Synopsys Synplify Pro for Lattice

User Guide

December 2016

SYNOPSYS®

Copyright Notice and Proprietary Information

© 2016 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.

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.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at
http://www.synopsys.com/Company/Pages/Trademarks.aspx.
All other product or company names may be trademarks of their respective owners.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.
690 East Middlefield Road
Mountain View, CA 94043
www.synopsys.com

December 2016

© 2016 Synopsys, Inc.

Contents

Chapter 1: Introduction

Synopsys FPGA and Prototyping Products 16
- FPGA Implementation Tools 16
- Synopsys FPGA Tool Features 17

Scope of the Document 21
- The Document Set 21
- Audience 21

Getting Started 22
- Starting the Software 22
- Getting Help 22

User Interface Overview 24

Chapter 2: FPGA Synthesis Design Flows

Logic Synthesis Design Flow 28

Hierarchical Project Management Flows 30
- Block-First Development Flow for Hierarchical Projects 31
- Top-First Development Flow for Hierarchical Projects 32
- Bottom-Up Synthesis for Hierarchical Projects 34
- Top-Down Synthesis for Hierarchical Projects 36
- Mixed Block Synthesis for Hierarchical Projects 38
- Hierarchical Project Management Limitations 39

Chapter 3: Preparing the Input

Setting Up HDL Source Files 42
- Creating HDL Source Files 42
- Using the Context Help Editor 44
- Checking HDL Source Files 45
- Editing HDL Source Files with the Built-in Text Editor 46
- Setting Editing Window Preferences 49
- Using an External Text Editor 51

© 2016 Synopsys, Inc.

Using Library Extensions for Verilog Library Files 53

Using Mixed Language Source Files 56

Working with Constraint Files 61
When to Use Constraint Files over Source Code 61
Tcl Syntax Guidelines for Constraint Files 61
Checking Constraint Files 63
Using a Text Editor for Constraint Files (Legacy) 63
Generating Constraint Files for Forward Annotation (Legacy) 64

Chapter 4: Setting Up a Logic Synthesis Project

Setting Up Project Files 68
Creating a Project File 68
Opening an Existing Project File 72
Making Changes to a Project 72
Setting Project View Display Preferences 73
Updating Verilog Include Paths in Older Project Files 75

Managing Project File Hierarchy 77
Creating Custom Folders 77
Manipulating Custom Project Folders 80
Manipulating Custom Files 81

Setting Up Implementations 83

Setting Logic Synthesis Implementation Options 85
Setting Device Options 85
Setting Optimization Options 87
Specifying Global Frequency and Constraint Files 89
Specifying Result Options 91
Specifying Timing Report Output 93
Setting Verilog and VHDL Options 93

Specifying Attributes and Directives 99
Specifying Attributes and Directives in VHDL 100
Specifying Attributes and Directives in Verilog 101
Specifying Attributes Using the SCOPE Editor 102
Specifying Attributes in the Constraints File 106
Handling Properties with Attributes or Directives 106

Working with Hierarchical Projects 108
Block-Based Subprojects 108
Creating Hierarchical Subprojects by Exporting Blocks 108
Creating Nested Subprojects 111
Working with Multiple Implementations in Hierarchical Projects 117

Working with Multiple Instances and Parameterized Modules 118
Configuring Synthesis Runs for Hierarchical Projects 121
Analyzing Synthesis Results for Hierarchical Projects 125

Searching Files 127
Identifying the Files to Search 128
Filtering the Files to Search 128
Initiating the Search 129
Search Results 129

Archiving Files and Projects 130
Archive a Project 130
Un-Archive a Project 134
Copy a Project 137
Support for Hierarchical Include Paths 140

Chapter 5: Specifying Constraints

Using the SCOPE Editor 144
Creating Constraints in the SCOPE Editor 144
Creating Constraints With the FDC Template Command 149

Specifying SCOPE Constraints 151
Entering and Editing SCOPE Constraints 151
Setting Clock and Path Constraints 153
Defining Input and Output Constraints 155
Specifying Standard I/O Pad Types 156
Using the TCL View of SCOPE GUI 157
Guidelines for Entering and Editing Constraints 159

Specifying Timing Exceptions 162
Defining From/To/Through Points for Timing Exceptions 162
Defining Multicycle Paths 166
Defining False Paths 167

Finding Objects with Tcl find and expand 168
Specifying Search Patterns for Tcl find 168
Refining Tcl Find Results with -filter 170
Using the Tcl Find Command to Define Collections 172
Using the Tcl expand Command to Define Collections 173
Checking Tcl find and expand Results 174
Using Tcl find and expand in Batch Mode 176

Using Collections 177
Creating and Using SCOPE Collections 178
Creating Collections using Tcl Commands 180

© 2016 Synopsys, Inc.

Viewing and Manipulating Collections with Tcl Commands 183

Converting SDC to FDC 187

Using the SCOPE Editor (Legacy) 188
Entering and Editing SCOPE Constraints (Legacy) 190
Specifying SCOPE Timing Constraints (Legacy) 191
Defining Input and Output Constraints (Legacy) 203
Defining False Paths (Legacy) 204

Chapter 6: Synthesizing and Analyzing the Results

Synthesizing Your Design 208
Running Logic Synthesis 208
Using Up-to-date Checking for Job Management 208

Checking Log File Results 213
Viewing and Working with the Log File 213
Accessing Specific Reports Quickly 217
Accessing Results Remotely 219
Analyzing Results Using the Log File Reports 223
Using the Watch Window 224
Checking Resource Usage 225

Handling Messages 227
Checking Results in the Message Viewer 227
Filtering Messages in the Message Viewer 229
Filtering Messages from the Command Line 231
Automating Message Filtering with a Tcl Script 232
Log File Message Controls 234
Handling Warnings 237
Working with Downgradable Errors and Critical Warnings 237

Chapter 7: Analyzing with HDL Analyst

Working in the Schematic (Beta) 242
Opening the Views 243
Cloning Schematics 245
Viewing Object Properties 246
Selecting Objects in the Schematic 248
Grouping Objects in the Schematic 249
Moving Between Views in a Schematic Window 254
Setting Schematic Preferences 254

Exploring Design Hierarchy (Beta) 255
Traversing Design Hierarchy with the Hierarchy Browser 255

Exploring Object Hierarchy with Push/Pop Commands 257

Finding Objects (Beta) 262
Browsing to Find Objects in HDL Analyst Views 262
Using Wildcards with the Find Command 264

Crossprobing (Beta) 265
Crossprobing within a View 265
Crossprobing from an HDL Analyst View 266
Crossprobing from the Text Editor Window 268

Analyzing With the HDL Analyst Tool (Beta) 270
Viewing Design Hierarchy and Context 270
Filtering Schematics 274
Expanding Pin and Net Logic 276
Dissolving and Partial Dissolving of Buses and Pins 282
Flattening Schematic Hierarchy 283
Using the FSM Viewer 286

Working in the Standard Schematic 291
Differentiating Between the HDL Analyst Views 292
Opening the Views 292
Viewing Object Properties 294
Selecting Objects in the RTL/Technology Views 299
Working with Multisheet Schematics 300
Moving Between Views in a Schematic Window 302
Setting Schematic Preferences 303
Managing Windows 304

Exploring Design Hierarchy 306
Traversing Design Hierarchy with the Hierarchy Browser 306
Exploring Object Hierarchy by Pushing/Popping 307
Exploring Object Hierarchy of Transparent Instances 313

Finding Objects 314
Browsing to Find Objects in HDL Analyst Views 314
Using Find for Hierarchical and Restricted Searches 316
Using Wildcards with the Find Command 319
Combining Find with Filtering to Refine Searches 323
Using Find to Search the Output Netlist 324

Crossprobing 327
Crossprobing within an RTL/Technology View 327
Crossprobing from the RTL/Technology View 328
Crossprobing from the Text Editor Window 330
Crossprobing from the Tcl Script Window 333

© 2016 Synopsys, Inc.

Crossprobing from the FSM Viewer 333

Analyzing With the HDL Analyst Tool 335
Viewing Design Hierarchy and Context 336
Filtering Schematics 339
Expanding Pin and Net Logic 341
Expanding and Viewing Connections 345
Flattening Schematic Hierarchy 346
Minimizing Memory Usage While Analyzing Designs 351

Using the FSM Viewer 352

Chapter 8: Analyzing Timing

Analyzing Timing in Schematic Views 358
Viewing Timing Information 358
Annotating Timing Information in the Schematic Views 359
Analyzing Clock Trees in the RTL View 361
Viewing Critical Paths 361
Handling Negative Slack 364

Generating Custom Timing Reports with STA 365

Using Analysis Design Constraints 368
Scenarios for Using Analysis Design Constraints 369
Creating an ADC File 370
Using Object Names Correctly in the adc File 374

Using Auto Constraints 375
Results of Auto Constraints 377

Chapter 9: Inferring High-Level Objects

Defining Black Boxes for Synthesis 382
Instantiating Black Boxes and I/Os in Verilog 382
Instantiating Black Boxes and I/Os in VHDL 384
Adding Black Box Timing Constraints 386
Adding Other Black Box Attributes 390

Defining State Machines for Synthesis 391
Defining State Machines in Verilog 391
Defining State Machines in VHDL 392
Specifying FSMs with Attributes and Directives 393

Specifying Safe FSMs 396
Implementing Safe Case FSMs 396

Automatic RAM Inference 398

Block RAM 398
RAM Attributes 399
Inferring Block RAM 401

Initializing RAMs 407
Initializing RAMs in Verilog 407
Initializing RAMs in VHDL 408

Chapter 10: Specifying Design-Level Optimizations

Tips for Optimization 414
General Optimization Tips 414
Optimizing for Area 415
Optimizing for Timing 416

Pipelining 418
Prerequisites for Pipelining 418
Pipelining the Design 418

Retiming 422
Controlling Retiming 422
Retiming Example 424
Retiming Report 425
How Retiming Works 426

Preserving Objects from Being Optimized Away 429
Using syn_keep for Preservation or Replication 430
Controlling Hierarchy Flattening 432
Preserving Hierarchy 433

Optimizing Fanout 434
Setting Fanout Limits 434
Controlling Buffering and Replication 436

Sharing Resources 438

Inserting I/Os 439

Optimizing State Machines 441
Deciding when to Optimize State Machines 441
Running the FSM Compiler 442

Inserting Probes 446
Specifying Probes in the Source Code 446
Adding Probe Attributes Interactively 447

© 2016 Synopsys, Inc.

Chapter 11: Working with Compile Points

Compile Point Basics 450
Advantages of Compile Point Design 450
Manual Compile Points 452
Nested Compile Points 453
Compile Point Types 453

Compile Point Synthesis Basics 458
Compile Point Constraint Files 458
Interface Logic Models 460
Interface Timing for Compile Points 461
Compile Point Synthesis 464
Incremental Compile Point Synthesis 465
Forward-annotation of Compile Point Timing Constraints 466

Synthesizing Compile Points 468
The Manual Compile Point Flow 468
Creating a Top-Level Constraints File for Compile Points 471
Defining Manual Compile Points 472
Setting Constraints at the Compile Point Level 475
Analyzing Compile Point Results 477

Resynthesizing Incrementally 479
Resynthesizing Compile Points Incrementally 479

Chapter 12: Working with IP Input

Generating IP with SYNCore 484
Specifying FIFOs with SYNCore 484
Specifying RAMs with SYNCore 490
Specifying Byte-Enable RAMs with SYNCore 497
Specifying ROMs with SYNCore 504
Specifying Adder/Subtractors with SYNCore 509
Specifying Counters with SYNCore 516

The Synopsys FPGA IP Encryption Flow 524
Overview of the Synopsys FPGA IP Flow 524
Encryption and Decryption 525

Working with Encrypted IP 529
Encrypting Your IP 529
Encrypting IP with the encryptP1735.pl Script 531
Encrypting IP with the encryptIP Script 536
Specifying the Script Output Method 537
Preparing the IP Package 540

Using Hyper Source 543
Using Hyper Source for Prototyping 543
Using Hyper Source for IP Designs 543
Threading Signals Through the Design Hierarchy of an IP 544

Working with Lattice IP 548

Chapter 13: Optimizing Processes for Productivity

Using Batch Mode 550
Running Batch Mode on a Project File 550
Running Batch Mode with a Tcl Script 552
Queuing Licenses 553

Working with Tcl Scripts and Commands 556
Using Tcl Commands and Scripts 556
Generating a Job Script 557
Setting Number of Parallel Jobs 557
Creating a Tcl Synthesis Script 558
Using Tcl Variables to Try Different Clock Frequencies 560
Using Tcl Variables to Try Several Target Technologies 561
Running Bottom-up Synthesis with a Script 562

Automating Flows with synhooks.tcl 563

Chapter 14: Improving Runtime

Multiprocessing With Compile Points 568
Setting Maximum Parallel Jobs 568
Specifying Licenses for Multiprocessing 569

Chapter 15: Clock Conversion

Working with Gated Clocks 574
Obstacles to Conversion 576
Prerequisites for Gated Clock Conversion 577
Defining Clocks Properly 579
Synthesizing a Gated-Clock Design 584
Accessing the Clock Conversion Report 585
Analyzing the Clock Conversion Report 586
Interpreting Gated Clock Error Messages 588
Disabling Individual Gated Clock Conversions 592
Integrated Clock Gating Cells 592
Using Gated Clocks for Black Boxes 597
OR Gates Driving Latches 598
Obstructions to Optimization 599

© 2016 Synopsys, Inc.

Unsupported Constructs 600
Other Potential Workarounds to Solve Clock-Conversion Issues 601
Restrictions on Using Gated Clocks 601

Optimizing Generated Clocks 604
Enabling Generated-Clock Optimization 604
Conditions for Generated-Clock Optimization 604
Generated-Clock Optimization Examples 605

Chapter 16: Optimizing for Lattice Designs

Optimizing Lattice Designs 610
Instantiating Lattice Macros 610
Using Lattice GSR Resources 612
Inferring Lattice PIC Latches 612
Controlling I/O Insertion in Lattice Designs 620
Forward-Annotating Lattice Constraints 621

Optimizing Lattice iCE40 Designs 623
Using Sequential Logic 623
Using Combinational Logic 624
Handling Tristates 624
Handling I/Os and Buffers 625

Chapter 17: Working with Synthesis Output

Passing Information to the P&R Tools 628
Specifying Pin Locations 628

Generating Vendor-Specific Output 629
Targeting Output to Your Vendor 629

Invoking Third-Party Vendor Tools 630
Configuring Tool Tags 630
Invoking a Third-Party Tool 631

Chapter 18: Running Post-Synthesis Operations

Running P&R Automatically after Synthesis 636
Integrating Synthesis and Place-and-Route in One Run 636
Releasing the Synthesis License During Place and Route 636

Simulating with the VCS Tool 638

CHAPTER 1

Introduction

This introduction to the Synplify Pro® tool describes the following:

- Synopsys FPGA and Prototyping Products, on page 16
- Scope of the Document, on page 21
- Getting Started, on page 22
- User Interface Overview, on page 24

© 2016 Synopsys, Inc.

Synopsys FPGA and Prototyping Products

The following figure displays the Synopsys FPGA and Prototyping family of products.

FPGA Implementation Tools

The Synplify Pro and Synplify Premier products are RTL synthesis tools especially designed for FPGAs (field programmable gate arrays) and CPLDs (complex programmable logic devices).

Synplify Pro Synthesis Software

The Synplify Pro FPGA synthesis software is the de facto industry standard for producing high-performance, cost-effective FPGA designs. Its unique Behavior Extracting Synthesis Technology® (B.E.S.T.™) algorithms, perform high-level optimizations before synthesizing the RTL code into specific FPGA logic. This approach allows for superior optimizations across the FPGA, fast runtimes, and the ability to handle very large designs. The Synplify Pro software supports the latest VHDL and Verilog language constructs, including SystemVerilog and VHDL 2008. The tool is technology independent allowing quick and easy retargeting between FPGA devices and vendors from a single design project.

Synplify Premier Synthesis Software

The Synplify Premier functionality is a superset of the Synplify Pro tool, providing the ultimate FPGA implementation and debug environment. It includes a comprehensive suite of tools and technologies for advanced FPGA designers, and also serves as the synthesis engine for ASIC prototypers targeting single FPGA-based prototypes.

The Synplify Premier product offers both FPGA designers and ASIC prototypers targeting single FPGAs with the most efficient method of design implementation and debug. On the design implementation side, it includes functionality for timing closure, logic verification, IP usage, ASIC compatibility, and DSP implementation, as well as a tight integration with FPGA vendor back-end tools. On the debug side, it provides for in-system verification of FPGAs which dramatically accelerates the debug process, and also includes a rapid and incremental method for finding elusive design problems.

Synopsys FPGA Tool Features

This table distinguishes between major functionality in the Synplify Pro, Synplify, Synplify Premier, and Synplify Premier with Design Planner products.

© 2016 Synopsys, Inc.

	Synplify	Synplify Pro	Synplify Premier	Synplify Premier DP
Performance				
Behavior Extracting Synthesis Technology® (BEST™)	x	x	x	x
Vendor-Generated Core/IP Support (certain technologies)		x	x	x
FSM Compiler	x	x	x	x
FSM Explorer		x	x	x
Gated Clock Conversion		x	x	x
Register Pipelining		x	x	x
Register Retiming		x	x	x
SCOPE® Constraint Entry	x	x	x	x
High Reliability Features		x	x	x
Integrated Place-and-Route	x	x	x	x
Analysis				
HDL Analyst®	Option	x	x	x
Timing Analyzer – Point-to-point		x	x	x
Timing Report View			x	x
FSM Viewer		x	x	x
Crossprobing		x	x	x
Probe Point Creation		x	x	x
Identify® Instrumentor		x	x	x
Identify Debugger			x	x
Power Analysis (SAIF)			x	x
Physical Design				
Design Plan File				x

	Synplify	Synplify Pro	Synplify Premier	Synplify Premier DP
Logic Assignment to Regions				x
Area Estimation and Region Capacity				x
Pin Assignment				x
Physical Optimizations				x
Physical Analyst			x	x
Synopsys DesignWare® Foundation Library			x	x
Runtime				
Hierarchical Design		x	x	x
Advanced Synthesis			x	x
Fast Synthesis			x	x
Multiprocessing			x	x
Compile on Error			x	x
Team Design				
Mixed Language Design		x	x	x
Compile Points		x	x	x
Hierarchical Design		x	x	x
True Batch Mode (Floating licenses only)		x	x	x
GUI Batch Mode (Floating licenses)	x	x	x	x
Batch Mode P&R	-	x	x	x
Back-annotation of P&R Data	-	-	-	x
Formal Verification		x	Logic synthesis mode	Logic synthesis mode
Identify Integration	Limited	x	x	x

© 2016 Synopsys, Inc.

	Synplify	Synplify Pro	Synplify Premier	Synplify Premier DP
Back-annotation of P&R Data				x
Design Environment				
Text Editor View	x	x	x	x
Watch Window		x	x	x
Message Window		x	x	x
Tcl Window		x	x	x
Multiple Implementations		x	x	x
Vendor Technology Support	x	x	Selected	Selected
Prototyping Features				
Runtime features			x	x
Compile Points		x	x	x
Gated Clock Conversion			x	x
Compile on Error			x	x

Scope of the Document

The following explain the scope of this document and the intended audience.

The Document Set

This user guide is part of a document set that includes reference manuals. It is intended for use with the other documents in the set. It concentrates on describing how to use the Synopsys FPGA software to accomplish typical tasks. This implies the following:

- The user guide only explains the options needed to do the typical tasks described in the manual. It does not describe every available command and option. For complete descriptions of all the command options and syntax, refer to the Tcl Commands chapter in the *Synopsys FPGA Synthesis Command Reference Manual*.
- The user guide contains task-based information. For a breakdown of how information is organized, see *Getting Help*, on page 22.

Audience

The Synplify Pro software tool is targeted towards the FPGA system developer. It is assumed that you are knowledgeable about the following:

- Design synthesis
- RTL
- FPGAs
- Verilog/VHDL

© 2016 Synopsys, Inc.

Getting Started

This section shows you how to get started with the Synopsys FPGA synthesis software. It describes the following topics, but does not supersede the information in the installation instructions about licensing and installation:

- Starting the Software, on page 22
- Getting Help, on page 22

Starting the Software

1. If you have not already done so, install the Synopsys FPGA synthesis software according to the installation instructions.
2. Start the software.
   - If you are working on a Windows platform, select Programs->Synopsys->*product version* from the Start button.
   - If you are working on a UNIX platform, type the appropriate command at the command line:

     ```
     synplify_pro
     ```

   The command starts the synthesis tool, and opens the Project window. If you have run the software before, the window displays the previous project. For more information about the interface, see the User Interface Overview chapter of the *Reference Manual*.

Getting Help

Before you call Synopsys Support, look through the documented information. You can access the information online from the Help menu, or refer to the PDF version. The following table shows you how the information is organized.

For help with...	Refer to the...
Using software features	*Synopsys FPGA Synthesis User Guide*
How to...	*Synopsys FPGA Synthesis User Guide*, application notes on the support web site
Flow information	*Synopsys FPGA Synthesis User Guide*, application notes on the support web site
Error messages	Online help (select Help->Error Messages)
Licensing	Synopsys Website
Attributes and directives	*Synopsys FPGA Synthesis Attribute Reference Manual*
Synthesis features	*Synopsys FPGA Synthesis Reference Manual*
Language and syntax	*Synopsys FPGA Synthesis Reference Manual*
Tcl syntax	Online help (select Help->Tcl Help)
Tcl synthesis commands	*Synopsys FPGA Synthesis Command Reference Manual*
Product updates	*Synopsys FPGA Synthesis Reference Manual* (Web menu commands)

© 2016 Synopsys, Inc.

User Interface Overview

The user interface (UI) consists of a main window, called the Project view, and specialized windows or views for different tasks. For details about each of the features, see Chapter 2, *User Interface Overview* of the *Synopsys FPGA Synthesis Reference Manual*.

Synplify Pro Interface

© 2016 Synopsys, Inc.

CHAPTER 2

FPGA Synthesis Design Flows

This describes the following tool flows:

- Logic Synthesis Design Flow, on page 28
- Hierarchical Project Management Flows, on page 30

© 2016 Synopsys, Inc.

Logic Synthesis Design Flow

The Synopsys FPGA tools synthesize logic by first compiling the RTL source into technology-independent logic structures, and then optimizing and mapping the logic to technology-specific resources. After logic synthesis, the tool generates a vendor-specific netlist and constraint file that you can use as inputs to the place-and-route (P&R) tool.

The following figure shows the phases and the tools used for logic synthesis and some of the major inputs and outputs. The interactive timing analysis is optional. Although the flow shows the vendor constraint files as direct inputs to the P&R tool, you should add these files to the synthesis project for timing black boxes.

Logic Synthesis Procedure

The following steps summarize the procedure for synthesizing the design, which is also illustrated in the figure that follows.

1. Create a project.
2. Add the source files to the project.
3. Set attributes and constraints for the design.

4. Set options for the implementation in the Implementation Options dialog box.
5. Click Run to run logic synthesis.
6. Analyze the results, using tools like the log file, the HDL Analyst schematic views, the Message window and the Watch Window.

   After you have completed the design, you can use the output files to run place-and-route with the vendor tool and implement the FPGA.

The following figure lists the main steps in the flow:

© 2016 Synopsys, Inc.

Hierarchical Project Management Flows

Some Lattice iCE40 Technologies

As designs grow in size and complexity, the industry uses team design and parallel development techniques to ensure that designs are finished on time. Typically, designs are split into smaller subprojects or blocks, and different teams work on different blocks.

The team design approach could be either top-first or block-first. The hierarchical project management capabilities (HPM) in the FPGA synthesis tools facilitate both the top-first or block-first team design approaches. See the following for details:

- Block-First Development Flow for Hierarchical Projects, on page 31
- Top-First Development Flow for Hierarchical Projects, on page 32
- Top-Down Synthesis for Hierarchical Projects, on page 36
- Bottom-Up Synthesis for Hierarchical Projects, on page 34
- Mixed Block Synthesis for Hierarchical Projects, on page 38

Hierarchical Project Flows and Compile Points

The compile point and the hierarchical project management flows are both modular and support the team design approach, but the focus is slightly different. Compile point flows are block-based, but hierarchical project management flows focus on managing the entirety of the design. Hierarchical project management can use compile points to implement a hierarchical design.

- Hierarchical project management
  Hierarchical project management consists of various features and methodologies that help you develop and automatically manage a single FPGA project using multiple teams across different geographical areas. These flows are intended for distributed design development, and for parallel development of portions of the design. You create partitions at the RTL level, but you do not have to floorplan them. The hierarchical project management flows are modular, and can use compile points for its block-level components. Hierarchical project management includes features like a GUI that supports hierarchical project management and module import and export, top-first, block-first and mixed development flows, and both top-down and bottom-up synthesis flows (see the list in

Hierarchical Project Management Flows, on page 30 for information on these flows).

- Compile points
  Compile points implement block-based flows, based on RTL partitions that are defined prior to synthesis. Compile points are often used to implement incremental team design changes. They can also be used to reduce runtime. For manual compile points, you manually define these compile points and synthesize them in logic synthesis flows.

Block-First Development Flow for Hierarchical Projects

In this design approach, blocks are developed independently and then stitched together at the top level. The advantage to this approach is independence, parallel development, and fewer design constraints at the team level. In the following example, different teams work on each block and the blocks are then integrated into the top level.

1. Create independent projects from blocks. For each block, do the following:
   - Add files.
   - Synthesize the block and check that results meet block-level goals. Iterate as needed.
2. Create the top-level project.
   - Add top-level source files, with black boxes defined for subproject blocks.
   - Compile the top level to create a hierarchical view.
3. Create and link subproject to the top level.

© 2016 Synopsys, Inc.

- To create a subproject, select an instance or design block from the Design Hierarchy view. Then, right-click and select Create Subproject (Design Block) respectively.
- To link the subproject to the top-level design, right-click and select Insert & Link Subproject to Design Block.

4. To experiment with different settings, use multiple implementations at the subproject level or at the top level.

   For details, see *Working with Multiple Implementations in Hierarchical Projects*, on page 117.

5. Synthesize the top level.
   - For top-down synthesis, the tool runs the design flat. For details about running top-down synthesis, see *Top-Down Synthesis for Hierarchical Projects*, on page 36.
   - For bottom-up synthesis, the tool first synthesizes the blocks and generates output netlists for them. It then synthesizes the top level, based on the top-level RTL files and the netlist files from the subproject blocks.For details about running bottom-up synthesis, see *Bottom-Up Synthesis for Hierarchical Projects*, on page 34.
   - If your design contains a combination of top-down and bottom-up blocks, the tool first compiles the top-down blocks, then works on the bottom-up blocks, and finally synthesizes the top level. See *Mixed Block Synthesis for Hierarchical Projects*, on page 38 for details.

Top-First Development Flow for Hierarchical Projects

In this flow, you start with the top-level design and then divide it up into subprojects (blocks). The block RTL is included in the top-level project. The advantage to using this approach is that you can take advantage of boundary optimizations to improve QOR.

1. Create a top-level project.
   - Add project files. You can have black boxes for the subproject blocks.
   - Compile the design.
2. Open the hierarchical view and define the subprojects.
   - Define the subprojects. as described in See *Creating Hierarchical Subprojects by Exporting Blocks*, on page 108 for information about creating the subprojects.

     You must repeat the process for each subproject block you want to create. If your top level has multiple instances of the same block with different parameter values, you must create subprojects for each instance as you need separate output netlists for each instance at the top level. You do not need to create subprojects for each instance if multiple instances have the same parameters or no parameters, unless you want them to have their own output netlist files.
   - Save the top-level design.
   - Send out the blocks you created to be developed independently.
3. Develop each block independently. The teams must do the following for each block they are working on:
   - Add source files.
   - Add constraints.
   - Synthesize the block and check that results meet block-level goals. Iterate as needed.
4. Optionally, create multiple implementations for your hierarchical project. You might want to do this if you want to experiment with different settings. See *Working with Multiple Implementations in Hierarchical Projects*, on page 117.
5. Synthesize the top level.
   - For top-down synthesis, the tool runs the design flat. For details about running top-down synthesis, see *Top-Down Synthesis for Hierarchical Projects*, on page 36.
   - For bottom-up synthesis, the tool first synthesizes the blocks and generates output netlists for them. It then synthesizes the top level, based on the top-level RTL files and the netlist files from the subproject blocks. For details about running bottom-up synthesis, see *Bottom-Up Synthesis for Hierarchical Projects*, on page 34.

© 2016 Synopsys, Inc.

- If your design contains a combination of top-down and bottom-up blocks, the tool first compiles the top-down blocks, then works on the bottom-up blocks and finally synthesizes the top level. See *Mixed Block Synthesis for Hierarchical Projects*, on page 38 for details.

Bottom-Up Synthesis for Hierarchical Projects

In this flow, you synthesize your design from the bottom up. This flow is independent of the approach used to develop the design; you can use bottom-up synthesis for a design developed with either the top-first or block-first design approach.

1. Open the top-level project.

   The top-level project can be developed from the top down or the bottom up (see *Top-First Development Flow for Hierarchical Projects*, on page 32 and *Block-First Development Flow for Hierarchical Projects*, on page 31 for details).

2. Check that the top-level entity or module is specified in the Implementation Options dialog box.

3. Configure the synthesis run parameters as described in *Configuring Synthesis Runs for Hierarchical Projects*, on page 121.
   - Select Hierarchical Project Options and specify which block implementation to run.
   - Set Run Type for the block to bottom_up.

