Copyright Notice and Proprietary Information
Copyright © 2002 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

Right to Copy Documentation
The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

“This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of ____________________________ and its employees. This is copy number ___________.”

Destination Control Statement
All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader’s responsibility to determine the applicable regulations and to comply with them.

Disclaimer
SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Registered Trademarks (®)
Synopsys, AMPS, Arcadia, C-Level Design, C2HDL, C2V, C2VHDL, CoCentric, COSSAP, CSim, DelayMill, DesignPower, DesignSource, DesignWare, Eagle®, EPIC, Formality, in-Sync, LEDA, ModelAccess, ModelTools, PathBlazer, PathMill, PowerArc, PowerMill, PrimeTime, RailMill, SmartLogic, SmartModel, SmartModels, SNUG, Solve-It, SolveNet, Stream Driven Simulator, System Compiler, TestBench Manager, TetraMAX, TimeMill, and VERA are registered trademarks of Synopsys, Inc.

Trademarks (™)
BCView, Behavioral Compiler, BOA, BRT, Cedar, ClockTree Compiler, DC Expert, DC Expert Plus, DC Professional, DC Ultra, DC Ultra Plus, Design Advisor, Design Analyzer, Design Compiler, DesignSphere, DesignTime, Direct RTL, Direct Silicon Access, DW8051, DWPCI, ECL Compiler, ECO Compiler, ExpressModel, Floorplan Manager, FoundryModel, FPGA Compiler II, FPGA Express, Frame Compiler, HDL Advisor, HDL Compiler, Integrator, Interactive Waveform Viewer, Liberty, Library Compiler, ModuleSource, Module Compiler, MS-3200, MS-3400, NanoSim, OpenVera, Physical Compiler, Power Compiler, PowerCODE, PowerGate, ProFPGA, Protocol Compiler, RoadRunner, Route Compiler, RTL Analyzer, Schematic Compiler, Scirocco, Shadow Debugger, SmartLicense, SmartModel Library, Source-Level Design, SWIFT, Synopsys Eagle®, SystemC, SystemC (logo), Test Compiler, TestGen, TimeTracker, Timing Annotator, Trace-On-Demand, VCS, VCS Express, VCSi, VHDL Compiler, VHDL System Simulator, VirSim, and VMC are trademarks of Synopsys, Inc.

Service Marks (SM)
DesignSphere, SVP Café, and TAP-in are trademarks of Synopsys, Inc.

All other product or company names may be trademarks of their respective owners.

Printed in the U.S.A.
Document Order Number: 36624-000 MA
FPGA Compiler II User Guide, v2002.05-FC3.7.1
Contents

What’s New in This Release ................................................. xii
About This Guide .............................................................. xv
Customer Support ........................................................... xviii

1. About FPGA Compiler II
   Features and Benefits .................................................. 1-3
   Methodology .............................................................. 1-5

2. DesignWizard Synthesis Flow
   Understanding the DesignWizard Flow ............................ 2-2
   Adding Design Source Files ........................................... 2-3
   Selecting Target Devices .............................................. 2-4
   Specifying Optimization and Output Options .................... 2-5
   Project File Location .................................................. 2-8

3. Advanced Synthesis Flow
   Creating a Project ..................................................... 3-3
Setting Project Synthesis Options ........................................... 3-5
Adding Design Source Files .................................................... 3-5
Debugging Design Source Files ................................................ 3-8
Creating an Implementation .................................................... 3-10
Understanding the Constraints Editor ....................................... 3-14
Importing and Exporting Constraints ....................................... 3-15
Specifying Constraints .......................................................... 3-15
  Specifying Clock Constraints ................................................. 3-16
  Specifying I/O Constraints .................................................... 3-17
  Creating Subpaths ............................................................... 3-17
Defining Multicycle Timing Constraints .................................... 3-18
  Defining False Paths .......................................................... 3-22
Exporting a Netlist .............................................................. 3-23
Generating a Report ............................................................. 3-25
Using Quartus II ..................................................................... 3-27

4. Analyzing Design Timing Results

Checking the Results of Optimization ......................................... 4-2
Viewing Schematics ............................................................... 4-5
  Viewing a Schematic of an RTL Design. ................................. 4-6
  Viewing a Schematic of an Optimized (Mapped) Design. .......... 4-7
Using the TimeTracker Timing Analysis .................................... 4-8

Contents

iv
5. Using the FPGA Compiler II Shell

- Creating FPGA and Design Compiler Scripts .................. 5-2
- Running Command Scripts From the Shell .................. 5-3
- Understanding FST Commands .......................... 5-4
  - FST Command Requirements .......................... 5-6
- Project Variables .................................. 5-7
- FST Commands ........................................ 5-11
  - Constraint Commands ................................ 5-11
  - Reports Commands .................................. 5-12
  - Browsing Objects Commands ......................... 5-13
  - Timing Commands .................................. 5-13
  - Source Design Management Commands ............. 5-14
  - Target Management Commands ...................... 5-14
  - Logical Library Management Commands .......... 5-14
  - Chip Management Commands ........................ 5-14
  - Source File Management Commands ............... 5-15
  - Project Management Commands .................... 5-15
  - Built-In Tcl Commands ............................ 5-15
  - Other Commands .................................. 5-18

6. Using Block-Level Incremental Synthesis (BLIS)

- Identifying Blocks ........................................ 6-2
- Defining Block Roots Using the GUI ........................ 6-3
- Defining Block Roots From the Shell ..................... 6-5
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Updating an Implementation Using BLIS</td>
<td>6-6</td>
</tr>
<tr>
<td>Choosing Block Definitions</td>
<td>6-8</td>
</tr>
<tr>
<td>Planning a Design for BLIS</td>
<td>6-9</td>
</tr>
<tr>
<td>Using BLIS With Altera Quartus II</td>
<td>6-9</td>
</tr>
<tr>
<td>7. Using DesignWare Components</td>
<td></td>
</tr>
<tr>
<td>Installing DesignWare Foundation</td>
<td>7-2</td>
</tr>
<tr>
<td>Instantiating DesignWare Foundation Components</td>
<td>7-4</td>
</tr>
<tr>
<td>Synthesizing DesignWare Foundation</td>
<td>7-5</td>
</tr>
<tr>
<td>8. Implementing Memory Elements</td>
<td></td>
</tr>
<tr>
<td>Using Memory Elements With Altera Devices</td>
<td>8-2</td>
</tr>
<tr>
<td>Using Memory Elements With Lucent Devices</td>
<td>8-5</td>
</tr>
<tr>
<td>Synthesizing Designs with RAM Elements</td>
<td>8-5</td>
</tr>
<tr>
<td>Synthesizing Designs With ROM Elements</td>
<td>8-8</td>
</tr>
<tr>
<td>Using Memory Devices With Xilinx Devices</td>
<td>8-10</td>
</tr>
<tr>
<td>Instantiating a LogiBLOX Element in a VHDL Design</td>
<td>8-10</td>
</tr>
<tr>
<td>Instantiating a LogiBLOX Element in a Verilog Design</td>
<td>8-11</td>
</tr>
<tr>
<td>Appendix A. Using ASCII Constraint Files</td>
<td></td>
</tr>
<tr>
<td>About the ASCII Constraint File Format</td>
<td>A-3</td>
</tr>
<tr>
<td>Header Information</td>
<td>A-4</td>
</tr>
<tr>
<td>Clock Constraints</td>
<td>A-4</td>
</tr>
<tr>
<td>Object Groups</td>
<td>A-5</td>
</tr>
<tr>
<td>Path Constraints</td>
<td>A-6</td>
</tr>
</tbody>
</table>
Port Constraints ................................................................. A-7
Module Constraints ......................................................... A-8
Register Constraints ......................................................... A-10
Vendor Options ................................................................. A-10
Exporting an ASCII Constraint File ................................. A-11
Importing an ASCII Constraint File ................................. A-12

Appendix B. Migrating From ASICs to FPGAs

HDL Source Code Compatibility ........................................ B-3
Design Compiler Shell Script Translation ............................. B-4
Synopsys Database Support ............................................... B-5
Register Retiming ............................................................. B-5
Integration With Place and Route Tools ............................... B-8

Appendix C. Mixing HDL and Netlist Inputs

Glossary

Index
# Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1-1</td>
<td>FPGA Compiler II Design Flow Overview</td>
<td>1-2</td>
</tr>
<tr>
<td>Figure 1-2</td>
<td>FPGA Compiler II in Your Design Environment</td>
<td>1-6</td>
</tr>
<tr>
<td>Figure 2-1</td>
<td>DesignWizard Project Dialog Box</td>
<td>2-2</td>
</tr>
<tr>
<td>Figure 2-2</td>
<td>Adding Source Files</td>
<td>2-3</td>
</tr>
<tr>
<td>Figure 2-3</td>
<td>Selecting Target Devices</td>
<td>2-4</td>
</tr>
<tr>
<td>Figure 2-4</td>
<td>Optimization Common Settings</td>
<td>2-5</td>
</tr>
<tr>
<td>Figure 2-5</td>
<td>Changing Output Options</td>
<td>2-6</td>
</tr>
<tr>
<td>Figure 2-6</td>
<td>Finishing the Project</td>
<td>2-7</td>
</tr>
<tr>
<td>Figure 3-1</td>
<td>The Project Window After You Add Design Source Files</td>
<td>3-7</td>
</tr>
<tr>
<td>Figure 3-2</td>
<td>HDL Editor Window</td>
<td>3-9</td>
</tr>
<tr>
<td>Figure 3-3</td>
<td>Drop-Down List of Top-Level Designs in the Toolbar</td>
<td>3-10</td>
</tr>
<tr>
<td>Figure 3-4</td>
<td>Create Implementation Dialog Box</td>
<td>3-11</td>
</tr>
<tr>
<td>Figure 3-5</td>
<td>Creating the Design Implementation</td>
<td>3-13</td>
</tr>
<tr>
<td>Figure 3-6</td>
<td>Constraint Tables</td>
<td>3-16</td>
</tr>
<tr>
<td>Figure 3-7</td>
<td>Shift Register With Multicycle Timing Constraint Assignment</td>
<td>3-19</td>
</tr>
</tbody>
</table>
Preface

This preface includes the following sections:

• What's New in This Release
• About This Guide
• Customer Support
What’s New in This Release

This section describes the new features and enhancements in FPGA Compiler II version 3.7.1. Unless otherwise noted, you can find more information about these changes later in this book.

New Features

This section presents new features for versions 3.7.1.

HDL Compiler (Presto Verilog)

In FPGA Compiler II version 3.7.1 the default compiler for Verilog HDL designs is HDL Compiler (Presto Verilog). In this release, the following Verilog 2000 features are supported:

- Casting operators
- Parameter passing by name
- Implicit event expression list; for example, @ *
- ANSI-C-style port declaration; for example, module t (input [1:0] in, output [2:0] out);
- Signed or unsigned parameters, nets, registers, and constants
- Multidimensional arrays and array of nets
- Variable part-select and part-select addressing operators ( [+:] and [-:] )
- Power operator (**)
- Arithmetic shift operators (<<< and >>>)
For details about supported Verilog 2000 constructs, see Appendix C, “Differences Between the New HDL Compiler and the Original HDL Compiler,” in the *HDL Compiler (Presto Verilog) Reference Manual*.

**Support for search_path Variable in Shell Mode**

Support added for the `search_path` variable in fc2_shell mode. This variable improves the handling of Verilog include files and source file management.

For example, to define the search path from the shell, enter

```
search_path="/u/rch/designs /u/abc/tests ../../temp/tests"
```

When you use this variable, it is not necessary to include the full path for the source file, as in this example:

```
add_file -format Verilog /u/rch/designs/test1.v
```

Instead, you can use the following command:

```
add_file -format Verilog test1.v
```

More than one path can be specified with spaces in the search path.

The Verilog `include` directive can be one of the following three types:

- **Absolute path:** `include /u/abc/tests/test2.v`
- **Relative path:** `include ../../tests/test2.v`
- **The file name only:** `include test2.v`

In the last case, it makes use of the search path.
Enhancements

FPGA version 3.7.1 includes the following enhancements.

• The *FPGA Compiler II User Guide* has been improved. The chapters have been rearranged to fit the flow of using FPGA Compiler II. You can find updated information about the following topics:

  - Advanced Synthesis Flow
  - Using Scripts
  - Using Block-Level Incremental Synthesis (BLIS)
  - DesignWare Components
  - Using Memory Elements

Known Limitations and Workarounds

Information about known problems, limitations, and workarounds is in the *FPGA Compiler II Release Note*, which is available from the Help menu in the GUI.

To view the Release Note,

• Choose Help > Release Note.

The Release Notes are also available in SolvNet at the Synopsys website. For more information, see “Accessing SolvNet” on page xvii.
About This Guide

This guide provides information for users of FPGA Compiler II. The guide explains how to apply the program’s basic features to your chosen design flow and programmable logic device architecture.

Audience

This guide is for logic designers or engineers who use FPGA Compiler II to implement FPGA designs.

Related Publications

For additional information, see

- The *Installation Guide* (delivered with the program in portable document format (PDF) files)
- The *VHDL Reference Manual* (online) (delivered with the program in portable document format (PDF) files)
- The *Verilog HDL Reference Manual* (online) (delivered with the program in portable document format (PDF) files)
- Synopsys Online Documentation (SOLD), which is included with the software for CD users or is available through the Synopsys Electronic Software Transfer (EST) System
- Documentation on the Web, which is available through SolvNet at http://solvnet.synopsys.com
- The Synopsys MediaDocs Shop, from which you can order printed copies of Synopsys documents, at http://mediadocs.synopsys.com
• The program’s Help menu, which provides access to a selection of FPGA-vendor documents from FPGA vendors

• Vendor-provided documents: A selection of documents provided by FPGA vendors is available from the Help menu (You might find these documents helpful in answering your vendor-specific questions.)
# Conventions

The following conventions are used in Synopsys documentation.

