except clock)	register cell	register setup time from specified port(s) with respect to flip-flop's clock pin.
register cell	register cell	register-to-register delay (cycle time), regardless of each register's clock source (overrides create_clock period constraint covering same registers
register cell	output or I/O port	register clock-to-output delay from flip-flop's clock pin to output pad.
clock input port	output or I/O port	register clock-to-output delay from the specified clock input to specified output port
clock input port	register cell	not used for CPLD designs

UCF/NCF File

TIMESPEC TS*id***=FROM:***source***:TO:***destination***:***delay***;**

set_output_delay

The **set_output_delay** command establishes clock-to-output delay specifications based on values specified in the **create_clock** or **set_max_delay** constraints or creates tighter constraints for named output ports as follows:

set_output_delay *delay* **–clock** *clock output_port*

When the named output ports are driven by registers covered by a **create_clock** period constraint, the **set_output_delay** constraint specifies how much time (*delay*) before the next clock edge the named outputs need to become stable.

In this case, the **set_output_delay** constraint specifies the delay path between the clock input pin of the CPLD device and the named output pin(s) according to the following relationship:

set_output_delay_value = *create_clock_period_value* – *cpld_clock_to_output_delay*

where *set_output_delay_value* is the *delay* value specified in the **set_output_delay** constraint, *create_clock_period_value* is the *period* value specified in a previous **create_clock** constraint, and *cpld_clock_to_output_delay* is the desired worst-case propagation delay between the clock input pin and output pin(s) of the CPLD.

This command also changes the values of pad-to-pad or clock-to-output delay specifications created by the **set_max_delay** command, by making the constraints tighter by the amount specified by the *delay* value for the named outputs.

Note: When using the **set_output_delay** constraint, the named clock must be explicitly declared as a global clock input port by using the **set_pad_type -exact BUFG** command.

UCF/NCF File

NET *output_port* **OFFSET=OUT:***delay***:BEFORE:***clock***;**

set_input_delay

The **set_input_delay** command establishes register setup time specifications based on **create_clock** or **set_max_delay** commands or creates tighter constraints on named input ports as follows:

set_input_delay *delay* **–clock** *clock input_port*

When the named input ports feed into registers covered by a **create_clock** period constraint, the **set_input_delay** constraint specifies how much time (*delay*) after the previous clock edge the named inputs are expected to become stable.

In this case, the set_input_delay constraint specifies the setup time requirements between data input pin(s) and the clock input pin of the CPLD device according to the following relationship:

set_input_delay_value = *create_clock_period_value* – *cpld_external_setup_time*

where *set_input_delay_value* is the *delay* value specified in the **set_input_delay** constraint, *create_clock_period_value* is the *period* value specified in a previous **create_clock** constraint, and *cpld_external_setup_time* is the desired worst case setup time between the data input pin(s) and the clock input pin of the CPLD.

This command also changes the values of pad-to-pad delay or register setup time specifications created by the **set_max_delay** command, by making the constraints tighter by the amount specified by the *delay* value for the named inputs.

When using the **set_input_delay** constraint, the named clock must be explicitly declared as a global clock input port by using the **set_pad_type -exact BUFG** command as described in the section *Special I/O Ports* in chaper 2.

UCF/NCF File

NET *input_port* **OFFSET=IN:***delay***:AFTER:***clock***;**

Appendix C

Fitter Command and Option Summary

This appendix describes how to invoke the CPLD fitter, and the commands used to prepare functional and timing simulation models. All of the available fitter options are described. This appendix contains the following sections:

- "Design Manager" section
- "CPLD Command" section

Design Manager

The Design Manager invokes the Flow Engine (fitter) and option templates to control the fitting of your design.

Invoking the Fitter

1. From the Design Manager select the file you want to process.

   File → Open Project

   Select a file from the template's list or use the **Browse** key to search your directories for the file you want to process. If the file is listed on the template, highlight the file and click once on **Open**.

2. Go to the Flow Engine and select options:

   Tools → Flow Engine

   Setup → Options

3. The Design Implementation Option menu appears. Select:

`Edit Template`

4. Then select from the five tabs all the options you want to use and press **`OK`**.
5. To run the fitter, click once on the **`run`** key found in the Flow Engine.

Fitter Options

This section describes fitter parameters that can be entered from the Design Manager.