- If necessary, synchronize block options with the top level with the Sync all Options to Top and Syn Required Options buttons. If the block options do not match the top-level settings, you could get errors.
- Click OK.

4. If the subproject teams are not working in the default directories, do one of the following before running synthesis:
   - Use source control management and update the block projects before running top-level synthesis.
   - Copy the block output netlist and constraint files to the subproject directories.
5. Synthesize the top-level design. Click Run to synthesize the top-level design.

   By default, the tool synthesizes the active implementations of the top level and blocks. If you want to run all implementations, select Run->Run all Implementations instead of clicking Run.

© 2016 Synopsys, Inc.

For the bottom-up flow, the tool does the following:

- It automatically uses the latest output netlists for the subprojects that were marked bottom-up when you configured settings for the run. It needed, it resynthesizes the blocks and generates the output netlists for them.
- It then compiles the top level, based on the top-level RTL files and the netlist files from the subproject blocks.
- Finally, it maps the top level based on the compiled `srs` for the top level.

6. Analyze the results using the log file and the RTL view, as described in *Analyzing Synthesis Results for Hierarchical Projects*, on page 125.

Top-Down Synthesis for Hierarchical Projects

In this flow, you synthesize the design from the top down. Note that this flow is independent of the approach used to develop the design; you can use top-down synthesis for a design developed with either the top-first or block-first approach.

1. Open the top-level project.

   The top-level project can be developed from the top down or the bottom up (see *Top-First Development Flow for Hierarchical Projects*, on page 32 and *Block-First Development Flow for Hierarchical Projects*, on page 31 for details).

2. Check that the top-level entity or module is specified in the Implementation Options dialog box.

3. Configure the synthesis run parameters as described in *Configuring Synthesis Runs for Hierarchical Projects*, on page 121.
   - Select Hierarchical Project Options and specify which block implementation to run.
   - Set Run Type for the block to top_down.
   - If necessary, synchronize block options with the top level with the Sync all Options to Top and Syn Required Options buttons. If the block options do not match the top-level settings, you could get errors.
   - Click OK.

4. Click Run to synthesize the top-level design.

   By default, the tool synthesizes the active implementations of the top level and blocks. If you want to run all implementations, select Run->Run all Implementations instead of clicking Run.

   For the top-down flow, the tool runs the design flat. It compiles the block RTL and constraint files along with the top-level files, as if they were part of the top-level project.

5. Analyze the results using the log file and the RTL view, as described in *Analyzing Synthesis Results for Hierarchical Projects*, on page 125.

© 2016 Synopsys, Inc.

Mixed Block Synthesis for Hierarchical Projects

In this flow, you have some blocks that need to be synthesized from the top down and others that must be synthesized from the bottom up. Note that this flow refers to the synthesis process and is independent of the approach used to develop the design.

1. Open the top-level project.

   The top-level project and blocks can be developed from the top down or the bottom up (see *Top-First Development Flow for Hierarchical Projects*, on page 32 and *Block-First Development Flow for Hierarchical Projects*, on page 31 for details).

2. Check that the top-level entity or module is specified in the Implementation Options dialog box.

3. Configure the synthesis run parameters as described in *Configuring Synthesis Runs for Hierarchical Projects*, on page 121.

	Top: camera_open_source	Sub: camera_wb_if	Sub: camera_io_calc
Implementation:	rev_1	rev	rev_1
Run Type:		top_down	bottom_up
Options:			
Technology	LatticeiCE40	LatticeiCE40	LatticeiCE40
Technology part	iCE40LP1K	iCE40LP1K	iCE40LP1K
Technology package	CM121	CM121	CM121
Fanout Guide	10000	10000	10000
Disable I/O Insertion	☑	☑	☑
Pipelining	☐	☐	☐
Retiming	☐	☐	☐
Fix Gated Clocks	0	0	0
Fix Generated Clocks	0	0	0
Auto Constrain IO	☐	☐	☐

   – Select Hierarchical Project Options and specify the block implementations to run.

   – Set Run Type for each block appropriately. In a mixed design, some blocks will be top_down and others bottom_up.

- If necessary, synchronize block options with the top level with the Sync all Options to Top and Syn Required Options buttons. If the block options do not match the top-level settings, you could get errors. Click OK.

4. Click Run to synthesize the top-level design.

   By default, the tool synthesizes the active implementations of the top level and blocks. If you want to run all implementations, select Run->Run all Implementations instead of clicking Run.

   The synthesis tool does the following to synthesize a hierarchical design with mixed blocks:

   - The tool compiles the top-down blocks to generate `srs` files for them.
   - It automatically picks up the latest output netlists for the bottom-up blocks. If necessary, it synthesizes these blocks and generates output netlists for them.
   - It then compiles the top level, using the top-level RTL files, the `srs` files for the top-down blocks, and the netlist files for the bottom-up blocks.
   - Finally, it maps the top level based on the compiled top level results (`srs`) from the previous phase.

5. Analyze the results using the log file and the RTL view, as described in *Analyzing Synthesis Results for Hierarchical Projects*, on page 125.

Hierarchical Project Management Limitations

Note the following hierarchical project management (HPM) limitations.

Secure Core Instantiation in Multiple Subprojects

HPM does not support the instantiation of the same secure core in multiple subprojects of the hierarchical bottom-up synthesis flow for supported devices.

© 2016 Synopsys, Inc.

CHAPTER 3

Preparing the Input

When you synthesize a design, you need to set up two kinds of files: HDL files that describe your design, and project files to manage the design. This chapter describes the procedures to set up these files and the project. It covers the following:

- Setting Up HDL Source Files, on page 42
- Using Mixed Language Source Files, on page 56
- Working with Constraint Files, on page 61

© 2016 Synopsys, Inc.

Setting Up HDL Source Files

This section describes how to set up your source files; project file setup is described in *Setting Up Project Files*, on page 68. Source files can be in Verilog or VHDL. For information about structuring the files for synthesis, refer to the *Reference Manual*. This section discusses the following topics:

- Creating HDL Source Files, on page 42
- Using the Context Help Editor, on page 44
- Checking HDL Source Files, on page 45
- Editing HDL Source Files with the Built-in Text Editor, on page 46
- Setting Editing Window Preferences, on page 49
- Using an External Text Editor, on page 51
- Using Library Extensions for Verilog Library Files, on page 53

Creating HDL Source Files

This section describes how to use the built-in text editor to create source files, but does not go into details of what the files contain. For details of what you can and cannot include, as well as vendor-specific information, see the *Reference Manual*. If you already have source files, you can use the text editor to check the syntax or edit the file (see *Checking HDL Source Files*, on page 45 and *Editing HDL Source Files with the Built-in Text Editor*, on page 46).

You can use Verilog or VHDL for your source files. The files have `v` (Verilog) or `vhd` (VHDL) file extensions, respectively. You can use Verilog and VHDL files in the same design. For information about using a mixture of Verilog and VHDL input files, see *Using Mixed Language Source Files*, on page 56.

1. To create a new source file either click the HDL file icon ( ) or do the following:
   - Select File->New or press Ctrl-n.
   - In the New dialog box, select the kind of source file you want to create, Verilog or VHDL. Note that you can use the Context Help Editor for designs that contain Verilog, SystemVerilog, or VHDL constructs in the source file. For more information, see *Using the Context Help Editor*, on page 44.

If you are using Verilog 2001 format or SystemVerilog, make sure to enable the Verilog 2001 or System Verilog option before you run synthesis (Project->Implementation Options->Verilog tab). The default Verilog file format for new projects is SystemVerilog.

– Type a name and location for the file and Click OK. A blank editing window opens with line numbers on the left.

2. Type the source information in the window, or cut and paste it. See *Editing HDL Source Files with the Built-in Text Editor*, on page 46 for more information on working in the Editing window.

   For the best synthesis results, check the *Reference Manual* and ensure that you are using the available constructs and vendor-specific attributes and directives effectively.

3. Save the file by selecting File->Save or the Save icon ( ).

Once you have created a source file, you can check that you have the right syntax, as described in *Checking HDL Source Files*, on page 45.

© 2016 Synopsys, Inc.

Using the Context Help Editor

When you create or open a design file, use the Context Help button displayed at the bottom of the window to help you code with Verilog/SystemVerilog/VHDL constructs in the source file or Tcl constraint commands into your Tcl file.

To use the Context Help Editor:

1. Click on the Context Help button to display this text editor.

2. When you select a construct in the left-side of the window, the online help description for the construct is displayed. If the selected construct has this feature enabled, the online help topic is displayed on the top of the window and a generic code or command template for that construct is displayed at the bottom.

3. The Insert Template button is also enabled. When you click the Insert Template button, the code or command shown in the template window is

inserted into your file at the location of the cursor. This allows you to easily insert the code or command and modify it for the design that you are going to synthesize.

4. If you want to copy only parts of the template, select the code or command you want to insert and click Copy. You can then paste it into your file.

Checking HDL Source Files

The software automatically checks your HDL source files when it compiles them, but if you want to check your source code before synthesis, use the following procedure. There are two kinds of checks you do in the synthesis software: syntax and synthesis.

1. Select the source files you want to check.
   - To check all the source files in a project, deselect all files in the project list, and make sure that none of the files are open in an active window. If you have an active source file, the software only checks the active file.
   - To check a single file, open the file with File->Open or double-click the file in the Project window. If you have more than one file open and want to check only one of them, put your cursor in the appropriate file window to make sure that it is the active window.
2. To check the syntax, select Run->Syntax Check or press Shift+F7.

   The software detects syntax errors such as incorrect keywords and punctuation and reports any errors in a separate log file (syntax.log). If no errors are detected, a successful syntax check is reported at the bottom of this file.
3. To run a synthesis check, select Run->Synthesis Check or press Shift+F8.

   The software detects hardware-related errors such as incorrectly coded flip-flops and reports any errors in a separate log file (syntax.log). If there are no errors, a successful syntax check is reported at the bottom of this file.
4. Review the errors by opening the syntax.log file when prompted and use Find to locate the error message (search for @E). Double-click on the 5-character error code or click on the message text and push F1 to display online error message help.

© 2016 Synopsys, Inc.

5. Locate the portion of code responsible for the error by double-clicking on the message text in the syntax.log file. The Text Editor window opens the appropriate source file and highlights the code that caused the error.

6. Repeat steps 4 and 5 until all syntax and synthesis errors are corrected.

Messages can be categorized as errors, warnings, or notes. Review all messages and resolve any errors. Warnings are less serious than errors, but you must read through and understand them even if you do not resolve all of them. Notes are informative and do not need to be resolved.

Editing HDL Source Files with the Built-in Text Editor

The built-in text editor makes it easy to create your HDL source code, view it, or edit it when you need to fix errors. If you want to use an external text editor, see *Using an External Text Editor*, on page 51.

1. Do one of the following to open a source file for viewing or editing:
   - To automatically open the first file in the list with errors, press F5.
   - To open a specific file, double-click the file in the Project window or use File->Open (Ctrl-o) and specify the source file.

   The Text Editor window opens and displays the source file. Lines are numbered. Keywords are in blue, and comments in green. String values are in red. If you want to change these colors, see *Setting Editing Window Preferences*, on page 49.

2. To edit a file, type directly in the window.

This table summarizes common editing operations you might use. You can also use the keyboard shortcuts instead of the commands.

To ...	Do ...
Cut, copy, and paste; undo, or redo an action	Select the command from the popup (hold down the right mouse button) or Edit menu.
Go to a specific line	Press Ctrl-g or select Edit->Go To, type the line number, and click OK.
Find text	Press Ctrl-f or select Edit ->Find. Type the text you want to find, and click OK.
Replace text	Press Ctrl-h or select Edit->Replace. Type the text you want to find, and the text you want to replace it with. Click OK.
Complete a keyword	Type enough characters to uniquely identify the keyword, and press Esc.
Indent text to the right	Select the block, and press Tab.
Indent text to the left	Select the block, and press Shift-Tab.
Change to upper case	Select the text, and then select Edit->Advanced ->Uppercase or press Ctrl-Shift-u.
Change to lower case	Select the text, and then select Edit->Advanced ->Lowercase or press Ctrl-u.
Add block comments	Put the cursor at the beginning of the comment text, and select Edit->Advanced->Comment Code or press Alt-c.
Edit columns	Press Alt, and use the left mouse button to select the column. On some platforms, you have to use the key to which the Alt functionality is mapped, like the Meta or diamond key.

3. To cut and paste a section of a PDF document, select the T-shaped Text Select icon, highlight the text you need and copy and paste it into your file. The Text Select icon lets you select parts of the document.

4. To create and work with bookmarks in your file, see the following table.

   Bookmarks are a convenient way to navigate long files or to jump to points in the code that you refer to often. You can use the icons in the

© 2016 Synopsys, Inc.

Edit toolbar for these operations. If you cannot see the Edit toolbar on the far right of your window, resize some of the other toolbars.

To ...	Do ...
Insert a bookmark	Click anywhere in the line you want to bookmark. Select Edit->Toggle Bookmarks, press Ctrl-F2, or select the first icon in the Edit toolbar. The line number is highlighted to indicate that there is a bookmark at the beginning of that line.
Delete a bookmark	Click anywhere in the line with the bookmark. Select Edit->Toggle Bookmarks, press Ctrl-F2, or select the first icon in the Edit toolbar. The line number is no longer highlighted after the bookmark is deleted.
Delete all bookmarks	Select Edit->Delete all Bookmarks, press Ctrl-Shift-F2, or select the last icon in the Edit toolbar. The line numbers are no longer highlighted after the bookmarks are deleted.
Navigate a file using bookmarks	Use the Next Bookmark (F2) and Previous Bookmark (Shift-F2) commands from the Edit menu or the corresponding icons from the Edit toolbar to navigate to the bookmark you want.

5. To fix errors or review warnings in the source code, do the following:
   - Open the HDL file with the error or warning by double-clicking the file in the project list.
   - Press F5 to go to the first error, warning, or note in the file. At the bottom of the Editing window, you see the message text.
   - To go to the next error, warning, or note, select Run->Next Error/Warning or press F5. If there are no more messages in the file, you see the message “No More Errors/Warnings/Notes” at the bottom of the Editing window. Select Run->Next Error/Warning or press F5 to go to the the error, warning, or note in the next file.
   - To navigate back to a previous error, warning, or note, select Run->Previous Error/Warning or press Shift-F5.

6. To bring up error message help for a full description of the error, warning, or note:
   - Open the text-format log file (click View Log) and either double click on the 5-character error code or click on the message text and press F1.
   - Open the HTML log file and click on the 5-character error code.
   - In the Tcl window, click the Messages tab and click on the 5-character error code in the ID column.
7. To cross probe from the source code window to other views, open the view and select the piece of code. See *Crossprobing from the Text Editor Window*, on page 330 for details.
8. When you have fixed all the errors, select File->Save or click the Save icon to save the file.

Setting Editing Window Preferences

You can customize the fonts and colors used in a Text Editing window.

1. Select Options->Editor Options and either Synopsys Editor or External Editor. For more information about the external editor, see *Using an External Text Editor*, on page 51.
2. Then depending on the type of file you open, you can to set the background, syntax coloring, and font preferences to use with the text editor.

Note: Thereafter, text editing preferences you set for this file will apply to all files of this file type.

The Text Editing window can be used to set preferences for project files, source files (Verilog/VHDL), log files, Tcl files, constraint files, or other default files from the Editor Options dialog box.

3. You can set syntax colors for some common syntax options, such as keywords, strings, and comments. For example in the log file, warnings and errors can be color-coded for easy recognition.

   Click in the Foreground or Background field for the corresponding object in the Syntax Coloring field to display the color palette.

© 2016 Synopsys, Inc.

You can select basic colors or define custom colors and add them to your custom color palette. To select your desired color click OK.

4. To set font and font size for the text editor, use the pull-down menus.
5. Check Keep Tabs to enable tab settings, then set the tab spacing using the up or down arrow for Tab Size.

6. Click OK on the Editor Options form.

Using an External Text Editor

You can use an external text editor like vi or emacs instead of the built-in text editor. Do the following to enable an external text editor. For information about using the built-in text editor, see *Editing HDL Source Files with the Built-in Text Editor*, on page 46.

1. Select Options->Editor Options and turn on the External Editor option.
2. Select the external editor, using the method appropriate to your operating system.
   - If you are working on a Windows platform, click the ... (Browse) button and select the external text editor executable.
   - From a UNIX or Linux platform for a text editor that creates its own window, click the ... Browse button and select the external text editor executable.
   - From a UNIX platform for a text editor that does not create its own window, do not use the ... Browse button. Instead type xterm -e *editor*. The following figure shows VI specified as the external editor.

© 2016 Synopsys, Inc.

– From a Linux platform, for a text editor that does not create its own window, do not use the ... Browse button. Instead, type gnome-terminal -x *editor*. To use emacs for example, type gnome-terminal -x emacs.

The software has been tested with the emacs and vi text editors.

3. Click OK.

Using Library Extensions for Verilog Library Files

Library extensions can be added to Verilog library files included in your design for the project. When you provide search paths to the directories that contain the Verilog library files, you can specify these new library extensions as well as the Verilog and SystemVerilog (`.v` and `.sv`) file extensions.

To do this:

1. Select the Verilog tab of the Implementation Options panel.
2. Specify the locations of the Library Directories for the Verilog library files to be included in your design for the project.
3. Specify the Library Extensions.

   Any library extensions can be specified, such as `.av`, `.bv`, `.cv`, `.xxx`, `.va`, `.vas` (separate library extensions with a space).

   The following figure shows you where to enter the library extensions on the dialog box.

© 2016 Synopsys, Inc.

The Tcl equivalent for this example is the following command:

```
set_option -libext .av .bv .cv .dv .ev
```

For details, see libext, on page 72 in the *Command Reference*.

4. After you compile the design, you can verify in the log file that the library files with these extensions were loaded and read. For example:

```
@N: Running Verilog Compiler in SystemVerilog mode
@I::"C:\dir\top.v"
@N: CG1180 :"C:\dir\top.v":8:0:8:3|Loading file
C:\dir\lib1\sub1.av from specified library directory
C:\dir\lib1
@I::"C:\dir\lib1\sub1.av"
@N: CG1180 :"C:\dir\top.v":10:0:10:3|Loading file
C:\dir\lib2\sub2.bv from specified library directory
C:\dir\lib2
@I::"C:\dir\lib2\sub2.bv"
@N: CG1180 :"C:\dir\top.v":12:0:12:3|Loading file
C:\dir\lib3\sub3.cv from specified library directory
```

```
C:\dir\lib3
@I::"C:\dir\lib3\sub3.cv"
@N: CG1180 :"C:\dir\top.v":14:0:14:3|Loading file
C:\dir\lib4\sub4.dv from specified library directory
C:\dir\lib4
@I::"C:\dir\lib4\sub4.dv"
@N: CG1180 :"C:\dir\top.v":16:0:16:3|Loading file
C:\dir\lib5\sub5.ev from specified library directory
C:\dir\lib5
@I::"C:\dir\lib5\sub5.ev"
Verilog syntax check successful!
```


© 2016 Synopsys, Inc.

Using Mixed Language Source Files

With the synthesis software, you can use a mixture of VHDL and Verilog input files in your project. For examples of the VHDL and Verilog files, see the *Reference Manual*. You cannot use Verilog and VHDL files together in the same design with the Synplify tool.

1. Remember that Verilog does not support unconstrained VHDL ports and set up the mixed language design files accordingly.

2. If you want to organize the Verilog and VHDL files in different folders, select Options->Project View Options and toggle on the View Project Files in Folders option.

   When you add the files to the project, the Verilog and VHDL files are in separate folders in the Project view.

3. When you open a project or create a new one, add the Verilog and VHDL files as follows:
   - Select the Project->Add Source File command or click the Add File button.
   - On the form, set Files of Type to HDL Files (*.vhd, *.vhdl, *.v).
   - Select the Verilog and VHDL files you want and add them to your project. Click OK. For details about adding files to a project, see *Making Changes to a Project*, on page 72.

The files you added are displayed in the Project view. This figure shows the files arranged in separate folders.

4. When you set device options (Implementation Options button), specify the top-level module. For more information about setting device options, see *Setting Logic Synthesis Implementation Options*, on page 85.
   - If the top-level module is Verilog, click the Verilog tab and type the name of the top-level module.
   - If the top-level module is VHDL, click the VHDL tab and type the name of the top-level entity. If the top-level module is not located in the default work library, you must specify the library where the compiler can find the module. For information on how to do this, see VHDL Panel, on page 227.

© 2016 Synopsys, Inc.

You must explicitly specify the top-level module, because it is the starting point from which the mapper generates a merged netlist.

5. Select the Implementation Results tab on the same form and select one output HDL format for the output files generated by the software. For more information about setting device options, see *Setting Logic Synthesis Implementation Options*, on page 85.
   - For a Verilog output netlist, select Write Verilog Netlist.
   - For a VHDL output netlist, select Write VHDL Netlist.
   - Set any other device options and click OK.

   You can now synthesize your design. The software reads in the mixed formats of the source files and generates a single `srs` file that is used for synthesis.

6. If you run into problems, see *Troubleshooting Mixed Language Designs*, on page 59 for additional information and tips.

Troubleshooting Mixed Language Designs

This section provides tips on handling specific situations that might come up with mixed language designs.

VHDL File Order

For VHDL-only designs or mixed designs where the top level is not specified, the FPGA synthesis tools automatically re-arrange the VHDL files so that the VHDL packages are compiled in the correct order.

However, if you have a mixed-language design where you have specified the top level, you must specify the VHDL file order for the tool. You only need to do this once, by selecting the Run->Arrange VHDL files command. If you do not do this, you get an error message

VHDL Global Signals

Currently, you cannot have VHDL global signals in mixed language designs, because the tool only implements these signals in VHDL-only designs.

Passing VHDL Boolean Generics to Verilog Parameters

The tool infers a black box for a VHDL component with Boolean generics, if that component is instantiated in a Verilog design. This is because Verilog does not recognize Boolean data types, so the Boolean value must be represented correctly. If the value of the VHDL Boolean generic is TRUE and the Verilog literal is represented by a 1, the Verilog compiler interprets this as a black box.

To avoid inferring a black box, the Verilog literal for the VHDL Boolean generic set to TRUE must be 1'b1, not 1. Similarly, if the VHDL Boolean generic is FALSE, the corresponding Verilog literal must be 1'b0, not 0. The following example shows how to represent Boolean generics so that they correctly pass the VHDL-Verilog boundary, without inferring a black box.

VHDL Entity Declaration	Verilog Instantiation

`Entity abc is`
`Generic`
`(`
`  Number_Bits : integer := 0;`
`  Divide_Bit : boolean := False;`
`);`
	`abc #(`
	`  .Number_Bits (16),`
	`  .Divide_Bit (1'b0)`
	`)`

© 2016 Synopsys, Inc.

Passing VHDL Generics Without Inferring a Black Box

In the case where a Verilog component parameter, (for example [0:0] RSR = 1'b0) does not match the size of the corresponding VHDL component generic (RSR : integer := 0), the tool infers a black box.

You can work around this by removing the bus width notation of [0:0] in the Verilog files. You must use a VHDL generic of type integer because the other types do not allow for the proper binding of the Verilog component.

Working with Constraint Files

Constraint files are text files that are automatically generated by the SCOPE interface (see *Specifying SCOPE Constraints*, on page 151), or which you create manually with a text editor. They contain Tcl commands or attributes that constrain the synthesis run. Alternatively, you can set constraints in the source code, but this is not the preferred method.

This section contains information about

- When to Use Constraint Files over Source Code, on page 61
- Tcl Syntax Guidelines for Constraint Files, on page 61
- Checking Constraint Files, on page 63
- Using a Text Editor for Constraint Files (Legacy), on page 63
- Generating Constraint Files for Forward Annotation (Legacy), on page 64

When to Use Constraint Files over Source Code

You can add constraints in constraint files (generated by SCOPE interface or entered in a text editor) or in the source code. In general, it is better to use constraint files, because you do not have to recompile for the constraints to take effect. It also makes your source code more portable. See *Using the SCOPE Editor*, on page 144 for more information.

However, if you have black box timing constraints like syn_tco, syn_tpd, and syn_tsu, you must enter them as directives in the source code. Unlike attributes, directives can only be added to the source code, not to constraint files. See *Specifying Attributes and Directives*, on page 99 for more information on adding directives to source code.

Tcl Syntax Guidelines for Constraint Files

This section covers general guidelines for using Tcl for constraint files:

- Tcl is case-sensitive.
- For naming objects:
  - The object name must match the name in the HDL code.
  - Enclose instance and port names within curly braces {}.

© 2016 Synopsys, Inc.

- Do not use spaces in names.
- Use the dot (.) to separate hierarchical names.
- In Verilog modules, use the following syntax for instance, port, and net names:

  v:*cell*[*prefix*:]*objectName*

  Where *cell* is the name of the design entity, *prefix* is a prefix to identify objects with the same name, *objectName* is an instance path with the dot (.) separator. The prefix can be any of the following:

Prefix (Lower-case)	Object
`i:`	Instance names
`p:`	Port names (entire port)
`b:`	Bit slice of a port
`n:`	Net names

- In VHDL modules, use the following syntax for instance, port, and net names in VHDL modules:

  v:*cell*[.*view*] [*prefix*:]*objectName*

  Where `v:` identifies it as a view object, `lib` is the name of the library, *cell* is the name of the design entity, *view* is a name for the architecture, *prefix* is a prefix to identify objects with the same name, and *objectName* is an instance path with the dot (.) separator. *View* is only needed if there is more than one architecture for the design. See the table above for the prefixes of objects.

- Name matching wildcards are * (asterisk matches any number of characters) and ? (question mark matches a single character). These characters do not match dots used as hierarchy separators. For example, the following string identifies all bits of the statereg instance in the statemod module:

```
i:statemod.statereg[*]
```

Checking Constraint Files

You can check syntax and other pertinent information on your constraint files using the Constraint Check command. To generate a constraint report, do the following:

1. Create a constraint file and add it to your project.
2. Select Run->Constraint Check.

   This command generates a report that checks the syntax and applicability of the timing constraints in the FPGA synthesis constraint files for your project. The report is written to the *projectName*_cck.rpt file and lists the following information:

   – Constraints that are not applied
   – Constraints that are valid and applicable to the design
   – Wildcard expansion on the constraints
   – Constraints on objects that do not exist

   For details on this report, see Constraint Checking Report, on page 281 of the *Reference Manual*.

Using a Text Editor for Constraint Files (Legacy)

You can use the Legacy SCOPE editor for the SDC constraint files created before release version G-2012.09. However, it is recommended that you translate your SDC files to FDC files to enable the latest version of the SCOPE editor and to utilize the enhanced timing constraint handling in the tool.

If you choose to use the legacy SCOPE editor, this section shows you how to manually create a Tcl constraint file. The software automatically creates this file if you use the legacy SCOPE editor to enter the constraints. The Tcl constraint file only contains general timing constraints. Black box constraints must be entered in the source code. For additional information, see *When to Use Constraint Files over Source Code*, on page 61.

1. Open a file for editing.
   – Make sure you have closed the SCOPE window, or you could overwrite previous constraints.
   – To create a new file, select File->New, and select the Constraints File (SCOPE) option. Type a name for the file and click OK.

© 2016 Synopsys, Inc.

– To edit an existing file, select File->Open, set the Files of Type filter to Constraint Files (sdc) and open the file you want.

2. Follow the syntax guidelines in *Tcl Syntax Guidelines for Constraint Files*, on page 61.

3. Enter the timing constraints you need. For the syntax, see the *Reference Manual*. If you have black box timing constraints, you must enter them in the source code.

4. You can also add vendor-specific attributes in the constraint file using define_attribute. See *Specifying Attributes in the Constraints File*, on page 106 for more information.

5. Save the file.

6. Add the file to the project as described in *Making Changes to a Project*, on page 72, and run synthesis.

Generating Constraint Files for Forward Annotation (Legacy)

The tool automatically generates vendor-specific constraint files that you can use for forward-annotation. The synthesis constraints are mapped to the appropriate vendor constraints.You can control this process with some attributes as described in the following procedure.

1. For some Lattice designs, set the -from and -to false path and multi-cycle constraints on the Others tab of the SCOPE interface.

   For details about these attributes, see the *Attribute Reference Manual*.

2. Select Project->Implementation Options, and check Write Vendor Constraints in the Implementation Results tab.

3. Click OK and run synthesis.

   The software converts the synthesis define_input_delay, define_output_delay, define_clock (including the define_clock constraints generated by auto constraining), define_multicycle_path, define_false_path, define_max_delay, and global-frequency constraints into corresponding commands in the $DESIGN_synplify.lpf file for Lattice.

Open the Lattice ispLEVER place-and-route tool, then import the $DESIGN_synplify.lpf file before running PAR. If a user-created `lpf` file already exists, ispLEVER backs it up into lpf.bak and copies the contents of $DESIGN_synplify.lpf into the $DESIGN.lpf file which is then used for all computations.

The software converts the following synthesis and `global frequency` constraints into corresponding commands in the `*.scf` file for Lattice iCE designs.

- define_clock
- define_input_delay
- define_output_delay
- define_false_path
- define_multicycle_path
- define_path_delay

See the *Reference Manual* for details about forward annotation.

© 2016 Synopsys, Inc.