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Courier</td>
<td>Indicates command syntax.</td>
</tr>
<tr>
<td><em>Courier italic</em></td>
<td>Indicates a user-defined value in Synopsys syntax, such as <code>object_name</code>.</td>
</tr>
<tr>
<td></td>
<td>(A user-defined value that is not Synopsys syntax, such as a user-defined</td>
</tr>
<tr>
<td></td>
<td>value in a Verilog or VHDL statement, is indicated by regular text font</td>
</tr>
<tr>
<td></td>
<td>italic.)</td>
</tr>
<tr>
<td><em>Courier bold</em></td>
<td>Indicates user input—text you type verbatim—in Synopsys syntax and examples.</td>
</tr>
<tr>
<td></td>
<td>(User input that is not Synopsys syntax, such as a user name or password</td>
</tr>
<tr>
<td></td>
<td>you enter in a GUI, is indicated by regular text font bold.)</td>
</tr>
<tr>
<td>[]</td>
<td>Denotes optional parameters, such as <code>pin1 [pin2 ... pinN]</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>_</td>
<td>Connects terms that are read as a single term by the system, such as <code>set_annotated_delay</code>.</td>
</tr>
<tr>
<td>Control-c</td>
<td>Indicates a keyboard combination, such as holding down the Control key and</td>
</tr>
<tr>
<td></td>
<td>pressing c.</td>
</tr>
<tr>
<td>\</td>
<td>Indicates a continuation of a command line.</td>
</tr>
<tr>
<td>/</td>
<td>Indicates levels of directory structure.</td>
</tr>
<tr>
<td>Edit &gt; Copy</td>
<td>Indicates a path to a menu command, such as opening the Edit menu and</td>
</tr>
<tr>
<td></td>
<td>choosing Copy.</td>
</tr>
</tbody>
</table>
Customer Support

You can access customer support through SolvNet online customer support or through the Synopsys Technical Support Center.

Accessing SolvNet

SolvNet includes an electronic knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. Also, SolvNet gives you access to a wide range of Synopsys online services including software downloads, documentation on the Web, and “Enter a Call With the Support Center.”

To access SolvNet,


2. If prompted, enter your user name and password. (If you do not have a Synopsys user name and password, click New Synopsys User Registration.)

If you need help using SolvNet, click SolvNet Help in the column on the left side of the SolvNet Web page.
Contacting the Synopsys Technical Support Center

If you have problems, questions, or suggestions, you can contact the Synopsys Technical Support Center in the following ways.

• Open a call to your local support center from the Web by going to http://solvnet.synopsys.com (you need a Synopsys user name and password). Then click “Enter a Call With the Support Center.”

• Send an e-mail message to support_center@synopsys.com

• Telephone your local support center.
  - From the continental United States, call (800) 245-8005.
  - From Canada, call (650) 584-4200.
  - Find other local support center telephone numbers at http://www.synopsys.com/support/support_ctr.

For more information, see the FPGA Web page at http://www.synopsys.com/products/fpga.
About FPGA Compiler II

FPGA Compiler II brings a combination of synthesis and optimization technology, high-level design methodology, and easy-to-use interfaces to the design desktop.

This chapter contains the following sections:

• Features and Benefits
• Methodology
FPGA Compiler II is for FPGA architectures. With this tool, you can create optimized netlists from VHDL code, Verilog HDL code, or from EDIF netlists.

Figure 1-1 shows how FPGA Compiler II fits into your design flow.

**Figure 1-1  FPGA Compiler II Design Flow Overview**
Features and Benefits

FPGA Compiler II has features specifically for creating high-density FPGAs:

- Architecture-specific mapping and optimization for the most popular programmable logic vendors
- Integrated static timing analysis (using TimeTracker)
- Schematic viewing (with links to TimeTracker)
- Tcl-based language for scripting
- Support for encrypted intellectual property (IP)
- Block-level incremental synthesis (BLIS) for Altera and Xilinx architectures
- Register retiming and pipelining (to balance delays between input to register, register to register, and register to output)
- Design Compiler shell (dc_shell) scripting support (for migration between ASICs and FPGAs)
- Support of instantiated DesignWare Foundation components
- Export of Synopsys database format (.db) files (for integration of FPGA Compiler II projects into a design flow that includes other Synopsys tools)
If you are migrating from a schematic-based methodology to an HDL-based methodology, FPGA Compiler II adds HDL logic synthesis and optimization to your Design environment. You can define a design completely with HDL source code or use a mixture of schematics and HDL source code to enter a design into FPGA Compiler II.

Using an HDL-based design methodology increases productivity, because HDL source code is vendor independent, reusable, and you can retarget it toward various technologies. FPGA Compiler II optimization algorithms are specifically tuned for multiple vendor devices.

The FPGA Compiler II GUI lets you use standard commands and procedures to input values. It does not require command scripts. Also, it allows multiple synthesis design flows, which lets you use a variety of design styles and performance goals.
Methodology

The following general methodology explains how FPGA Compiler II fits into your design flow.

You can use any of the following design creation methods:

- For an HDL-based design methodology, you write the HDL source code for the design.
- For a schematic-based methodology, you capture the design schematics and export them to an HDL format, netlist format, or a combination of these.
- For a mixed (HDL and schematic) design methodology, you write the HDL source code for the parts of the design you want to describe in HDL. You capture the schematics for the rest of the design.

If you want to verify the design functionality, you use an HDL simulator for HDL code and use a gate-level simulator for schematics.

*Figure 1-2* shows how FPGA Compiler II relates to other tools when you use these design methods.
Figure 1-2  FPGA Compiler II in Your Design Environment

Figure 1-2 shows the flow of FPGA Compiler II with other tools in these design environments. The steps in this procedure are similar for each environment; only the method of entering the design into FPGA Compiler II differs. Note that simulation is optional.
You use the following general procedure when using FPGA Compiler II:

1. Create design source files.

2. Launch FPGA Compiler II.

3. Set up the design project and analyze the design source files. You can setup your project files by using the DesignWizard, see “Understanding the DesignWizard Flow” on page 2-2, or use the procedure described in Chapter 3, “Advanced Synthesis Flow”.

4. Elaborate the design.

In this step, you identify the top-level design, choose the target device, set the clock frequency, enter any constraints, and create the implementation.

The tool elaborates each subdesign module as it creates and links the design hierarchy. The resulting implementation is a generic gate-level representation of your design.

After elaboration, the tool generates a schematic that represents the hierarchy of your design.

5. Optimize the design implementation.

During optimization, the tool uses architecture-specific algorithms to analyze the timing of your design, compare timing to your requirements, and make changes to meet your design goals.

Also, the tool can generate reports and an optimized schematic.

6. Analyze timing information to verify circuit performance.
The tool displays timing information in tables alongside your design constraints and highlights timing violations. The tool links timing information directly to the schematic for debugging. See Chapter 4, “Analyzing Design Timing Results.”

7. Export an optimized netlist for place and route.

The tool can generate an optimized netlist ready for place and route using vendor tools.

8. Generate VHDL or Verilog netlists for functional simulation.

By using a standard HDL simulator, you can verify that the optimized netlist is functionally identical to the RTL input.

9. Export the .db file so that you can use the design with other Synopsys tools.

To finish the design process, complete the following steps outside of FPGA Compiler II:

1. Place and route the design by using the FPGA vendor’s development system.

2. (Optional) Simulate the design with post-place and route timing delays.

3. Program the device.
This chapter introduces you to the DesignWizard. The DesignWizard allows you to create a project, synthesize your design, and to obtain an optimized netlist in as few as two steps. This chapter includes the following sections:

- Understanding the DesignWizard Flow
- Adding Design Source Files
- Selecting Target Devices
- Specifying Optimization and Output Options
- Project File Location
Understanding the DesignWizard Flow

You can quickly and easily implement your design source files using the DesignWizard from the FPGA Compiler II startup screen, or you can start the DesignWizard from the FPGA Compiler II File menu.

The DesignWizard screen, as shown in Figure 2-1, lets you

• Create a new project or open an existing one (For more information, see “Creating a Project” on page 3-3.)

• Disable the DesignWizard for subsequent starts of FPGA Compiler II

Figure 2-1  DesignWizard Project Dialog Box
Adding Design Source Files

In the DesignWizard Step 1 window, you can add the design source files by selecting the Add button. When you add design source files, FPGA Compiler II extracts the module or entity names from these source files. When you are finished adding your source files you must select the top level module/entity. See Figure 2-2.

Figure 2-2 Adding Source Files

To go to Step 2 of the DesignWizard, click on the Next button.
Selecting Target Devices

In Step 2 of the DesignWizard, you select the FPGA vendor, family, device, and speed grade that you want the design to target to, see Figure 2-3.

**Figure 2-3  Selecting Target Devices**

Before you finish with Step 2, you can specify Optimization and Output Options. For details, see “Specifying Optimization and Output Options” on page 2-5.
Specifying Optimization and Output Options

From the DesignWizard Step 2 of 2 window, select the Optimization button to change the clock frequency, effort levels and other constraints. The most common settings are shown in Figure 2-4.

Figure 2-4 Optimization Common Settings

![Optimization Options dialog box](image)

When you set the effort to Fast, it reduces your runtime. It is recommended that you use the Fast mode for most runs. For maximum clock speed, set the optimization to Speed and the Effort level to High, and do not preserve hierarchy.

To specify project settings, select the Output Options dialog box (see Figure 2-3). You can change the project name and the location where you want the project files to be stored. It is recommended to keep all files related to the project in one folder. You can specify the netlist settings for your place and route tool and generate a .db file to use with Design Compiler. These files are saved to the Results Files Location that you specified in Project Settings.
When you have finished with Step 2, click the Run button to finish synthesis (see Figure 2-3 on page 2-4).
Once the files are synthesized, you can make changes to your project by adding other source files, editing constraints, or updating an implementation. For more information, see Chapter 3, "Advanced Synthesis Flow".
Project File Location

After synthesis is done, a new directory is created under the directory (folder) of the source files. This directory contains all the information used by FPGA Compiler II to synthesize the design. You will find the EDIF netlist and constraint files exported by FPGA Compiler II in this directory.

The project directory name is the top level entity/module name appended with _fc2_proj. For example, micro_fc2_proj.
Advanced Synthesis Flow

This chapter describes how to control the synthesis process by setting constraints, attributes, and control options. It includes the following sections:

• Creating a Project
• Adding Design Source Files
• Debugging Design Source Files
• Understanding the Constraints Editor
• Importing and Exporting Constraints
- Specifying Constraints
- Defining Multicycle Timing Constraints
- Defining False Paths
- Exporting a Netlist
- Generating a Report
- Using Quartus II
Creating a Project

A project is a directory (folder) created by FPGA Compiler II. This working directory holds a project file with a .exp extension and subdirectories created by the tool for internal and intermediate use. The key components of a project are the design source files that describe the FPGA design or designs. The type of design source files that the tool supports are VHDL, Verilog, and EDIF netlists.

To create a project in FPGA,

1. Launch FPGA Compiler II.
   To do so, choose Start > Programs > Synopsys > FPGA Compiler II version > FPGA Compiler II.

   The FPGA tool opens the main window, which includes the toolbar, the tip bar, the Design Sources window, the Chips window, and the Output window.

   (If the DesignWizard window appears, click Cancel. The DesignWizard is another way to create a project. For information about the DesignWizard, see Chapter 2, “DesignWizard Synthesis Flow.”)

2. Create a new project.
   To do so, do one of the following:
   - Click the New Project button in the toolbar.
   - Choose New Project from the File menu.

   In the main window, FPGA displays the Create New Project dialog box.
The dialog box includes a default location and name for the project. You can change the location, name, or both. If you want to do this, use the drop-down list in the Save In field to navigate through your directory tree or click the Create New Folder icon to create a new directory.

3. In the Name field, type a name for your project.
   
   FPGA stores the project files in a directory that has the same name as your project.

4. Click the Create button.
   
   After the tool creates the project, the Add Sources dialog box opens.
Setting Project Synthesis Options

You can configure the appearance and behavior of FPGA Compiler II by choosing Synthesis > Options and clicking the General tab.

The default HDL compiler in FPGA Compiler II is the Presto Verilog compiler. The benefits of using Presto Verilog is the ability to implement fastest and smallest, and safest FSM’s in the same way it does for VHDL designs. For details, see the HDL Compiler (Presto Verilog) Reference Manual.

To use the original HDL compiler, choose the Synthesis/Options menu and click on the Project tab. In the Verilog Synthesis section, choose “Compatible HDL Compiler - HDLC.” The options that you select apply to all subsequent projects.

Adding Design Source Files

After you create the project, a default library named WORK appears in the project folder. You add the design source files to the WORK library. You can add any combination of VHDL, Verilog, EDIF netlist files. When you add files, FPGA Compiler II analyzes them automatically to detect errors. If the design source files contain errors, the Output window helps you find and correct problems. You can use the integrated text editor (the HDL Editor) to debug the files. For more information, see “Debugging Design Source Files” on page 3-8. For explanations of error messages, see the online Help.

VHDL environments sometimes require multiple libraries. You can add a library at any time by right-clicking in the Design Sources window and choosing New Library. Then, you add design source files to that library.
Note:
FPGA does not copy design source files to another directory. When you add files, the tool analyzes them in their current location. If you changed the files (a red question mark ? icon appears next to the file name that indicates a change), you must reanalyze them by clicking the Update button on the toolbar or right-clicking in the Design Sources window and Update File.

To add design source files,

1. Open the Add Sources dialog box.

To do this, do one of the following:

- Click on the toolbar.
- Choose Synthesis > Add Source Files.
- Right-click WORK in the Design Sources window and choose Add Sources in WORK.

The Add Sources dialog box appears.

2. In the Add Sources dialog box, click or shift-click to highlight the design source file or files.

3. Click Open.

Note:
On a Windows PC, you can also add files to the project by dragging them into the Design Sources window.