The Implementation Options menu contains five tabs of options for the fitter. The following summarizes fitter options:

- **`Basic`** → **`Default Output Slew Rate`** — sets default output slew-rate to **`FAST`** or **`SLOW`** (default is **`FAST`**).
- **`Basic`** → **`Macrocell Power Setting`** — Sets default power mode for all macrocells in the design to standard or low-power (default is **`Std`** power).
- **`Basic`** → **`Create Programmable Ground Pins`** — creates additional ground pins on unused I/Os (default is **`OFF`**).
- **`Basic`** → **`Use Design Location Constraints`**— if this is not checked, the program temporarily ignores all LOC attributes in the design, allowing the fitter to assign the locations of all I/O pins (default is **`ON`**).
- **`Basic`** → **`Use Timing Constraints`** — turn this selection off if you want to temporarily ignore all timing specification attributes in the design (default is **`ON`**).
- **`Basic`** → **`Use Global Clock(s)`** — Select this option to automatically use global clocks (**`GCK`**) for ordinary input signals used as clocks. The global clock may allow you to meet your timing constraints more easily. By default, this option is **`ON`**.
- **`Basic`** → **`Use Global Output Enable(s)`** — Select this option to automatically use global output enable (GTS) for ordinary input signals used as output enable constraints. Global output enable may allow you to meet your timing constraints more easily. By default, this option is **`ON`**.

- **`Basic → Use Global Set/Reset`** — Select this option to automatically use global set/reset (GSR) for ordinary input signals used as asynchronous clear or preset. By default, this option is **`ON`**.
- **`Advanced → Collapsing Input Limit`** — The maximum number of function block inputs allowed as a result of logic collapsing. Default is 36.
- **`Advanced → Collapsing Pterm Limit`** — The maximum number of product terms allowed as a result of collapsing (default=20 on **`Optimize Speed`** template; 90 on **`Optimize Density`** template).
- **`Advanced → Use Multilevel Logic Optimization`** — Spends additional time transforming the logic in your design to new logical structures that achieve better performance and density (default=**`ON`**).
- **`Advanced → Use Timing Optimization`** — enables the global timing optimization performed by the fitter; if this option is not selected, only paths with T-specs specified in the design are optimized to improve timing (default is **`ON`** in **`Optimize Speed`** template, **`OFF`** in **`Optimize Density`** template).
- **`Advanced → Enable D to T-Type Transform Optimization`** — if this box is checked (default), the fitter transforms between D-type and T-type registers.
- **`Advanced → Use Advanced Fitting`** — Select this option to enable an advanced fitting strategy that favors placing signals with common inputs in the same function block. This usually allows you to pack more logic into the same device. Disable this option if the software has trouble fitting a design that used to fit with an older version of software (by default, this option is **`ON`**).
- **`Advanced → Use Local Macrocell Feedback`** — enables the software to use local feedback in XC9500 devices (except XC9536) whenever possible. The local feedback path takes less time than the global feedback path. Using local feedback can speed up your design but can make it difficult to keep the same timing after a design change (default is **`OFF`**).
- **`Advanced → Use Local Pin Feedback`** — enables the software to use local I/O pin feedback in XC9500 devices whenever possible. The software uses the pin feedback path instead of the

FastCONNECT path for output pin signals that do not have 3-state control or slow slew rate (by default, this option is **`OFF`**).

- **`Interface`** → **`Macro Search Path`** — Use this option to add the specified search path to the list of directories to search when resolving instantiated Macros. Specify a macro search path or click **`Browse`** to look for a path to add as a macro search path. To specify multiple search paths, type in each directory name separated by a colon (:). A semicolon is automatically appended when you use the **`Browse`** button to select multiple search paths.
- **`Timing Reports`** → **`Produce Post Layout Timing Report`** — generates static timing report.
- **`Timing Reports`** → **`Timing Report Format`** — Select **`Summary`** to generate a report that contains summary information and design statistics. Select **`Detailed`** to generate a report that lists delay information for all nets and paths.
- **`Programming`** → **`Signature/User Code`** — Enter a unique text string in this field to identify the signature data. You can enter a string of up to four alphanumeric characters. The device programmer can read the signature, and the person running the device programmer can verify that the correct configuration data file is loaded. Use the JTAG Programmer to identify the configuration data signature (usercode) of a programmed XC9500 device.

CPLD Command

The **`cpld`** command invokes the CPLD design implementation software (the fitter). The command is run in a UNIX command window. Your current working directory must be set to the project directory which contains your design source netlist files before invoking **`cpld`**.

Invoking the Fitter

The format of the cpld command is:

`cpld` [*options*] *design_name*

Invoking the **`cpld`** command with no parameters produces a listing of all available command-line options.