CHAPTER 4

Setting Up a Logic Synthesis Project

When you synthesize a design with the Synopsys FPGA synthesis tool, you must set up a project for your design. The following describe the procedures for setting up a project for logic synthesis:

- Setting Up Project Files, on page 68
- Managing Project File Hierarchy, on page 77
- Setting Up Implementations, on page 83
- Setting Logic Synthesis Implementation Options, on page 85
- Specifying Attributes and Directives, on page 99
- Working with Hierarchical Projects, on page 108
- Searching Files, on page 127
- Archiving Files and Projects, on page 130

© 2016 Synopsys, Inc.

Setting Up Project Files

This section describes the basics of how to set up and manage a project file for your design, including the following information:

- Creating a Project File, on page 68
- Opening an Existing Project File, on page 72
- Making Changes to a Project, on page 72
- Setting Project View Display Preferences, on page 73
- Updating Verilog Include Paths in Older Project Files, on page 75

For information about working with hierarchical projects, refer to *Working with Hierarchical Projects*, on page 108.

Creating a Project File

You must set up a project file for each project. A project contains the data needed for a particular design: the list of source files, the synthesis results file, and your device option settings. The following procedure shows you how to set up a project file using individual commands.

1. Start by selecting one of the following: File->Build Project, File->Open Project, or the P icon. Click New Project.

   The Project window shows a new project. Click the Add File button, press F4, or select the Project->Add Source File command. The Add Files to Project dialog box opens.

2. Add the source files to the project.
   - Make sure the Look in field at the top of the form points to the right directory. The files are listed in the box. If you do not see the files, check that the Files of Type field is set to display the correct file type. If you have mixed input files, follow the procedure described in *Using Mixed Language Source Files*, on page 56.

– To add all the files in the directory at once, click the Add All button on the right side of the form. To add files individually, click on the file in the list and then click the Add button, or double-click the file name.

  You can add all the files in the directory and then remove the ones you do not need with the Remove button.

  If you are adding VHDL files, select the appropriate library from the VHDL Library popup menu. The library you select is applied to all VHDL files when you click OK in the dialog box.

Your project window displays a new project file. If you click on the plus sign next to the project and expand it, you see the following:

– A folder (two folders for mixed language designs) with the source files. If your files are not in a folder under the project directory, you can set this preference by selecting Options->Project View Options and checking the View project files in folders box. This separates one kind of file from another in the Project view by putting them in separate folders.

© 2016 Synopsys, Inc.

– The implementation, named rev_1 by default. Implementations are revisions of your design within the context of the synthesis software, and do not replace external source code control software and processes. Multiple implementations let you modify device and synthesis options to explore design options. Each implementation has its own synthesis and device options and its own project-related files.

3. Add any libraries you need, using the method described in the previous step to add the Verilog or VHDL library file.

   – For vendor-specific libraries, add the appropriate library file to the project. Note that for some families, the libraries are loaded automatically and you do not need to explicitly add them to the project file.

     To add a third-party VHDL package library, add the appropriate vhd file to the design, as described in step 2. Right click the file in the Project view and select File Options, or select Project-> Set VHDL library. Specify a library name that is compatible with the simulators. For example, MYLIB. Make sure that this package library is before the top-level design in the list of files in the Project view.

     For information about setting Verilog and VHDL file options, see *Setting Verilog and VHDL Options*, on page 93. You can also set these file options later, before running synthesis. For additional information about using vendor macro libraries and black boxes, see Chapter 16, *Optimizing for Lattice Designs*.

   – For generic technology components, you can either add the technology-independent Verilog library supplied with the software (*install_dir*/lib/generic_ technology/gtech.v) to your design, or add your

own generic component library. Do not use both together as there may be conflicts.

4. Check file order in the Project view. File order is especially important for VHDL files.
   - For VHDL files, you can automatically order the files by selecting Run->Arrange VHDL Files. Alternatively, manually move the files in the Project view. Package files must be first on the list because they are compiled before they are used. If you have design blocks spread over many files, make sure you have the following file order: the file containing the entity must be first, followed by the architecture file, and finally the file with the configuration.
   - In the Project view, check that the last file in the Project view is the top-level source file. Alternatively, you can specify the top-level file when you set the device options.
5. Select File->Save, type a name for the project, and click Save. The Project window reflects your changes.
6. To close a project file, select the Close Project button or File->Close Project.

© 2016 Synopsys, Inc.

Opening an Existing Project File

There are two ways to open a project file: the Open Project and the generic File ->Open command.

1. If the project you want to open is one you worked on recently, you can select it directly: File->Recent Projects-> *projectName*.
2. Use one of the following methods to open any project file:

Open Project **Command**	File->Open **Command**
Select File->Open Project, click the Open Project button on the left side of the Project window, or click the P icon. To open a recent project, double-click it from the list of recent projects. Otherwise, click the Existing Project button to open the Open dialog box and select the project.	Select File->Open. Specify the correct directory in the Look In: field. Set File of Type to Project Files (*.prj). The box lists the project files. Double-click on the project you want to open.

The project opens in the Project window.

Making Changes to a Project

Typically, you add, delete, or replace files.

1. To add source or constraint files to a project, select the Add Files button or Project->Add Source File to open the Select Files to Add to Project dialog box. See *Creating a Project File*, on page 68 for details.
2. To delete a file from a project, click the file in the Project window, and press the Delete key.
3. To replace a file in a project,
   - Select the file you want to change in the Project window.
   - Click the Change File button, or select Project->Change File.
   - In the Source File dialog box that opens, set Look In to the directory where the new file is located. The new file must be of the same type as the file you want to replace.

- If you do not see your file listed, select the type of file you need from the Files of Type field.
- Double-click the file. The new file replaces the old one in the project list.

4. To specify how project files are saved in the project, right click on a file in the Project view and select File Options. Set the Save File option to either Relative to Project or Absolute Path.

5. To check the time stamp on a file, right click on a file in the Project view and select File Options. Check the time that the file was last modified. Click OK.

Setting Project View Display Preferences

You can customize the organization and display of project files.

1. Select Options->Project View Options.

   The Project View Options form opens.

© 2016 Synopsys, Inc.

2. To organize different kinds of input files in separate folders, check View Project Files in Folders.

   Checking this option creates separate folders in the Project view for constraint files and source files.

3. Control file display with the following:
   - Automatically display all the files, by checking Show Project Library. If this is unchecked, the Project view does not display files until you click on the plus symbol and expand the files in a folder.
   - Check one of the boxes in the Project File Name Display section of the form to determine how filenames are displayed. You can display just the filename, the relative path, or the absolute path.

4. To view project files in customized custom folders, check View Project Files in Custom Folders. For more information, see *Creating Custom Folders*, on page 77. Type folders are only displayed if there are multiple types in a custom folder.

5. To open more than one implementation in the same Project view, check Allow Multiple Projects to be Opened.

6. Control the output file display with the following:
   - Check the Show all Files in Results Directory box to display all the output files generated after synthesis.
   - Change output file organization by clicking in one of the header bars in the Implementation Results view. You can group the files by type or sort them according to the date they were last modified.
7. To view file information, select the file in the Project view, right-click, and select File Options. For example, you can check the date a file was modified.

Updating Verilog Include Paths in Older Project Files

If you have a project file created with an older version of the software (prior to 8.1), the Verilog include paths in this file are relative to the results directory or the source file with the `include statements. In releases after 8.1, the project file `include paths are relative to the project file only. The GUI in the more

© 2016 Synopsys, Inc.

recent releases does not automatically upgrade the older PRJ files to conform to the newer rules. To upgrade and use the old project file, do one of the following:

- Manually edit the PRJ file in a text editor and add the following on the line before each set_option -include_path:

  ```
  set_option -project_relative_includes 1
  ```

- Start a new project with a newer version of the software and delete the old project. This will make the new PRJ file obey the new rule where includes are relative to the PRJ file.

Managing Project File Hierarchy

The following sections describe how you can create and manage customized folders and files in the Project view:

- Creating Custom Folders
- Manipulating Custom Project Folders
- Manipulating Custom Files

This information applies to file management; for information about managing and working with hierarchical projects, see *Working with Hierarchical Projects*, on page 108.

Creating Custom Folders

You can create logical folders and customize files in various hierarchy groupings within your Project view. These folders can be specified with any name or hierarchy level. For example, you can arbitrarily match your operating system file structure or HDL logic hierarchy. Custom folders are distinguished by their blue color.

There are several ways to create custom folders and then add files to them in a project. Use one of the following methods:

1. Right-click on a project file or another custom folder and select Add Folder from the popup menu. Then perform any of the following file operations:
   - Right-click on a file or files and select Place in Folder. A sub-menu displays so that you can either select an existing folder or create a new folder.

© 2016 Synopsys, Inc.

Note that you can arbitrarily name the folder, however do not use the character (/) because this is a hierarchy separator symbol.

– To rename a folder, right-click on the folder and select Rename from the popup menu. The Rename Folder dialog box appears; specify a new name.

2. Use the Add Files to Project dialog box to add the entire contents of a folder hierarchy, and optionally place files into custom folders corresponding to the OS folder hierarchies listed in the dialog box display.

– To do this, select the Add File button in the Project view.

– Select any requested folders such as dsp from the dialog box, then click the Add button. This places all the files from the dsp hierarchy into the custom folder you just created.

- To automatically place the files into custom folders corresponding to the OS folder hierarchy, check the option called Add Files to Custom Folders on the dialog box.
- By default, the custom folder name is the same name as the folder containing files or folder to be added to the project. However, you can modify how folders are named, by clicking on the Folders Option button. The following dialog box is displayed.

To use:

- Only the folder containing files for the folder name, click on Use OS Folder Name.
- The path name to the selected folder to determine the level of hierarchy reflected for the custom folder path.

3. You can drag and drop files and folders from an OS Explorer application into the Project view. This feature is available on Windows and Linux desktops running KDE.
   - When you drag and drop a file, it is immediately added to the project. If no project is open, the software creates a project.

© 2016 Synopsys, Inc.

- When you drag and drop a file over a folder, it will be placed in that folder. Initially, the Add Files to Project dialog box is displayed asking you to confirm the files to be added to the project. You can click OK to accept the files. If you want to make changes, you can click the Remove All button and specify a new filter or option.

Note: To display custom folders in the Project view, select the Options->Project View Options menu, then enable/disable the check box for View Project Files in Custom Folders on the dialog box.

Manipulating Custom Project Folders

The following procedure describes how you can remove files from folders, delete folders, and change the folder hierarchy.

1. To remove a file from a custom folder, either:
   - Drag and drop it into another folder or onto the project.
   - Highlight the file, right-click and select Remove from Folder from the popup menu.

     Do not use the Delete (DEL) key, as this removes the file from the project.
2. To delete a custom folder, highlight it then right-click and select Delete from the popup menu or press the DEL key. When you delete a folder, make one of the following choices:
   - Click Yes to delete the folder and the files contained in the folder from the project.
   - Click No to just delete the folder.
3. To change the hierarchy of the custom folder:
   - Drag and drop the folder within another folder so that it is a sub-folder or over the project to move it to the top-level.
   - To remove the top-level hierarchy of a custom folder, drag and drop the desired sub-level of hierarchy over the project. Then delete the empty root directory for the folder.

For example, if the existing custom folder directory is:

`/Examples/Verilog/RTL`

Suppose you want a single-level RTL hierarchy only, then drag and drop `RTL` over the project. Thereafter, you can delete the `/Examples/Verilog` directory.

Manipulating Custom Files

Additionally, you can perform the following types of custom file operations:

1. To suppress the display of files in the Type folders, right-click in the Project view and select Project View Options or select Options->Project View Options. Disable the option View Project Files in Type Folders on the dialog box.

2. To display files in alphabetical order instead of project order, check the Sort Files button in the Project view control panel. Click the down arrow key in the bottom-left corner of the panel to toggle the control panel on and off.

© 2016 Synopsys, Inc.

3. To change the order of files in the project:
   - Make sure to disable custom folders and sorting files.
   - Drag and drop a file to the desired position in the list of files.
4. To change the file type, drag and drop it to the new type folder. The software will prompt you for verification.

Setting Up Implementations

An implementation is a version of a project, implemented with a specific set of constraints and other settings. A project can contain multiple implementations, each one with its own settings.

Working with Multiple Implementations

The synthesis tools let you create multiple implementations of the same design and then compare results. This lets you experiment with different settings for the same design. Implementations are revisions of your design within the context of the synthesis software, and do not replace external source code control software and processes.

1. Click the Add Implementation button or select Project->New Implementation and set new device options (Device tab), new options (Options tab), or a new constraint file (Constraints tab).

   The software creates another implementation in the project view. The new implementation has the same name as the previous one, but with a different number suffix. The following figure shows two implementations, rev1 and rev2, with the current (active) implementation highlighted.

   The new implementation uses the same source code files, but different device options and constraints. It copies some files from the previous implementation: the `tlg` log file, the `srs` RTL netlist file. The software keeps a repeatable history of the synthesis runs.

2. Run synthesis again with the new settings.
   - To run the current implementation only, click Run.
   - To run all the implementations in a project, select Run->Run All Implementations.

   You can use multiple implementations to try a different part or experiment with a different frequency. See *Setting Logic Synthesis Implementation Options*, on page 85 for information about setting options.

© 2016 Synopsys, Inc.

The Project view shows all implementations with the active implementation highlighted and the corresponding output files generated for the active implementation displayed in the Implementation Results view on the right; changing the active implementation changes the output file display. The Watch window monitors the active implementation. If you configure this window to watch all implementations, the new implementation is automatically updated in the window.

3. Compare the results.
   - Use the Watch window to compare selected criteria. Make sure to set the implementations you want to compare with the Configure Watch command. See *Using the Watch Window*, on page 224 for details.

Log Parameter	rev_1	rev_2
eight_bit_uc|clock - Estimated Frequency	47.0 MHz	201.6 MHz
eight_bit_uc|clock - Requested Frequency	55.3 MHz	237.1 MHz
eight_bit_uc|clock - Slack	-3.191	-0.744

   - To compare details, compare the log file results.

4. To rename an implementation, click the right mouse button on the implementation name in the project view, select Change Implementation Name from the popup menu, and type a new name.

   Note that the current UI overwrites the implementation; releases prior to 9.0 preserve the implementation to be renamed.

5. To copy an implementation, click the right mouse button on the implementation name in the project view, select Copy Implementation from the popup menu, and type a new name for the copy.

6. To delete an implementation, click the right mouse button on the implementation name in the project view, and select Remove Implementation from the popup menu.

Setting Logic Synthesis Implementation Options

You can set global options for your synthesis implementations, some of them technology-specific. This section describes how to set global options like device, optimization, and file options with the Implementation Options command. For information about setting constraints for the implementation, see *Specifying SCOPE Constraints*, on page 151. For information about overriding global settings with individual attributes or directives, see *Specifying Attributes and Directives*, on page 99.

This section discusses the following topics:

- Setting Device Options, on page 85
- Setting Optimization Options, on page 87
- Specifying Global Frequency and Constraint Files, on page 89
- Specifying Result Options, on page 91
- Specifying Timing Report Output, on page 93
- Setting Verilog and VHDL Options, on page 93

Setting Device Options

Device options are part of the global options you can set for the synthesis run. They include the part selection (technology, part and speed grade) and implementation options (I/O insertion and fanouts). The options and the implementation of these options can vary from technology to technology, so check the vendor chapters of the *Reference Manual* for information about your vendor options.

1. Open the Implementation Options form by clicking the Implementation Options button or selecting Project->Implementation Options, and click the Device tab at the top if it is not already selected.

2. Select the technology, part, package, and speed. Available options vary, depending on the technology you choose.

© 2016 Synopsys, Inc.

3. Set the device mapping options. The options vary, depending on the technology you choose.
   - If you are unsure of what an option means, click on the option to see a description in the box below. For full descriptions of the options, click F1 or refer to the appropriate vendor chapter in the *Reference Manual*.
   - To set an option, type in the value or check the box to enable it.

   For more information about setting fanout limits, pipelining, and retiming, see *Setting Fanout Limits*, on page 434, *Pipelining*, on page 418, and *Retiming*, on page 422, respectively. For details about other vendor-specific options, refer to the appropriate vendor chapter and technology family in the *Reference Manual*.

Device Mapping Options

Option	Value
Fanout Guide	1000
Disable I/O Insertion	☐
Update Compile Point Timing Data	☐
Force GSR Usage	no
Read Write Check on RAM	☑
Resolve Mixed Drivers	☐

4. Set other implementation options as needed (see *Setting Logic Synthesis Implementation Options*, on page 85 for a list of choices). Click OK.

5. Click the Run button to synthesize the design. The software compiles and maps the design using the options you set.

6. To set device options with a script, use the set_option Tcl command. The following table contains an alphabetical list of the device options on the Device tab mapped to the equivalent Tcl commands. Because the options are technology- and family-based, all of the options listed in the table may not be available in the selected technology. All commands begin with set_option, followed by the syntax in the column as shown. Check the *Reference Manual* for the most comprehensive list of options for your vendor.

The following table shows a majority of the device options.

Option	Tcl Command (set_option ...)
Annotated Properties for Analyst This option is supported for Lattice iCE devices.	`-run_prop_extract {1|0}`
Fanout Guide	`-fanout_limit` *fanout_value*
Package	`-package` *pkg_name*
Part	`-part` *part_name*
Resolve Mixed Drivers	`-resolve_multiple_driver {1|0}`
Speed	`-speed_grade` *speed_grade*
Technology	`-technology` *keyword*
Update Compile Point Timing Data	`-update_models_cp {0|1}`

Setting Optimization Options

Optimization options are part of the global options you can set for the implementation. This section tells you how to set options like frequency and global optimization options like resource sharing. You can also set some of these options with the appropriate buttons on the UI.

1. Open the Implementation Options form by clicking the Implementation Options button or selecting Project->Implementation Options, and click the Options tab at the top.
2. Click the optimization options you want, either on the form or in the Project view. Your choices vary, depending on the technology. If an option is not available for your technology, it is greyed out. Setting the option in one place automatically updates it in the other.

© 2016 Synopsys, Inc.

Optimization Options

Project View

Implementation Options->Options

For details about using these optimizations refer to the following sections:

FSM Compiler	*Optimizing State Machines*, on page 441
Resource Sharing	*Sharing Resources*, on page 438
Pipelining	*Pipelining*, on page 418
Retiming	*Retiming*, on page 422

The equivalent Tcl set_option command options are as follows:

Option	set_option Tcl Command Option
FSM Compiler	`-symbolic_fsm_compiler {1|0}`
Resource Sharing	`-resource_sharing {1|0}`
Pipelining	`-pipe {0|1}`
Retiming	`-retiming {1|0}`

3. Set other implementation options as needed (see *Setting Logic Synthesis Implementation Options*, on page 85 for a list of choices). Click OK.
4. Click the Run button to run synthesis.

   The software compiles and maps the design using the options you set.

Specifying Global Frequency and Constraint Files

This procedure tells you how to set the global frequency and specify the constraint files for the implementation.

1. To set a global frequency, do one of the following:
   - Type a global frequency in the Project view.
   - Open the Implementation Options form by clicking the Implementation Options button or selecting Project->Implementation Options, and click the Constraints tab.

   The equivalent Tcl set_option command is -frequency *frequencyValue*.

   You can override the global frequency with local constraints, as described in *Specifying SCOPE Constraints*, on page 151. In the Synplify Pro tool, you can automatically generate clock constraints for your design instead of setting a global frequency. See *Using Auto Constraints*, on page 375 for details.

© 2016 Synopsys, Inc.

Global Frequency and Constraints

Project View

Implementation Options->Constraints

2. To specify constraint files for an implementation, do one of the following:
   - Select Project->Implementation Options->Constraints. Check the constraint files you want to use in the project.
   - From the Implementation Options->Constraints panel, you can also click to add a constraint file.
   - With the implementation you want to use selected, click Add File in the Project view, and add the constraint files you need.

To create constraint files, see *Specifying SCOPE Constraints*, on page 151.

3. To remove constraint files from an implementation, do one of the following:
   - Select Project->Implementation Options->Constraints. Click off the checkbox next to the file name.
   - In the Project view, right-click the constraint file to be removed and select Remove from Project.

   This removes the constraint file from the implementation, but does not delete it.

4. Set other implementation options as needed (see *Setting Logic Synthesis Implementation Options*, on page 85 for a list of choices). Click OK.

   When you synthesize the design, the software compiles and maps the design using the options you set.

Specifying Result Options

This section shows you how to specify criteria for the output of the synthesis run.

1. Open the Implementation Options form by clicking the Implementation Options button or selecting Project->Implementation Options, and click the Implementation Results tab at the top.

© 2016 Synopsys, Inc.

2. Specify the output files you want to generate.
   - To generate mapped netlist files, click Write Mapped Verilog Netlist or Write Mapped VHDL Netlist.
   - To generate a vendor-specific constraint file for forward annotation, click Write Vendor Constraint File. See *Generating Constraint Files for Forward Annotation (Legacy)*, on page 64 for more information.
3. Set the directory to which you want to write the results.
4. Set the format for the output file. The equivalent Tcl command for scripting is project -result_format *format*.

   You might also want to set attributes to control name-mapping. For details, refer to the appropriate vendor chapter in the *Reference Manual*.
5. Set other implementation options as needed (see *Setting Logic Synthesis Implementation Options*, on page 85 for a list of choices). Click OK.

   When you synthesize the design, the software compiles and maps the design using the options you set.

Specifying Timing Report Output

You can determine how much is reported in the timing report by setting the following options.

1. Selecting Project->Implementation Options, and click the Timing Report tab.
2. Set the number of critical paths you want the software to report.

3. Specify the number of start and end points you want to see reported in the critical path sections.
4. Set other implementation options as needed (see *Setting Logic Synthesis Implementation Options*, on page 85 for a list of choices). Click OK.

   When you synthesize the design, the software compiles and maps the design using the options you set.

Setting Verilog and VHDL Options

When you set up the Verilog and VHDL source files in your project, you can also specify certain compiler options.

Setting Verilog File Options

You set Verilog file options by selecting either Project->Implementation Options-> Verilog, or Options->Configure Verilog Compiler.

© 2016 Synopsys, Inc.

1. Specify the Verilog format to use.
   - To set the compiler globally for all the files in the project, select Project->Implementation Options->Verilog. If you are using Verilog 2001 or SystemVerilog, check the *Command Reference Manual* for supported constructs.
   - To specify the Verilog compiler on a per file basis, select the file in the Project view. Right-click and select File Options. Select the appropriate compiler. The default Verilog file format for new projects is SystemVerilog.

2. Specify the top-level module if you did not already do this in the Project view.

3. To extract parameters from the source code, do the following:
   - Click Extract Parameters.
   - To override the default, enter a new value for a parameter.

   The software uses the new value for the current implementation only. Note that parameter extraction is not supported for mixed designs.

4. Type in the directive in Compiler Directives, using spaces to separate the statements.

   You can type in directives you would normally enter with 'ifdef and ‘define statements in the code. For example, `ABC=30` results in the software writing the following statements to the project file:

```
set_option -hdl_define -set "ABC=30"
```


© 2016 Synopsys, Inc.

5. In the Include Path Order, specify the search paths for the include commands for the Verilog files that are in your project. Use the buttons in the upper right corner of the box to add, delete, or reorder the paths.

6. In the Library Directories, specify the path to the directory which contains the library files for your project. Use the buttons in the upper right corner of the box to add, delete, or reorder the paths.

7. Set other implementation options as needed (see *Setting Logic Synthesis Implementation Options*, on page 85 for a list of choices). Click OK.

   When you synthesize the design, the software compiles and maps the design using the options you set.

Setting VHDL File Options

You set VHDL file options by selecting either Project->Implementation Options->VHDL, or Options->Configure VHDL Compiler.

For VHDL source, you can specify the options described below.

1. Specify the top-level module if you did not already do this in the Project view. If the top-level module is not located in the default work library, you must specify the library where the compiler can find the module. For information on how to do this, see VHDL Panel, on page 227.

   You can also use this option for mixed language designs or when you want to specify a module that is not the actual top-level entity for HDL Analyst displaying and debugging in the schematic views.

2. For user-defined state machine encoding, do the following:
   - Specify the kind of encoding you want to use.
   - Disable the FSM compiler.

   When you synthesize the design, the software uses the compiler directives you set here to encode the state machines and does not run the FSM compiler, which would override the compiler directives. Alternatively, you can define state machines with the syn_encoding attribute, as described in *Defining State Machines in VHDL*, on page 392.

3. To extract generics from the source code, do this:
   - Click Extract Generic Constants.
   - To override the default, enter a new value for a generic.

   The software uses the new value for the current implementation only. Note that you cannot extract generics if you have a mixed language design.

© 2016 Synopsys, Inc.

4. To push tristates across process/block boundaries, check that Push Tristates is enabled. For details, see Push Tristates Option, on page 241in the *Reference Manual*.

5. Determine the interpretation of the synthesis_on and synthesis_off directives:
   - To make the compiler interpret synthesis_on and synthesis_off directives like translate_on/translate_off, enable the Synthesis On/Off Implemented as Translate On/Off option.
   - To ignore the synthesis_on and synthesis_off directives, make sure that this option is not checked. See translate_off/translate_on, on page 280 in the *Reference Manual* for more information.

6. Set other implementation options as needed (see *Setting Logic Synthesis Implementation Options*, on page 85 for a list of choices). Click OK.

   When you synthesize the design, the software compiles and maps the design using the options you set.

Specifying Attributes and Directives

Attributes and directives are specifications that you assign to design objects to control the way your design is analyzed, optimized, and mapped.

Attributes control mapping optimizations and directives control compiler optimizations. Because of this difference, you must specify directives in the source code or the compiler directives file. This table describes the methods that are available to create attribute and directives specifications:

	Attributes	Directives
VHDL	Yes	Yes
Verilog	Yes	Yes
SCOPE Editor	Yes	No
Constraints File	Yes	No

It is better to specify attributes in the SCOPE editor or the constraints file, because you do not have to recompile the design first. For directives, you must compile the design for them to take effect.

If SCOPE/constraints file and the HDL source code are specified for a design, the constraints has the highest priority when there are conflicts. Then, the `cdc` compiler directives take precedence over the HDL source code.

For further details, refer to the following:

- Specifying Attributes and Directives in VHDL, on page 100
- Specifying Attributes and Directives in Verilog, on page 101
- Specifying Attributes Using the SCOPE Editor, on page 102
- Specifying Attributes in the Constraints File, on page 106

© 2016 Synopsys, Inc.

Specifying Attributes and Directives in VHDL

You can use other methods to add attributes to objects, as listed in *Specifying Attributes and Directives*, on page 99. However, you can specify directives only in the source code. There are two ways of defining attributes and directives in VHDL:

- Using the predefined attributes package
- Declaring the attribute each time it is used

For details of VHDL attribute syntax, see VHDL Attribute and Directive Syntax, on page 308 in the *HDL Reference Manual.*

Using the Predefined VHDL Attributes Package

The advantage to using the predefined package is that you avoid redefining the attributes and directives each time you include them in source code. The disadvantage is that your source code is less portable. The attributes package is located in *installDirectory*/lib/vhd/synattr.vhd.

1. To use the predefined attributes package included in the software library, add these lines to the syntax:

   ```
   library synplify;
   use synplify.attributes.all;
   ```

2. Add the attribute or directive you want after the design unit declaration.

   declarations;
   attribute *attribute_name* **of** *objectName* **:** *objectType* **is** *value***;**

   For example:

   ```
   entity simpledff is
      port (q: out bit_vector(7 downto 0);
            d : in bit_vector(7 downto 0);
            clk : in bit);
   attribute syn_noclockbuf of clk : signal is true;
   ```

   For details of the syntax conventions, see VHDL Attribute and Directive Syntax, on page 308 in the *Reference Manual.*

3. Add the source file to the project.

Declaring VHDL Attributes and Directives

If you do not use the attributes package, you must redefine the attributes each time you include them in source code.

1. Every time you use an attribute or directive, define it immediately after the design unit declarations using the following syntax:

   design_unit_declaration;
   attribute *attributeName* : *dataType*;
   attribute *attributeName* **of** *objectName* : *objectType* **is** *value*;

   For example:

   ```
   entity simpledff is
      port (q: out bit_vector(7 downto 0);
            d : in bit_vector(7 downto 0);
            clk : in bit);
   attribute syn_noclockbuf : boolean;
   attribute syn_noclockbuf of clk :signal is true;
   ```

2. Add the source file to the project.

Specifying Attributes and Directives in Verilog

You can use other methods to add attributes to objects, as described in *Specifying Attributes and Directives*, on page 99. However, you can specify directives only in the source code.

Verilog does not have predefined synthesis attributes and directives, so you must add them as comments. The attribute or directive name is preceded by the keyword synthesis. Verilog files are case sensitive, so attributes and directives must be specified exactly as presented in their syntax descriptions. For syntax details, see Verilog Attribute and Directive Syntax, on page 102in the *Reference Manual.*

1. To add an attribute or directive in Verilog, use Verilog line or block comment (C-style) syntax directly following the design object. Block comments must precede the semicolon, if there is one.

Verilog Block Comment Syntax	Verilog Line Comment Syntax
/* **synthesis** *attributeName* = *value* */	// **synthesis** *attributeName* = *value*
/* **synthesis** *directoryName* = *value* */	// **synthesis** *directoryName* = *value*

© 2016 Synopsys, Inc.