The FPGA tool displays the project window and extends the menu bar. The project window has two windows—the Design Sources and Chips window (see Figure 3-1).
The Design Sources window displays the name, location, and status of each of the design source files. The Chips window displays information (such as name and device type) for individual design implementations. The project window’s title bar displays the name of the project.

In the Design Sources window, the icon to the left of each file name indicates the results of the analysis. For example, in the figure, the red cross indicates that the counter file has at least one error. The green check marks next to the other files indicate that the files have no errors or warnings.

The error in the design source file is reflected up the hierarchy. Therefore, the library and the project icons are also marked with red crosses. (For a list of analysis status icons and their explanations, see the online Help.)
Chapter 3: Advanced Synthesis Flow

3-8

Debugging Design Source Files

You use the Errors, Warnings, and Messages tabs of the Output window to find and correct errors and warnings in design source files. The Output window shows the file name, line number, and type of each error or warning. After viewing the messages, you can use the HDL Editor to investigate and fix the errors and warnings.

To view and correct errors and warnings,

1. In the Design Sources window, click to highlight the design and read its errors and warnings in the Output window.

2. Open the HDL Editor and correct any errors.

   You can open the HDL Editor in either of the following ways:

   - Right-click in the Design Sources window and choose Edit File.
   - Double-click an error or warning message in the Output window.

   The FPGA tool opens the HDL Editor so that you can edit the file. You can use the HDL Editor Edit menu to navigate in the file. Also, you can use the pop-up menu that appears when you right-click in the HDL Editor window to navigate.

   You can use an editor other than the HDL Editor as the default design source file editor. To do this, choose Synthesis > Options, click the General tab, and deselect the Use FPGA Compiler II internal source editor check box. If you deselect the check box, choosing Edit File opens the file in the editor associated with the file type.

   Figure 3-2 shows the HDL Editor window and its pop-up menu.
In line 20 of Figure 3-2, the HDL Editor indicates that a misspelling in the if statement caused the error in the design named counter.

3. Save the file.

When a file is out of date, the file icon contains a question mark.

4. Update the file.

To update the file, do one of the following:

- Go to the Design Sources window and click on the toolbar.
- Right-click in the HDL Editor window and choose Analyze File (before you close the HDL Editor window).
Depending on the extent of your changes, FPGA updates the file, the library, or the project. FPGA reanalyzes only the files that you change.

After adding, analyzing, and debugging source files, you can elaborate the design.

Creating an Implementation

To start building an implementation from analyzed source files, you identify the top-level entity (for VHDL), module (for Verilog), or schematic netlist as the top-level design. FPGA Compiler II uses the top-level design that you identify to build design hierarchy and interconnections.

Identify the top-level design in any of the following ways:

- From the drop-down list of top-level designs (entities, modules, and netlists) in the toolbar, select the top-level design that you want to use (see Figure 3-3).

Figure 3-3  Drop-Down List of Top-Level Designs in the Toolbar

- In the Design Sources window, double-click the design source file that you want to use. The file expands and displays an icon for the file. Then, right-click the icon and choose Create Implementation.
• Click to highlight the icon of the top-level design and click the Create Implementation button on the toolbar.

After you identify the top-level design (in any of the three ways above), the Create Implementation dialog box appears (see Figure 3-4).

Figure 3-4 Create Implementation Dialog Box

When you create an implementation, the tool elaborates logic for each source file. It determines the complete hierarchical structure and topology of the design (including multiple-level links and references between subdesigns). The tool uses this information to produce an intermediate, unoptimized implementation.

To create the implementation,

1. In the Implementation Name field, enter an implementation name.

   If you do not enter a name, FPGA Compiler II uses the name of the top-level design to create a unique implementation name.
2. Select the target device and speed grade for the design.

3. Set the default clock frequency.

   All clocks in the design use this default frequency. (After you elaborate the design, you can use the design constraint tables to change default clock frequencies later.)

   Overconstraining a design can degrade the place and route results. Therefore, you should specify only what you need.

4. Choose the optimization goal and optimization effort.

   You can optimize for speed or area and high, fast, or low CPU effort.

5. Select or deselect the Do Not Insert I/O Pads option.

   This option determines whether the tool inserts I/O pads during optimization. Use this option for implementations that are modules in larger designs.

6. Select or deselect the Skip Constraint Entry option.

   This option enables or disables constraint specification for the design.

7. Select or deselect the Preserve Hierarchy option.

   This option controls whether the tool preserves or eliminates the module’s boundary during optimization. For more information, see the online Help.

8. Click OK.
The design is elaborated. An implementation icon, implementation name, target device, and speed grade appear in the Chips window. The icon indicates the implementation status (see Figure 3-5).

**Figure 3-5 Creating the Design Implementation**

9. View the Output window to investigate any errors and warnings.

After performing an implementation you can edit constraints for clocks, paths, ports, and modules. For more information, see “Understanding the Constraints Editor” on page 3-14.”
Understanding the Constraints Editor

Before you start to optimize a design for a target device, you can set performance constraints. Design constraints guide FPGA Compiler II with specific optimization requirements. For determining timing violations, see Chapter 4, “Analyzing Design Timing Results.”

Using the graphical user interface, you can enter constraints for your design in editable tables. See Figure 3-6 on page 3-16.

FPGA Compiler II separates constraint entries into logically related groups (for example, clocks, paths, ports, and modules). The tool automatically extracts design-specific information, such as clock names, port names, and design hierarchy, from the design and displays it in tables. You enter performance constraints, attributes, and optimization options directly into the tables.

Each set of constraint tables and dialog boxes is specific to a particular target architecture. Controls for some target technologies are available when you select the Vendor Options tab in the Constraint Table dialog box.

Specifying constraints is optional, but highly recommended. Entering your requirements in the constraint tables can significantly improve place and route results.

For example, if you enter constraints for an output port with restrictive speed requirements, it is easier for the place and route tool to fulfill those requirements. If a design is very large and has many hierarchical levels, you can improve place and route results by entering hierarchy constraints. Note, however, that overconstraining
a design can adversely affect place and route results. Therefore, constrain only critical parts of the design and add constraint values that are realistic.

---

**Importing and Exporting Constraints**

You can import or export an ASCII file of constraint commands for the current chip by using menu commands or the scripting tool. For more information, see Appendix A, “Using ASCII Constraint Files.”

---

**Specifying Constraints**

To enter design constraints,

1. Right-click the elaborated implementation and choose Edit Constraints to open the Constraints window.

   Figure 3-6 shows how the constraints and synthesis controls are logically grouped by function into separate Clocks, Paths, Ports, and Modules tabs. An additional tab, called Registers, is available for selected FPGA technologies.
2. Click the tabs to display tables for the constraint groups.

3. Explore the tables for the implementation.

   The contents of the tables reflect the architecture you chose earlier. Note that the Clocks and Paths constraint tables are preloaded with the clock frequency (and corresponding period) that you entered for the target clock frequency.

---

**Specifying Clock Constraints**

To specify clock constraints, do the following:

1. Right-click on the preoptimized chip in the Chips window and select Edit Constraints.

2. Click on the Clock column cell and expand the cell.

3. Select Define. In the Define Clock box that appears, enter the period, rise time, and fall time of the clock signal.
Specifying I/O Constraints

You can enter input delay, output delay, and pad locations in the Port column of the Edit Constraints window.

To define the I/O constraints:

1. Click the Input Delay cell for a port and select Define. In the Define dialog box enter the input delay.
2. Click the Output Delay cell for a port and select Define. In the Define dialog box enter the output delay.
3. Specify the pad location for a port. You cannot specify pad locations for a design that has the Do Not Insert I/O Pads option selected.

Creating Subpaths

To create subpaths for a path:

1. Right-click on the path in the Path table. Select one of the following: New Subpath, Edit Subpath, or Delete Subpath.

   The Create/Edit Timing Sub Path dialog box appears, displaying the primary path and the components in that path. See Figure 3-9 on page 3-21.

2. Select New Subpath.
3. Enter a name for the subpath (Subpath name).
4. Select startpoints and endpoints for each subpath group by double-clicking the object icons.

   The names of subpath groups must be unique.
5. Specify the delay for the subpath. You can specify different constraints for each subpath.

You can use the Select All button to make multiple startpoint and endpoint selections. You can use the Clear all button to clear all startpoint and endpoint selections. You can enter common expressions such as DI* to make multiple selections.

To define false paths, see "Defining False Paths" on page 3-22.

6. When you click OK, the path constraint table is updated to reflect the new subpath groups.

---

**Defining Multicycle Timing Constraints**

You can use FPGA Compiler II to generate timing groups and path groups for logic that uses clock enable signals. This feature is useful when parts of a design run at a slower speed than the rest (with the enable signal controlling the slower flip-flops).

Using enable signals with a fundamental clock eliminates clock skew, which additional clock signals can introduce. You can set multicycle timing constraints for specified paths, which makes the slower logic easier to place and route. You can apply multicycle timing constraints to subpath groups that you create.

The following example shows how the 10-MHz enable signal might align with the system clock when the rising edge of the 40-MHz system clock is the active edge. The 40-MHz clock is distributed to the clock input of each flip-flop, and the enable signal is distributed to each flip-flop clock enable input. The primary clock period is 25 ns, but the 10-MHz enabled logic must satisfy a period of only 100 ns:
The simple shift register circuit in the logic diagram in Figure 3-7 shows how the multicycle timing constraints are assigned.

Figure 3-7 is an example in which a multicycle timing constraint is appropriate. The FPGA contains high-speed interface logic that must run at 40 MHz and low-speed interface and core logic that must run at 10 MHz. The FPGA has a 40-MHz system clock and uses the clock to generate a 10-MHz enable signal for internal distribution.

Register reg1 is a 4-bit, serial-input, parallel-output register. Register reg2 is a holding register that is loaded with the clock enable signal ena. The paths from the output of reg1 to the input of reg2 (net q) are multicycle paths, because the data bits have four clock cycles to reach their destinations. The register-to-register timing constraint is 25 ns (1/40 MHz), but the multicycle timing constraint is 100 ns (4 x 25 ns).
To create a subpath group of the register-to-register paths,

1. Open the Paths constraint table.

2. Right-click the register-to-register path groups.

3. Choose New Sub path.

   The Create/Edit Timing Sub Path window appears.

4. Use the Create/Edit Timing Sub Path window to construct your own path group by choosing specific startpoints and endpoints.

   Figure 3-8 shows the Create/Edit Timing Sub Path window.

   **Figure 3-8  New Sub Path Dialog Box**

   ![New Sub Path Dialog Box](image)

   The new path group is called a subpath, because it is a subset of another path group; in this case, the register-to-register paths in the design.

   In the example in Figure 3-9, the outputs of reg1 are the startpoints, and the inputs of reg2 are the endpoints for the subpath. A delay of 100 ns is assigned to the subpath.
For more information about using the Create/Edit Timing Sub Path window, see the online Help.

After you create a subpath and apply the multicycle timing constraint, the subpath appears in the Paths constraint table, as shown in Figure 3-10.
You can include an enabled flip-flop in two path groups—those that include clock-to-clock paths and those that include clock-to-enabled-clock paths. This implies that there are two overlapping constraints. The constraint for clock-to-clock timing (25 ns in this case) conflicts with the constraint for clock-to-enabled-clock timing, which is 100 ns.

Most vendor place and route tools assign different priorities to these two constraints by placing a higher priority on the more specific one. Because the subpath constraint is more specific than the clock-to-clock constraint, it takes precedence, and the corresponding paths can be optimized for the slower speed.

**Defining False Paths**

You can use FPGA Compiler II to specify false paths in your design. False paths are members of larger path groups that do not need to be constrained. Specifying false paths reduces the number of paths
that must meet performance constraints. Therefore, the synthesis and place and route tools are more likely to produce satisfactory results.

To define false paths for a design, use the procedure described in “Creating Subpaths” on page 3-17. In step 5, specify a value of 999 for the delay of the false paths. Setting very large delay values for the subpaths effectively defines them as false paths to the synthesis and place and route tools.

---

**Exporting a Netlist**

FPGA Compiler II generates EDIF netlists that place and route tools from FPGA vendors can process directly. The tool can also generate VHDL and Verilog netlists for simulation.

To generate netlist files,

1. Either select the optimized implementation and click on the toolbar, or right-click the implementation and choose Export Netlist. The Export Netlist dialog box appears (see Figure 3-11).
2. Choose an export directory for the netlist files.

3. To change directories, either type the new directory name or click Browse.

4. Specify whether to export timing constraints with the netlist, using the Export Timing Specifications check box.

5. Select an output format for your netlist:
   - To export only the netlist for place and route, select NONE in the Output Format list box.
   - To also export a netlist for simulation, select Verilog or VHDL.

6. Specify whether to export the Synopsys design database with the netlist, using the Generate Synopsys db Files check box.
   The .db format enables integration with other Synopsys tools.
7. Click OK.

**Caution!**
Many files might be exported from a single design. To avoid overwriting your source files, always export netlists into a separate directory.

---

**Generating a Report**

You can generate the following reports:

- **Library report**
  A library report shows errors, warnings, and messages for each design source file in the library.

- **File report**
  A file report shows errors, warnings, and messages for the selected design source file.

- **Chip report**
  A chip report shows implementation settings such as target device, synthesis goal, optimization effort, clock frequency, and other timing information.

- **Project report**
  A project report contains all the information in the library, file, and chip reports.
To generate a report,

1. In the project window, choose the project, library, file, or chip and click on the toolbar.

   Or, right-click the project, library, file, or chip and choose Report from the pop-up menu.

2. In the dialog box that appears (see Figure 3-12), specify the name and location for the report.

   Figure 3-12  Generate Project Report Dialog Box

3. Click Save. A text file containing summary information for the whole project, the library, the design file, or the chip is created.

4. Open the report file in a text editor or word processor.
You can launch Quartus II from within the GUI. All the files that Quartus II requires are automatically generated in the directory specified in the Place and Route dialog box in Figure 3-13.