The *design_name* is the name of the top-level design netlist file, without path qualifiers, and either with or without extension. If *design_name* is specified without extension, the **`cpld`** command searches for source files in the following order:

1. Synopsys Design Compiler or FPGA Compiler netlist (*design_name*.sxnf)
2. Xilinx PLUSASM equation file (*design_name*.pld)
3. XNF netlist (*design_name*.xnf)
4. Synopsys Design/FPGA Compiler EDIF netlist (*design_name*.sedif)
5. EDIF netlist (*design_name*.edn, *design_name*.edf or *design_name*.edif)
6. Xilinx NGO (unexpanded) database file (*design_name*.ngo)
7. Xilinx NGD (expanded) database file (*design_name*.ngd)

Fitter Options

The *[options]* field of the cpld command represents an optional list of one or more command-line parameters. Invoking the **`cpld`** command with just the design name and no option parameters runs the fitter with all default conditions, including automatic device selection.

The following are the **`cpld`** command-line parameters that apply to synthesis design entry:

- **`-autoslewpwr`** — reduces slew rate before reducing power mode if t-specs still met.
- **`-autopwrslew`** — reduces power mode and/or slew rate if timespecs can still be met.
- **`-detail`** — produces a detailed path timing report (*design_name*.tim) in addition to the default summary report.
- **`-grounds`** — creates programmable ground pins on unused I/Os.
- **`-ignoreloc`** — temporarily ignores all LOC attributes in the schematic, allowing the fitter to assign the locations of all I/O pins.

- **`-ignorets`** — temporarily ignores all timing specification attributes in the schematic.
- **`-inputs`** <*n*> — maximum number of function block inputs allowed as a result of logic collapsing. Default is 36.
- **`-localfbk`** — uses local feedback. Enables the software to use local feedback whenever possible. The local feedback path takes less time than the global feedback path. Using local feedback can speed up your design but can make it difficult to keep the same timing after a design change. XC9500 only.
- **`-loweffort`** — low fitting effort, to save processing time.
- **`-lowpwr`** — uses the low-power mode by default for all macrocells in the design (default is normally standard power).
- **`-nodt`** — disables transformation between D-type and T-type registers.
- **`-nogck`** — disables global clock optimization.
- **`-nogsr`** — disables global set/reset optimization
- **`-nogts`** — disables global output-enable (GTS) optimization.
- **`-nomlopt`** — disables multi-level logic optimization.
- **`-nota`** — do not generate a summary static timing report.
- **`-notiming`** — inhibits the default global timing optimization performed by the fitter; only paths with T-specs specified in the schematic are optimized to improve timing.
- **`-notsim`** — disables generation of timing simulation file (.nga).
- **`-nouim`** — disables implementation of AND functions in FAST-connect. XC9500 only.
- **`-noxor`** — disables transformation of sum-of-product XOR logic into macrocell XOR gates.
- **`-p`** ***part_type*** — specifies the target device type or set of devices from which to choose (default is automatic device selection from the XC9500 family); where *part_type* can be:
  - 9500 = any XC9500 family device (auto selection)
  - 9500xl = any XC9500XL family device (auto selection)
  - 9500XV = any XC9500XV family device (auto selection)

    - "*95ddd[xl][-ss][-pppp]*" — where *95ddd* is the device code (such as 95108), *ss* is the speed grade, *pppp* is the package code (such as PQ160), and an asterisk (*) can be used as a wildcard string (quotes required around *part_type* when asterisk is used).

- **`-pinfbk`** — uses pin feedback. Enables pin feedback whenever possible. The software uses the pin feedback path instead of the FastCONNECT path for output pin signals that do not have 3-state control or slow slew rate. XC9500 only.
- **`-pinlock`** — uses the guide file (*design_name*.gyd) from the last successful invocation of the fitter to reproduce the same pin locations (default is automatic pin assignment).
- **`-pterms`** ***nn*** — the maximum number of product terms allowed as a result of collapsing (default=20).
- **`-s`** ***signature*** — specifies the user signature string (up to 4 alphanumeric characters) to be programmed into the device for identification purposes (default is the design name).
- **`-slowslew`** — applies slow output slew-rate as default (default is fast).
- **`-ucf`** — reads user constraints from *filename*.ucf. By default, *design_name*.ucf is read if it exists.
- **`-xactfit`** — Use this option only if you have a design implemented in XACT v6 and cannot get the same pinout using the current software. The default is advanced fitting.