For details of the syntax rules, see Verilog Attribute and Directive Syntax, on page 102 in the *HDL Reference Manual*. The following are examples:

```
module fifo(out, in) /* synthesis syn_hier = "hard" */;

module b_box(out, in); // synthesis syn_black_box
```

2. To attach multiple attributes or directives to the same object, separate the attributes with white spaces, but do not repeat the synthesis keyword. Do not use commas. For example:

   ```
   case state /* synthesis full_case parallel_case */;
   ```

3. If multiple registers are defined using a single Verilog reg statement and an attribute is applied to them, then the synthesis software only applies the last declared register in the reg statement. For example:

   ```
   reg [5:0] q, q_a, q_b, q_c, q_d /* synthesis syn_preserve=1 */;
   ```

   The syn_preserve attribute is only applied to q_d. This is the expected behavior for the synthesis tools. To apply this attribute to all registers, you must use a separate Verilog reg statement for each register and apply the attribute.

Specifying Attributes Using the SCOPE Editor

The SCOPE window provides an easy-to-use interface to add any attribute. You cannot use it for adding directives, because they must be added to the source files. (See *Specifying Attributes and Directives in VHDL*, on page 100 or *Specifying Attributes and Directives in Verilog*, on page 101). The following procedure shows how to add an attribute directly in the SCOPE window.

1. Start with a compiled design and open the SCOPE window. To add the attributes to an existing constraint file, open the SCOPE window by clicking on the existing file in the Project view. To add the attributes to a new file, click the SCOPE icon and click Initialize to open the SCOPE window.

2. Click the Attributes tab at the bottom of the SCOPE window.

   You can either select the object first (step 3) or the attribute first (step 4).

	Enable	Object Type	Object	Attribute	Value	Value Type	Description	Comment
1	✔	<any>	<Global>	syn_noclockbuf	1	boolean	Use normal input buffer	
2								
3								

Attributes

3. To specify the object, do one of the following in the Object column. If you already specified the attribute, the Object column lists only valid object choices for that attribute.
   - Select the type of object in the Object Filter column, and then select an object from the list of choices in the Object column. This is the best way to ensure that you are specifying an object that is appropriate, with the correct syntax.
   - Drag the object to which you want to attach the attribute from the RTL or Technology views to the Object column in the SCOPE window. For some attributes, dragging and dropping may not select the right object. For example, if you want to set syn_hier on a module or entity like an and gate, you must set it on the view for that module. The object would have this syntax: v:*moduleName* in Verilog, or v:*library.moduleName* in VHDL, where you can have multiple libraries.
   - Type the name of the object in the Object column. If you do not know the name, use the Find command or the Object Filter column. Make sure to type the appropriate prefix for the object where it is needed. For example, to set an attribute on a view, you must add the v: prefix to the module or entity name. For VHDL, you might have to specify the library as well as the module name.
4. If you specified the object first, you can now specify the attribute. The list shows only the valid attributes for the type of object you selected. Specify the attribute by holding down the mouse button in the Attribute column and selecting an attribute from the list.

© 2016 Synopsys, Inc.

If you selected the object first, the choices available are determined by the selected object and the technology you are using. If you selected the attribute first, the available choices are determined by the technology.

When you select an attribute, the SCOPE window tells you the kind of value you must enter for that attribute and provides a brief description of the attribute. If you selected the attribute first, make sure to go back and specify the object.

5. Fill out the value. Hold down the mouse button in the Value column, and select from the list. You can also type in a value.

6. Save the file.

   The software creates a Tcl constraint file composed of define_attribute statements for the attributes you specified. See How Attributes and Directives are Specified, on page 8 of the *Reference Manual* for the syntax description.

7. Add it to the project, if it is not already in the project.
   - Choose Project -> Implementation Options.
   - Go to the Constraints panel and check that the file is selected. If you have more than one constraint file, select all those that apply to the implementation.

The software saves the SCOPE information in a Tcl constraint file, using define_attribute statements. When you synthesize the design, the software reads the constraint file and applies the attributes.

© 2016 Synopsys, Inc.

Specifying Attributes in the Constraints File

When you use the SCOPE window (*Specifying Attributes Using the SCOPE Editor*, on page 102), the attributes are automatically written to a constraint file using the Tcl define_attribute syntax. This is the preferred method for defining constraints as the syntax is determined for you.

However, the following procedure explains how you can specify attributes directly in the constraint file.

1. Open a file in a text editor.
2. Enter the desired attributes. For example,

   define_attribute {*objectName***}** *attributeName value*

   For commands and syntax, see Summary of Attributes and Directives, on page 11 in the *Attribute Reference Manual*.
3. Save the constraints in a file using the FDC file extension.

The following code excerpt provides an example of attributes defined in the constraint file:

```
# Use a regular buffer instead of a clock buffer for clock "clk_slow".
   define_attribute {clk_slow} syn_noclockbuf 1

# Relax timing by not buffering "clk_slow", because it is the slow clock
# Set the maximum fanout to 10000.
   define_attribute {clk_slow} syn_maxfan 10000
```

For information about editing constraints, see *Using a Text Editor for Constraint Files (Legacy)*, on page 63.

Handling Properties with Attributes or Directives

Any property added to an object (i.e. net or instance) that is preserved or kept during the flow will be annotated in the netlist. Only properties for the syn_* attributes/directives are processed by the tool; while all other properties are simply annotated in the netlist when the object is available in the flow.

Examples

Suppose top_property_handling.v contains MyProp=Value, which is associated with out and is annotated in the netlist. If you apply the property on the net intermediate_net, then the property is not annotated in the netlist since this net is not preserved/kept in the flow.

Using the syn_keep Attribute

The syn_keep attribute helps preserve the specified net. If top_syn_keep.v applies syn_keep on the net intermediate_net, then this same net is kept.

Using Properties with the syn_keep Attribute

For this example, top.v applies the syn_keep to the net intermediate_net where the MyProp=Value is associated with intermediate_net. In this case, the property is annotated in the netlist since syn_keep instructs the tool to preserve/keep the net in the flow.

© 2016 Synopsys, Inc.

Working with Hierarchical Projects

You can use the hierarchical project management (HPM) top-down or bottom-up flows to develop hierarchical designs. You can also use top-down or bottom-up synthesis or a combination of the two to synthesize these designs. For more information about these flows, see *Hierarchical Project Management Flows*, on page 30.

This section describes how to set up and work with hierarchical projects in the flows mentioned above. The following topics provide more detail:

- Block-Based Subprojects, on page 108
- Creating Hierarchical Subprojects by Exporting Blocks, on page 108
- Creating Nested Subprojects, on page 111
- Working with Multiple Implementations in Hierarchical Projects, on page 117
- Working with Multiple Instances and Parameterized Modules, on page 118
- Configuring Synthesis Runs for Hierarchical Projects, on page 121
- Analyzing Synthesis Results for Hierarchical Projects, on page 125

Block-Based Subprojects

Any changes made to a block-based subproject affect all instances of the block. You cannot assign timing budgets or resources to block-based subprojects.

Creating Hierarchical Subprojects by Exporting Blocks

When you use a top-first approach to team design (*Top-First Development Flow for Hierarchical Projects*, on page 32), you must partition the design and specify subprojects. This procedure shows you how to designate a block as a subproject; see *Creating Hierarchical Subprojects by Exporting Blocks*, on page 108 for information about creating subprojects based on instances.

Unlike an instance-based subproject, any changes made to a block-based subproject affect all instances of the block.

1. Open the compiled top-level design.

   You must compile the design because you cannot export subprojects from an uncompiled design. It does not matter if the top level has black boxes.

2. Export a subproject.
   - Go to the Design Hierarchy view, and right-click the block you want to make into a subproject; then select Create Subproject (Design Block) from the popup menu.

The Export "*blockname*" as subproject dialog box opens.

© 2016 Synopsys, Inc.

– Specify a name for the project and a location.

– Specify source files for the subproject. Check the selected files in the list. The tool automatically enables needed source files from the top-level design, but the list might not be complete. If a source file is not listed, click the Add File button and add the missing file to the list. The file is automatically selected when it is added with Add File. When the tool creates the subproject, it adds the selected files to it.

  You must have at least one file in order to create a subproject. Some files may overlap with the parent project or a sibling project.

– To link updates to the projects, enable the Link sub projects to the parent project option.

– Click OK.

The tool creates a new subproject for the specified block and automatically links it to the top-level project. The block icon in the Design Hierarchy view changes to a green rectangle with a P in it, to indicate that it is now

defined as a subproject. In the Project Files view, you see a Subprojects directory that contains the new block project. The implementation options for the block project match the options of the active implementation of the parent project.

See *Working with Multiple Implementations in Hierarchical Projects*, on page 117 and *Working with Multiple Instances and Parameterized Modules*, on page 118 for information about creating additional implementations for the block.

3. Save the design. Until you do so, the hierarchy you created is not saved.
4. Synthesize the design, following the instructions in *Top-Down Synthesis for Hierarchical Projects*, on page 36 and *Bottom-Up Synthesis for Hierarchical Projects*, on page 34.

Creating Nested Subprojects

With hierarchical design, you can have subprojects that contain other subprojects. You can use the block and instance exporting techniques to recursively export parts of the design as subprojects.

© 2016 Synopsys, Inc.

The following procedure shows how to export nested subprojects, and is based on a design containing three instances of ablock. The ablock module instantiates two bblock instances (b1, b2), bblock instantiates two instances of cblock (c1, c2), and cblock instantiates two instances of dblock (d1, d2).

1. Load the top-level project and compile it with Run->Compile Only.

   You must compile the design before you can export an instance as a subproject. The Design Hierarchy tab and the RTL view show the a1, a2, and a3 instances at ablock at the top level.

2. Create a subproject.
   - In the Design Hierarchy view, right-click an instance (a3) and select the Create Subproject command. For details about using the command to export subprojects, see *Creating Hierarchical Subprojects by Exporting Blocks*, on page 108.
   - A dialog box shows you the files needed for the selected instance. Edit the list as needed to reflect the file list for the instance.
   - Click OK. The view now shows a green P icon next to the instance to indicate it is an exported subproject.

The Project Files view shows the newly-created subproject. It has its own implementation, separate from the top-level implementation.

© 2016 Synopsys, Inc.

3. Save the design.

   Until you save the design, the subproject you just created is not written to disk.

4. Create a subproject within a subproject.
   - Compile the subproject under which you want to create another subproject (a3), by selecting the subproject implementation and selecting Run->Compile Only.

     You must compile again at this level, because you cannot export a subproject from an uncompiled design, and at this point the a3 subproject is not compiled.

– In the Design Hierarchy view, right-click an instance and select the Create Subproject command to export it. You can select any instance in the hierarchy. The following figure shows the current subproject expanded, the next level skipped (bblock), and a cblock instance selected from two levels below:

– Check that the list of dependent files for the instance is accurate in the window that opens, and click OK when you are done. The Design Hierarchy and Project Files views reflect the creation of a new cblock subproject under the parent ablock subproject.

5. Save the design.

© 2016 Synopsys, Inc.

6. Compile and map the entire design by selecting the top-level implementation and clicking Run.

   The top-level RTL shows exported subprojects in green, to indicate that they were exported. The following figure shows the a3 instance that was exported from the top level.

   The RTL view indicates exported subprojects relative to the current level. If you push into the a3 instance from the top level, and then descend into the bblock instance, the exported cblock instance is not in green, because the reference point is the top level. However, if you select the ablock subproject and then generate an RTL view, the cblock instance is in green when you push down to it. This is because the cblock instance was exported relative to the ablock subproject.

Working with Multiple Implementations in Hierarchical Projects

The following procedure show you how to work with multiple implementations in hierarchical projects. For example, you might want to create multiple implementations with different run configurations if you are experimenting with different implementation options.

1. Open the top-level hierarchical project and compile it (F7).

   This updates the hierarchical project views.

2. To add an implementation to the top-level project or a block-level project, do the following in the Project Files view:
   - Right-click the top level or the block and select Add Implementation.
   - Set the new implementation options you want.

3. To add subproject implementations to the top-level project, do one of the following:
   - In the Project Files view, right-click a top-level implementation and select Add Subproject Implementations.
   - Select a top-level implementation in the Project Files view. Select the Project-> Add Subproject Implementation command.

© 2016 Synopsys, Inc.

The tool automatically adds a new implementation to each defined subproject, and updates the top-level implementation to use the new block implementations. The new block implementations are based on the selected top-level implementation.

4. To specify which implementation to run, do the following:
   - Select Project->Edit Run Configuration.
   - In the dialog box, set Implementation to the block implementation you want to synthesize. To prevent synthesis from being run set this option to <off>. If there are red cells in the table, synchronize options across implementations as needed with the Sync buttons. Click OK.
   - To run all implementations, select the Run->Run all Implementations instead of clicking the Run button to synthesize.

   By default the tool runs the active implementation.

Working with Multiple Instances and Parameterized Modules

A module can have various parameters, such as SIZE or WIDTH. When multiple instances of a module have different parameter values, they are called parameterized modules. The following procedure shows you how to work with multiple instantiations of parameterized modules.

1. Create your design and instantiate the parameterized modules.

   In the design example used to illustrate this procedure, cblock is instantiated four times. The instantiations have different values defined for DWIDTH and USE_AND:

```
`include "meminfo.h"
parameter USE_AND = 1;
//parameter USE_AND = 0;
//parameter DWIDTH = 2;
parameter DWIDTH = 6;
```

   The following table shows the parameter values for the four cblock instances in the design. It also shows that there are three unique sets of parameters between the instances; cblock_1 and cblock_3 share the same parameter set.

cblock Instances	USE_AND	DWIDTH	Parameter Set
cblock_1	0	6	1
cblock_2	1	6	2
cblock_3	0	6	1
dblock_4	1	2	3

2. Export one instance of the parameterized module.
   - Compile the design.
   - Open the Design Hierarchy tab, and select one instance of the parameterized module. You can select any instance of the module in the design to export.
   - Export the subproject by right-clicking the module and selecting the Create Subproject command.

     A block-based export creates a single subproject that affects all instances of the module.
   - If required, add any additional files needed for the subproject to the automatically generated list that displays, and click OK.

After you set the files for the subproject, the module is marked with a green icon in the Design Hierarchy view. Module-based subprojects are

© 2016 Synopsys, Inc.

denoted by rev_#(%), as shown in the following figure, where cblock was exported as a module.

– Check the Project Files view. It shows that a subproject was created for cblock, and that it is a parameterized implementation:

3. Click the Save All icon ( ) in the top bar to save all the exported subproject and project file information.

4. Set options and constraints for synthesis.

5. Run the project.

   All exported subprojects are synthesized from the bottom up by default. If you want to allow the subproject to be optimized, you can change the default, as described in *Configuring Synthesis Runs for Hierarchical Projects*, on page 121.

6. If you make changes to parameters in a design, do the following:
   - Right-click the top-level project implementation and select Subproject Parameter Sync.
   - Rerun synthesis.

Configuring Synthesis Runs for Hierarchical Projects

Before you synthesize the hierarchical project, set options for the synthesis run. This procedure shows you how to use the Hierarchical Project Options command and set synthesis options for the top level design or for the blocks. You can set different run configurations for different implementations.

1. Select the Hierarchical Project Options command.
   - Select the top level or the block in the Project Files view.
   - Either right-click and select Hierarchical Project Options from the popup menu or select Project->Hierarchical Project Options.

© 2016 Synopsys, Inc.

2. Set options in the dialog box that opens:
   - In the block column, set Implementations to the implementation you want to run. If you do not want to synthesize a block, set it to <off>. When you set a block to <off>, that subproject is treated as a black box.

– Set Run Type for the block to reflect how you want it synthesized: top_down or bottom_up. For details of the differences, see *Bottom-Up Synthesis for Hierarchical Projects*, on page 34 and *Top-Down Synthesis for Hierarchical Projects*, on page 36.

– Set any other implementation options you want for the synthesis run.

3. If necessary, synchronize the top-level and block settings. If you see a red cell in the table, it means that option must be synchronized before synthesis.

   The following figure shows that pink highlighting in the device option field. This indicates that you must synchronize the subproject with the top level setting. The Disable I/O Insertion option is yellow and synchronization with the top level is optional.

© 2016 Synopsys, Inc.

– Click the Synchronize All Options with Top Level button at the bottom of the box to match the required block device settings to the settings at the top level. If the required device options match, this button is grayed out and unavailable.

4. Click OK.

5. Synthesize the design.

See *Bottom-Up Synthesis for Hierarchical Projects*, on page 34, *Top-Down Synthesis for Hierarchical Projects*, on page 36 and *Mixed Block Synthesis for Hierarchical Projects*, on page 38 for details.

Analyzing Synthesis Results for Hierarchical Projects

After synthesizing a hierarchical project, you can analyze the results as described below.

1. To view the synthesis results, check the RTL view.

   The following figure of the top-level RTL view shows the linked netlists from subprojects.

© 2016 Synopsys, Inc.

This is an example of an RTL view of the subprojects:

2. Check the log file for hierarchical reports for each subproject.

3. For resource usage, check the corresponding sections in the top-level and subproject log files.

Searching Files

A find-in-files feature is available to perform string searches within a specified set of files. Advantages to using this feature include:

- Ability to restrict the set of files to be searched to a project or implementation.
- Ability to cross probe the search results.

The find-in-files feature uses a dialog box to specify the search pattern, the criteria for selecting the files to be searched, and any search options such as match case or whole word. The files that meet the criteria are searched for the pattern, and a list of the files containing the search pattern are displayed at the bottom of the dialog box.

To use the find-in-files feature, open the Find in Files dialog box by selecting Edit->Find in Files and enter the search pattern in the Find what field at the top of the dialog box.

© 2016 Synopsys, Inc.

Identifying the Files to Search

The Find In section at the top of the dialog box identifies the files to be searched:

- Project Files – Searches the files included in the selected project (use the drop-down menu to select the project). By default, the files in the active project are searched. The files can reside anywhere on the disk; any project ‘include files are also searched.
- Implementation Directory – Searches all files in the specified implementation directory (use the drop-down menu to select the implementation). By default, the files in the active implementation are searched. You can search all implementations by selecting <All Implementations> from the drop-down menu. If Include sub-folders for directory searches is also selected, all files in the implementation directory hierarchy are searched.
- Subproject – Searches all files of a subproject included in the top-level project file, when Include SubProject files is selected.
- Directory – Searches all files in the specified directory (use the browser button to select the directory). If Include sub-folders for directory searches is also selected, all files in the directory hierarchy are searched.

All of the above selection methods can be applied concurrently when searching for a specified pattern.

The Result Window selection is used after any of the above selection methods to search the resulting list of files for a subsequent sub-pattern.

Filtering the Files to Search

A file filter allows the file set to be searched to be further restricted based on the matching of patterns entered into the File filter field.

- A pattern without a wildcard or a “.” (period) is interpreted as a filename extension. For example, `fdc` restricts the search to only constraint files.
- Multiple patterns can be specified using a semicolon delimiter. For example, v;vhd restricts the files searched to only Verilog and VHDL files.
- Wildcard characters can be used in the pattern to match file names. For example, a*.vhd restricts the files searched to VHDL files that begin with an “a” character.

- Leaving the File filter field empty searches all files that meet the Find In criteria.
- The Match Case, Whole Word, and Regular Expressions search options can be used to further restrict searches.

Initiating the Search

After entering the search criteria, click the Find button to initiate the search. All matches found are listed in the results area at the bottom of the dialog box; the status line just below the Find button reports the number of matches found in the indicated number of files and the total number of files searched.

While the find operation is running, the status line is continually updated with how many matches are found in how many files and how many files are being searched.

Search Results

The search results are displayed in the results window at the bottom of the dialog box. For each match found, the entire line of the file is the displayed in the following format:

fullpath_to_file(*lineNumber*): *matching_line_text*

For example, the entry

```
C:\Designs\leon\dcache.vhd(487): wdata := r.wb.data1;
```

indicates that the search pattern (data1) was found on line 487 of the dcache.vhd file.

To open the target file at the specified line, double-click on the line in the results window.

© 2016 Synopsys, Inc.

Archiving Files and Projects

Use the archive utility to archive, extract (unarchive), or copy design projects. Archived files are in a proprietary format and saved to a file name using the `sar` extension. The archive utility is available through the Project menu in the GUI or using the project command in the Tcl window.

This document provides a description of how to use the utility.

- Archive a Project
- Un-Archive a Project
- Copy a Project
- Support for Hierarchical Include Paths

Archive a Project

Use the archive utility to store the files for a design project into a single archive file in a proprietary format (`sar`). You can archive an entire project or selected files from a project. If you want to create a copy of a project without archiving the files, see *Copy a Project*, on page 137.

Here are the steps to create an archive:

1. In the Project view, select Project->Archive Project to bring up the wizard.

   The Tcl command equivalent is project -archive. For a complete description of the project Tcl command options for archiving, see project, on page 54 of the *Reference Manual*.

   The archive utility automatically runs a syntax check on the active project (Run->Syntax Check command) to ensure that a complete list of project files is generated. If you have Verilog 'include files in your project, the utility includes the complete list of Verilog files. It also checks the syntax automatically for each implementation in the project to ensure that the file list is complete for each implementation as well. The wizard displays the name of the project to archive, the top-level directory where the project file is located (root directory), and other information.

2. Do the following on the first page of the wizard:
   - Fill in Destination File with a location for the archive file.
   - Set Archive Style. You can archive all the project files with all the implementations or selectively archive files and implementations
   - To archive only the active implementation, enable Active Implementation.
   - To selectively archive files, enable Customized file list, click Next, and use the check boxes to include files in or exclude files from the archive. Use the Add Extra Files button on the this page to include additional files in the project.
3. Click Next.

   If you did not select Customized file list, the tool summary displays all the files in the archive and shows the full uncompressed file size as shown in step 5 (the actual size is smaller after the archiving operation as there is no duplication of files). When you select Customized file list, the following interim menu is displayed to allow you to exclude specific file from the archive.

© 2016 Synopsys, Inc.

4. Click next to advance to the next screen (step 3).

5. Verify that the current archive contains the files that you want, then click Archive which creates the project archive `sar` file. If the list of files is incorrect, click Back and include/exclude any desired files.

6. Click Archive if you are finished. The synthesis tool reports the archive success and the path location of the archive file.

© 2016 Synopsys, Inc.

Un-Archive a Project

Uses this procedure to extract design project files from an archive file (`sar`).

1. In the Project view, select Project->Un-Archive Project to display the wizard

   The Tcl command equivalent is project -unarchive. For a complete description of the project Tcl command options for archiving, see project, on page 54 of the *Reference Manual*.

2. In the wizard, enter the following:
   - Name of the `sar` file containing the project files.
   - Name of project to extract (un-archive). This field is automatically extracted from the `sar` file and cannot be changed.
   - Pathname of directory in which to write the project files (destination).
   - Click Next.

3. Make sure all the files that you want to extract are checked and references to these files are resolved.
   - If there are files in the list that you do not want to include when the project is un-archived, uncheck the box next to the file. The un-checked files will be commented out in the project file (`prj`) when project files are extracted.
   - If you need to resolve a file in the project before un-archiving, click the Resolve button and fill out the dialog box.
   - If you want to replace a file in the project, click the Change button and fill out the dialog box. Put the replacement files in the directory you specify in Replace directory. You can replace a single file, any unresolved files, or all the files. You can also undo the replace operation.

© 2016 Synopsys, Inc.

4. Click Next and verify that the project files you want are displayed in the Un-Archive Summary.

5. If you want to load this project in the UI after files have been extracted, enable the Load project into Synplify Pro after un-archiving option.

6. When the Add extra input path to project file option is enabled, the archive utility finds all include files and copies them into a directory called extra_input. This directory is added to the unarchived project file.

The Tcl command equivalent is set_option -include_path "./extra_input/".

If the archive files contain relative or absolute include paths, the _SEARCHFILENAMEONLY_ directive can have the compiler remove the relative/absolute paths from the 'include and search only for the file names. To use the _SEARCHFILENAMEONLY_ directive, all include files must have unique names. For details, see _SEARCHFILENAMEONLY_, on page 237.

7. Click Un-Archive.

   A message dialog box is displayed while the files are being extracted.

8. If the destination directory already contains project files with the same name as the files you are extracting, you are prompted so that the existing files can be overwritten by the extracted files.

Copy a Project

Use this utility to create an unarchived copy of a design project. You can copy an entire project or just selected files from the project. However, if you want to create an archive of the project, where the entire project is stored as a single file, see *Archive a Project*, on page 130.

Here are the steps to create a copy of a design project:

1. From the Project view, select Project->Copy Project.

   The Tcl command equivalent is project -copy. For a complete description of the project Tcl command options for archiving, see project, on page 54 of the *Reference Manual*.

   This command automatically runs a syntax check on the active project (Run->Syntax Check command) to ensure that a complete list of project files is generated. If you have Verilog include files in your project, they are included. The utility runs this check for each implementation in the

© 2016 Synopsys, Inc.

project to ensure that the file list is complete for each implementation and then displays the wizard, which contains the name of the project and other information.

2. Do the following in the wizard:
   - Specify the destination directory where you want to copy the files.
   - Select the files to copy. You can choose to copy all the project files; one or more individual files, input files only, or customize the list to be copied.
   - To specify a custom list of files, enable Customized file list. Use the check boxes to include or exclude files from the copy. Enable SRS if you want to copy all `srs` files (RTL schematics). You cannot enable the Source Files option if you select this. Use the Add Extra Files button to include additional files in the project.

– Click **Next**.

© 2016 Synopsys, Inc.

3. Do the following:
   - Verify the copy information.
   - Enter a destination directory. If the directory does not exist it will be created.
   - Click Copy.

   This creates the project copy.

Support for Hierarchical Include Paths

The archive utility can support various forms of include path hierarchies to locate files for a project. For example:

- The include path can be relative to the location of the source file.

```
block_a/
block_a.v -> `include "block_a.h"
```

- The include path can be a relative path outside of the project.

  ```
  block_b.v -> `include "../../block_b.h"
  ```

  The archive utility can determine the absolute path for the file from the relative path as shown below:

  ```
  remote/sbg_pe/tests/feature_flow/include/block_b.h
  ```

  After unarchiving the project, you can see the directory structure for the equivalent absolute path relative to the project.

  ```
  "./remote/sbg_pe/tests/feature_flow/include/block_b.h"
  ```

- The file location can be specified by include_path in the project file.

  ```
  block_c/
  block_c.v -> `include "block_c.h"
  ```

  Where this file is located in the directory /include1/.

- The include path can be an absolute path outside of the project.

  ```
  block_d/
  block_d.v -> `include "/slowfs/sbg/tests/include2/block_d.h"
  ```

  When you archive the project, the absolute path becomes a relative path. After unarchiving the project, you can see the directory structure for the relative path to the project.

  ```
  "./slowfs/sbg/tests/include2/block_d.h"
  ```

- The file location can be specified by include_path in the project file.

  ```
  top_block/
  top_block.v -> `include "top_block.h"
  ```

  Where the top_block.v file is located in the directory /include2/.

- Any additional search paths specified in the project file are copied and included as relative paths to the project.

After you archive and unarchive the project, the relative paths in the original project become absolute paths in the new unarchived project. In the project file, the set_option -include_path preserves the original search order for the files.

© 2016 Synopsys, Inc.

Using the __SEARCHFILENAMEONLY__ Compiler Directive

Whenever you have a SAR file that contains relative or absolute include paths for the files in the project, you can also use the _SEARCHFILENAMEONLY_ directive to have the compiler remove the relative/absolute paths from the 'include and search only for the file names. Otherwise, you may have problems using the archive utility. For details, see _SEARCHFILENAMEONLY_, on page 237.

CHAPTER 5

Specifying Constraints

This chapter describes how to specify constraints for your design. It covers the following:

- Using the SCOPE Editor, on page 144
- Specifying SCOPE Constraints, on page 151
- Specifying Timing Exceptions, on page 162
- Finding Objects with Tcl find and expand, on page 168
- Using Collections, on page 177
- Converting SDC to FDC, on page 187
- Using the SCOPE Editor (Legacy), on page 188

the following chapters discuss related information :

- Chapter 4, *Constraints* (*Reference Manual)* for an overview of constraints
- Chapter 5, *SCOPE Constraints Editor*, (*Reference Manual)* for a description of the SCOPE editor

© 2016 Synopsys, Inc.

Using the SCOPE Editor

The SCOPE (Synthesis Constraints OPtimization Environment®) presents a spreadsheet-like editor with a number of panels for entering and managing timing constraints and synthesis attributes. The SCOPE GUI is good for editing most constraints, but there are some constraints (like black box constraints) which can only be entered as directives in the source files. The SCOPE GUI also includes an advanced text editor that can help you edit constraints easily.

These constraints are saved to the FPGA Design Constraint (FDC) file. The FDC file contains *Synopsys SDC Standard* timing constraints (for example, create_clock, set_input_delay, and set_false_path), along with the non-timing constraints (design constraints) (for example, define_attribute, define_scope_collection, and define_io_standard). When working with these constraints, use the following processes:

- For existing designs, run the sdc2fdc script to translate legacy SDC constraints and create a constraint file that contains Synopsys SDC standard timing constraints and design constraints. For details about this script, see *Converting SDC to FDC*, on page 187.
- For new designs, use the SCOPE editor. See *Creating Constraints in the SCOPE Editor*, on page 144 for more information.
- For new designs, use the create_fdc_template Tcl command. See *Creating Constraints With the FDC Template Command*, on page 149 for details.

Creating Constraints in the SCOPE Editor

The following procedure shows you how to use the SCOPE editor to create constraints for the FDC constraint file.

1. To create a new constraint file, follow these steps:
   - Compile the design (F7).
   - Open the SCOPE window by:

     Clicking the SCOPE icon in the toolbar ( ).

     This brings up the New Constraint File dialog box.

     OR

Pressing Ctrl-n or selecting File -> New. This brings up the New dialog box.