To launch Quartus II,

1. Either select the optimized implementation and click on the toolbar or right-click the implementation and choose Place and Route Chip. The Place and Route dialog box appears.

2. Choose a place and route directory.

3. To change directories, either type the new directory name or click Browse.

4. Specify whether to export timing specifications with the netlist, using the Timing Specifications check box.
This chapter describes how to analyze the synthesis and optimization results of your design.

This chapter includes the following sections:

- Checking the Results of Optimization
- Viewing Schematics
- Using the TimeTracker Timing Analysis
Checking the Results of Optimization

After you create an elaborated implementation, you can determine circuit performance by checking the results of optimization and analyzing timing information. The post-synthesis timing data is displayed in the same formats you used to enter constraints.

To view the results of optimization,

1. To see a complete list of paths, right-click an optimized implementation and choose View Results.

2. Check the Clocks constraint table to see the maximum clock frequencies that were estimated for each of the clocks in the design. Clock frequency violations appear in red. Figure 4-1 shows the Clocks constraint table after optimization.

Figure 4-1 Optimization Results in the Clocks Constraint Table

3. Check the Paths constraint table for more information about timing violations:
   - Select a path group to see a list of paths in that group.
   - Select a path from the list to see the details of path composition, cumulative delays, and fanout.

Figure 4-2 shows the Paths constraint table after optimization.
Note:
All pins of primitive cells on the timing path are displayed. Therefore two rows on the right window correspond to a single cell (source and load).

**Figure 4-2 Optimization Results in the Paths Constraint Table**

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Reg Delay</th>
<th>Est. Delay</th>
<th>Slack</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Input Ports</td>
<td>RC - N_CLK</td>
<td>40</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>1 NOTCLR</td>
<td>GOUT_reg7</td>
<td>10.5</td>
<td>20.2</td>
<td></td>
</tr>
<tr>
<td>2 NOTCLR</td>
<td>GOUT_reg8</td>
<td>10.3</td>
<td>29.7</td>
<td></td>
</tr>
<tr>
<td>3 NOTCLR</td>
<td>GOUT_reg9</td>
<td>9.2</td>
<td>30.8</td>
<td></td>
</tr>
<tr>
<td>4 NOTCLR</td>
<td>GOUT_reg10</td>
<td>8.7</td>
<td>31.3</td>
<td></td>
</tr>
</tbody>
</table>

4. Check the Ports constraint table for information about input and output delays.

**Figure 4-3** shows the Ports constraint table. The results include the slack for input arrival time and output delay for each port.

**Figure 4-3 Optimization Results in the Ports Constraint Table**

Checking the Results of Optimization
5. Check the Modules constraint table for information about the device resources used. For details about cell count, click in any cell of the Area column and select the drop-down arrow.

Figure 4-4 shows the Modules constraint table after optimization.

*Figure 4-4  Optimization Results in the Modules Constraint Table*
Viewing Schematics

You can view and analyze your design graphically by using the integrated schematic viewer. You can view an RTL version of the design, as shown in Figure 4-5, or an optimized (mapped) version of the design, as shown in Figure 4-6.

Figure 4-5  RTL Version of a Design in the Schematic Viewer
Figure 4-6  Optimized (Mapped) Version of a Design in the Schematic Viewer

Viewing a Schematic of an RTL Design

You can use the schematic viewer to examine a variety of information about an RTL (generic) design. This information includes design hierarchy and the contents of a block.

To view the schematic of an RTL design,

1. In the Chips window, right-click the elaborated implementation.
2. From the pop-up menu, choose View Schematic.
The schematic viewer appears. The viewer windows arrange themselves to increase viewable area.

3. Maximize the main window.

4. Navigate within the schematic in any of the following ways:
   - To zoom the schematic, click the Zoom In, Zoom Out, Zoom In Tool, or Zoom Full-Fit buttons on the toolbar.
   - To zoom into a specific location, click the Zoom In Tool button on the toolbar and drag the pointer over the location.
   - To view the contents of a block, double-click the block.
   - To return to the next level up, right-double-click anywhere in the schematic.
   - To navigate around the hierarchy by using the Chips window, double-click a level of hierarchy to expand it or select one of the blocks to view that level.

---

**Viewing a Schematic of an Optimized (Mapped) Design**

You can use the schematic viewer to examine a variety of information about an optimized (mapped) design. The schematic viewer lets you view the project files, the TimeTracker window (see “Using the TimeTracker Timing Analysis” on page 4-8), and the schematic window at the same time.

To view the schematic of an optimized design,

1. In the Chips window, right-click the optimized implementation.

2. From the pop-up menu, choose View Schematic.
The windows arrange themselves to display the project window on the left and the TimeTracker and schematic windows on the right.

3. You can navigate within the schematic in any of the following ways:

- To zoom the schematic, click the Zoom In or Zoom Out Tool, or Zoom Full-Fit buttons on the toolbar.

- To zoom into a specific location, click the Zoom In Tool button on the toolbar and drag the pointer over the location.

---

Using the TimeTracker Timing Analysis

You can use TimeTracker to examine a variety of timing results for a specific path in a design. These results include critical path, cell type, delay, fanin and fanout, and slack values.

To view timing results for a path in the schematic viewer,

1. Open the TimeTracker by right-clicking on the optimized implementation and select View Results. Click the Paths tab.

   The Paths constraint table appears.

2. In the Paths window (the upper-left window), click to choose the row for the path group.

   The paths constrained by the path group appear in the paths area (the lower-left window). For any paths that fail timing, the timing values appear in red in the TimeTracker window. In the schematic, the paths are highlighted.
3. From the group of paths, select the path for which you want to view timing results.

   The right side of the TimeTracker window displays the cells in this path. The instance name, cell type, delay, and fanout for each cell is displayed.

   In the schematic window, the critical path appears in red.

4. View the information about the path in any of the following ways:

   - To view the cell name, cell type, pin numbers, delay, fanout, and slack values, choose the path and hold the pointer over the cell. A pop-up window displays this information.

   - To find the cell to which a pin is attached, choose the pin from the pin list. The schematic window highlights the cell in yellow.

   - To display the fanin and fanout logic cones, make sure that the cell is selected and highlighted in yellow, then click the Fan In and Fan Out buttons on the toolbar.

   - To move along the path, click the Previous Pin and Next Pin buttons on the toolbar or click each pin in the path in TimeTracker.
Using the FPGA Compiler II Shell

This chapter describes how to use the FPGA Compiler II Shell and the FPGA Scripting Tool (FST).

This chapter includes the following sections:

• Creating FPGA and Design Compiler Scripts
• Running Command Scripts From the Shell
• Understanding FST Commands
Creating FPGA and Design Compiler Scripts

FPGA Compiler II can automatically create shell scripts that you can use from the fc2_shell or dc_shell. To create FPGA Compiler scripts or Design Compiler scripts from the FPGA Compiler II GUI, select Script > Export FPGA Script or Export DC Script. You can use the dc_shell script to create an ASIC version of an FPGA chip in Design Compiler.

To create scripts of your project file from the fc2_shell, do the following:

1. Open the FPGA Compiler II Shell.
   
   In Windows, start the shell by choosing Start > Programs > Synopsys > Shell, or from a UNIX shell or command window by entering the command fc2_shell.

2. From the fc2_shell cd to the directory where the project_name.exp file is located.
   
   Enter the following commands,

   fc_shell> open_project project_name.exp
   fc_shell> current_chip chip_name
   fc_shell> script_chip -fc2

   To generate a Design Compiler script use the -dc option. For example,

   fc_shell> script_chip -dc
The following is an example script for a project and chip named micro:

```bash
create_project micro/micro.exp
add_file rtl/convsegs.vhd
add_file rtl/counter4.vhd

create_chip -progress -target FLEX10K -name micro-fc2 micro

set_clock -period 20 -rise 0 -fall 10 CLOCK
set_max_delay -path my_from:my_to 20
set_module_primitive preserve "'/AM2910/U4

optimize_chip -progress -name Optimized-micro-fc2 micro -fc2
export_chip -dir netlist Optimized-micro-fc2
script_chip -dc Optimized-micro-fc2
```

---

**Running Command Scripts From the Shell**

You can run a FPGA Compiler II script in one or two ways:

1. **From the fc2_shell interface**, using the `source` command to execute the script file

   The syntax is

   ```bash
   fc2_shell> source script_name.scr
   ```

2. **When you invoke the fc2_shell interface**, using the `-f` option to execute the script file.

   The fc2_shell command with the `-f` option executes a script file before displaying the initial fc2_shell prompt.

   The syntax is

   ```bash
   fc2_shell -f script_file
   ```
Understanding FST Commands

The FPGA scripting tool (FST) is a Tcl-based command-line interface to all of the synthesis and optimization features of FPGA Compiler II.

To see a list of FPGA commands and its related man pages, do the following:

• At the shell prompt enter

  fc2_shell> help

• To access the man pages, enter

  fc2_shell> man commandname

• To display Help about the man pages, enter

  fc2_shell> man help

Commands are organized into command groups, either by action or by object. A special built-in command group contains a list of supported Tcl keywords, as well as some generic command line utilities common to Synopsys command line interfaces.

Commands and options that are not in the built-in command group can be abbreviated as long as the abbreviation is unique. If the abbreviation is not unique, a list of matching commands is displayed.

FST commands are in the form

<action>_<object>_<modifier>
The following objects defined by the data model are implemented in FST:

- project
- library
- file
- chip
- module
- clock
- group
- path group

In addition, the following netlist objects are implemented in FST:

- design
- port
- padcell
- net
- pin
FST Command Requirements

Most FST commands require a project to be opened. Once a project is open, it is referred to as the *current project*.

Commands that operate on chips usually operate on the *current chip*. The current chip is set by the `current_chip` command.

Modifications to the current chip are generally not saved until you change the current chip or close the current project. The exceptions to this rule are commands that take a chip name as an argument. For example, some commands, such as `export_constraint` or `export_netlist`, export information from a chip to a file. These commands require a chip as an argument and will read a chip before it writes the necessary information. If the chip to be exported is the current chip, it is automatically saved to disk. This behavior ensures that a file correctly reflects the state of a chip in a project.

The data models defined by the GUI are preserved in command line form by FST. Command line conventions established by the PrimeTime and Design Compiler products are followed as long as they are consistent with this data model.

FST is capable of writing simple Design Compiler shell scripts that can be used as a starting point for synthesizing an ASIC version of an FPGA chip.

Conversely, you can translate Design Compiler scripts to FPGA scripts by using the command `translate_dc_script`. Note, this is a command line only feature. For more information, see Appendix B, “Migrating From ASICS to FPGAs.”
Project Variables

This section lists the project variables and their definitions. Project variables are available only when a project is opened.

proj_altera_insert_lcell {yes, no (default)}
proj_clock_default_frequency = integer
proj_compiler_type = {presto (default), hdlc}
proj_enable_vpp = {yes, no (default)}
proj_export_directory = /any/legal/directory
proj_export_timing_constraint = {yes, no (default)}
proj_fsm_coding_style = {onehot (default) binary zeroonehot}
proj_fsm_optimize_case_default = {yes (default), no}
proj_gsr_ignore_unlinked_cells = {yes, no (default)}

A detailed description of these variables follows:

proj_altera_insert_lcell

Inserts LCELL buffers: the style is WSIWYG (what you see is what you get). This variable is used for Altera FLEX only. The default for this variable is no.

For Altera FLEX devices, enabling this option means the following specifications at optimization: insert LCELL buffers in the EDIF netlist and specify WYSIWYG style in the assignment constraint file for place and route. If this option is not enabled, no LCELL buffers are inserted and the logic synthesis style is FAST. When you change this setting, the changes take effect the next time you create an implementation.
proj_clock_default_frequency

To set the default clock frequency when creating a new chip.
When you create a new implementation, FPGA Compiler II offers a default frequency for the primary clock of the design. Use this variable to set the default for your subsequent project.

This target frequency is used as the default value for all clocks in the design. After you elaborate the design implementation, you can change target clock frequencies by using the set_clock command.

proj_compiler_type

This variable controls the compiler that is used for analyzing the source Verilog files. If the variable is set to Presto (the default), all the subsequent source files added to the project are analyzed by using the HDL Compiler (Presto Verilog). If the variable is set to hdlc, all the subsequent source files added to the project are analyzed by using the old compiler.

Note:
You must complete a force update on your design source files for the change to the new HDL Compiler (Presto Verilog) to take effect. You can use the update_project command from the fc2_shell to update your project files.

proj_enable_vpp

This variable controls the use of the Verilog preprocessor by the HDL analyzer. Verilog preprocessor constructs such as 'ifdef, 'else, and 'endif are disabled by default. To enable the Verilog Preprocessor for subsequent HDL analyses of Verilog files, enable this option. (The default for this variable is, no.)
proj_export_directory

You can specify the default directory in which netlists are stored during Export Netlist.

All files are exported to the directory set by the proj_export_directory variable. The directory can be specified by either absolute or relative path and must be writable.

If a directory is specified with the export_chip command, files are exported to the directory specified by the export_chip command and not to the directory set by the proj_export_directory variable.

In the following example, the proj_export_directory is set to proj_dir. The export_chip command is used without the -dir option. Therefore, all files will be exported to proj_dir.

```
f2_shell >add_file prep4.vhd
f2_shell >analyze_file
f2_shell >create_chip -target VIRTEX -name p_v prep4
f2_shell >current_chip -name p_v
f2_shell >optimize_chip -name p_v_opt
f2_shell >proj_export_directory=proj_dir
f2_shell >export_chip
f2_shell >quit
```

proj_export_timing_constraint

This variable controls the export of timing constraints when a netlist is exported. If the variable is set to yes, timing constraints are written when the chip is exported.

proj_fsm_coding_style

This variable controls the encoding for FSMs in VHDL designs. Choices include one hot (default), binary, and zero one hot. The default value of proj_fsm_coding_style is onehot.
proj_fsm_optimize_case_default

This variable controls the implementation of the “when others” statement in the VHDL case statement describing the state logic. Choices include fastest and smallest (default yes, or safest no).