2. To open an existing file, do one of the following:
   - Double-click the file from the Project view.
   - Press Ctrl-o or select File->Open. In the dialog box, set the kind of file you want to open to Constraint Files (SCOPE) (fdc), and double-click to select the file from the list.

An empty SCOPE spreadsheet window opens. The tabs along the bottom of the SCOPE window list the different kinds of constraints you can add. For each kind of constraint, the columns contain specific data.

© 2016 Synopsys, Inc.

3. Select if you want to apply the constraint to the top-level or for modules from the Current Design option drop-down menu located at the top of the SCOPE editor.

4. You can enter or edit the following types of constraints:
   - Timing constraints – on the Clocks, Generated Clocks, Inputs/Outputs, Registers, or Delay Paths tab.
   - Design constraints – on the Collections, Attributes, I/O Standards, or Compile Points tab.

   For details about these constraints, see *Specifying SCOPE Constraints*, on page 151.

   For information about ways to enter constraints within the SCOPE editor, see *Guidelines for Entering and Editing Constraints*, on page 159.

5. The free form constraint editor is located in the TCL View tab, which is the last tab in SCOPE. The text editor has a help window on the right-

hand side. For more information about this text editor, see *Using the TCL View of SCOPE GUI*, on page 157.

6. Click on the Check Constraints button to run the constraint checker. The output provides information on how the constraints are interpreted by the tool.

All constraint information is saved in the same FPGA Design Constraint file (FDC) with clearly marked beginning and ending for each section. Do not manually modify these pre-defined SCOPE sections.

© 2016 Synopsys, Inc.

The following example shows the contents of an FDC file.

```
##############################################################################
# FDC constraints translated from Synplify Legacy Timing & Design Constraints
##############################################################################

set_rtl_ff_names {#}
###==== BEGIN Header

# Synopsys, Inc. constraint file
# D:\bugs\timing_88\clk_prior\scratch\top.fdc
# Written on Wed Jun 20 10:50:15 2012
# by Synplify Premier with Design Planner, G-2012.09 FDC Constraint Editor

# Custom constraint commands may be added outside of the SCOPE tab sections bounded with BEGIN/END.
# These sections are generated from SCOPE spreadsheet tabs.

###==== END Header

###==== BEGIN Clocks - (Populated from tab in SCOPE, do not edit)
create_clock -name {clka} {p:clka} -period 10 -waveform {0 5.0}
create_clock -name {clkb} {p:clkb} -period 6.667 -waveform {0 3.3335}
set_clock_groups -derive -name default_clkgroup_0 -asynchronous -group [get_clocks {clka}]
set_clock_groups -derive -name default_clkgroup_1 -asynchronous -group [get_clocks {clkb}]
###==== END Clocks

###==== BEGIN "Generated Clocks" - (Populated from tab in SCOPE, do not edit)
###==== END "Generated Clocks"

###==== BEGIN Collections - (Populated from tab in SCOPE, do not edit)
define_scope_collection all_inputs_fdc {find -port * -filter @direction==input}
define_scope_collection all_outputs_fdc {find -port * -filter @direction==output}
define_scope_collection all_clocks_fdc {find -hier  -clock *}
define_scope_collection all_registers_fdc {find -hier -seq *}
define_scope_collection all_grp {define_collection  [find -inst {i:FirstStbcPhase}] [find -inst {i:Norml
define_scope_collection fdc_cmd_0 {find -seq {*y*.q[*]}}
define_scope_collection fdc_cmd_1 {find {n:foo}}
define_scope_collection fdc_cmd_2 {expand -hier -seq -from $fdc_cmd_1}
###==== END Collections

###==== BEGIN Inputs/Outputs - (Populated from tab in SCOPE, do not edit)
set_input_delay -clock {c:clka} -clock_fall -add_delay 0.000 $all_inputs_fdc
set_output_delay -clock {c:clka} -add_delay 0.000 $all_outputs_fdc
set_input_delay -clock {c:clka} -add_delay 2.00 {p:a[7:0]}
set_input_delay -clock {c:clka} -add_delay 0 {p:rst}
###==== END Inputs/Outputs

###==== BEGIN "Delay Paths" - (Populated from tab in SCOPE, do not edit)
set_multicycle_path 3 -end  -from  $fdc_cmd_0
set_false_path -comment {false foo[0] free{iddhh}} -from {b:ena}
set_false_path -from  $fdc_cmd_2  -to {i:abc.def.g_reg}  -through  {n:bar}
set_false_path -from {$boing} -to {c:dcm|clk0fx_derived_clock[2]}-through  {n:fudge}
set_false_path -from  {c:dcm|clk0fx_derived_clock}  -to  {i:abc.def.g_reg[0] i:abc}  -through {n:fudge}
###==== END "Delay Paths"
```

Creating Constraints With the FDC Template Command

Use the Tcl command create_fdc_template to create an initial constraint file (fdc) for your specific design. This command lets you specify port clocks, I/O delays, and initial set_clock_groups for the clocks for which text headers are generated that can help guide you when creating this constraint.

The following procedure shows you how to create constraints in the FDC constraints file with the create_fdc_template command:

1. Create a project for your design.
2. Compile the design.
3. At the command line, for example, you can specify the following:

   ```
   create_fdc_template -period 10 -out_delay 1.5
   ```

   The command automatically updates your project to reflect the new constraint file(s). Do Ctrl+s to save the new settings.

4. If you open the SCOPE editor, you can check that the clock period and output delay values were added to the constraint file as shown in the following figure.

© 2016 Synopsys, Inc.

	Enable	Name	Object	Period	Waveform	Add	Clock Group	Latency	Uncertainty	Comment
1	☑	clock	[get_ports {p:clock}]	10	0 5.0	☐	<default>			
2										
3										
4										
5										
6										
7										

Clocks

	Enable	Delay Type	Port	Rise	Fall	Max	Min	Clock	Clock Fall	Add Delay	Value
1	☑	input	p:porta[7:0]	☐	☐	☐	☐	c:clock	☐	☑	0
2	☑	input	p:portb[7:0]	☐	☐	☐	☐	c:clock	☐	☑	0
3	☑	input	p:portc[7:0]	☐	☐	☐	☐	c:clock	☐	☑	0
4	☑	input	p:resetn	☐	☐	☐	☐	c:clock	☐	☑	0
5	☑	output	p:porta[7:0]	☐	☐	☐	☐	c:clock	☐	☑	1.5
6	☑	output	p:portb[7:0]	☐	☐	☐	☐	c:clock	☐	☑	1.5
7	☑	output	p:portc[7:0]	☐	☐	☐	☐	c:clock	☐	☑	1.5

Inputs/Outputs

5. Each port clock includes a set_clock_groups header with details displayed in the TCL View, which can help you determine whether clocks have been optimized away or if there are any derived clocks.

   However, if there is only one clock port and no derived clocks, no explicit clock groups are created since they are not needed, as shown below.

```
##############################################################################
### Individual "set_clock_groups" commands for all "clock" derived clocks
### appear at the end of this file.  Enabling a given command will make the
### given clock asynchronous to all other clocks.  If a given clock (below) does not
### appear in the final Performance Summary (in the *.srr file after synthesis),
### the clock may have been optimized away due to Gated/Generated Clock Conversion.
### See the "CLOCK OPTIMIZATION REPORT" in the *.srr file.
### Below is a list of any clocks derived from "clock":
###  clock DERIVED CLOCKS:
##############################################################################
```

   For details about the command syntax, see create_fdc_template, on page 32.

6. You can continue using the SCOPE editor to create other constraints.

7. Save the constraint file.

Specifying SCOPE Constraints

Timing constraints define the performance goals for a design. The FPGA synthesis tool supports a subset of the Synopsys SDC Standard timing constraints (for example, create_clock, set_input_delay, and set_false_path). For additional support, see *Synopsys Standard Timing Constraints*, on page 152.

Design constraints let you add attributes, define collections and specify constraints for them, and select specific I/O standard pad types for your design.

You can define both timing and design constraints in the SCOPE editor. For the different types of constraints, see the following topics:

- Entering and Editing SCOPE Constraints
- Setting Clock and Path Constraints
- Defining Input and Output Constraints
- Specifying Standard I/O Pad Types

To set constraints for timing exceptions like false paths and multicycle paths, see *Specifying Timing Exceptions*, on page 162.

For information about collections, see *Using Collections*, on page 177.

Entering and Editing SCOPE Constraints

This section contains a description of the timing and design constraints you can enter in the SCOPE GUI that are saved to an FDC file. The SCOPE timing constraint panels include:

SCOPE Panel	See ...	Tcl Commands
Clocks	Clocks	create_clock set_clock_groups set_clock_latency set_clock_uncertainty
Generated Clocks	Generated Clocks	create_generated_clock
Collections	Collections	define_scope_collection

© 2016 Synopsys, Inc.

SCOPE Panel	See ...	Tcl Commands
Inputs/Outputs	Inputs/Outputs	set_input_delay set_output_delay
Registers	Registers	set_reg_input_delay set_reg_output_delay
Delay Paths	Delay Paths	set_false_path set_max_delay set_multicycle_path
Attributes	Attributes	define_attribute define_global_attribute
I/O Standards	I/O Standards	define_io_standard
Compile Points	Compile Points	define_compile_point define_current_design
TCL View	TCL View	--

Synopsys Standard Timing Constraints

The FPGA synthesis tools support Synopsys standard timing constraints for a subset of the clock definition (Clocks and Generated Clocks), I/O delay (Inputs/Outputs), and timing exception constraints (Delay Paths).

Setting Clock and Path Constraints

The following table summarizes how to set different clock and path constraints from the SCOPE window.

To define ...	Pane	Do this to set the constraint ...
Clocks	Clock	Select the clock object (Clock). Specify a clock name (Clock Alias), if required. Type a period (Period). Change the rise and fall edge times for the clock. waveforms of the clock in nanoseconds, if needed. Change the default clock group, if needed. Check the Enabled box. See *Defining Clocks*, on page 194 for information about clock attributes.
Generated Clocks	Generated Clocks	Select the generated clock object. Specify the master clock source (a clock source pin in the design). Specify whether to use invert for the generated clock signal. Specify whether to use: edges, divide_by, or multiply_by. Check the Enabled box.
Input/output delays	Inputs/ Outputs	See *Defining Input and Output Constraints (Legacy)*, on page 203 for information about setting I/O constraints.
Maximum path delay	Delay Paths	Select the Delay Type path of Max Delay. Select the start/from point for either a port or register (From/Through). See *Defining From/To/Through Points for Timing Exceptions*, on page 162 for more information. Select the end/to point for either an output port or register. Specify a through point for a net or hierarchical port/pin (To/Through). Set the delay value (Max Delay). Check the Enabled box.
Multicycle paths	Delay Paths	See *Defining Multicycle Paths*, on page 166.

© 2016 Synopsys, Inc.

To define ...	Pane	Do this to set the constraint ...
False paths	Delay Paths	See *Defining False Paths*, on page 167 for details.
Global attributes	Attributes	Set Object Type to <global>. Select the object (Object). Set the attribute (Attribute) and its value (Value). Check the Enabled box.
Attributes	Attributes	Do either of the following: • Select the type of object (Object Type). Select the object (Object). Set the attribute (Attribute) and its value (Value). Check the Enabled box. • Set the attribute (Attribute) and its value (Value). Select the object (Object). Check the Enabled box.

Defining Input and Output Constraints

In addition to setting I/O delays in the SCOPE window as described in *Setting Clock and Path Constraints, on page 192*, you can also set the Use clock period for unconstrained IO option.

- Open the SCOPE window, click Inputs/Outputs, and select the port (Port). You can set the constraint for
  - All inputs and outputs (globally in the top-level netlist)
  - For a whole bus
  - For single bits

  You can specify multiple constraints for the same port. The software applies all the constraints; the tightest constraint determines the worst slack. If there are multiple constraints from different levels, the most specific overrides the more global. For example, if there are two bit constraints and two port constraints, the two bit constraints override the two port constraints for that bit. The other bits get the two port constraints.
- Specify the constraint value in the SCOPE window:
  - Select the type of delay: input or output (Type).
  - Type a delay value (Value).
  - Check the Enabled box, and save the constraint file in the project.

  Make sure to specify explicit constraints for each I/O path you want to constrain.
- To determine how the I/O constraints are used during synthesis, do the following:
  - Select Project->Implementation Options, and click Constraints.
  - To use only the explicitly defined constraints disable Use clock period for unconstrained IO.
  - To synthesize with all the constraints, using the clock period for all I/O paths that do not have an explicit constraint enable Use clock period for unconstrained IO.
  - Synthesize the design. When you forward-annotate the constraints, the constraints used for synthesis are forward-annotated for place-and-route.

© 2016 Synopsys, Inc.

- Input or output ports with explicitly defined constraints, but without a reference clock (-ref option) are included in the System clock domain and are considered to belong to every defined or inferred clock group.
- If you do not meet timing goals after place-and-route and you need to adjust the input constraints; do the following:
  - Open the SCOPE window with the input constraint.
  - Use the set_clock_route_delay command to translate the -route option for the constraint, so that you can specify the actual route delay in nanoseconds, as obtained from the place-and-route results. Adding this constraint is equivalent to putting a register delay on the input register. Use it as a fudge factor to force the synthesis engine to accommodate a routing delay that is larger than expected.
  - Resynthesize your design.

Specifying Standard I/O Pad Types

You can specify a standard I/O pad type to use in the design. The equivalent Tcl command is define_io_standard.

1. Open the SCOPE window and go to the I/O Standard tab.
2. In the Port column, select the port. This determines the port type in the Type column.
3. Enter an appropriate I/O pad type in the I/O Standard column. The Description column shows a description of the I/O standard you selected.

   For details of supported I/O standards, see Industry I/O Standards, on page 181.
4. Where applicable, set other parameters like drive strength, slew rate, and termination.

   You cannot set these parameter values for industry I/O standards whose parameters are defined by the standard.

   The software stores the pad type specification and the parameter values in the syn_pad_type attribute. When you synthesize the design, the I/O specifications are mapped to the appropriate I/O pads within the technology.

Using the TCL View of SCOPE GUI

The TCL View of the SCOPE GUI is an advanced text file editor used for FPGA timing and design constraints. This text editor provides the following capabilities:

- Uses dynamic keyword expansion and tool tips for commands that
  - Automatically completes the command from a popup list
  - Displays complete command syntax as a tool tip
  - Displays parameter options for the command from a popup list
  - Includes a keyword command syntax help
- Checks command syntax and uses color indicators that
  - Validates commands and command syntax
  - Distinguishes between FPGA design constraints and SCOPE legacy constraints
- Allows for standard editor commands, such as copy, paste, comment/un-comment a group of lines, and highlighting of keywords

To use the TCL View of the SCOPE GUI:

1. Click on the TCL View of the SCOPE GUI.
2. You can specify FPGA design constraints as follows:
   - Type the command; after you type three characters a popup menu displays the design constraint command list. Select a command.
   - When you type a dash (-), the options popup menu list is displayed. Select an option.
   - When you hover over a command, a tool tip is displayed for the selected commands.

© 2016 Synopsys, Inc.

Command List Popup Menu

Options List Popup Menu

Command Tool Tip

3. You can also specify a command by using the constraints browser that displays a constraints command list and associated syntax.
   - Double-click on the specified constraint to add the command to the editor window.
   - Then, use the constraint syntax window to help you specify the options for this command.
   - Click on the Hide Syntax Help button at the bottom of the editor window to close the syntax help browser.

4. When you save this file, the constraint file is added to your project in the Constraint directory if the Add to Project option is checked on the New dialog box. Thereafter, you can double-click on the FDC constraint file to open it in the text editor.

Guidelines for Entering and Editing Constraints

1. Enter or edit constraints as follows:
   - For attribute cells in the spreadsheet, click in the cell and select from the pull-down list of available choices.
   - For object cells in the spreadsheet, click in the cell and select from the pull-down list. When you select from the list, the objects automatically have the proper prefixes in the SCOPE window.

© 2016 Synopsys, Inc.

Alternatively, you can drag and drop an object from an HDL Analyst view into the cell, or type in a name. If you drag a bus, the software enters the whole bus (busA). To enter busA[3:0], select the appropriate bus bits before you drag and drop them. If you drag and drop or type a name, make sure that the object has the proper prefix identifiers:

Prefix Identifiers	Description for ...
v:*design_name*	hierarchies or “views” (modules)
c:*clock_name*	clocks
i:*instance_name*	instances (blocks)
p:*port_name*	ports (off-chip)
t:*pin_name*	hierarchical ports, and pins of instantiated cells
b:*name*	bits of a bus (port)
n:*net_name*	internal nets

– For cells with values, type in the value or select from the pull-down list.

– Click the check box in the Enabled column to enable the constraint or attribute.

– Make sure you have entered all the essential information for that constraint. Scroll horizontally to check. For example, to set a clock constraint in the Clocks tab, you must fill out Enabled, Clock, Period, and Clock Group. The other columns are optional. For details about setting different kinds of constraints, go to the appropriate section listed in *Specifying SCOPE Constraints*, on page 151.

2. For common editing operations, refer to this table:

To ...	Do ...
Cut, copy, paste, undo, or redo	Select the command from the popup (hold down the right mouse button to get the popup) or from the Edit menu.

To ...	Do ...
Copy the same value down a column	Select Fill Down (Ctrl-d) from the Edit or popup menus.
Insert or delete rows	Select Insert Row or Delete Rows from the Edit or popup menus.
Find text	Select Find from the Edit or popup menus. Type the text you want to find, and click OK.

3. Edit your constraint file if needed. If your naming conventions do not match these defaults, add the appropriate command specifying your naming convention to the beginning of the file, as shown in these examples:

	Default	You use	Add this to your file
Hierarchy separator	A.B	Slash: A/B	set_hierarchy_separator {/}
Naming bit 5 of bus ABC	ABC[5]	Underscore	bus_naming_style {%s_%d}
Naming row 2 bit 3 of array ABC [2x16]	ABC [2] [3]	Underscore ABC[2_3]	bus_dimension_separator_style {_}

© 2016 Synopsys, Inc.

Specifying Timing Exceptions

You can specify the following timing exception constraints, either from the SCOPE interface or by manually entering the Tcl commands in a file:

- Multicycle Paths – Paths with multiple clock cycles.
- False Paths – Clock paths that you want the synthesis tool to ignore during timing analysis and assign low (or no) priority during optimization.
- Max Delay Paths – Point-to-point delay constraints for paths.

The following shows you how to specify timing exceptions in the SCOPE GUI. For the equivalent Tcl syntax, see Chapter 2, *Tcl Commands* in the *Reference Manual*.

- Defining From/To/Through Points for Timing Exceptions, on page 162
- Defining Multicycle Paths, on page 166
- Defining False Paths, on page 167

For information about resolving timing exception conflicts, see Conflict Resolution for Timing Exceptions, on page 197 in the *Reference Manual*.

Defining From/To/Through Points for Timing Exceptions

For multi-cycle path, false path, and maximum path delay constraints, you must define paths with a combination of From/To/Through points. Whenever the tool encounters a conflict in the way timing-exception constraints are written, see Conflict Resolution for Timing Exceptions, on page 197 to determine how resolution occurs based on the priorities defined.

The following guidelines provide details for defining these constraints. You must specify at least one From, To, or Through point.

- In the From field, identify the starting point for the path. The starting point can be a clock, input or bidirectional port, or register. Only black box output pins are valid. To specify multiple starting points:
  - Such as the bits of a bus, enclose them in square brackets: A[15:0] or A[*].
  - Select the first start point from the HDL Analyst view, then drag and drop this instance into the From cell in SCOPE. For each subsequent

instance, press the Shift key as you drag and drop the instance into the From cell in SCOPE. For example, valid Tcl command format include:

```
set_multicycle_path -from {i:aq i:bq} 2
```

```
set_multicycle_path -from [i:aq i:bq} -through {n:xor_all} 2
```

- In the To field, identify the ending point for the path. The ending point can be a clock, output or bidirectional port, or register. Only black box input pins are valid. To specify multiple ending points, such as the bits of a bus, enclose them in square brackets: B[15:0].
- A single through point can be a combinational net, hierarchical port or instantiated cell pin. To specify a net:
  - Click in the Through field and click the arrow. This opens the Product of Sums (POS) interface.
  - Either type the net name with the n: prefix in the first cell or drag the net from an HDL Analyst view into the cell.
  - Click Save.

  For example, if you specify n:net1, the constraint applies to any path passing through net1.
- To specify an OR when constraining a list of through points, you can type the net names in the Through field or you can use the POS UI. To do this:
  - Click in the Through field and click the arrow. This opens the Product of Sums interface.
  - Either type the first net name in a cell in a Prod row or drag the net from an HDL Analyst view into the cell. Repeat this step along the same row, adding other nets in the Sum columns. The nets in each row form an OR list.

© 2016 Synopsys, Inc.

– Alternatively, select Along Row in the SCOPE POS interface. In an HDL Analyst view, select all the nets you want in the list of through points. Drag the selected nets and drop them into the POS interface. The tool fills in the net names along the row. The nets in each row form an OR list.

– Click Save.

The constraint works as an OR function and applies to any path passing through any of the specified nets. In the example shown in the previous figure, the constraint applies to any path that passes through net1 *or* net2.

- To specify an AND when constraining a list of through points, type the names in the Through field or do the following:

  – Open the Product of Sums interface as described previously.

  – Either type the first net name in the first cell in a Sum column or drag the net from an HDL Analyst view into the cell. Repeat this step down the same Sum column.

– Alternatively, select Down Column in the SCOPE POS interface. In an HDL Analyst view, select all the nets you want in the list of through points. Drag the selected nets and drop them into the POS interface. The tool fills in the net names down the column.

The constraint works as an AND function and applies to any path passing through all the specified nets. In the previous figure, the constraint applies to any path that passes through net1 *and* net3.

- To specify an AND/OR constraint for a list of through points, type the names in the Through field (see the following figure) or do the following:
  – Create multiple lists as described previously.
  – Click Save.

© 2016 Synopsys, Inc.

In this example, the synthesis tool applies the constraint to the paths through all points in the lists as follows:

```
net1 AND net3
OR net1 AND net4
OR net2 AND net3
OR net2 AND net4
```

Defining Multicycle Paths

To define a multicycle path constraint, use the Tcl set_multicycle_path command, or select the SCOPE Delay Paths tab and do the following;

1. From the Delay Type pull-down menu, select Multicycle.
2. Select a port or register in the From or To columns, or a net in the Through column. You must set at least one From, To, or Through point. You can use a combination of these points. See *Defining From/To/Through Points for Timing Exceptions*, on page 162 for more information.
3. Select another port or register if needed (From/To/Through).
4. Type the number of clock cycles or nets (Cycles).
5. Specify the clock period to use for the constraint by going to the Start/End column and selecting either Start or End.

   If you do not explicitly specify a clock period, the software uses the end clock period. The constraint is now calculated as follows:

   multicycle_distance = clock_distance + (cycles -1) * reference_clock_period

   In the equation, clock_distance is the shortest distance between the triggering edges of the start and end clocks, cycles is the number of clock cycles specified, and reference_clock_period is either the specified start clock period or the default end clock period.
6. Check the Enabled box.

Defining False Paths

You define false paths by setting constraints explicitly on the Delay Paths tab or implicitly on the Clock tab. See *Defining From/To/Through Points for Timing Exceptions*, on page 162 for object naming and specifying through points.

- To define a false path between ports or registers, select the SCOPE Delay Paths tab, and do the following:
  - From the Delay Type pull-down menu, select False.
  - Use the pull-down to select the port or register from the appropriate column (From/To/Through).
  - Check the Enabled box.

  The software treats this as an explicit false constraint and assigns it the highest priority. Any other constraints on this path are ignored.
- To define a false path between two clocks, select the SCOPE Clocks tab, and assign the clocks to different clock groups:

  The software implicitly assumes a false path between clocks in different clock groups. This false path constraint can be overridden by a maximum path delay constraint, or with an explicit constraint.
- To set an implicit false path on a path to/from an I/O port, do the following:
  - Select Project->Implementation Options->Constraints.
  - Disable Use clock period for unconstrained IO.

© 2016 Synopsys, Inc.

Finding Objects with Tcl find and expand

The Tcl find and expand commands are powerful search tools that you can use to quickly identify the objects you want. The following sections describe how to use these commands effectively:

- Specifying Search Patterns for Tcl find, on page 168
- Refining Tcl Find Results with -filter, on page 170
- Using the Tcl Find Command to Define Collections, on page 172
- Using the Tcl expand Command to Define Collections, on page 173
- Checking Tcl find and expand Results, on page 174
- Using Tcl find and expand in Batch Mode, on page 176

Once you have located objects with the find or expand commands, you can group them into collections, as described in *Using Collections*, on page 177, and apply constraints to all the objects in the collection at the same time.

Specifying Search Patterns for Tcl find

The usage tips in the following table apply for Tcl find search patterns, regardless of whether you specify the find command in the SCOPE window or as a Tcl command. For full details of the command syntax, refer to Tcl Find Syntax, on page 105 of the *Reference Manual*.

Case rules	Use the case rules for the language from which the object was generated: • VHDL: case-insensitive • Verilog: case-sensitive. Make sure that the object name you type in the SCOPE window matches the Verilog name. For mixed language designs, use the case rules for the parent module. The top level for this example is VHDL, so the following command finds any object in the current view that starts with either a or A: `find {a*} -nocase`
Pattern matching	You have two pattern-matching choices: • Specify the -regexp argument, and then use regular expressions for pattern matching. • Do not specify -regexp, and use only the * and ? wildcards for pattern matching. For hierarchical instance names that use dots as separators, the dots must be escaped with a backward slash (\). For example: abc\.d.
Scope of the search	The scope of the search varies, depending on where you enter the command. If you enter it in the SCOPE environment, the scope of the search is the entire database, but if it is entered in the Tcl window, the default scope of the search is the current HDL Analyst view. See *Comparison of Methods for Defining Collections*, on page 177 for a list of the differences. To set the scope to include the hierarchial levels below the current view in HDL Analyst, use the -hier argument. This example finds all objects below the current view that begin with a: `find {a*} -hier`

© 2016 Synopsys, Inc.

Restricting search by type of object	Use the *-object_type* argument. The following command finds all nets that contain syn: `find -net {*syn*}`
Restricting search by object property	Use the -filter option, as described in *Refining Tcl Find Results with -filter*, on page 170.
Extending search through the hierarchy	Use the -flat option. With this option, the * wildcard matches hierarchy separators as well as regular characters. In the following example, the command finds all instances that include fft_stages in their name, whether it just matches an instance name (inst1_fft_stages_2) or matches a hierarchical name that includes the hierarchy separator (a1.fft_stages_xy): `find -seq -flat *fft_stages* -print`

Refining Tcl Find Results with -filter

The -filter option of the find command lets you further refine the objects located by the find command, according to their properties. When used with other commands, it can be a powerful tool for generating statistics and for evaluation. To filter your find results, follow these steps:

1. Enable property annotation.
   - Select Project->Implementation Options. On the Device tab, enable Annotated Properties for Analyst. Alternatively, use the equivalent Tcl command: set_option -run_prop_extract 1.

– Compile or synthesize the design. After compilation, the tool annotates the design with properties that you can specify with the -filter option, like clock pins.

2. Specify the command using the find pattern as usual, and then specify the -filter option as the last argument:

   find *searchPattern* -filter *expression*
   find *searchPattern* -filter !*expression*

   With this command, the tool first finds objects that match the find *search-Pattern*, and then further filters the found objects the according to the property criteria specified in -filter *expression.* Use the ! character before *expression* if you want to select objects that do not match the properties specified in the filter *expression*.

   expression can be a property name, specified as *@propertyName*, or a property name and value pair, specified as *@propertyName operator value*.

   The following example finds registers in the current view that are clocked by myclk:

   ```
   find -seq {*} -filter {@clock==myclk}
   ```

   For further information about the command, see the following:

For ...	See
Tips on using find search patterns	*Specifying Search Patterns for Tcl find*, on page 168
find syntax details	find, on page 104 in the Reference Manual
find -filter syntax details	find -filter, on page 118in the Reference Manual

© 2016 Synopsys, Inc.

Examples of Useful Find -filter Commands

To find ...	Use a command like this example ...
Instances by slack value	set slack [find –hier –inst {*} –filter @slack <= {-1.000}]
Instances with negative slack	set negFF [find –hier –inst {*} –filter @slack <= {0.0}]
Instances within a slack range	set slackRange [find –hier –inst {*} –filter @slack <= {-1.000} && @slack >= {+1.000}]
Pins by fanout value	set pinResult [find –pin *.CE –hier –filter {@fanout > 15 && @slack < 0.0} -print]
Sequential elements within a clock domain	set clk1FF [find –hier -seq * –filter {@clock==clk1]
Sequential components by primitive type	set fdrse [find –hier –seq {*} –filter @view=={FDRSE}

Using the Tcl Find Command to Define Collections

It is recommended that you use the SCOPE window rather than the Tcl window described here to specify the find command, for the reasons described in *Comparison of Methods for Defining Collections*, on page 177.

The Tcl find command returns a collection of objects. If you want to create a collection of connectivity-based objects, use the Tcl expand command instead of find (*Specifying Search Patterns for Tcl find*, on page 168). This section lists some tips for using the Tcl find command.

1. Create a collection by typing the set command and assigning the results to a variable. The following example finds all instances with a primitive type DFF and assigns the collection to the variable $result:

   ```
   set result [find -hier -inst {*} -filter @ view == DFF]
   ```

   The result is a random number like s:49078472, which is the collection of objects found. The following table lists some usage tips for specifying the find command. For full details of the syntax, refer to Tcl Find Syntax, on page 105 of the *Reference Manual*.