Fastest and smallest (default yes) provides the best implementation when illegal state transitions can be ignored. The optimization function performs CASE DEFAULT optimization to reduce the redundant logic.

When recovery from illegal state transition is necessary, select safest (no).

To ensure the fastest and smallest FSM implementation, FPGA Compiler II ignores the “when others” statement from your FSM descriptions.

proj_gsr_ignore_unlinked_cells

This variable is for use with Xilinx devices only. This variable enables and disables the ignore unlinked cells option for global set or reset mapping. Without any options, global set or reset mapping ignores unlinked cells. With the -off option, global set or reset mapping does not ignore unlinked cells. This variable affects the implementation for the current chip.
FST Commands

This section contains lists of commands for working with constraints, generating reports, browsing design objects, managing designs, and so on. For more information, see the command's man pages.

Constraint Commands

Constraints commands let you import constraints from a file, export constraints to a file, and set constraints.

export_constraint
import_constraint
set_actel_option_chip_max_fanout
set_altera_option_chip_packed_mode
set_altera_option_map_to_memory_blocks
set_apex20k_option_module_loc
set_cell_max_fanout
set_chip_advanced_opt
set_chip_constraint_driven
set_chip_effort
set_chip_gsr_ignore_unlinked_cells
set_chip_hierarchy
set_chip_lpm
set_chip_max_fanout
set_chip_mem_map
set_chip_objective
set_chip_primitive
set_chip_sharing
set_lucent_option_pad_direct_in
set_lucent_option_pad_direct_out
set_module_dont_touch
set_module_effort
set_module_hierarchy
set_module_mem_map
set_module_objective
set_module_primitive
set_module_rmdup_cells
set_module_sharing
set_open_drain
set_pad_buffer
set_pad_dir
set_pad_drv
set_pad_input_voltage
set_pad_io_standard
set_pad_loc
set_pad_output_voltage
set_pad_reg_delay
set_pad_register
set_pad_resistance
set_pad_slew_rate
set_xilinx_option_chip_buf_internal_nets

---

Reports Commands

Reports commands let you generate reports about chips, files, libraries, and projects.

report_chip
report_file
report_library
report_project

Chapter 5: Using the FPGA Compiler II Shell

5-12
Browsing Objects Commands

Object browsing commands let you traverse objects in a design.

get_cell
get_chip
get_clock
get_design
get_device
get_file
get_library
get_module
get_net
get_pathgroup
get_pin
get_port
get_register
get_speed
get_target

Timing Commands

Timing commands let you create and delete subpaths, get timing information, set the clock speed, and set delays.

create_subpath
delete_subpath
report_timing
set_clock
set_disable_timing
set_input_delay
set_max_delay
set_output_delay
Source Design Management Commands

The `list_design` command lets you list all design names.

`list_design`

Target Management Commands

The `list_target` command lets you list all design targets.

`list_target`

Logical Library Management Commands

Logical library management commands let you create libraries, delete libraries, and generate lists of libraries.

`create_library`
`delete_library`
`list_library`

Chip Management Commands

Chip management commands let you read Standard Delay Format (SDF) files, create chips, list the current chip, delete chips, export chips, generate lists of chips, optimize chips, set the chip export directory, and so on.

`chip_place_route`
`chip_read_sdf`
`create_chip`
`current_chip`
`delete_chip`
`export_chip`
`export_sdf`
`list_chip`
optimize_chip
script_chip
set_chip_export_directory
set_chip_retiming

Source File Management Commands
Source file management commands let you add design source files, analyze files, generate lists of files, and delete files.

add_file
analyze_file
list_file
remove_file

Project Management Commands
Project management commands let you open a project, close the current project, and create a project.

open_project
close_project
create_project

Built-In Tcl Commands
The FPGA scripting language includes many commands in the Tcl standard.

after
alias
append
apropos
array
break
catch
cd
clock
close
concat
continue
define_proc_attributes
echo
eof
error
error_info
eval
eexec
exit
expr
fblocked
fconfigure
file
fileevent
flush
for
foreach
format
gets
glob
global
help
history
if
incr
info
interp
join
lappend
lindex
linsert
list
llength
load
lrange
lreplace
lsearch
lsort
man
open
package
parse_proc_arguments
pid
print_suppressed_messages
printvar
proc
puts
pwd
quit
read
redirect
regexp
regsub
return
scan
seek
set
socket
source
split
string
subst
suppress_message
switch
tell
time
trace
unalias
unset
unsuppress_message
update
uplevel
upvar
vwait
which
while
Other Commands

This section lists miscellaneous FST commands.

list_message
list_status
translate_dc_script
update_chip
update_file
update_library
update_project
Block-level incremental synthesis (BLIS) is a feature that lets you change part of a design and resynthesize only that part. This decreases overall compilation time for synthesis and place and route. BLIS preserves the post-place and route timing behavior for the unchanged parts.

This chapter includes the following sections:

- Identifying Blocks
- Defining Block Roots Using the GUI
- Defining Block Roots From the Shell
- Updating an Implementation Using BLIS
Identifying Blocks

The subset of a design to which you can apply BLIS is called a block. A block is composed of one or more verilog modules, VHDL entities, or EDIF netlists in the design’s hierarchy.

The top-level component of a block is called a block root. The components of a block include the block root and all parts of the design hierarchy below it that do not include another block root.

Before you use BLIS, you must divide the design into blocks. FPGA Compiler II generates a separate netlist for each block. The netlist of a block does not change unless you change its corresponding design.

For example, the top-level module TOP in Figure 6-1 has two components, A and B. Components C and D are under A in the hierarchy, and components E and F are in B's hierarchy.

This means that the top-level design TOP is a block root. If you want to designate A and E as block roots, then the design has three blocks:

Block 1: TOP, B, F
Block 2: A, C, D
Block 3: E
If you change a module, entity, or netlist in a block, the entire block is resynthesized. For example, if you change F, every member of Block 1 (TOP, B, and F) is resynthesized, even though you did not change TOP or B.

**Defining Block Roots Using the GUI**

To identify blocks by using the graphical user interface (GUI), right-click an elaborated implementation in the Chips window, choose Edit Constraints, and click the Modules tab. The Modules constraint table (**Figure 6-2**) appears.
Then, use the Modules constraint table to specify any components as block roots in the Block Partition column.

To remove a block root designation, click the cell and choose Remove. (The top-level design is always a block root, which means that you cannot remove it.)
Defining Block Roots From the Shell

To identify blocks by using the shell, use the command `set_module_block` followed by the option `true` and the path to the module, entity, or netlist that you want to specify as the block root. For example:

```
fc2_shell> set_module_block true /TOP/A
```

For more information about `set_module_block`, see the man pages.

The following is an example fc2_shell script to run BLIS. See the man pages for usage and syntax.

```
create_project TOP

add_file -library WORK -format VHDL top.vhd
add_file -library WORK -format VHDL clock.vhd
add_file -library WORK -format VHDL cntr.vhd
add_file -library WORK -format VHDL countd.vhd
add_file -library WORK -format VHDL countu.vhd
add_file -library WORK -format VHDL fsm.vhd
add_file -library WORK -format VHDL arith.vhd

analyze_file -progress

create_chip -progress -name TOP -target APEX20KE -preserve TOP

current_chip TOP

set_module_block true /TOP/A
set_module_block true /TOP/B/E

optimize_chip -name TOP-Optimized -progress

export_chip -timing_constraint -dir project
```
Caution!

The concept of block and block root applies only when the target architecture supports BLIS. Attempting to apply this feature on an architecture that is not supported by BLIS results in the following error message:

Error: block assignments are not supported for the target technology of this chip

Updating an Implementation Using BLIS

This section describes how to update an implementation after you change a design. Before you update an implementation, make sure that you have defined blocks, created an optimized implementation, and generated place and route netlists.

In the GUI, a question mark (?) next to a design source file in the Design Sources window indicates that the file has changed. To reanalyze the file, right-click the file and choose Update Project. A red question mark next to an elaborated implementation in the Chips window shows that the implementation is out-of-date with respect to its design source files. To update the implementation, right-click the elaborated implementation and choose Update Project. See Figure 6-3 on page 6-7.
Figure 6-3  Update Project

Note:
It is not recommended to export a netlist after updating your project. Doing so will cause all netlists to be regenerated. This step is recommended only when the design is synthesized for the first time.

To update your project from the fc2_shell, use the `update_proj` command.
Choosing Block Definitions

BLIS is only applied to designs with at least two blocks defined. BLIS incrementally synthesizes your design when you update your source files (detected via time stamps), or if you change the block root definitions. You cannot use BLIS when you perform other operations (such as changing timing constraints).

BLIS uses time stamps to determine if an implementation is out-of-date. If the time stamp of any analyzed design source file is newer than the elaborated implementation, then you must update the elaborated implementation.

You cannot use BLIS if you describe all modules, entities, or netlists in a single design source file. This means that when planning a design for BLIS, you must describe each module, entity, or netlist in its own design source file. This ensures that changing a module, entity, or netlist does not affect the time stamps of other modules, entities, or netlists that might be members of other blocks.

For example, in Figure 6-1, a single design source file describes A and B. Changing A not only makes the tool resynthesize the entire Block 2, but it also changes the time stamp of B, because it is in the same design source file as A. The newer time stamp of B makes the tool resynthesize the entire Block 1.
Planning a Design for BLIS

Because FPGA Compiler II does not optimize incrementally across block boundaries, you should not separate combinational logic paths into different blocks. You must group related combinational logic within a module or entity and register all of the module or entity outputs.

BLIS is most effective when you make changes within a module or entity, because changing the partition or the pins of the modules or entities of the design causes the tool to resynthesize the entire design. This is why thorough planning early in the design process is vital.

Using BLIS With Altera Quartus II

When BLIS is used with the LogicLock feature from Altera Quartus II version 1.1 or later, it enables you to perform a place and route on the modified block only, reducing incremental design cycle time and preserving the timing of unmodified blocks. For more information see the Altera Application Note 161, “Using the LogicLock Methodology in the Quartus II Design Software.”

Note:
To set up your design correctly make sure the implementation is up-to-date.
Chapter 6: Using Block-Level Incremental Synthesis (BLIS)

6-10
This chapter describes using DesignWare components in FPGA Compiler II.

This chapter contains the following sections:

- Installing DesignWare Foundation
- Instantiating DesignWare Foundation Components
- Synthesizing DesignWare Foundation
DesignWare Foundation is a library of verified components that you can synthesize, parameterize, and reuse. The library contains smaller functions (such as adders, multipliers, and shifters) to larger and more complex functions (such as FIFOs, JTAGs, and microcontrollers). This section describes how to use Synopsys DesignWare Foundation in FPGA Compiler II.

Basic DesignWare Foundation components are mapped to technology-specific primitives. You can use instantiated DesignWare Foundation components that are created for Design Compiler and use FPGA Compiler II to synthesize designs that contain these components.

For a list of supported DesignWare components, go to the Synopsys DesignWare Web page at http://www.synopsys.com/products/designware/dw_fl_ds.html

**Installing DesignWare Foundation**

The Typical installation option of FPGA Compiler II installs DesignWare Foundation libraries by default. When you choose the Custom installation option, you can choose not to install DesignWare Foundation libraries by deselecting DesignWare in the FPGA Vendors and Families dialog box (Figure 7-1).
Using the entire DesignWare Foundation library requires a DesignWare Foundation license. However, you can implement basic DesignWare Foundation components without a license. The basic components are

- DW01_cmp2
- DW01_cmp6
- DW01_absval
- DW01_add
- DW01_sub
• DW01_addsub
• DW01_inc
• DW01_dec
• DW01_incdec
• DW02_mult

**Instantiating DesignWare Foundation Components**

You can use DesignWare Foundation components in FPGA Compiler II by direct instantiation. Following are two examples.

In Verilog:

```verilog
DW02_mult #(inst_A_width, inst_B_width)
U1 (.A(inst_A), .B(inst_B), .TC(inst_TC),
   .PRODUCT(inst_PRODUCT));
```

In VHDL:

```vhdl
U1: DW02_mult generic map ( A_width => inst_A_width, B_width => inst_B_width )
   port map (A => inst_A, B => inst_B, TC => inst_TC, PRODUCT => inst_PRODUCT);
```

Note:

You cannot infer DesignWare Foundation components in FPGA Compiler II.
Synthesizing DesignWare Foundation

Unlike Design Compiler (in which you can choose a specific architecture for a DesignWare Foundation component), FPGA Compiler II automatically selects the implementation for the FPGA technology that you are using.

FPGA Compiler II uses vendor-specific architectures. For example, when it encounters an instantiated DW01_add component, FPGA Compiler II maps it to carry chains (if the target FPGA technology supports them).
Implementing Memory Elements

This chapter describes how to implement RAM and ROM elements in design source files to use with FPGA Compiler II.

This chapter includes the following sections:

- Using Memory Elements With Altera Devices
- Using Memory Elements With Lucent Devices
- Using Memory Devices With Xilinx Devices
Using Memory Elements With Altera Devices

You can synthesize designs with RAM or ROM elements by using these steps:

Generate the type of memory you want using the Genmem utility, Altera’s memory generator.

For Verilog designs:

1. Use the `memfile.v` file for port information for the memory instance. Instantiate a memory instance in your Verilog source code. See Example 8-1 on page 8-3.