2. Check your find constraints. See *Checking Tcl find and expand Results*, on page 174.

3. Once you have defined the collection, you can view the objects in the collection, using one of the following methods, which are described in more detail in *Viewing and Manipulating Collections with Tcl Commands, on page 183*:
   - Print the collection using the -print option to the find command.
   - Print the collection without carriage returns or properties, using c_list.
   - Print the collection in columns, with optional properties, using c_print.
4. To manipulate the objects in the collection, use the commands described in *Viewing and Manipulating Collections with Tcl Commands, on page 183*.
5. Combine the Tcl find command with other commands:

To ...	Combine with ...
Create or copy objects; create collections	set define_collection
Generate reports for evaluation	c_list c_print
Generate statistics	c_info

Using the Tcl expand Command to Define Collections

The Tcl expand command returns a list of objects that are logically connected between the specified expansion points. This section contains tips on using the Tcl expand command to generate a collection of objects that are related by their connectivity. For the syntax details, refer to expand, on page 125 in the *Command Reference Manual*.

1. Specify at least one from, to, or thru point as the starting point for the command. You can use any combination of these points.

   The following example expands the cone of logic between reg1 and reg2.

   ```
   expand -from {i:reg1} -to {i:reg2}
   ```

   If you only specify a thru point, the expansion stops at sequential elements. The following example finds all elements in the transitive fanout and transitive fanin of a clock-enable net:

   ```
   expand -thru {n:cen}
   ```


© 2016 Synopsys, Inc.

2. To specify the hierarchical scope of the expansion, use the -hier argument.

   If you do not specify this argument, the command only works on the current view. The following example expands the cone of logic to reg1, including instances below the current level:

   ```
   expand -hier -to {i:reg1}
   ```

   If you only specify a thru point, you can use the -level argument to specify the number of levels of expansion. The following example finds all elements in the transitive fanout and transitive fanin of a clock-enable net across one level of hierarchy:

   ```
   expand -thru {n:cen} -level 1
   ```

3. To restrict the search by type of object, use the -*object_type* argument.

   The following command finds all pins driven by the specified pin.

   ```
   expand -pin -from {t:i_and3.z}
   ```

4. To print a list of the objects found, either use the -print argument to the expand command, or use the c_print or c_list commands (see *Creating Collections using Tcl Commands*, on page 180).

Checking Tcl find and expand Results

You must check the validity of the find constraints you set. Use the methods described below.

1. Run the Constraints Checker, either from the UI or at the command line:
   - From the UI, select Run->Constraint Check.
   - At the command line specify the -run constraint_check option to the synthesis tool command. For example: synplify_pro -batch *design*.prj -run constraint_check.
   - If there are issues, the tool reports them in the *design*_cck.rpt report file. Check the Summary and Inapplicable Constraints sections in this file.

2. To list objects selected by the find or expand commands, use one of these methods:
   - List the results by specifying the -print option to the command.

- List the results with the c_list command.
- Print out the results one item per line, using the c_print command.

3. To visually validate the objects selected by the find or expand commands, do the following:
   - Run the command and save the results as a collection.
   - On the SCOPE Collections tab, select the collection.
   - Right-click and choose Select in Analyst. The objects in the collection are highlighted in the RTL view. The example below shows high fanout nets that drive more than 20 destinations.

© 2016 Synopsys, Inc.

Using Tcl find and expand in Batch Mode

When you use the Tcl find command in batch mode, you must specify the open_design command before the find or expand commands.

1. Create the Tcl file to be run in batch mode, making sure that the open_design command precedes the find/expand commands you want.

   You cannot include the Tcl find command in Timing Analyzer scripts. Instead, run Tcl Find to TXT command and use the results.

2. Run the script at the command line. For example, if the file created in step 1 was called analysis.tcl, specify it at the command line, as shown below:

   ```
   synplify_pro -batch analysis.tcl
   ```

   The tool generates two text files as specified, with the results of the two searches.

Using Collections

A collection is a defined group of objects. The advantage offered by collections is that you can operate on all the objects in the collection at the same time. A collection can consist of a single object, multiple objects, or even other collections. You can either define collections in the SCOPE window or type the commands in the Tcl script window.

- Creating and Using SCOPE Collections, on page 178
- Creating Collections using Tcl Commands, on page 180
- Viewing and Manipulating Collections with Tcl Commands, on page 183

Comparison of Methods for Defining Collections

You can enter the find and expand Tcl commands that are used to define collections in either the Tcl script window or in the SCOPE window. It is recommended that you use the SCOPE interface for the reasons outlined below:

	SCOPE Window	Tcl Window
Database used	Top level; includes all objects. See the example below.	Current Analyst view, which might be a lower-level view. If the current view is the Technology view after mapping, objects might be renamed, replicated, or removed.
Persistence	Collection saved in project file.	Collection only valid for the current session; you must redefine it the next time you open the project.
Constraints	Can apply to collection.	Cannot apply to collection.

In the design shown below, if you push down into B, and then type `find -hier a*` in the Tcl window, the command finds a3 and a4. However if you cut and paste the same command into the SCOPE Collections tab, your results would include a1, a2, a3, and a4, because the SCOPE interface uses the top-level database and searches the entire hierarchy.

© 2016 Synopsys, Inc.

Creating and Using SCOPE Collections

The following procedure shows you how to define collections in the SCOPE window. The SCOPE method is preferred over typing the commands in the Tcl window (*Creating Collections using Tcl Commands*, on page 180) for the reasons described in *Comparison of Methods for Defining Collections*, on page 177.

1. Define a collection by doing the following:
   - Open the SCOPE window and click the Collections tab.
   - In the Name column, type a name for the collection.

	Enable	Name	Command
1	☑	find_all	find -hier -inst {*usbSlaveControl.u_endpMux.*}
2	☑	find_reg	find -hier -seq {*usbSlaveControl.u_endpMux.*}
3	☑	find_comb	find -hier -inst {*usbSlaveControl.u_endpMux.*} -filter@is_combination
4			

Collections

   - In the Command column, enter the command. See the *Command Reference* for complete syntax details. Additional information about specifying search patterns is described in *Specifying Search Patterns for Tcl find*, on page 168.

     You can also paste in a command. If you cut and paste a Tcl Find command from the Tcl window into the SCOPE Collections tab, remember that the SCOPE interface works on the top-level database, while the find command in the Tcl window works on the current level displayed in the Analyst view.

Objects in a collection do not have to be of the same type. The collections shown in the preceding figure do the following:

Collection	Finds ...
find_all	All components in the module endpMux
find_reg	All registers in the module endpMux
find_comb	All combinational components under endpMux

The collections you define appear in the SCOPE pull-down object lists, so you can use them to define constraints.

You can crossprobe the objects selected by the find and expand commands by right-clicking and choosing Select in Analyst column. The schematic views highlight the objects located by these commands. For other viewing operations, see *Viewing and Manipulating Collections with Tcl Commands*, on page 183.

Note: Using collections with Tcl control constructs (such as if, for, foreach, and while) can produce unexpected synthesis results. Avoid defining constraints for collections with control constructs, especially since the constraint checker does not recognize these built-in Tcl commands.

2. To create a collection that is made up of other collections, do this:
   - Define the collections as described in the previous step. These collections must be defined before you can concatenate them or add them together in a new collection.
   - To concatenate collections or add to collections, type a name for the new collection in the Name column. Type the appropriate operator command like c_union or c_diff in the Command column. See *Creating Collections using Tcl Commands*, on page 180 for a list of available commands and the *Command Reference* for their syntax.

   The software saves the collection information in the constraint file for the project.

3. To apply constraints to a collection do the following:
   - Define a collection as described in the previous steps.

© 2016 Synopsys, Inc.

– Go to the appropriate SCOPE tab and specify the collection name where you would normally specify the object name. Collections defined in the SCOPE interface are available from the pull-down object lists. The following figure shows the collections defined in step 1 available for setting a false path constraint.

– Specify the rest of the constraint as usual. The software applies the constraint to all the objects in the collection.

Creating Collections using Tcl Commands

This section describes how to use the Tcl collection commands at the command line or in a script instead of entering them in the SCOPE window (*Creating and Using SCOPE Collections*, on page 178). There are differences in operation depending on where the collection commands are entered, and it is recommended that you use the SCOPE window, for the reasons described in *Comparison of Methods for Defining Collections*, on page 177.

For details of the syntax for the commands described here, refer to *Collections, on page 163* in the *Reference Manual*.

1. To create a collection using a Tcl command line command, name it with the set command and assign it to a variable.

   A collection can consist of individual objects, Tcl lists (which can consist of a single element), or other collections. You can embed the Tcl find and expand commands in the set command to locate objects for the collection (see *Using the Tcl Find Command to Define Collections*, on page 172 and *Specifying Search Patterns for Tcl find*, on page 168). The following example creates a collection called my_collection which consists of all the modules (views) found by the embedded find command:

```
set my_collection [find -view {*}]
```

2. To create collections derived from other collections, do the following:
   - Define a new variable for the collection.
   - Create the collection with one of the operator commands from this table:

To ...	Use this command ...
Add objects to a collection	c_union. See *Examples: c_union Command*, on page 182
Concatenate collections	c_union. See *Examples: c_union Command*, on page 182.
Isolate differences between collections	c_diff. See *Examples: c_diff Command*, on page 182.
Find common objects between collections	c_intersect. See *Examples: c_intersect Command*, on page 182.
Find objects that belong to just one collection	c_symdiff. See *Examples: c_symdiff Command*, on page 183.

3. If your Tcl collection includes instances that use special characters, make sure to use extra curly braces or use a backslash to escape the special character.

Curly Braces { }	define_scope_collection GRP_EVENT_PIPE2 {find -seq {EventMux\[2\].event_inst?_sync[*]} -hier} define_scope_collection mytn {find -inst {i:count1.co[*]}}
Backslash Escape Character (\)	define_scope_collection mytn {find -inst i:count1.co\[*\]}

Once you have created a collection, you can do various operations on the objects in the collection (see *Viewing and Manipulating Collections with Tcl Commands*, on page 183), but you cannot apply constraints to the collection.

© 2016 Synopsys, Inc.

Examples: c_union Command

This example adds the reg3 instance to collection1, which contains reg1 and reg2 and names the new collection sumCollection.

```
set sumCollection [c_union $collection1 {i:reg3}]
c_list $sumCollection
   {"i:reg1" "i:reg2" "i:reg3"}
```

If you added reg2 and reg3 with the c_union command, the command removes the redundant instances (reg2) so that the new collection would still consist of reg1, reg2, and reg3.

This example concatenates collection1and collection2 and names the new collection combined_collection:

```
set combined_collection [c_union $collection1 $collection2]
```

Examples: c_diff Command

This example compares a list to a collection (collection1) and creates a new collection called subCollection from the list of differences:

```
set collection1 {i:reg1 i:reg2}
set subCollection [c_diff $collection1 {i:reg1}]
c_print $subCollection
   "i:reg2"
```

You can also use the command to compare two collections:

```
set reducedCollection [c_diff $collection1 $collection2]
```

Examples: c_intersect Command

This example compares a list to a collection (collection1) and creates a new collection called interCollection from the objects that are common:

```
set collection1 {i:reg1 i:reg2}
set interCollection [c_intersect $collection1 {i:reg1 i:reg3}]
c_print $interCollection
   "i:reg1"
```

You can also use the command to compare two collections:

```
set common_collection [c_intersect $collection1 $collection2]
```

Examples: c_symdiff Command

This example compares a list to a collection (collection1) and creates a new collection called diffCollection from the objects that are different. In this case, reg1 is excluded from the new collection because it is in the list and collection1.

```
set collection1 {i:reg1 i:reg2}
set diffCollection [c_symdiff $collection1 {i:reg1 i:reg3}]
c_list $diffCollection
   {"i:reg2" "i:reg3"}
```

You can also use the command to compare two collections:

```
set symdiff_collection [c_symdiff $collection1 $collection2]
```

Examples: Names with Special Characters

Your instance names might include special characters, as for example when your HDL code uses a generate statement. If your instance names have special characters, do the following:

Make sure that you include extra curly braces {}, as shown below:

```
define_scope_collection GRP_EVENT_PIPE2 {find -seq
   {EventMux\[2\].event_inst?_sync[*]} -hier}
define_scope_collection mytn {find -inst {i:count1.co[*]}}
```

Alternatively, use a backslash to escape the special character:

```
define_scope_collection mytn {find -inst i:count1.co\[*\]}
```

Viewing and Manipulating Collections with Tcl Commands

The following section describes various operations you can do on the collections you defined. For full details of the syntax, see Collections, on page 163 in the *Reference Manual*.

1. To view the objects in a collection, use one of the methods described in subsequent steps:
   - Select the collection in an HDL Analyst view (step 2).
   - Print the collection without carriage returns or properties (step 3).
   - Print the collection in columns (step 4).
   - Print the collection in columns with properties (step 5).

© 2016 Synopsys, Inc.

2. To select the collection in an HDL Analyst view, type select *<collection>*.

   For example, select $result highlights all the objects in the $result collection.

3. To print a simple list of the objects in the collection, uses the c_list command, which prints a list like the following:

```
{i:EP0RxFifo.u_fifo.dataOut[0]} {i:EP0RxFifo.u_fifo.dataOut[1]}
{i:EP0RxFifo.u_fifo.dataOut[2]} ...
```

   The c_list command prints the collection without carriage returns or properties. Use this command when you want to perform subsequent Tcl commands on the list. See *Example: c_list Command*, on page 186.

4. To print a list of the collection objects in column format, use the c_print command. For example, c_print $result prints the objects like this:

```
{i:EP0RxFifo.u_fifo.dataOut[0]}
{i:EP0RxFifo.u_fifo.dataOut[1]}
{i:EP0RxFifo.u_fifo.dataOut[2]}
{i:EP0RxFifo.u_fifo.dataOut[3]}
{i:EP0RxFifo.u_fifo.dataOut[4]}
{i:EP0RxFifo.u_fifo.dataOut[5]}
```

5. To print a list of the collection objects and their properties in column format, use the c_print command as follows:

   – Annotate the design with a full list of properties by selecting Project->Implementation Options, going to the Device tab, and enabling Annotated Properties for Analyst. Synthesize the design. If you do not enable the annotation option, properties like clock pins will not be annotated as properties.

   – Check the properties available by right-clicking on the object in the HDL Analyst view and selecting Properties from the popup menu. You see a window with a list of the properties that can be reported.

   – In the Tcl window, type the c_print command with the -prop option. For example, typing c_print -prop slack -prop view -prop clock $result lists the objects in the $result collection, and their slack, view and clock properties.

```
Object Name                      slack  view  clock
{i:EP0RxFifo.u_fifo.dataOut[0]} 0.3223 "FDE" clk
{i:EP0RxFifo.u_fifo.dataOut[1]} 0.3223 "FDE" clk
{i:EP0RxFifo.u_fifo.dataOut[2]} 0.3223 "FDE" clk
{i:EP0RxFifo.u_fifo.dataOut[3]} 0.3223 "FDE" clk
```

```
{i:EP0RxFifo.u_fifo.dataOut[4]} 0.3223 "FDE" clk
{i:EP0RxFifo.u_fifo.dataOut[5]} 0.3223 "FDE" clk
{i:EP0RxFifo.u_fifo.dataOut[6]} 0.3223 "FDE" clk
{i:EP0RxFifo.u_fifo.dataOut[7]} 0.3223 "FDE" clk
{i:EP0TxFifo.u_fifo.dataOut[0]} 0.1114 "FDE" clk
{i:EP0TxFifo.u_fifo.dataOut[1]} 0.1114 "FDE" clk
```

– To print out the results to a file, use the c_print command with the -file option. For example, c_print -prop slack -prop view -prop clock $result -file results.txt writes out the objects and properties listed above to a file called results.txt. When you open this file, you see the information in a spreadsheet format.

6. You can do a number of operations on a collection, as listed in the following table. For details of the syntax, see Collections, on page 163 in the *Reference Manual*.

To ...	Do this ...
Copy a collection	Create a new variable for the copy and copy the original collection to it with the set command. When you make changes to the original, it does not affect the copy, and vice versa. **set** *my_collection_copy* **$***my_collection*
List the objects in a collection	Use the c_print command to view the objects in a collection, and optionally their properties, in column format: "v:top" "v:block_a" "v:block_b" Alternatively, you can use the -print option to an operation command to list the objects.
Generate a Tcl list of the objects in a collection	Use the c_list command to view a collection or to convert a collection into a Tcl list. You can manipulate a Tcl list with standard Tcl commands. In addition, the Tcl collection commands work on Tcl lists. This is an example of c_list results: {"v:top" "v:block_a" "v:block_b"} Alternatively, you can use the -print option to an operation command to list the objects.

© 2016 Synopsys, Inc.

Example: c_list Command

The following provides a practical example of how to use the c_list command. This example first finds all the CE pins with a negative slack that is less than 0.5 ns and groups them in a collection:

```
set get_components_list [c_list [find -hier -pin {*.CE} -filter
@slack < {0.5}]]
```

The c_list command returns a list:

```
{t:EP0RxFifo.u_fifo.dataOut[0].CE}
{t:EP0RxFifo.u_fifo.dataOut[1].CE}
{t:EP0RxFifo.u_fifo.dataOut[2].CE} ...
```

You can use the list to find the terminal (pin) owner:

```
proc terminal_to_owner_instance {terminal_name terminal_type} {
   regsub -all $terminal_type$ $terminal_name {} suffix
   regsub -all {^t:} $suffix {i:} prefix
   return $prefix
   }

foreach get_component $get_components_list {
   append owner [terminal_to_owner_instance $get_component {.CE}]
" "
   }

puts "terminal owner is $owner"
```

This returns the following, which shows that the terminal (pin) has been converted to the owning instance:

```
terminal owner is i:EP0RxFifo.u_fifo.dataOut[0]
i:EP0RxFifo.u_fifo.dataOut[1] i:EP0RxFifo.u_fifo.dataOut[2]
```

Converting SDC to FDC

The sdc2fdc Tcl shell command translates legacy FPGA timing constraints to Synopsys FPGA timing constraints. From the Tcl command line in the synthesis tool, the sdc2fdc command scans the input SDC files and attempts to convert constraints for the implementation.

To run the sdc2fdc Tcl shell command:

1. Load your Project file.
2. From the Tcl command line, type:

   ```
   sdc2fdc
   ```

3. Check the constraint results directory for details about this translation.
4. The new constraints file is automatically updated for your project. Save the new settings.

   The constraint results directory is created at

   projectDir/FDC_constraints/*implName*

   This directory includes the following results files:

   – *topLevel*_translated.fdc – Contains the Synopsys FPGA design constraints (FPGA design constraints and the Synopsys standard timing constraints)

   – *topLevel*|*compilePoint*_translate.log – Contains details about the translation. Translation error messages explain issues and how to fix them. Any translation errors not addressed when you run synthesis appear in the SRR log file, but does not stop synthesis from running.

5. Open the FDC file resulting from translation in the FPGA SCOPE editor to check these constraints and make any changes to them.
6. Run the constraints checker.
7. Save this version of the FDC to run synthesis.

For information about the FDC file, see FDC Constraints, on page 140.

For details about the translated files and troubleshooting guidelines, see sdc2fdc Conversion, on page 143.

© 2016 Synopsys, Inc.

Using the SCOPE Editor (Legacy)

You can use the Legacy SCOPE editor for the SDC constraint files created before release version G-2012.09. However, it is recommended that you translate your SDC files to FDC files to enable the latest version of the SCOPE editor and to utilize the enhanced timing constraint handling in the tool. The latest version of the SCOPE editor automatically formats timing constraints using Synopsys Standard syntax (such as create_clock, and set_multicyle_path).

To do this, add your SDC constraint files to your project and run the following at the command line:

```
% sdc2fdc
```

This feature translates all SDC files in your project.

If you choose to do so, the following procedure shows you how to use the legacy SCOPE editor to create constraints for the constraint file (SDC).

1. Open an existing file for editing.
   - Make sure you have closed the SCOPE window, or you could overwrite previous constraints.
   - Double-click on an existing constraint file (`sdc`) in the project.
   - Select File->Open, set the Files of Type filter to Constraint Files (`sdc`) and open the file you want.
2. Enter the timing or design constraints you need.

Use SCOPE ...	To Define ...
Clocks	Clock frequencies define_clock. See *Defining Clocks, on page 194* for additional information.
	Clock frequency other than the one implied by the signal on the clock pin syn_reference_clock (attribute). See *Defining Clocks, on page 194* for additional information
	Clock domains with asymmetric duty cycles define_clock. See *Defining Clocks, on page 194* for additional information

Use SCOPE ...	To Define ...
Clock to Clock	Edge-to-edge clock delays define_clock_delay. See *Defining Clocks*, on page 194 for additional information
Collections	Set constraints for a group of objects you have defined as a collection with the Tcl command.
Inputs/Outputs	Speed up paths feeding into a register define_reg_input_delay.
	Speed up paths coming from a register define_reg_output_delay.
Registers	Input delays from outside the FPGA define_input_delay. See *Defining Input and Output Constraints (Legacy)*, on page 203 for additional information
	Output delays from your FPGA define_output_delay. See *Defining Input and Output Constraints (Legacy)*, on page 203 for additional information
Delay Paths	Paths with multiple clock cycles define_multicycle_path. See *Defining Multicycle Paths*, on page 166 for additional information
	False paths (certain technologies) define_false_path. See *Defining False Paths (Legacy)*, on page 204 for additional information.
	Path delays define_path_delay. See *Defining From/To/Through Points for Timing Exceptions*, on page 162 for additional information
Attributes	Assign attributes for objects specifying their values
I/O Standards	Define an I/O standard for ports
Compile Points	Specify compile points for your design
Other	Enter newly-supported constraints for advanced users.

© 2016 Synopsys, Inc.

Entering and Editing SCOPE Constraints (Legacy)

Enter constraints directly in the SCOPE window. You can use the Initialize Constraint panel to enter default constraints, and then use the direct method to modify, add, or delete constraints.

The tool also lets you add constraints automatically. For information about auto constraints, see *Using Auto Constraints*, on page 375.

1. Click the appropriate tab at the bottom of the window to enter the kind of constraint you want to create:

To define ...	Click ...
Clock frequency for a clock signal output of clock divider logic A specific clock frequency that overrides the global frequency	Clocks
Edge-to-edge clock delay that overrides the automatically calculated delay.	Clock to Clock
Constraints for a group of objects you have defined as a collection with the Tcl command. For details, see *Creating and Using SCOPE Collections*, on page 178.	Collections
Input/output delays that model your FPGA input/output interface with the outside environment	Inputs/ Outputs
Delay constraints for paths feeding into/out of registers	Registers
Paths that require multiple clock cycles	Delay Paths
Paths to ignore for timing analysis (false paths)	Delay Paths
Maximum delay for paths	Delay Paths
Attributes, like syn_reference_clock, that were not entered in the source files	Attributes
I/O standards for any port in the I/O Standard panel of the SCOPE window.	I/O Standard
Compile points in a top-level constraint file. See *Synthesizing Compile Points*, on page 468 for more information about compile points.	Compile Points
Place and route tool constraints Other constraints not used for synthesis, but which are passed to other tools. For example, multiple clock cycles from a register or input pin to a register or output pin	Other

The SCOPE window displays columns appropriate to the kind of constraint you picked. You can now enter constraints using the wizard, or work directly in the SCOPE window.

2. Save the file by clicking the Save icon and naming the file.

   The software creates a TCL constraint file (`sdc`). See *Working with Constraint Files*, on page 61 for information about the commands in this file.

3. To apply the constraints to your design, you must add the file to the project now or later.
   - Add it immediately by clicking Yes in the prompt box that opens after you save the constraint file.
   - Add it later, following the procedure for adding a file described in *Making Changes to a Project*, on page 72.

Specifying SCOPE Timing Constraints (Legacy)

You can define timing constraints in the SCOPE GUI, which automatically generates a Tcl constraints file, or manually with a text editor, as described in *Using a Text Editor for Constraint Files (Legacy)*, on page 63.

The SCOPE GUI is much easier to use, and you can define various timing constraints in it. For the equivalent Tcl syntax, see Chapter 2, *Tcl Commands* in the *Reference Manual*. See the following for different timing constraints:

- Entering Default Constraints, on page 192
- Setting Clock and Path Constraints, on page 192
- Defining Clocks, on page 194
- Defining Input and Output Constraints (Legacy), on page 203
- Specifying Standard I/O Pad Types, on page 156

To set constraints for timing exceptions like false paths and multicycle paths, see *Specifying Timing Exceptions*, on page 162.

© 2016 Synopsys, Inc.

Entering Default Constraints

To edit or set individual constraints, or to create constraints in the Other tab, work directly in the SCOPE window (*Setting Clock and Path Constraints*, on page 192). For auto constraints in the Synplify Pro tool, see *Using Auto Constraints*, on page 375. To apply the constraints, add the file to the project according to the procedure described in *Making Changes to a Project*, on page 72. The constraints file has an `sdc` extension. See *Working with Constraint Files*, on page 61 for more information about constraint files.

Setting Clock and Path Constraints

The following table summarizes how to set different clock and path constraints from the SCOPE window. For information about setting compile point constraints or attributes, see *Synthesizing Compile Points*, on page 468 for more information about compile points and *Specifying Attributes Using the SCOPE Editor*, on page 102. For information about setting default constraints, see *Entering Default Constraints*, on page 192.

To define ...	Pane	Do this to set the constraint ...
Clocks	Clock	Select the clock object (Clock). Specify a clock name (Clock Alias), if required. Type a frequency value (Frequency) or a period (Period). Change the default Duty Cycle or set Rise/Fall At, if needed. Change the default clock group, if needed Check the Enabled box. See *Defining Clocks*, on page 194 for information about clock attributes.
Virtual clocks	Clock	Set the clock constraints as described for clocks, above. Check the Virtual Clock box.
Route delay	Clock Inputs/ Outputs Registers	Specify the route delay in nanoseconds. Refer to *Defining Clocks*, on page 194, *Defining Input and Output Constraints (Legacy)*, on page 203 and the Register Delays section of this table details.
Edge-to-edge clock delay	Clock to Clock	Select the starting edge for the delay constraint (From Clock Edge). Select the ending edge for the constraint (To Clock Edge). Enter a delay value. Mark the Enabled check box.

To define ...	Pane	Do this to set the constraint ...
Input/output delays	Inputs/ Outputs	See *Defining Input and Output Constraints (Legacy)*, on page 203 for information about setting I/O constraints.
Register delays	Registers	Select the register (Register). Select the type of delay, input or output (Type). Type a delay value (Value). Check the Enabled box. If you do not meet timing goals after place-and-route, adjust the clock constraint as follows: • In the Route column for the constraint, specify the actual route delay (in nanoseconds), as obtained from the place-and-route results. Adding this constraint is equivalent to putting a register delay on that input register. • Resynthesize your design.
Maximum path delay	Delay Path	Select the Delay Type path of Max Delay. Select the port or register (From/Through). See *Defining From/To/Through Points for Timing Exceptions*, on page 162 for more information. Select another port or register if needed (To/Through). Set the delay value (Max Delay). Check the Enabled box.
Multi-cycle paths	Delay Paths	See *Defining Multicycle Paths*, on page 166.
False paths	Delay Paths Clock to Clock	See *Defining False Paths (Legacy)*, on page 204 for details.

© 2016 Synopsys, Inc.

To define ...	Pane	Do this to set the constraint ...
Global attributes	Attributes	Set Object Type to <global>. Select the object (Object). Set the attribute (Attribute) and its value (Value). Check the Enabled box.
Attributes	Attributes	Do either of the following: • Select the type of object (Object Type). Select the object (Object). Set the attribute (Attribute) and its value (Value). Check the Enabled box. • Set the attribute (Attribute) and its value (Value). Select the object (Object). Check the Enabled box.
Other	Other	Type the TCL command for the constraint (Command). Enter the arguments for the command (Arguments). Check the Enabled box.

Defining Clocks

Clock frequency is the most important timing constraint, and must be set accurately. If you are planning to auto constrain your design (*Using Auto Constraints*, on page 375), do not define any clocks. The following procedures show you how to define clocks and set clock groups and other constraints that affect timing:

- Defining Clock Frequency, on page 196
- Constraining Clock Enable Paths, on page 200
- Defining Other Clock Requirements, on page 201

Defining Clock Frequency

This section shows you how to define clock frequency either through the GUI or in a constraint file. See *Defining Other Clock Requirements*, on page 201 for other clock constraints. If you want to use auto constraints, do not define your clocks.

1. Define a realistic global frequency for the entire design, either in the Project view or the Constraints tab of the Implementation Options dialog box.

   This target frequency applies to all clocks that do not have specified clock frequencies. If you do not specify any value, a default value of 1 MHz (or 1000 ns clock period) applies to all timing paths whenever the clock associated with both start and end points of the path is not specified. Each clock that uses the global frequency is assigned to its own clock group. See *Defining Other Clock Requirements*, on page 201 for more information about clock group settings.

   The global frequency also applies to any purely combinational paths. The following figure shows how the software determines constraints for specified and unspecified start or end clocks on a path:

If clkA is ...	And clkB is ...	The effect for logic C is ...
Undefined	Defined	The path is unconstrained unless you specify that clkB be constrained to the inferred clock domain for clkA
Defined	Undefined	The path is unconstrained unless you specify that clkA be constrained to the inferred clock domain for clkB.
Defined	Defined	For related clocks in the same clock group, the relationship between clocks is calculated; all other paths between the clocks are treated as false paths.
Undefined	Undefined	The path is unconstrained.