2. Include the simulation model `memfile.v` in your project.

3. Create an implementation.

4. In the Modules constraint table for the implementation, choose Preserve Hierarchy column for each memory instance.

For VHDL designs:

1. Include the `memfile.cmp` component declaration in your VHDL source code.

2. Instantiate a VHDL memory instance in your VHDL source code. You do not need to preserve the instance in FPGA Compiler II when using VHDL. See Example 8-2 on page 8-4.
Example 8-1  Asynchronous RAM Instance in Verilog

module asyn_ram_16x8 (Q, Data, WE, Address);
parameter LPM_FILE = "UNUSED";
parameter Width = 8;
parameter WidthAd = 4;
parameter NumWords = 16;

input [WidthAd-1:0] Address;
input [Width-1:0] Data;
input WE;
output [Width-1:0] Q;

input [3:0] Address;
input [7:0] Data;
input WE;
output [7:0] Q;

asyn_ram_16x8 u1(
  .Address(Address),
  .WE(WE),
  .Data(Data),
  .Q(Q)
);

Note:
Choose the Preserve Hierarchy option for the instance u1.
Example 8-2  Asynchronous RAM Instance in VHDL

component asyn_ram_16x8
pragma translate_off
generic ( LPM_FILE : string );
pragma translate_on
port ( Data    : in std_logic_vector(7 downto 0);
      Address : in std_logic_vector(3 downto 0);
      WE      : in std_logic;
      Q       : out std_logic_vector(7 downto 0)
      );
end component;
begin  --  rtl
u1 : asyn_ram_16x8 port map(
      Data => d,
      Address => a,
      WE => w,
      Q => qu
      );
end rtl;

Note:
It is not necessary to choose the Preserve Hierarchy option for the instance u1 when using VHDL.
You can synthesize designs with RAM or ROM elements in FPGA Compiler II by using the steps described in the following sections.

**Synthesizing Designs with RAM Elements**

To synthesize designs with RAM elements:

1. Generate the RAM netlist using SCUBA, Lucent’s Synthesis Compiler for User programmable Arrays. To place the RAM into a VHDL design, choose VHDL output from SCUBA. To place the RAM into a Verilog design, choose Verilog output from SCUBA. SCUBA generates a VHDL entity/architecture pair or Verilog module composed of ORCA library primitives.

2. For VHDL, instantiate the RAM from SCUBA in your design, create a component declaration for the RAM, and place the declaration in the same architecture as the instance. For Verilog, you only need to instantiate the RAM. For an example of a VHDL and Verilog design containing an SSPRAM, see Example 8-3 and Example 8-4.

3. Read the HDL files for your design and the HDL netlists generated by SCUBA into FPGA Compiler II.

4. Synthesize the design.
Example 8-3  RAM Example of a SCUBA Generated VHDL Netlist

-- begin netlist

library IEEE;
use IEEE.std_logic_1164.all;

entity my_sspram is
  port (addr: in std_logic_vector(3 downto 0); datain: in std_logic_vector(3 downto 0);
        clk: in std_logic; wpe: in std_logic; wren: in std_logic;
        dataout: out std_logic_vector(3 downto 0));
end my_sspram;

architecture Structure of my_sspram is

  -- internal signal declarations
  signal wpe_inv: std_logic;

  -- local component declarations
  component RCF16X4Z
    port (AD0: in std_logic; AD1: in std_logic; AD2: in std_logic;
          AD3: in std_logic; DI0: in std_logic; DI1: in std_logic;
          DI2: in std_logic; DI3: in std_logic; CK: in std_logic;
          WREN: in std_logic; WPE: in std_logic; TRI: in std_logic;
          DO0: out std_logic; DO1: out std_logic; DO2: out std_logic;
          DO3: out std_logic);
  end component;
  component INV
    port (A: in std_logic; Z: out std_logic);
  end component;

begin

  -- component instantiation statements
  INV_0: INV
    port map (A=>wpe, Z=>wpe_inv);

  mem_0: RCF16X4Z
    port map (AD0=>addr(0), AD1=>addr(1), AD2=>addr(2), AD3=>addr(3),
              DI0=>datain(0), DI1=>datain(1), DI2=>datain(2), DI3=>datain(3),
              CK=>clk, WREN=>wren, WPE=>wpe, TRI=>wpe_inv, DO0=>dataout(0),
              DO1=>dataout(1), DO2=>dataout(2), DO3=>dataout(3));

end Structure;

-- end netlist
Use the port information in this netlist to generate the instance and component declaration for the instantiating architecture:

```verilog
architecture struct of mem_top is

    component my_sspram
        port (addr: in std_logic_vector(3 downto 0);
             datain: in std_logic_vector(3 downto 0);
             clk: in std_logic; wpe: in std_logic; wren: in std_logic;
             dataout: out std_logic_vector(3 downto 0));
    end component;

begin  --  struct

    u0 : my_sspram
        port map(addr, datain, clk, wpe, wren, dataout);

Example 8-4  RAM Example of a SCUBA Generated Verilog Netlist

module my_sspram (addr, datain, clk, wpe, wren, dataout);
    input [3:0] addr;
    input [3:0] datain;
    input clk;
    input wpe;
    input wren;
    output [3:0] dataout;

    INV INV_0 (.A(wpe), .Z(wpe_inv));
    RCF16X4Z mem_0 (.AD0(addr[0]), .AD1(addr[1]),
                     .AD2(addr[2]), .AD3(addr[3]), .DI0(datain[0]),
                     .DI1(datain[1]), .DI2(datain[2]), .DI3(datain[3]),
                     .CK(clk), .WREN(wren), .WPE(wpe), .TRI(wpe_inv),
                     .DO0(dataout[0]), .DO1(dataout[1]),
                     .DO2(dataout[2]), .DO3(dataout[3]));
endmodule

Use the port information in this netlist to generate the instance for the instantiating module:

```
Synthesizing Designs With ROM Elements

To synthesize designs with ROM elements:

1. Prepare a ROM memfile for the SCUBA program. This file contains the values for each location in the ROM. Example 8-5 shows an example of a memfile for a 16x4 ROM. It is an ASCII file with the format addr: value value value.

Example 8-5 Memfile for a 16x4 ROM

<table>
<thead>
<tr>
<th>Addr</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0 1 2 3</td>
</tr>
<tr>
<td>04</td>
<td>4 5 6 7</td>
</tr>
<tr>
<td>08</td>
<td>8 9 A B</td>
</tr>
<tr>
<td>0C</td>
<td>C D E F</td>
</tr>
</tbody>
</table>

For detailed information, see the *Lucent SCUBA Reference Manual*.

2. Generate the ROM netlist using SCUBA. Choose BusA0 as the bus style and EDIF as the output format. SCUBA generates an EDIF netlist of ORCA primitives. See Example 8-6.

3. For VHDL, instantiate the ROM from SCUBA in your design, create a component declaration for the ROM, and place the declaration in the same architecture as the instance. For Verilog, you only need to instantiate the ROM. See Example 8-7.

4. Read the HDL files for your design and the EDIF netlists generated by SCUBA into FPGA Compiler II.

5. Synthesize the design.

Use the port information in the EDIF netlist to generate the instance and component declaration for the instantiating architecture.
Example 8-6  ROM Example of a SCUBA Generated EDIF Netlist

architecture struct of top is

    component my_rom
        port(addr3, addr2, addr1, addr0: in std_logic;
             dataout3, dataout2, dataout1, dataout0: out std_logic);
    end component;

begin  --  struct

    rom_i : my_rom
        port map(addr3 => a(3),
                  addr2 => a(2),
                  addr1 => a(1),
                  addr0 => a(0),
                  dataout3 => z(3),
                  dataout2 => z(2),
                  dataout1 => z(1),
                  dataout0 => z(0));

    -- rest of architecture

    For Verilog, instantiate the ROM entity from SCUBA in your design.

Example 8-7  ROM Example of a SCUBA Generated Verilog Netlist

my_rom rom_i(
    .addr0(a[0]),
    .addr1(a[1]),
    .addr2(a[2]),
    .addr3(a[3]),
    .dataout0(z[0]),
    .dataout1(z[1]),
    .dataout2(z[2]),
    .dataout3(z[3])
);

Using Memory Elements With Lucent Devices

8-9
Using Memory Devices With Xilinx Devices

You can synthesize designs with memory elements when the target devices are Xilinx devices. The methodology describes instantiating LogiBLOX components in VHDL and Verilog but can be applied to any other kind of component such as LogiCORE.

Note:
LogiBLOX elements must be generated with Xilinx’s LogiBLOX software and instantiated in the VHDL or Verilog design that is synthesized by FPGA Compiler II.

Instantiating a LogiBLOX Element in a VHDL Design

You can Instantiate a LogiBLOX Element in a VHDL design by following these steps:

1. Generate the LogiBLOX element using the Xilinx LogiBLOX software.
   a. In the Setup window, set the vendor name to Synopsys and the Bus Notation to B<l>. In the Options tab, select VHDL Template.
   b. Define the LogiBLOX module attributes. The module name specified for the LogiBLOX module is used as the name of the instantiated component in the VHDL code.
   c. Click the OK button. The software creates a number of files, including an .ngo file (the file merged by ngdbuild) and a .vhi file (used as an instantiation reference).
2. Instantiate the LogiBLOX element:

Using the .vhi file for the port names of the instantiated component, write the VHDL code to instantiate the LogiBLOX element and declare the corresponding component in your VHDL design.

3. Synthesize your design:

Read your VHDL design into FPGA Compiler II, synthesize it, and export the netlist file.

4. Place and route your design:

Using the netlist written by FPGA Compiler II and the .ngo file written by LogiBLOX, process the design through the Xilinx place and route tool.

---

**Instantiating a LogiBLOX Element in a Verilog Design**

You can Instantiate a LogiBLOX Element in a Verilog design by following these steps:

1. Generate the LogiBLOX element using the Xilinx LogiBLOX software:

   a. In the Setup window, set the vendor name to Synopsys and the Bus Notation to B<l>. In the Options tab, select Verilog Template.

   b. Define the LogiBLOX module attributes. The module name specified for the LogiBLOX module is used as the name of the instantiated component in the Verilog code.

   c. Click the OK button. The software creates a number of files, including an .ngo file (the file merged by ngdbuild) and a .vei file (used as an instantiation reference).
2. Instantiate the LogiBLOX element:

   Using the .vei file for the port names of the instantiated module, write the Verilog code to instantiate the LogiBLOX element in your design. Note that you must also declare an empty Verilog module for this component. This module passes port direction information to FPGA Compiler II (this is realized by the component statement in VHDL, but there is no equivalent statement in Verilog).

3. Synthesize your design:

   Read your Verilog design in FPGA Compiler II, synthesize it, and export the netlist.

4. Place and route your design:

   Using the netlist written by FPGA Compiler II and the .ngo file written by LogiBLOX, process the design through the Xilinx place and route tool.
A

Using ASCII Constraint Files

This appendix describes how to use constraint files with FPGA Compiler II. It includes the following sections:

• About the ASCII Constraint File Format
• Exporting an ASCII Constraint File
• Importing an ASCII Constraint File

FPGA Compiler II can export and import constraint files in ASCII format.

When you export a constraint file, you create a file that contains the constraint settings of the current chip. You can edit the ASCII constraint file and reapply the new constraint information to the current chip. When you import a constraint file, FPGA Compiler II sets the constraints on the current chip.
This method of exporting, editing, and importing the constraint file is useful if you use a scripting methodology. However, GUI users can find this method of editing constraints useful as well—for example, in order to avoid keyboard and mouse tasks required by data entry in the GUI constraint tables.

Exporting or importing a constraint file always acts on the current chip. Set the current chip with the `current_chip` command before exporting or importing a constraint file.

**Important:**

To support current methodologies that require doing so, you can continue to import constraint files in binary format. Exporting binary constraint files is no longer supported.
About the ASCII Constraint File Format

In the ASCII constraint file you can find

- Header information that identifies the chip name and target architecture
- Constraint information and commands in sections that correspond to the tabs in the constraint tables
- Comments that show valid constraint settings and other information

The ASCII constraint file contains constraint information, comments, and defaults for

- Clocks
- Object groups (as commented information in the export file)
- Paths
- Ports
- Modules
- Registers
- Vendor options

Comments in the file begin with a pound sign (#).
Header Information

Header information consists of the chip name and the architecture. The header appears at the beginning of the file. Header information is optional when you are importing an ASCII constraint file.

Following is an example:

```plaintext
chip_name = tutor;
tutor_name = APEX20K;
```

Clock Constraints

The clock constraints portion of the file contains default information and user settings. The first line defines the default clock period, rise time, and fall time. Subsequent lines contain the clock names followed by the clock period, rise time, and fall time for each clock. If the clock uses a default value, -1 appears.

Following is an example:

```plaintext
######## Clock Constraints ########

clock <default> 20/0/10
clock data_clk1 -1/-1/-1
clock clk2 -1/-1/-1
```
Object Groups

Logically related groups of design objects appear as comments in the ASCII constraint file. In the following example, (I) is the group of input objects, (O) is the group of output objects, and (RC, data_clk1) is the group of objects triggered by the rising clock edge of data_clk1.

Example

```
# Members of Group  (I):
#   /d/vcc
#   /d/data_clk1
#   /d/clk2
#   /d/clrn
#   /d/ena1
#   /d/ena2
#   /d/data[3]
#   /d/data[2]
#   /d/data[1]
#   /d/data[0]

# Members of Group  (O):
#   /d/regout
#   /d/vcc
#   /d/data[3]
#   /d/data[2]
#   /d/data[1]
#   /d/data[0]
#   /d/done

# Members of Group  (RC, data_clk1):
#   /d/u1/q_reg
```
Path Constraints

The path constraints section sets the required delay between objects or object groups.

In the following example, the first line defines required timing in nanoseconds between all inputs (I) and outputs (O). The second line defines required timing in nanoseconds between all inputs and objects triggered by the rising edge of the data_clk1 clock.

Example

```
######## Path Constraints ########
path -from (I) -to (O) -delay 33
path -from (I) -to (RC,data_clk1) -delay 34
```

To define subpaths, you define a name for the subpath and the explicit sources and destinations of the subpath.

In the following example, the first path command defines a subpath called my_from:my_to between five input objects and one output object.

In the second path command, the subpath is called m_f:m_t. The subpath starts at input /d/ena2 and terminates at output /d/regout.
For clarity, the export file prints the -from, -to, and -delay objects value on separate lines.

Example

path my_from:my_to
  -from
    (I:
      /d/data_clk1,
      /d/clk2,
      /d/clrn,
      /d/ena1,
      /d/ena2)
  -to
    (O:
      /d/regout)
  -delay 43;

path m_f:m_t
  -from (I:
    /d/ena2)
  -to (O:
    /d/regout)
  -delay 53;

Note:
Make sure that the path you define exists. The tool does not validate the path you define.

Port Constraints

The port constraints section of the constraint file contains user-defined defaults and port constraints.
Example

######## Default Port Constraints ########

    port  <default>   use_io_reg  "OFF";
# Valid 'use_io_reg' values are:
#       <default>
#       ON
#       OFF

    port  <default>   slew_rate  "FAST";
# Valid 'slew_rate' values are:
#       <default>
#       SLOW
#       FAST

Example

######## Port Constraints ########

# Port clk2, direction = input

    port  clk2  input_delay  34/(RC,data_clk1);
# Port clrn, direction = input

    port  clrn  input_delay  34/(RC,data_clk1);
    port  clrn  location  "data location";

# Port done, direction = output

    port  done  output_delay  20/(RC,data_clk1);

Module Constraints

The module constraints section of the constraint file contains user-defined defaults and module constraints.
### Default module constraints

```plaintext
module <default> hierarchy "preserve"
# Valid module ‘hierarchy’ values are:
#   inherit
#   preserve
#   eliminate
```

```plaintext
module <default> primitive "preserve";
# Valid module ‘primitive’ values are:
#   inherit
#   preserve
#   optimize
```

### Module Constraints

```plaintext
# module test - Root design
module d hierarchy "inherit";
module d primitive "inherit";
module d dont_touch "false";
module d optimize_for "inherit";
module d effort "inherit";
module d dup_reg_merge "<default>";
```

```plaintext
# module dflipflop - u1
module /d/u1 hierarchy "inherit";
module /d/u1 primitive "inherit";
module /d/u1 dont_touch "inherit";
module /d/u1 optimize_for "inherit";
module /d/u1 effort "inherit";
module /d/u1 dup_reg_merge "<default>";
```
Register Constraints

The register constraints section displays information from the Registers constraint table. The estimated fanout for registers is shown in comment lines.

Example

```plaintext
#  Register Constraints :
#------------------------
 register <default> max_fanout 3;
  register  sdatc_reg<11>  max_fanout  5;
#  register  sdatc_reg<11>  estimated fanout = 1
#  register  sdatb_reg<4>  estimated fanout = 1
```

Vendor Options

The vendor options section contains options specific to the available target architectures.

Example

```plaintext
lucent_option  ignore_unlinked_cells "disable";
xilinx_option  ignore_unlinked_cells  "enable";
xilinx_option  buffer_internal_nets  "enable";
actel_option  max_fanout  16;
actel_option  advanced_optimization  "disable";
genral_option  use_lpm  "enable";
```
**Exporting an ASCII Constraint File**

You can export an ASCII file of constraint commands for the current implementation by using menu commands or the command line (scripting tool).

To use GUI commands to export an ASCII constraint file,

1. Right-click the elaborated implementation in the Chips window.
2. Choose Edit Constraints from the pop-up menu.

   Steps 1 and 2 set the current implementation and enable the menu items in step 3.


   The Export Constraint File dialog box opens.

4. Enter the file name you want and click Save.

   When saving the file, the tool appends the .scf ending for an ASCII constraint file.

To use the shell to export an ASCII file of constraint commands for the current implementation,

1. Run the scripting tool.

   For instructions, see Chapter 5, “Using the FPGA Compiler II Shell.”

2. Set the current chip. At the shell prompt, enter

   > current_chip my_chip
3. At the shell prompt, enter

```
> export_constraint -ascii file_name
```

where `file_name` is the output file name you choose. When saving the file, the tool appends the `.scf` ending for a constraint file in ASCII format.

---

**Importing an ASCII Constraint File**

You can import an ASCII file of constraint commands for the current chip by using menu commands or the scripting tool.

To use GUI commands to import an ASCII constraint file for application to the current implementation,

1. Right-click the elaborated implementation in the Chips window.
2. Choose Edit Constraints from the pop-up menu.
   
   Steps 1 and 2 set the current chip and enable the menu items in step 3.
   
   The Import Constraints File dialog box opens.
4. Enter the file name and click OK.

To use the shell to import an ASCII file of constraint commands for the current implementation,

1. Run the scripting tool.
For instructions, see Chapter 5, “Using the FPGA Compiler II Shell.”

2. Set the current implementation. At the shell prompt, enter

```
> current_chip my_chip
```

3. At the shell prompt, enter

```
> import_constraint -ascii -f file_name
```

where `file_name` is the name of the file of constraint commands to apply to the current implementation.

When importing a constraint file, the tool skips any lines that contain errors and issues an error message. For debugging the constraint file, a line number appears with each error message. Lines with errors are not processed.

Messages are issued at the command line for the following problems:

- The constraint is unknown (misspelled, for example).
- The constraint is not supported for the object. For example, the architecture does not support the constraint, or the constraint name is misspelled.
- The target object cannot be found. For example, the object name is misspelled.
- The constraint is outside the valid range.
- The object is invalid for the constraint. For example, the object and constraint both exist, but are inappropriate for each other.
This appendix gives an overview of how to use FPGA Compiler II to convert ASIC designs to FPGAs. It discusses the features in the synthesis process that are related to migration and optimization.

This appendix includes the following sections:

- HDL Source Code Compatibility
- Design Compiler Shell Script Translation
- Synopsys Database Support
- Register Retiming
- Integration With Place and Route Tools
FPGA Compiler II has architecture-specific FPGA synthesis technology that is compatible with the Synopsys design flow. This technology has capabilities and features that provide a smooth transition between ASICs and FPGAs, and it lets you use FPGAs without altering your ASIC design flow.

FPGA Compiler II includes many features that let you have the benefits of FPGAs while reusing designs that previously targeted ASIC technologies. *Figure B-1* shows a typical FPGA Compiler II design flow.

*Figure B-1  Typical FPGA Compiler II Design Flow*
HDL Source Code Compatibility

The first part of migration is bringing the design files into FPGA Compiler II. Because FPGA Compiler II optimizes the design specifically for the targeted programmable architecture when you import it, you must start with the HDL source files (rather than a netlist optimized for another technology). However, FPGA Compiler II uses the same HDL compiler for source code analysis as Design Compiler, which lets you reuse the same industry-standard Verilog or VHDL source code in the FPGA Compiler II environment. For more information, see “Setting Project Synthesis Options” on page 3-5. This is also true for designs that contain instantiated DesignWare Foundation components. For more information, see Chapter 7, “Using DesignWare Components.”
Design Compiler Shell Script Translation

The command line interface (fc2_shell) uses a Tcl-based scripting language. This interface lets you use every GUI feature of FPGA Compiler II on the command line, which is useful for repetitive project runs. Also, it lets you change project parameters by eliminating the sequential steps that the GUI requires.

You do not need to rewrite existing dc_shell scripts, because FPGA Compiler II translates them to the fc2_shell format. The command to do this is translate_dc_script. For example, enter the following at the fc2_shell prompt:

```shell
> translate_dc_script -input_script run.scr -output_script run.fc2
```

where `run.scr` is the dc_shell script and `run.fc2` is the output in fc2_shell format.

FPGA Compiler II ignores dc_shell constructs that do not apply to the current FPGA design.

For more information, see Chapter 5, “Using the FPGA Compiler II Shell.”
Synopsys Database Support

You can use FPGA Compiler II to export .db files. This feature lets you create projects in FPGA Compiler II and bring them directly into other Synopsys tools (such as Design Compiler and PrimeTime).

FPGA Compiler II creates two .db files. For example, for a design called test, it creates a file named test-Optimized_des.db, which describes the design. It creates another file named test-Optimized_lib.db, which contains the technology library primitives for the design. Technology library primitives contain both timing and functional information. For more information, see “Specifying Optimization and Output Options” on page 2-5” and “Exporting a Netlist” on page 3-23.

Register Retiming

FPGA Compiler II has a feature for design optimization called register retiming. This feature optimizes at a high level to improve design performance and timing closure. This feature is tuned for high-density and high-performance programmable devices.

You use retiming when at least one critical path does not meet timing requirements. Retiming positions the registers to minimize delays from input to register, register to register, and register to output. Retiming lets you maximize design performance without editing the HDL source files.

You can use retiming for the Altera APEX20K and FLEX10K and the Xilinx Virtex and XC4000 architectures. You can use retiming for only flattened (nonhierarchical) designs.
In the example in Figure B-2, the required clock period is 10 ns. The design implementation before retiming has a critical path of 11 ns and another path with a delay of 7.5 ns.

**Figure B-2  Register Retiming**

Retiming moves the registers forward, which improves the critical path. In this example, moving the left register set forward would not improve the design performance. However, moving the middle register set increases the delay on the fastest path group (on the left), but it also reduces the delay on the slowest path group (on the right). Therefore, by moving the middle register set forward, you can improve the critical path from 11 ns to 9.8 ns.

Retiming also lets you pipeline combinational logic by adding registers at its input. In the example in Figure B-3, the original design has a combinational portion with a delay of 18 ns.
You can add a register bank (a bank of three registers in the example) at the input of the design by editing the HDL source file. With the retiming option enabled, FPGA Compiler II moves the registers automatically through the design to achieve timing closure.

To use retiming in the GUI, right-click the preoptimized chip in the Chips window and choose Edit Constraints. Then, click the Vendor Options tab and select the Perform Retiming check box.

To use retiming in the FPGA Compiler II shell, use the `set_chip_retiming` command on the preoptimized chip:

```
> current_chip top
> set_chip_retiming -enable
```
Integration With Place and Route Tools

After you generate a netlist and constraint files for the place and route tools, FPGA Compiler II generates the appropriate interface files for the target technology. These include the netlist, the constraint file, and the library file.

To generate the interface files by using the GUI, right-click the optimized chip in the Chips window and choose Export Netlist. The tool generates the necessary files in the directory that you specify. For more information, see “Specifying Optimization and Output Options” on page 2-5.

To generate the interface files by using the shell, use the `export_chip` command on the optimized chip:

```
> optimize_chip -name top-Optimized
> export_chip
```
This appendix shows how to mix HDL and netlist inputs.

To mix HDL and netlist inputs,

1. Create the appropriate netlist format.
   
   If your design contains schematics, you must export them to EDIF format in your schematic editor.

2. Instantiate the netlist and HDL modules.

   Case 1: HDL module instantiating netlist modules

   The netlist modules must be instantiated in the HDL code as black boxes.

   - In VHDL: Declare the corresponding components and instantiate them. Do not declare any architecture block for these modules—the description is derived from the netlist.
- In Verilog: Instantiate the corresponding modules. Also include empty modules to provide port direction information.

Case 2: Netlist module instantiating HDL modules

The HDL modules must appear in the netlist as black boxes. Their functionality will be derived from synthesis.

3. Synthesize your design.
   
a. Add all the netlist and HDL files to your project.
   b. Select the top-level design and create an implementation.
Glossary

active edge
The active edge of a clock signal at a sequential element is the edge at which the output signal becomes stable.

For flip-flops, the active edge is the clocking edge. For example, the rising edge is the active edge for rising-edge-triggered flip-flops.

For level-sensitive latches, the active edge is the end of the enabled time. For example, the falling edge is the active edge for positively enabled latches.

Active edge is an important concept for understanding how default timing values are assigned to path groups.

analysis order
The order in which design source files are analyzed; that is, the files are analyzed in the order in which you add them to a project.

Analysis order is important only for VHDL files and for files in which packages are defined and then referenced. For example, if a package defined in file1 uses another package defined in file2, add file2 before adding file1. This order of addition causes the tool to analyze file2 before file1. Be sure to add design source files to the project in the order in which they must be analyzed.

See also design source files and project.
analyze
What the tool does to check for syntax errors and verify that the HDL source file contents conform to Synopsys HDL policy.

In the GUI, the HDL design source files are automatically analyzed when you add them to a library. The shell commands for adding and analyzing design source files are `add_file` and `analyze_file`.

See also design source files.

back-annotation
The updating of a circuit design using simulation and other post-processing information that reflects implementation-dependent characteristics of the design such as pin selection, component location, or parasitic electrical characteristics.

bidirectional
A signal or port that can act as either an input to or an output from an electronic circuit.

clock
A signal with a periodic behavior. In synchronous circuit designs, clocks are used to synchronize the propagation of data signals by controlling sequential elements. It is important that you accurately specify all clocks so that a synchronous circuit can be optimized efficiently. The tool automatically analyzes the circuit and lists all the signals that require clock definitions in the Clocks constraint table.

A clock is defined by its waveform, with rising and falling edge times in the interval between 0 and its period. Because the interval always starts at time 0, the precise relationship between all the clocks in the design are known by the tool.

A clock can also be specified by its frequency, in megahertz. In this case, the tool converts the frequency into a waveform rising at time 0 and falling at the specified fall time.

debugging
Finding and eliminating errors in designs.
**delay**
The time it takes a signal to propagate from one point to another. All delays specified are in nanoseconds and must be expressed as integers.

**design source files**
Text files that contain the design description, which can be VHDL, Verilog HDL, EDIF. Source files can be created using any text editor.

**elaboration**
The process of mapping a text-based design to technology-independent library cells. Elaboration is part of the Create Implementation process.

During elaboration, logic is inferred from the design source code logic (for example, an if-then-else statement is translated into an AND-OR network, a “+” is translated into an addition operator). Only designs in files that are analyzed can be elaborated.

See also *analyze*.