© 2016 Synopsys, Inc.

2. Define frequency for individual clocks on the Clocks tab of the SCOPE window (define_clock constraint).
   - Specify the frequency as either a frequency in the Frequency column (-freq Tcl option) or a time period in the Period column (-period Tcl option). When you enter a value in one column, the other is calculated automatically.
   - For asymmetrical clocks, specify values in the Rise At (-rise) and Fall At (-fall) columns. The software automatically calculates and fills out the Duty Cycle value.

   The software infers all clocks, whether declared or undeclared, by tracing the clock pins of the flip-flops. However, it is recommended that you specify frequencies for all the clocks in your design. The defined frequency overrides the global frequency. Any undefined clocks default to the global frequency.

3. Define internal clock frequencies (clocks generated internally) on the SCOPE Clocks tab (define_clock constraint). Apply the constraint according to the source of the internal clock.

Source	Add SCOPE constraint/define_clock to ...
Register	Register.
Instance, like a PLL or clock DLL	Instance. If the instance has more than one clock output, apply the clock constraints to each of the output nets, making sure to use the n: prefix (to signify a net) in the SCOPE table.
Combinational logic	Net. Make sure to use the n: prefix in the SCOPE interface.

4. For signals other than clocks, define frequencies with the syn_reference_clock attribute. You can add this attribute on the SCOPE Attributes tab, as follows:
   - Define a dummy clock on the Clocks tab (define_clock constraint).
   - Add the syn_reference_clock attribute (Attributes tab) to the affected registers to apply the clock. In the constraint file, you can use the Find command to find all registers enabled by a particular signal and then apply the attribute:

```
define_clock -virtual dummy -period 40.0
define_attribute {find -seq * -hier -filter @(enable == en40)}
   syn_reference_clock dummy
```

In earlier releases, limited clocking resources might have forced you to use an enable signal as a clocking signal, and use the syn_reference_clock attribute to define an enable frequency. However, because of changes in the reporting of clock start and end points, it is recommended that you use a multicycle path constraint instead for designs that use an enable signal and a global clock, and where paths need to take longer than one clock cycle. See *Constraining Clock Enable Paths*, on page 200 for a detailed explanation.

Note: This method is often used for designs that have an enable signal and a global clock, and where paths need to take longer than one clock cycle. The registers in the design are actually connected to the global clock; however, the tool treats the registers as having a virtual clock at the frequency of the enable signal.

Using this method to constrain paths for technologies with clock buffer delays requires careful analysis with the Timing Analysis Reports (STA). The virtual clock does not include clock buffer delays. However, non-virtual clocks that pass through clock buffers do include clock buffer delays. The register that generates the enable signal is on the non-virtual clock domain, whereas the registers connected to the enable signal are on the virtual clock domain. Timing analysis shows that the enable signal is on the path between the non-virtual and virtual clock domains. For the actual design, the enable signal is on a path in the non-virtual clock domain. Any paths between virtual and non-virtual clocks are reported with a clock buffer delay on the non-virtual clock. This may result in the critical path reporting negative slack.

In the following example, the path comes from a register on a non-virtual clock and goes to a register on a virtual clock.

```
Path information for path number 1:
   Requested Period:3.125
   - Setup time: 0.229
   = Required time: 2.896

   - Propagation time: 1.448
   - Clock delay at starting point: 1.857
   = Slack (critical: -0.409

   Number of logic level(s): 0
```


© 2016 Synopsys, Inc.

```
   Starting point:  SourceFlop / Q
   Ending point:  DestinationFlop / CE
   The start point is clocked by Non-VirtualClock [rising] on pin C
   The end point is clocked by VirtualClock [rising] on pin C

The path is reported with a negative slack of -0.49.
```

Timing analysis specifies a Clock delay at starting point that is the delay in the clock buffers of the non-virtual clock, but not a Clock delay at ending point. In the actual design, this delay exists at the end point. Since the clock end point is a virtual clock, the clock buffer delay creates a negative slack that does not exist in the actual design.

It is recommended that you use a multicycle path constraint instead to constrain all registers driven by the enable signal in the design.

5. After synthesis, check the Performance Summary section of the log file for a list of all the defined and inferred clocks in the design.

6. If you do not meet timing goals after place-and-route, adjust the clock constraint as follows:
   - Open the SCOPE window with the clock constraint.
   - In the Route column for the constraint, specify the actual route delay (in nanoseconds), as obtained from the place-and-route results. Adding this constraint is equivalent to putting a register delay on all the input registers for that clock.
   - Resynthesize your design.

Constraining Clock Enable Paths

You might use an enable signal as a clocking signal if you have limited clocking resources. If the enable is slower than the clock, you can ensure more accuracy by defining the enable frequency separately, instead of slowing down the clock frequency. If you slow down the clock frequency, it affects all other registers driven by the clock, and can result in longer run times as the tool tries to optimize a non-critical path.

There are two ways to define clock enables:

- By setting a multicycle path constraint to constrain all flip-flops driven by the clock enable signal (see *Defining Multicycle Paths*, on page 166). This is the recommended method.
- Using the syn_reference_clock attribute, as described in step 4 of *Defining Clock Frequency*, on page 196. Although this method was used in earlier releases, it is not recommended any more because of changes in the way the clock start and end points are reported. An explanation of the clock start and end points reporting follows.

Clock Domains for Clock Enables Defined with syn_reference_clock

When you use the syn_reference_clock attribute to constrain an enable signal, you are telling the tool to treat the flip-flops as if they had a virtual clock at the frequency of the enable signal, when the flip-flops are actually connected to the global clock. This could result in critical paths being reported with negative slack.

The flip-flop that generates the enable signals is in the non-virtual clock domain. The flip-flops that are connected to the enable signal are in the virtual clock domain. The timing analyst considers the enable signal to be on a path that goes between a non-virtual clock domain and a virtual clock domain. In the actual circuit, the enable signal is on a path within a non-virtual clock domain. The timing analyst reports any paths between virtual and non-virtual clocks with a clock buffer delay on the non-virtual clock. This is why critical paths might be reported with negative slack.

If you use this method to constrain paths in a technology that includes clock buffer delays, you must carefully analyze the timing analysis reports. The virtual clock does not include clock buffer delays, but any non-virtual clock that passes through clock buffers will include clock buffer delays.

© 2016 Synopsys, Inc.

The following is an example report of a path from a clock enable, starting from a flip-flop on a non-virtual clock to a flip-flop on a virtual clock. The path is reported with a negative slack of -0.49.

```
Path information for path number 1:
   Requested Period: 3.125
   - Setup time: 0.229
   = Required time:2.896

   - Propagation time:  1.448
   - Clock delay at starting point:     1.857
   = Slack (critical) :  -0.409

   Number of logic level(s):     0
   Starting point:SourceFlop/ Q
   Ending point:DestinationFlop / CE

The start point is clocked by Non-VirtualClock [rising]on pin C

The end point is clocked by VirtualClock [rising] on pin C
```

This timing analysis report includes a Clock delay at starting point, but does not include Clock delay at ending point. The clock delay at the starting point is the delay in the clock buffers of the non-virtual clock. In the actual circuit, this delay would also be at the ending point and not affect the calculation of slack. However as the ending clock is a virtual clock, the clock buffer delay ends up creating a negative slack that does not exist in the actual circuit.

This report is a result of defining the clock enables with the syn_reference_clock attribute. This is why it is recommended that you use multicycle paths to constrain all the flip-flops driven by the enable signal.

Defining Other Clock Requirements

Besides clock frequency (described in *Defining Clock Frequency*, on page 196), you can also set other clock requirements, as follows:

- If you have limited clock resources, define clocks that do not need a clock buffer by attaching the syn_noclockbuf attribute to an individual port, or the entire module/architecture.
- Define the relationship between clocks by setting clock domains. By default, each clock is in a separate clock group named default_clkgroup<*n*> with a sequential number suffix.

  - On the SCOPE Clocks tab, group related clocks by putting them into the same clock group. Use the Clock Group field to assign all related clocks to the same clock group.
  - Make sure that unrelated clocks are in different clock groups. If you do not, the software calculates timing paths between unrelated clocks in the same clock group, instead of treating them as false paths.
  - Input and output ports that belong to the System clock domain are considered a part of every clock group and will be timed. See *Defining Input and Output Constraints (Legacy)*, on page 203 for more information.

  The software does not check design rules, so it is best to define the relationship between clocks as completely as possible.

- Define all gated clocks with the define_clock constraint.

  Avoid using gated clocks to eliminate clock skew. If possible, move the logic to the data pin instead of using gated clocks. If you do use gated clocks, you must define them explicitly, because the software does not propagate the frequency of clock ports to gated clocks.

  To define a gated clock, attach the define_clock constraint to the clock source, as described above for internal clocks. To attach the constraint to a keepbuf (a keepbuf is a placeholder instance for clocks generated from combinational logic), do the following:

  - Attach the syn_keep attribute to the gated clock to ensure that it retains the same name through changes to the RTL code.
  - Attach the define_clock constraint to the net or pin connected to the keepbuf instance generated for the gated clock.

- Specify edge-to-edge clock delays on the Clock to Clock tab (define_-clock_delay).
- After synthesis, check the Performance Summary section of the log file for a list of all the defined and inferred clocks in the design.

© 2016 Synopsys, Inc.

Defining Input and Output Constraints (Legacy)

In addition to setting I/O delays in the SCOPE window as described in *Setting Clock and Path Constraints*, on page 192, you can also set the Use clock period for unconstrained IO option.

- Open the SCOPE window, click Inputs/Outputs, and select the port (Port). You can set the constraint for
  - All inputs and outputs (globally in the top-level netlist)
  - For a whole bus
  - For single bits

  You can specify multiple constraints for the same port. The software applies all the constraints; the tightest constraint determines the worst slack. If there are multiple constraints from different levels, the most specific overrides the more global. For example, if there are two bit constraints and two port constraints, the two bit constraints override the two port constraints for that bit. The other bits get the two port constraints.
- Specify the constraint value in the SCOPE window:
  - Select the type of delay: input or output (Type).
  - Type a delay value (Value).
  - Check the Enabled box, and save the constraint file in the project.

  Make sure to specify explicit constraints for each I/O path you want to constrain.
- To determine how the I/O constraints are used during synthesis, do the following:
  - Select Project->Implementation Options, and click Constraints.
  - To use only the explicitly defined constraints disable Use clock period for unconstrained IO.
  - To synthesize with all the constraints, using the clock period for all I/O paths that do not have an explicit constraint enable Use clock period for unconstrained IO.
  - Synthesize the design. When you forward-annotate the constraints, the constraints used for synthesis are forward-annotated for place-and-route.

- Input or output ports with explicitly defined constraints, but without a reference clock (-ref option) are included in the System clock domain and are considered to belong to every defined or inferred clock group.
- If you do not meet timing goals after place-and-route and you need to adjust the input constraints; do the following:
  - Open the SCOPE window with the input constraint.
  - In the Route column for the input constraint, specify the actual route delay in nanoseconds, as obtained from the place-and-route results. Adding this constraint is equivalent to putting a register delay on the input register.
  - Resynthesize your design.

Defining False Paths (Legacy)

You define false paths by setting constraints explicitly on the Delay Paths tab or implicitly on the Clock and Clock to Clock tabs. See *Defining From/To/Through Points for Timing Exceptions*, on page 162 for object naming and specifying through points.

- To define a false path between ports or registers, select the SCOPE Delay Paths tab, and do the following:
  - From the Delay Type pull-down menu, select False.
  - Use the pull-down to select the port or register from the appropriate column (From/To/Through).
  - Check the Enabled box.

  The software treats this as an explicit false constraint and assigns it the highest priority. Any other constraints on this path are ignored.
- To define a false path between two clocks, select the SCOPE Clocks tab, and assign the clocks to different clock groups:

  The software implicitly assumes a false path between clocks in different clock groups. This false path constraint can be overridden by a maximum path delay constraint, or with an explicit constraint.
- To define a false path between two clock edges, select the SCOPE Clock to Clock tab, and do the following:
  - Specify one clock as the starting clock edge (From Clock Edge).
  - Specify the other clock as the ending clock edge (To Clock Edge).

© 2016 Synopsys, Inc.

  - Click in the Delay column, and select false.
  - Mark the Enabled check box.

  Use this technique to specify a false path between any two clocks, regardless of clock groups. This constraint can be overridden by a maximum delay constraint on the same path

- To override an implicit false path between any two clocks described previously, set an explicit constraint between the clocks by selecting the SCOPE Clock to Clock tab, and doing the following:
  - Specify the starting (From Clock Edge) and ending clock edges (To Clock Edge).
  - Specify a value in the Delay column.
  - Mark the Enabled check box.

  The software treats this as an explicit constraint. You can use this method to constrain a path between any two clocks, regardless of whether they belong to the same clock group.

- To set an implicit false path on a path to/from an I/O port, do the following:
  - Select Project->Implementation Options->Constraints.
  - Disable Use clock period for unconstrained IO.

CHAPTER 6

Synthesizing and Analyzing the Results

This chapter describes how to run synthesis, and how to analyze the log file generated after synthesis. See the following:

- Synthesizing Your Design, on page 208
- Checking Log File Results, on page 213
- Handling Messages, on page 227

© 2016 Synopsys, Inc.

Synthesizing Your Design

Once you have set your constraints, options, and attributes, running synthesis is a simple one-click operation. See the following:

- Running Logic Synthesis
- Using Up-to-date Checking for Job Management

Running Logic Synthesis

When you run logic synthesis, the tool compiles the design and then maps it to the technology target you selected.

1. If you want to compile your design without mapping it, select Run-> Compile Only or press F7.

   A compiled design has the RTL mapping, and you can view the RTL view. You might want to just compile the design when you are not ready to synthesize the design, but when you need to use a tool that requires a compiled design, like the SCOPE interface.

2. To synthesize the logic, set all the options and attributes you want, and then click Run.

Using Up-to-date Checking for Job Management

Synthesis is becoming more complex and consists of running many jobs. Often, part or all of the job flow is already up-to-date and rerunning the job may not be necessary. For large designs that may take hours to run, up-to-date checking can reduce the time for rerunning jobs.

Up-to-date checking is run for all synthesis design flows. However, for the Hierarchical Project Management flows, up-to-date checking is an essential feature. For example, if a project contains four sub-projects and only one project is modified, then the other three projects do not need to be rerun. This saves in overall runtime.

Up-to-date checking includes the following:

- The GUI launches mapper modules (pre-mapping and technology mapping) and saves the intermediate netlists and log files in the synwork and synlog folders, respectively.

- After each individual module run completes, the GUI optionally copies the contents of these intermediate log files from the synlog folder and adds them to the Project log file (rev_1/*projectName*.srr). To set this option, see *Copy Individual Job Logs to the SRR Log File*, on page 211.
- If you re-synthesize the design and there are no changes to the inputs (HDL, constraints, and Project options):
  - The GUI does not rerun pre-mapping and technology mapping and no new netlist files are created.
  - In the HTML log file, the GUI adds a link that points to the existing pre-mapping and mapping log files from the previous run. Double-click on this link (`@L:` indicates the link) to open the new text file window.

    If you open the text log file, the link is a relative path to the implementation folder for the pre-mapping and mapping log files from the previous run.

Note: Also, the GUI adds a note that indicates mapping will not be re-run and to use the Run->Resynthesize All option in the Project view to force synthesis to be run again.

```
##########################################################]

Synopsys Pre-mapping Report
@N: :  | premapping output is up to date. No run necessary.
To force a re-synthesis, select [Resynthesize All] in menu [Run].

Log file from previous run:
@L:top_premapping.srr
##########################################################]
```

```
ersion map610dev, Build
ll Rights Reserved

d\gated_clocks\test.sdc

le <C:\builds\syn201103_
table from file <C:\buil
```

```
@W:"c:\syn_hier\fixed\gated_clocks\myreg.v":18:14:18:22|Net U_reg
@W:"c:\syn_hier\fixed\gated_clocks\top.v":17:14:17:24|Net U_gt_re

Finished Pre Mapping Phase. (Time elapsed 0h:00m:00s; Memory used

@N: BN225 |Writing default property annotation file C:\syn_hier\f
Pre Mapping successful!
Process took 0h:00m:01s realtime, 0h:00m:01s cputime
# Mon Jan 24 18:39:06 2011

##########################################################]
```


© 2016 Synopsys, Inc.

As the job is running, you can click in the job status field of the Project view to bring up the Job Status display. When you rerun synthesis, the job status identifies which modules (pre-mapping or mapping) are up-to-date.

Job Status for Synthesis Run

Job Status for Re-synthesis Run

See also:

- Copy Individual Job Logs to the SRR Log File
- Limitations and Risks

Copy Individual Job Logs to the SRR Log File

By default, up-to-date checking uses links in the log file (`srr`) to individual job logs. To change this option so that individual job logs are always appended to the main log file (`srr`), do the following:

1. Select Options->Project View Options from the Project menu.
2. On the Project View Options dialog box, scroll down to the Use links in SRR log file to individual job logs option.
3. Use the pull-down menu, and select off.

© 2016 Synopsys, Inc.

Limitations and Risks

Up-to-date checking limitations and risks include the following:

- Compiler up-to-date checks are done internally by the compiler and with no changes to the compiler reporting structure.
- GUI up-to-date checks use timestamp information of its input files to decide when mapping is re-run. Be aware that:
  - The GUI uses netlist files (`srs` and `srd`) from the synwork folder for timestamp checks. If you delete an `srs` file from the implementation folder, this does not trigger compiler or mapper re-runs. You must delete netlist files from the synwork folder instead.
  - The copy command behaves differently on Windows and Linux. On Windows, the timestamp does not change if you copy a file from one directory to another. But on Linux (and MKS shell), the timestamp information gets changed.
- When running a design, the up-to-date checking feature automatically determines if the design needs to be re-synthesized. However, when you modify constraints in a Tcl file sourced within the constraints file, the software is not aware of these changes and does not force the design to be re-synthesized.

Checking Log File Results

You can check the log file for information about the synthesis run. In addition, the user interface has a Tcl Script window, that echoes each command as it is run. The following describe different ways to check the results of your run:

- Viewing and Working with the Log File, on page 213
- Accessing Specific Reports Quickly, on page 217
- Accessing Results Remotely, on page 219
- Analyzing Results Using the Log File Reports, on page 223
- Using the Watch Window, on page 224
- Checking Resource Usage, on page 225

Viewing and Working with the Log File

The log file contains the most comprehensive results and information about a synthesis run. The default log file is in HTML format, but there is a text version available too.

For users who only want to check a few critical performance criteria, it is easier to use the Watch Window (see *Using the Watch Window*, on page 224) instead of the log file. For details, read through the log file.

1. To open the log file, use one of these listed methods, according to the format you want:

HTML	• Select View->Log File. • Click the View Log button in the Project window. • Double-click the *designName*.htm file in the Implementation Results view.
Text	Double-click the *designName*.srr file in the Implementation Results view. To set the text file version to open by default instead of the HTML version, select Options->Project View Options, and toggle off the View log file in HTML option.

The log file lists the compiled files, details of the synthesis run, and includes color-coded errors, warnings and notes, and a number of

© 2016 Synopsys, Inc.

reports. For information about the reports, see *Analyzing Results Using the Log File Reports*, on page 223.

2. Navigate the log file to view specific pieces of information.

   For quicker access to specific log information, use alternative access methods, described in *Accessing Specific Reports Quickly*, on page 217 instead of the ones described here.

   – Use the panel on the left of the HTML log file to navigate to the section you want. You can use the Find button and the search field at the bottom of this panel to search the headings.

– To search the body of the log file, use Control-f or the Edit->Find command. See *Viewing and Working with the Log File*, on page 213 for details.

– To add bookmarks or for general information about working in an editing window, see *Editing HDL Source Files with the Built-in Text Editor*, on page 46.

The areas of the log file that are most important are the warning messages and the timing report. The log file includes a timing report that lists the most critical paths. The synthesis products also let you generate a report for a path between any two designated points, see Generating Custom Timing Reports with STA, on page 365. The following table lists places in the log file you can use when searching for information.

To find ...	Search for ...
Notes	@N or look for blue text
Warnings and errors	@W and @E, or look for purple and red text respectively
Performance summary	Performance Summary
The beginning of the timing report	START TIMING REPORT

© 2016 Synopsys, Inc.

To find ...	Search for ...
Detailed information about slack times, constraints, arrival times, etc.	Interface Information
Resource usage	Resource Usage Report. See *Checking Resource Usage*, on page 225.
Gated clock conversions	Gated clock report

3. Resolve any errors and check all warnings.

   You must fix errors, because you cannot synthesize a design with errors. Check the warnings and make sure you understand them. See *Checking Results in the Message Viewer*, on page 227 for information. Notes are informational and usually can be ignored. For details about crossprobing and fixing errors, see *Handling Warnings*, on page 237, *Editing HDL Source Files with the Built-in Text Editor*, on page 46, and Crossprobing from the Text Editor Window, on page 330.

   If you see Automatic dissolve at startup messages, you can usually ignore them. They indicate that the mapper has optimized away hierarchy because there were only a few instances at the lower level.

4. If you are trying to find and resolve warnings, you can bookmark them as shown in this procedure:
   - Select Edit->Find or press Ctrl-f.
   - Type @W as the criteria on the Find form and click Mark All. The software inserts bookmarks at every line with a warning. You can now page through the file from bookmark to bookmark using the commands in the Edit menu or the icons in the Edit toolbar. For more information on using bookmarks, see *Editing HDL Source Files with the Built-in Text Editor*, on page 46.

5. To crossprobe from the log file to the source code, click on the file name in the HTML log file or double-click on the warning text (not the ID code) in the ASCII text log file.

Accessing Specific Reports Quickly

The log file contains all the results from the synthesis run, but you might want to hone in on specific information. Instead of browsing the log file to find the information you need, you can use the techniques described below:

1. To quickly view specific pieces of log information, go to the Project Status window and click the appropriate links to display the corresponding reports or specific parts of the log file.

Timing reports	Click Detailed Report or Timing Report View in the Timing Summary panel.
Log at different stages	Click Detailed Report in the Run Status panel.
Area reports	Click Detailed Report or Hierarchical Area Report in the Area Summary panel.
High reliability reports	Click Detailed Report in the High Reliability Report panel.
Optimizations	Click Detailed Report in the Optimizations Summary panel.

The Detailed Report links display parts of the log file, and the other links go to special view windows for different kinds of reports. See The Project Results View, on page 35 for more information about different reports that can be accessed from the Project Results view.

© 2016 Synopsys, Inc.

2. To view timing information, use one of these methods:
   - Set important timing parameters to monitor in the Watch window, like slack and frequency. See *Using the Watch Window*, on page 224 for details.
   - Click View Log in the Project view and navigate to the appropriate section in the log file.

3. To view messages, use any of the following methods
   - From the Run Status panel in the Project Status window, click the link that lists the number of errors, warnings, or notes at different design stages. The Message window opens. Click the message ID to get more information about the error and how to fix it.

     This is the quickest method to narrow down the list of messages and access the one you want.

     The numbers of notes, errors, and warnings reported in the Run Status panel might not match the numbers displayed in the Messages window if the design contains compile points. The numbers reported are for the top level.

   - Click the Messages tab at the bottom of the Project view to open a window with a list of all the notes, errors and warnings. See *Checking Results in the Message Viewer*, on page 227 for more information about using this window.

– Open the log file, locate the message, and click the message ID. The log file includes all the results from the run, so it could be harder to locate the message you want.

Accessing Results Remotely

You can access the log file results remotely from various mobile devices. For example, you can use this feature to run synthesis for jobs with long run times and then check the results of the synthesis run later from anywhere. The Project Status report files can be accessed from any browser without bringing up the synthesis tool.

To access the log file remotely, do the following:

1. Select Options->Project Status Page Location from the Project menu and select the implementation for which you want the reports.

© 2016 Synopsys, Inc.

2. Set the location for storing the project status page, using either of these methods:
   – Enable Save to different location and specify a path for the location of the status page. This allows you to save the status reports in different locations.

– Use an environment variable by enabling Use Environment Variable SYNPLIFY_REMOTE_REPORT_LOCATION.

Windows	Enable Use Environment Variable SYNPLIFY_REMOTE_REPORT_LOCATION. Specify the variable name SYNPLIFY_REMOTE_REPORT_LOCATION and the location you want from the Control Panel on the Edit User Variable dialog box.
Linux	Specify setenv SYNPLIFY_REMOTE_REPORT_LOCATION *pathLocation* in the .cshrc file. Enable Use Environment Variable SYNPLIFY_REMOTE_REPORT_LOCATION.

If you use this option, you must restart the tool the first time, since the environment variable is not applied dynamically. This option always saves the status report to the location indicated by the variable.

© 2016 Synopsys, Inc.

3. Click OK.

4. Run synthesis.

   The status reports are saved to the location you specified for your project. For example:

   `C:\synResults\tutorial\rev_1`

5. Access the location you set up from any browser on a mobile device (for example, a smart phone or tablet).
   - Access the location you set in the previous steps.
   - Open the *projectName/implementationName*/index.html file with any browser.

     Your company may need to set up a location on its internal internet, where the status reports can be saved and later accessed with a URL address.

Analyzing Results Using the Log File Reports

The log file contains technology-appropriate reports like timing reports, resource usage reports, and net buffering reports, in addition to any notes, errors, and warning messages.

1. To analyze timing results, do the following:
   - View the Timing Report (Performance Summary section of the log file) and check the slack times. See Handling Negative Slack, on page 364 for details.
   - Check the detailed information for the critical paths, including the setup requirements at the end of the detailed critical path description. You can crossprobe and view the information graphically and determine how to improve the timing.
   - In the HTML log file, click the link to open up the HDL Analyst view for the path with the worst slack.

   To generate timing information for a path between any two designated points, see Generating Custom Timing Reports with STA, on page 365.

2. To check buffers, do the following:
   - Check the report by going to the Net Buffering Report section of the log file.
   - Check the number of buffers or registers added or replicated and determine whether this fits into your design optimization strategy.

3. To check logic resources, check the Resource Usage Report section at the end of the log file, as described in *Checking Resource Usage*, on page 225.

© 2016 Synopsys, Inc.

Using the Watch Window

The Watch window provides a more convenient viewing mechanism than the log file for quickly checking key performance criteria or comparing results from different runs. Its limitation is that it only displays certain criteria. If you need details, use the log file, as described in *Viewing and Working with the Log File*, on page 213.

1. Open the Watch window, if needed, by checking View->Watch Window.

   If you open an existing project, the Watch window shows the parameters set the last time you opened the window.

2. If you need a larger window, either resize the window or move the Watch Window as described below.

   – Hold down Ctrl or Shift, click on the window, and move it to a position you want. This makes the Watch window an independent window, separate from the Project view.

   – To move the window to another position within the Project view, right-click in the window border and select Float in Main Window. Then move the window to the position you want, as described above.

   See Watch Window, on page 46 in the *Reference Manual* for information about the popup menu commands.

3. Select the log parameter you want to monitor by clicking on a line and selecting a parameter from the resulting popup menu.

   The software automatically fills in the appropriate value from the last synthesis run. You can check the clock requested and estimated frequencies, the clock requested and estimated periods, the slack, and some resource usage criteria.

4. To compare the results of two or more synthesis runs, do the following:
   - If needed, resize or move the window as described above.
   - Click the right mouse button in the window and select Configure Watch from the popup.
   - Click Watch Selected Implementations and either check the implementations you want to compare or click Watch All Implementations. Click OK. The Watch window now shows a column for each implementation you selected.
   - In the Watch window, set the parameters you want to compare.

   The software shows the values for the selected implementations side by side. For more information about multiple implementations, see Tips for Optimization, on page 414.

Log Parameter	rev_1	rev_2	rev_3
Worst Slack	989.029	995.811	997.109
system|clk_inferred_clock - Requested Frequency	1.0 MHz	1.0 MHz	1.0 MHz
system|clk_inferred_clock - Estimated Frequency	91.1 MHz	238.7 MHz	345.9 MHz

Checking Resource Usage

Each FPGA architecture has a certain number of dedicated FPGA resources. Use the Resource Usage section of the log file to check whether you are exceeding the available resources.

1. Go to the Resource Usage report at the end of the log file (`srr`).
2. Check the number and types of components used to determine if you have used too much of your resources.

   The following is an example:

   Resource Usage Report for top

   Mapping to part: xc4vlx15sf363-10
   Register bits not including I/Os: 15000 (122%)
   Block Rams: 48 of 48 (100%)
   DSP primitives: 24 of 32 (75%)

   Mapping Summary:
   Total LUTs: 14740 (120%)

© 2016 Synopsys, Inc.

If your design is overutilized, you can manage usage with resource-specific attributes like syn_ramstyle, syn_dspstyle, and so on. For hierarchical designs you can set limits with attributes like syn_allowed_resources or the Allocate Timing and Resource Budgets command.

Handling Messages

This section describes how to work with the error messages, notes, and warnings that result after a run. See the following for details:

- Checking Results in the Message Viewer, on page 227
- Filtering Messages in the Message Viewer, on page 229
- Filtering Messages from the Command Line, on page 231
- Automating Message Filtering with a Tcl Script, on page 232
- Log File Message Controls, on page 234
- Handling Warnings, on page 237

Checking Results in the Message Viewer

The Tcl Script window includes a Message Viewer. By default, the Tcl window is in the lower left corner of the main window. This procedure shows you how to check results in the message viewer.