**ending group of a timing path**
The ending group of a path can be the set of all primary outputs of the design, all edge-sensitive sequential elements clocked by a specified periodic signal, or all level-sensitive sequential elements clocked by a specified periodic signal.

See also *starting group*.

**flip-flop**
An edge-triggered device controlled by periodic signals (clocks). Flip-flops synchronize the propagation of data signals.

See also *clock*. 
**global buffer**
Global buffers drive clocks or high fanout nets to make a design faster, minimize clock skew, or make the routing task easier. Some buffers can drive all nets, others can only drive clock nets. Buffers are architecture-dependent. Global buffers are usually a limited resource.

Global buffers are available only to designs that use Actel or Xilinx technologies and that do not have the Do Not Insert I/O Pads option selected.

See also *global buffer insertion*.

**global buffer insertion**
The process of inserting global buffers into the netlist to minimize clock skew and make designs faster and easier to route. Global buffer insertion takes into account architecture-dependent issues such as the number of buffers available for a particular architecture and the ability of a global buffer to drive nonclock signals. Global buffer insertion is part of the optimization process.

See also *global buffer*.

**global set/reset (GSR)**
A signal that asynchronously sets or resets all of the sequential elements in a design.

See also *sequential elements*.

**global set/reset (GSR) inferencing**
A built-in optimization that automatically detects the presence of global set/reset (GSR) signals in your design. The signal is marked with a target technology-specific GSR marker block. Place and route tools use this marker to identify the GSR signal and route it using dedicated routing.

Automatic GSR signal detection and dedicated routing assignment help improve the performance of your design.
hand-instantiated pad
A primitive pad cell that is instantiated by the designer in the netlist or HDL description is defined as a hand-instantiated pad cell.

HDL library
A collection of designs that are stored in the tool.

hierarchy
A method for describing and modeling an electronic system using different abstraction levels. The bottom level of the hierarchy is the physical layout of the design (a concrete level, not at all abstract). The top of the hierarchy is a functional description of the system or a block diagram (a very high level of abstraction). Intermediate levels include the register transfer level (RTL), the gate level, and the transistor level.

implementation
An implementation consists of analyzed source files which FPGA Compiler II uses to build the design hierarchy and interconnections.

input delay
The input delay of an input or inout port is the maximum delay from that port to a timing group.

inout port
A port whose net is driven by a three-state driver and loads internal logic or other ports in the design.

input register delay
The setup time for registers mapped to input pads or driven directly by input ports.

The shell command for specifying the use of an input delay inside an input register during optimization is set_pad_reg_delay.
**instantiation**
Instantiation is when you place a symbol that represents a primitive or a macro in your design or netlist.

See also *macro* or *primitive*.

**latch**
An enabled device usually controlled by a periodic signal. Latches allow propagation of data signals during a specific time interval.
Latches are also called *level-sensitive devices*.

**macro**
A macro is a component made of nets and primitives, flip-flops, or latches that implements high-level functions, such as adders, subtracters, and dividers. Soft macros and Relationally Placed Macros (RPMs) are types of macros.

**module**
A module is part of an implementation hierarchy and is the container for its submodules. A module’s label is the name of its top-level (root) design. The top-level design of an implementation is associated with the top level design of a hierarchy.

See also *module boundary* and *top-level design*.

**module boundary**
A module has a boundary defined by its ports. Module boundaries can be preserved or eliminated during optimization. You can use the Preserve and Eliminate settings in the Hierarchy column of the Modules constraint table to control whether boundaries are preserved during optimization.

**multicycle path**
A design might have path groups that are longer than a single cycle. To make a path group multicycle, change the default path delay computed by FPGA to the new delay representing the multicycle behavior. For example, if a path group is clocked at both ends by 20
ns clocks and its default path delay computed from the clock waveforms is 10 ns, changing the default delay from 10 ns to 30 ns will add one more cycle to this path group.

**netlist files**
FPGA netlist files are text files that contain the design netlist description. These files are in EDIF format.

**output delay**
The output delay of an output or inout port is the maximum delay from a timing group to that port.

**pad constraints**
When you specify pad constraints, you describe the type of pad cells that the pad-mapping optimization inserts at the ports of the top-level design. You can specify these pad characteristics: port pad type, resistance, input register delay, I/O registers, input voltage, and slew rate.

See also *input register delay*.

**pad mapping**
The optimization step that creates the I/O pads at each port in the top-level design. This optimization is controlled by the pad constraints and by the global pad controls you set when you create implementations.

See also *pad constraints* and *top-level design*.

**path delay**
Path delay for a path group is the maximum delay allowed from any point in the starting timing group to any point in the ending timing group.

**Note:**
To protect the integrity of encrypted IP cores, the Paths constraint table is disabled for implementations that include these cores.
**path group**
A path group is the set of combinational paths from one timing group (called the *starting group*) to another timing group (called the *ending group*).

**place and route**
A layout task that positions major functional blocks or electronic components within an IC or on a PC board (place) and the subsequent routing of appropriate electrical connections to those components (route).

**point-to-point timing**
A timing point is defined as either the input or output of a register or a primary I/O of the design.

**primitive**
A basic logic element, such as a gate (AND, OR, XOR, NAND, or NOR), inverter, flip-flop, or latch. A logic element that directly corresponds, or maps, to one of these basic elements in an optimized netlists. For information about the primitives in a specific architecture, see the documentation for the technology library.

You can also instantiate primitives in the HDL description of a design by using named association.

See also *instantiation*.

**project**
Is a directory (folder) created by the tool. This working directory holds a project file with an .exp extension and subdirectories created by the tool for internal and intermediate use. You do not need to look at the contents of the subdirectories; they are automatically maintained.

See also *project file*.
**project file**
The project file with an .exp extension. This file contains all the information that the tool needs to reopen an existing project.

See also *project*.

**project state**
The project state is defined by the status of the design source files and the status of the implementation. (See Source file status icons and the Implementation status icons in the online Help).

**resistor pad**
A pull-up or pull-down cell that causes the logic value of a three-state net to be either logic 1 or logic 0 when the net's three-state drivers are all driving a logic z or high impedance.

**register transfer level (RTL)**
A register-level description of a digital electronic circuit (see *hierarchy*). Registers store intermediate information between clock cycles in a digital circuit. An RTL description describes what intermediate information is stored, where it is stored within the design, and how that information moves through the design as it operates.

**sequential elements**
Flip-flops and latches are collectively referred to as sequential elements.

See also *flip-flop* and *latch*.

**slack**
Slack is the margin by which a delay requirement is met. Positive slack means that the requirement is met; negative slack means that the requirement is not met.
starting group of a timing path
The starting group of a path can be the set of all primary inputs of the design, all edge-sensitive sequential elements clocked by a specified periodic signal, or all level-sensitive sequential elements clocked by a specified periodic signal.

See also ending group of a timing path.

three-state output port
A port whose net is driven by a three-state driver and does not load other logic or ports in the design.

timing group
A timing group is a collection of sequential elements and ports in the top-level design that have common timing behavior.

These are timing groups:

• All input and inout ports belong to the INPUT group

• All output and inout ports belong to the OUTPUT group

• All flip-flops clocked by the same edge of a common clock belong to a group

• All latches enabled by the same value of a common clock belong to a group

• All flip-flops or latches enabled by the same value of a common signal belong to a group

A timing group is a central concept, because all timing-constraint specifications use timing groups as basic entities.
**timing diagram**
A graphical representation of the signals in an electronic circuit that shows how the signals change over time in relationship to each other.

**top-level design**
An HDL design can be partitioned into a set of smaller subdesigns. This partitioning produces a hierarchy of designs. The top-level design refers to the design at the top of this hierarchy.

See also *design source files*. 
Index

A
active edge, defined GL-1
adding design source files 2-3
analysis order, defined GL-1
analyze, defined GL-2

B
back-annotation 1-8, GL-2
benefits 1-3
bidirectional GL-2
black boxes C-2
boundary, module, defined GL-6
buffer, global, defined GL-4
buffer, insertion of global buffers, defined GL-4

C
chip, current, setting A-11
clock frequency, setting 3-12
clock, defined GL-2
clocks constraint table 4-2
command 5-4
commands
  create_project 6-5
  current_chip 5-6, 6-5, A-2
  export_chip B-8
  export_constraint 5-6
  export_netlist 5-6
  import_constraint A-13
  set_chip_retiming B-7
  set_clock 5-8
  set_module_block 6-5
  set_pad_reg_delay GL-5
  translate_dc_script 5-6
  update_proj 6-7
constraint file
  ASCII, exporting A-11
  ASCII, importing A-12
  error checking A-13
  subpath caution note A-7
  subpaths A-6
constraint tables 4-2
  how to open 3-15
constraints
  entering 3-14
  pad, defined GL-7
create_project command 6-5
creating a new project 3-3
current chip 5-6
current chip, setting A-11
current project 5-6
current_chip command 5-6, 6-5
D

databases, project 3-24
dc_shell scripting 1-3
debugging GL-2
debugging source files 3-8
defined GL-6
  primitive GL-8
delay
  input register delay, defined GL-5
  output, defined GL-7
  path, defined GL-7
delay extraction 4-2
delay, defined GL-3
delay, input, defined GL-5
Design Compiler scripts 5-6
design constraints
  entering 3-14
design flow 1-2
  debugging source files 3-8
  DesignWizard 2-2
  HDL-based 1-5
  synthesizing a design 3-10
design implementation 3-11
design methodology (see design flow)
design source files
  adding libraries of 2-3
  adding to project 3-6
  defined GL-3
  updating from project 3-10
design, top-level, defined GL-11
DesignWizard, using 2-2

E

edge, active, defined GL-1
elaboration
  defined GL-3
element, sequential, defined GL-9
ending group of a timing path, defined GL-3
error checking, constraint file, when importing A-13
errors and warnings 3-8
export_chip command B-8
export_constraint command 5-6
export_netlist command 5-6
exporting
  constraints A-1

F

files
  design source files, defined GL-3
  netlist files, defined GL-7
  project, defined GL-9
flip-flop, defined GL-3
FPGA A-1
FPGA scripting tool (FST) 5-4
FST, FPGA scripting tool 5-4

G

generating
  project databases 3-24
generating netlists 3-23
generating reports 3-25
genmen utility 8-2
global buffer insertion, defined GL-4
global buffer, defined GL-4
global set/reset (GSR) inferencing, defined GL-4
global set/reset (GSR), defined GL-4
group, path, defined GL-8
group, timing, defined GL-10
GSR (global set/reset) inferencing, defined GL-4
GSR (global set/reset), defined GL-4
H
hand-instantiated pad, defined GL-5
HDL design methodology, migrating to 1-4
HDL editor 3-8
HDL library, defined GL-5
HDL modules C-2
help
  Synopsys Technical Support Center xix
hierarchy GL-5

I
identifying top-level design 3-10
implementation GL-5
  current chip, setting the A-11
design, creating 3-11
import_constraint command A-13
importing
  constraints A-1
inferencing, global set/reset (GSR), defined
  GL-4
inout port, defined GL-5
input delay, defined GL-5
input register delay, defined GL-5
insertion of global buffers, defined GL-4
instantiation GL-6
instantiation, defined GL-6

K
known problems xiv

L
latch, defined GL-6
libraries of design source files 2-3
library, HDL, defined GL-5
LogiCORE 8-10

M
mapping, pad, defined GL-7
maximum clock speed 2-5
memory generator 8-2
methodology, using DesignWizard 2-2
mixing HDL and netlist inputs C-1
module boundary, defined GL-6
module, defined GL-6
modules constraint table 4-4
multicycle path, defined GL-6

N
Netlist Module C-2
netlists
  files, defined GL-7
  generating 3-23

O
optimization viewing results 4-2
output delay, defined GL-7
output port, three-state, defined GL-10
Output window
  information in 3-8

P
pad constraints, defined GL-7
pad mapping, defined GL-7
pad, hand-instantiated, defined GL-5
pad, resistor, defined GL-9
path
  delay, defined GL-7
  multicycle, defined GL-6
path group, defined GL-8
path, timing
  ending group, defined GL-3
  starting group, defined GL-10
paths constraint table 4-2, 4-8
place and route 1-8, 3-14, GL-8
point-to-point timing, defined GL-8
port, inout, defined GL-5
port, three-state output port, defined GL-10
ports constraint table 4-3
primitive, defined GL-8
project
  defined GL-8
  file, defined GL-9
  state, defined GL-9
  variables 5-7
project variables 5-7
project window 3-6

R
register delay, input, defined GL-5
register retiming B-5
register transfer level (RTL) GL-9
Release Note, viewing xiv
reports, generating 3-25
resistor pad, defined GL-9
RTL (register transfer level) GL-9

S
schematic viewer 1-3
scripts 1-3
  creating 5-2
dc_shell 1-3
  using 5-3
sequential element defined GL-9
set_chip_retiming command B-7
set_clock command 5-8
set_module_block command 6-5
set_pad_reg_delay command GL-5
setting clock frequency 3-12
simulation 1-5, 1-8, 3-23, 3-24
slack, defined GL-9
source files
debugging 3-8
source files, design source files defined GL-3
starting group of a timing path, defined GL-10
state, project, defined GL-9
subpath
  cautionary note A-7
  specifying in constraint file A-6
Support Center, Synopsys Technical xix
synthesizing a design 3-10
T
three-state output port, defined GL-10
timing analysis 4-2
timing diagram GL-11
timing group, defined GL-10
timing path
   ending group, defined GL-3
   starting group, defined GL-10
timing, point-to-point, defined GL-8
top-level design
   defined GL-11
   identifying 3-10
top-level design, identifying 3-10
translate_dc_script command 5-6

U
update_proj command 6-7
updating a project 6-7
updating source files 3-10

V
variables 5-7
viewing errors and warnings 3-8
viewing results of optimization 4-2
viewing schematics
   mapped 4-7
   optimized 4-7
   RTL or generic 4-6
viewing timing results 4-8