1. If you need a larger window, either resize the window or move the Tcl window. Click in the window border and move it to a position you want. You can float it outside the main window or move it to another position within the main window.

2. Click the Messages tab to open the message viewer.

   The window lists the errors, warnings, and notes in a spreadsheet format. See Message Viewer, on page 50 in the *Reference Manual* for a full description of the window.

© 2016 Synopsys, Inc.

Type	ID	Message	Source Location	Log Location	Time	Report
9	CD233	Using sequential encoding for type aluop_type	const_pkg.vhd	spcl_reqs.srr	07:37:47 Wed May 09	Vhdl Compiler
9	CD630	Synthesizing work.reg_file.first	-	spcl_reqs.srr	07:37:47 Wed May 09	Vhdl Compiler
8	FX271	Instance "DECODE.ALUOP[3]" with 30 loads has been...	-	spcl_reqs.srr	07:37:47 Wed May 09	SPARTAN3 Mapper
[4]	FX271	Instance "SPECIAL_REGS.INST[9]" with 19 loads has ...	spcl_reqs.vhd (44)	spcl_reqs.srr (194)	07:37:47 Wed May 09	SPARTAN3 Mapper
	FX271	Instance "SPECIAL_REGS.INST[11]" with 30 loads ha...	spcl_reqs.vhd (44)	spcl_reqs.srr (167)	07:37:47 Wed May 09	SPARTAN3 Mapper
	FX271	Instance "DECODE.ALUOP[1]" with 15 loads has been...	ins_decode.vhd (296)	spcl_reqs.srr (193)	07:37:47 Wed May 09	SPARTAN3 Mapper
	FX271	Instance "DECODE.ALUOP[3]" with 30 loads has been...	ins_decode.vhd (296)	spcl_reqs.srr (168)	07:37:47 Wed May 09	SPARTAN3 Mapper
	FX271	Instance "DECODE.ALUOP[0]" with 36 loads has been...	ins_decode.vhd (296)	spcl_reqs.srr (166)	07:37:47 Wed May 09	SPARTAN3 Mapper
	CD720	Setting time resolution to ns	std.vhd (123)	spcl_reqs.srr (16)	07:37:47 Wed May 09	Vhdl Compiler
	FX107	No read/write conflict check. Simulation mismatch pos...	reg_file.vhd (23)	spcl_reqs.srr (112)	07:37:47 Wed May 09	Mapper Report
	CL134	Found RAM mem, depth=32, width=8	reg_file.vhd (23)	spcl_reqs.srr (80)	07:37:47 Wed May 09	Vhdl Compiler
	CL201	Trying to extract state machine for register STACKLEV...	pc.vhd (34)	spcl_reqs.srr (82)	07:37:47 Wed May 09	Vhdl Compiler
	FX214	Generating ROM ROM.Data_1[11:0]	ins_rom.vhd (22)	spcl_reqs.srr (145)	07:37:47 Wed May 09	Mapper Report
	MT206	Autoconstrain Mode is ON	-	spcl_reqs.srr (108)	07:37:47 Wed May 09	Mapper Report
	MT197	Clock constraints cover only FF-to-FF paths associate...	-	spcl_reqs.srr (246)	07:37:47 Wed May 09	Timing Report

3. To reduce the clutter in the window and make messages easier to find and understand, use the following techniques:
   - Use the color cues. For example, when you have multiple synthesis runs, messages that have not changed from the previous run are in black; new messages are in red.
   - Enable the Group Common IDs option in the upper right. This option groups all messages with the same ID and puts a plus symbol next to the ID. You can click the plus sign to expand grouped messages and see individual messages.

     - There are two types of message groups. The same warning or note ID appears in multiple source files indicated by a dash in the source files column. Multiple warnings or notes in the same line of source code indicated by a bracketed number.
   - Sort the messages. To sort by a column header, click that column heading. For example, click Type to sort the messages by type. For example, you can use this to organize the messages and work through the warnings before you look at the notes.
   - To find a particular message, type text in the Find field. The tool finds the next occurrence. You can also click the F3 key to search forward, and the Shift-F3 key combination to search backwards.

4. To filter the messages, use the procedure described in *Filtering Messages in the Message Viewer*, on page 229. Crossprobe errors from the message window:
   - If you need more information about how to handle a particular message, click the message ID in the ID column. This opens the documentation for that message.
   - To open the corresponding source code file, click the link in the Source Location column. Correct any errors and rerun synthesis. For warnings, see *Handling Warnings*, on page 237.
   - To view the message in the context of the log file, click the link in the Log Location column.

Filtering Messages in the Message Viewer

The Message viewer lists all the notes, warnings, and errors. The following procedure shows you how to filter out the unwanted messages from the display, instead of just sorting it as described in *Checking Results in the Message Viewer*, on page 227. For the command line equivalent of this procedure, see *Filtering Messages from the Command Line*, on page 231.

1. Open the message viewer by clicking the Messages tab in the Tcl window as previously described.
2. Click Filter in the message window.

   The Warning Filter spreadsheet opens, where you can set up filtering expressions. Each line is one filter expression.

3. Set your display preferences.
   - To hide your filtered choices from the list of messages, click Hide Filter Matches in the Warning Filter window.

© 2016 Synopsys, Inc.

– To display your filtered choices, click Show Filter Matches.

4. Set the filtering criteria.

   – Set the columns to reflect the criteria you want to filter. You can either select from the pull-down menus or type your criteria. If you have multiple synthesis runs, the pull-down menu might contain selections that are not relevant to your design.

     The first line in the following example sets the criteria to show all warnings (Type column) with message ID FA188 (ID). The second set of criteria displays all notes that begin with MF.

   – Use multiple fields and operators to refine filtering. You can use wildcards in the field, as in line 2 of the example. Wildcards are case-sensitive and space-sensitive. You can also use ! as a negative operator. For example, if you set the ID in line 2 to !MF*, the message list would show all notes except those that begin with MF.

   – Click Apply when you have finished setting the criteria. This automatically enables the Apply Filter button in the messages window, and the list of messages is updated to match the criteria.

     The synthesis tool interprets the criteria on each line in the Warning Filter window as a set of AND operations (Warning and FA188), and the lines as a set of OR operations (Warning and FA188 or Note and MF*).

   – To close the Warning Filter window, click Close.

5. To save your message filters and reuse them, do the following:

   – Save the project. The synthesis tool generates a Tcl file called *projectName*.pfl (Project Filter Log) in the same location as the main project file. The following is an example of the information in this file:

```
log_filter -hide_matches
log_filter -field type==Warning
   -field message==*Una*
   -field source_loc==sendpacket.v
   -field log_loc==usbHostSlave.srr
   -field report=="Compiler Report"
log_filter -field type==Note
log_filter -field id==BN132
log_filter -field id==CL169
log_filter -field message=="Input *"
log_filter -field report=="Compiler Report"
```

– When you want to reuse the filters, source the *projectName*.pfl file.

You can also include this file in a synhooks Tcl script to automate your process.

Filtering Messages from the Command Line

The following procedure shows you how to use Tcl commands to filter out unwanted messages. If you want to use the GUI, see *Filtering Messages in the Message Viewer*, on page 229.

1. Type your filter expressions in the Tcl window using the log_filter command. For details of the syntax, see log_filter, on page 47 in the *Reference Manual*.

   For example, to hide all the notes and print only errors and warnings, type the following:

   ```
   log_filter -enable
   log_filter -hide_matches
   log_filter -field type==Note
   ```

2. To save and reuse the filter commands, do the following:
   – Type the log_filter commands in a Tcl file.
   – Source the file when you want to reuse the filters you set up.

3. To print the results of the log_filter commands to a file, add the log_report command at the end of a list of log_filter commands.

   ```
   log_report -print filteredMsg.txt
   ```


© 2016 Synopsys, Inc.

This command prints the results of the preceding log_filter commands to the specified text file, and puts the file in the same directory as the main project file. The file contains the filtered messages, for example:

```
@N MF138 Rom slaveControlSel_1 mapped in logic. Mapper Report
   wishbonebi.v (156) usbHostSlave.srr (819) 05:22:06 Mon Oct 18
@N(2) MO106 Found ROM, 'slaveControlSel_1', 15 words by 1 bits
   Mapper Report wishbonebi.v (156) usbHostSlave.srr (820)
   05:22:06 Mon Oct 18
@N MO106 Found ROM, 'slaveControlSel_1', 15 words by 1 bits Mapper
   Report wishbonebi.v (156) usbHostSlave.srr (820) 05:22:06 Mon
   Oct 18
@N MF138 Rom hostControlSel_1 mapped in logic. Mapper Report
   wishbonebi.v (156) usbHostSlave.srr (821) 05:22:06 Mon Oct 18
@N MO106 Found ROM, 'hostControlSel_1', 15 words by 1 bits Mapper
   Report wishbonebi.v (156) usbHostSlave.srr (822) 05:22:06 Mon
   Oct 18
@N Synthesizing module writeUSBWireData Compiler Report
   writeusbwiredata.v (59) usbHostSlave.srr (704) 05:22:06 Mon Oct 18
```

Automating Message Filtering with a Tcl Script

The following example shows you how to use a synhooks Tcl script to automatically load a message filter file when a project opens and to send email with the messages after a run.

1. Create a message filter file like the following. (See *Filtering Messages in the Message Viewer*, on page 229 or *Filtering Messages from the Command Line*, on page 231 for details about creating this file.)

   ```
   log_filter -clear
   log_filter -hide_matches
   log_filter -field report=="ECP3 MAPPER"
   log_filter -field type==NOTE
   log_filter -field message=="Input *"
   log_filter -field message=="Pruning *"
   puts "DONE!"
   ```

2. Copy the synhooks.tcl file and set the environment variable as described in *Automating Flows with synhooks.tcl*, on page 563.

3. Edit the synhooks.tcl file so that it reads like the following example. For syntax details, see *synhooks* *File Syntax, on page 384* in the *Reference Manual*.

– The following loads the message filter file when the project is opened. Specify the name of the message filter file you created in step 1. Note that you must source the file.

```
proc syn_on_open_project {project_path} {
set filter filterFilename
puts "FILTER $filter IS BEING APPLIED"
source d:/tcl/filters/$filterFilename
}
```

– Add the following to print messages to a file after synthesis is done:

```
proc syn_on_end_run {runName run_dir implName} {
set warningFileName "messageFilename"

if {$runName == "synthesis"} {
   puts "Mapper Done!"
   log_report -print $warningFileName
set f [open [lindex $warningFileName] r]
set msg ""
while {[gets $f warningLine]>=0} {
   puts $warningLine
   append msg $warningLine\n
   }
close $f
```

– Continue by specifying that the messages be sent in email. You can obtain the smtp email packages off the web.

```
source "d:/tcl/smtp_setup.tcl"
proc send_simple_message {recipient email_server subject body}{
   set token [mime::initialize -canonical text/plain -string
      $body]
   mime::setheader $token Subject $subject
   smtp::sendmessage $token -recipients $recipient -servers
      $email_server
   mime::finalize $token
}
puts "Sending email..."

send_simple_message {address1,address2}
   yourEmailServer subjectText> emailText
   }
}
```

When the script runs, an email with all the warnings from the synthesis run is automatically sent to the specified email addresses.

© 2016 Synopsys, Inc.

Log File Message Controls

The log file message control feature allows messages in the current session to be elevated in severity (for example, promoted to an error from a warning), lowered in severity (for example, demoting a warning to a note), or suppressed from the log file after the next run through the Log File Filter dialog box. This dialog box is displayed by opening the log file in HTML mode and selecting Log File Message Filter from the popup menu with the right mouse button.

Log File Filter Dialog Box

The Log File Filter dialog box is the primary control for changing a message priority or suppressing a message. When you initially open the dialog box, all of the messages from the log (`srr`) file for the active implementation are displayed in the upper section and the lower section is empty. To use the dialog box:

1. Select (highlight) the message to be promoted, demoted, or suppressed from the messages displayed in the upper section.

2. Select the Suppress Message, Make Error, Make Warning, or Make Note button to move the selected message from the upper section to the lower section. The selected message is repopulated in the lower section with the Override column reflecting the disposition of the message according to the button selected.

Allowed Severity Changes

Allowed severity levels and preference settings for warning, note, and advisory messages are:

- Promote – warning to error, note to warning, note to error
- Demote – warning to note
- Suppress – suppress warning, suppress note, suppress advisory

---

Note: Normal error messages (messages generated by default) cannot be suppressed or changed to a lesser severity level.

When using the dialog box:

- Use the control and shift keys to select multiple messages.
- If an `srr` file is not present (for example, if you are starting a new project) the table will be empty. Run the design at least once to generate an `srr` file.
- Clicking the OK button saves the message status changes to the *projectName*.pfl file in the project directory.

Message Reporting

The compiler and mapper must be rerun before the impact of the message status changes can be seen in the updated log file.

When a *projectName*.pfl input file is present at the start of the run, the message-status changes in the file are forwarded to the mapper and compiler which generate an updated log file. Depending on the changes specified:

- If an ID is promoted to an error, the mapper/compiler stops execution at the first occurrence of the message and prints the message in the @E:*msgID* :*messageText* format

© 2016 Synopsys, Inc.

- If an ID is promoted to a warning, the mapper/compiler prints the message in the @W:*msgID* :*messageText* format.
- If an ID is demoted to a note, the mapper/compiler prints the message in the @N:*msgID* :*messageText* format.
- If an ID is suppressed, the mapper/compiler excludes the message from the `srr` file.

Note: The online, error-message help documentation is unchanged by any message modification performed by the filtering mechanism. If a message is initially categorized as a warning in the synthesis tool, it continues to be reported as a warning in error-message help irrespective its promotion/demotion status.

Updating the *projectName*.pfl file

The *projectName*.pfl file in the top-level project directory stores the user message filter settings from the Log File Filter dialog box for that project. This file can be edited with a text editor. The file entry syntax is:

message_override -suppress *ID* [*ID* ...] | **-error** *ID* [*ID* ...] | **-warning** *ID* [*ID* ...] | **-note** *ID* [*ID* ...]

For example, to override the default message definition for note FX702 as a warning, enter:

```
message_override -warning FX702
```

Note: After editing the `pfl` file, close and reopen the project to update the overrides.

messagefilter.txt File

A messagefilter.txt file in the *implementation*/syntmp directory lists any changes made to message priority or suppression through the Log File Filter dialog box. This file, which is only generated when changes are made to the default status of a message, can be accessed outside of the GUI without consuming a license.

Handling Warnings

If you get warnings (@W prefix) after a synthesis run, do the following:

- Read the warning message and decide if it is something you need to act on, or whether you can ignore it.
- If the message is not self-explanatory or if you are unsure about how to handle the error, click the message ID in either the message window or HTML log file or double click the message ID in the ASCII text log file. These actions take you to online information about the condition that generated the warning.

Working with Downgradable Errors and Critical Warnings

You can temporarily change the classification for certain kinds of messages:

Downgradable errors (@DE)	Can be downgraded to warnings. This is a small set of *non-fatal* errors where you can temporarily postpone addressing the error and continue with the design flow and verification of other aspects of the design.
Critical warnings (@CW)	Can be upgraded to errors. This small set of warnings represent critical problems. Elevating them to errors ensures that they are recognized and dealt with, because the error status forces the tool to stop.

You can downgrade or upgrade these messages from the GUI or through a Tcl command.

Downgrading or Upgrading Messages from the GUI

1. Open the log file.
2. Right click within the log file and select Log File Message Filter from the pop-up menu to display the Log File Filter dialog box.
3. Highlight the DE or CW message from the list of messages displayed at the top of the dialog box. You can only downgrade messages with a DE prefix, or upgrade messages with a CE prefix.
4. Depending on the message type selected:
   - Click the Make Warning button to downgrade the error to a warning. This action moves the message from the top section of the message

© 2016 Synopsys, Inc.

filter to the bottom section. The Override column in the bottom section displays the updated status for the message (Warning).

– Click the Make Error button to upgrade the critical warning to an error. This action moves the message from the top section of the message filter to the bottom section. The Override column in the bottom section displays the updated status for the message (Error).

5. To see the changes reflected, click the Run button. You can verify that the message is now treated as:
   – A warning (the mapping operation continues past the initial point of the error condition)
   – An error (the operation stops at the initial point of the error).
6. To revert the DE or CW message:
   – Click on one of the downgraded DE warning IDs in the report and select Log File Message Filter from the pop-up menu to display the Log File Filter dialog box. This reverts the warning message to an error.
   – Click on one of the upgraded CW warning IDs in the report and select Log File Message Filter from the pop-up menu to display the Log File Filter dialog box. This reverts the error message to a warning.
7. From the dialog box, highlight the DE or CW message from the list of messages displayed at the top of the dialog box. Click either the:
   – Make Error button to return the message to its original error status.
   – Make Warning button to return the message to its original critical warning status.

Modifying the status of a message does not affect the message string. A message originally categorized as an error continues to be reported as an error regardless of its user-assigned status.

Downgrading or Upgrading Messages with a Tcl Command

Type the message_override command in the Tcl script window to change the classification of DE and CW messages.

1. To downgrade DE errors or upgrade CW warnings, use the appropriate message_override command:

   message_override -warning DE*messageID*
   message_override -error CW*messageID*

Enter the message ID without the @ prefix, as shown above.

2. To see the changes reflected, rerun the state from the database where you changed the message classification.

   After you have fixed the cause of the upgraded warning (critical warning) or completed the rest of the flow (downgradable error), you must change the message classification back to its original status.

3. To revert the messages back to their original status, use the appropriate message_override command:

   message_override -error DE*messageID*
   message_override -warning CW*messageID*

4. Rerun again to confirm that the message status has reverted to the original.

© 2016 Synopsys, Inc.

SYNOPSYS®
Silicon to Software™

CHAPTER 7

Analyzing with HDL Analyst

This chapter describes how to analyze logic in the HDL Analyst and FSM Viewer.

See the following for detailed procedures:

- Working in the Schematic (Beta), on page 242
- Exploring Design Hierarchy (Beta), on page 255
- Finding Objects (Beta), on page 262
- Crossprobing (Beta), on page 265
- Analyzing With the HDL Analyst Tool (Beta), on page 270
- Working in the Standard Schematic, on page 291
- Exploring Design Hierarchy, on page 306
- Finding Objects, on page 314
- Crossprobing, on page 327
- Analyzing With the HDL Analyst Tool, on page 335
- Using the FSM Viewer, on page 352

© 2016 Synopsys, Inc.

Working in the Schematic (Beta)

The HDL Analyst tool includes single-page schematics, which can help you graphically analyze and navigate your entire design easier. This section describes basic procedures you use in the schematics. These procedures include the following topics:

- Clone Schematic – See Cloning Schematics, on page 245
- Instance Groups – See Grouping Objects in the Schematic, on page 249
- Partial Dissolve – See Grouping Objects in the Schematic, on page 249
- Net Based Filtering – See Filtering Schematics, on page 274
- Unfilter – See Filtering Schematics, on page 274
- Multi-threaded Find – See Browsing to Find Objects in HDL Analyst Views, on page 262
- New Mouse Strokes with Cancel Display– See Mouse Stroke Conventions, on page 245
- New Push View Tab – See Cloning Schematics, on page 245
- Peek – See Viewing Design Hierarchy and Context, on page 270
- Show Contents – See Viewing Design Hierarchy and Context, on page 270
- Improved Bus Display and Handling – See Dissolving and Partial Dissolving of Buses and Pins, on page 282

For information on specific tasks like analyzing critical paths, see the following sections:

- Traversing Design Hierarchy with the Hierarchy Browser, on page 255
- Exploring Object Hierarchy with Push/Pop Commands, on page 257
- Crossprobing (Beta), on page 265
- Analyzing With the HDL Analyst Tool (Beta), on page 270

Opening the Views

The procedure for opening a view is the same at different design stages; the main difference is the content that is available at the different design database states.

1. Start at the database state you want.

RTL view	Start with a compiled design.
Technology view	Start with a mapped (synthesized) design.

2. To enable the new HDL Analyst, use one of the following methods:
   - From the UI: Click the New HDL Analyst (Beta) check box.
   - From the UI: Select HDL Analyst->Use New HDL Analyst (Beta) option
   - Tcl switch: toggle_new_analyst
3. Open the schematic using one of the following commands:

Hierarchical RTL or Technology view	Use one of these methods: • Select HDL Analyst->RTL->Hierarchical View. • Click the RTL View icon ( ) (a plus sign inside a circle). • Double-click the `srs` file in the Implementation Results view. To open a flattened RTL view, select HDL Analyst->RTL->Flattened View.
Hierarchical Technology view	Use one of these methods: • Select HDL Analyst ->Technology->Hierarchical View. • Click the Technology View icon (NAND gate icon). • Double-click the `srm` file in the Implementation Results view.
Flattened RTL or Technology view	Select HDL Analyst->RTL->Flattened View or HDL Analyst->Technology->Flattened View

All schematic views have the schematic on the right and a pane on the left that contains a hierarchical list of the objects in the design. This pane is called the Hierarchy Browser.

© 2016 Synopsys, Inc.

Compiled View

Mapped View

Dataflow View

Both the compiled and mapped views have a Dataflow View. Use this view to display objects from a left to right datapath flow as shown above. You can display a Clock View as well.

Clock View

To display all sequential elements connected to clock nets and debug the clocks in the design use this view. Select Clocks View from the drop-down menu in the upper right corner of the schematic view. Clock nets are displayed with the color green.

Mouse Stroke Conventions

Use the mouse strokes to control navigation and the display, which are listed at the bottom of the schematic window. They include:

- Zoom – Ctrl-scroll wheel
- Zoom Area – Ctrl-drag
- Pan – Middle-click drag or Alt-drag
- Push – Double-click
- Cancel Display – Press Esc
  (For large designs, you can cancel displaying the netlist but still use the netlist from the hierarchy browser, Tcl window, and Find dialog box.)

Cloning Schematics

Most operations performed in any of the HDL Analyst views (Clock View or Dataflow View) are displayed in the current view. To create a new view, use the clone commands.

1. To clone the current view displayed, right-click and select Clone Schematic from the drop-down menu. This view opens in a new window.

© 2016 Synopsys, Inc.

2. To push into an object and create a new view, select an object and right-click and Push in New Tab from the drop-down menu. For more information, see *Exploring Object Hierarchy with Push/Pop Commands, on page 257*.

3. To filter objects and create a new view, select the objects and right-click and Filter in a New Tab from the drop-down menu. For more information, see *Filtering Schematics, on page 274*.

Viewing Object Properties

There are a few ways in which you can view the properties of objects.

1. To temporarily display the properties of a particular object, hold the cursor over the object.

2. Select the object, right-click, and select Properties. The properties and their values are displayed in a table.

   For example, you can view the properties for instances and ports as shown below.

Instance Properties

Port Properties

Similarly, you can view the properties for pins and nets.

© 2016 Synopsys, Inc.

Net Properties

Pin Properties

Selecting Objects in the Schematic

For mouse selection, standard object selection rules apply:

To select ...	Do this ...
Single objects	Click on the object in the schematic, or click the object name in the Hierarchy Browser. Tcl equivalent: **select {i:***instanceName***}**

To select ...	Do this ...
Multiple objects	Use one of these methods: • Draw a rectangle around the objects. • Select an object, press Ctrl, and click other objects you want to select. • Select multiple objects in the Hierarchy Browser. See *Browsing With the Hierarchy Browser*, on page 262. • Use Find to select the objects you want. See *Finding Objects (Beta)*, on page 262. Tcl equivalent: **select {i:***instance1***} {i:***instance2***}**
Objects by type (instances, ports, nets)	Use Find to select the objects (see *Browsing With the Find Command*, on page 263), or use the Hierarchy Browser, which lists objects by type.
No objects (deselect all currently selected objects)	Click the left mouse button in a blank area of the schematic. Deselected objects are no longer highlighted.

The HDL Analyst view highlights selected objects in red. If you have other windows that are cloned, the selected object is highlighted in the other windows as well (crossprobing).

Grouping Objects in the Schematic

You can group objects in the schematic. Sometimes the HDL Analyst tool automatically determines groups shown with the color purple below. When you push into the group block, the content of its objects is displayed.

© 2016 Synopsys, Inc.

The HDL Analyst tool automatically groups instances with similar names at every level, when you enable the Allow Automatic Grouping option on the HDL Analyst Options dialog box. For example, suppose there are three registers with the names out_reg[1], out_reg[2], and out_reg[3]. A group will be created with the registers having the name out_reg[3:1].

To create your own groups:

1. Select the specified objects in the view.
2. Right-click and select the Group option from the pop-up menu.

The objects are now grouped together in the purple block.

© 2016 Synopsys, Inc.

3. Push into the purple block to see the grouped objects.

4. To ungroup selections, right-click and select Dissolve from the drop-down menu.
5. To remove individual instances of a group, right-click and select Partial Dissolve from drop-down menu. A dialog box is displayed where you can select the items to remove.

6. User-created groups are not saved when you close and re-open the same netlist.

© 2016 Synopsys, Inc.

Moving Between Views in a Schematic Window

When you filter or expand your design, you move through a number of different design views in the same schematic window. For example, you might start with a view of the entire design, then filter an object and finally expand a connection in the filtered view, for a total of three views. You can also move back after flattening a view.

1. To move back to the previous view, click the Back icon ( ).

   The software displays the last view.

2. To move forward again, click the Forward icon ( ).

   The software displays the next view in the display history.

Setting Schematic Preferences

You can set various preferences for the schematic from the user interface.

1. Select Options->HDL Analyst Options. For a description of all the options on this form, see New HDL Analyst Options Command, on page 327 in the *Command Reference Manual*.

2. The table details the following operations:

To ...	Do this ...
Specify how you want the schematic to display.	Select Clock View or Dataflow View (default).
Specify how the tool determines the detailed routing for the design.	Select Optimized, Detailed, Standard (default), or Quick (direct connection). The time to route the design increases as you go from Quick to Optimized mode. The schematic is the most clean and uncluttered for the Optimized mode.
Determine if you want automatic grouping for the design.	When Allow Automatic Grouping is enabled, the tool automatically groups instances with similar names at every level in the design.

Exploring Design Hierarchy (Beta)

Schematics generally have a certain amount of design hierarchy. You can move between hierarchical levels using the Hierarchy Browser mode. For additional information, see *Analyzing With the HDL Analyst Tool (Beta)*, on page 270. See *Traversing Design Hierarchy with the Hierarchy Browser*, on page 255.

Traversing Design Hierarchy with the Hierarchy Browser

The Hierarchy Browser is the list of objects on the left side of the schematic view. It is best used to get an overview, or when you need to browse and find an object. If you want to move between design levels of a particular object, using the Push command is more direct. Refer to *Exploring Object Hierarchy with Push/Pop Commands*, on page 257 for details.

The hierarchy browser allows you to traverse and select the following:

- Instances or Groups
- Ports
- Internal nets

The browser lists the objects by type. Use the expand ( ) and collapse ( ) signs to ascend or descend the hierarchy.

© 2016 Synopsys, Inc.

You can perform some similar operations as done in the schematic view, such as filtering an object from the Hierarchy Browser. For example, highlight an instance, then right-click and select Filter as shown below.

Exploring Object Hierarchy with Push/Pop Commands

To view the internal hierarchy of a specific instance, use the Push/Pop commands from the drop-down menu or mouse strokes. When combined with other commands like filtering and expansion commands, Push/Pop can be a very powerful tool for isolating and analyzing logic. See *Filtering Schematics*, on page 274 and *Expanding Pin and Net Logic*, on page 276 for details about filtering and expansion. See the following sections for information about pushing down and popping up in hierarchical design objects:

- Pushing into Objects, on page 257
- Popping up a Hierarchical Level, on page 260

Pushing into Objects

In the schematic, you can push into instances and view the lower-level hierarchy. You can use a mouse stroke or the command to push into objects:

1. To move down a level (push into an object) with a mouse stroke, put your cursor near the top of the object, hold down the right mouse button, and draw a vertical stroke from top to bottom. You can push into the following objects; see step 3 for examples of pushing into different types of objects.
   - Hierarchical instances. They can be displayed as pale yellow boxes (opaque instances).

© 2016 Synopsys, Inc.

- Technology-specific primitives. The primitives are listed in the Hierarchy Browser in the schematic, under i:*instanceNames*->Primitives.
- Inferred ROMs and state machines.

The remaining steps show you how to use the icon or command to push into an object.

2. Enable the Push/Pop command by doing one of the following:
   - Double-click on the object.
   - Right-click in the view and select Push/Pop from the drop-down menu.
   - Use the mouse strokes.

After pushing into an instance, the following schematic is displayed. Notice the purple block that groups gates with similar names together. You can push into this block as well.

3. To push (descend) into an object, double-click on the hierarchical object. The following figure shows the result of pushing into a ROM.

© 2016 Synopsys, Inc.

Popping up a Hierarchical Level

1. To move up a level (pop up a level), put your cursor anywhere in the design,
   - Use the Pop Hierarchy icon ( ).

Hold down the right mouse button, and draw a vertical mouse stroke, moving from the bottom upwards.

The software moves up a level, and displays the next level of hierarchy.

© 2016 Synopsys, Inc.

Finding Objects (Beta)

In the schematics, you can use the Hierarchy Browser or the Find command to find objects, as explained in these sections:

- Browsing to Find Objects in HDL Analyst Views, on page 262
- Using Wildcards with the Find Command, on page 264

Browsing to Find Objects in HDL Analyst Views

You can always zoom in to find an object in the schematic. Use Zoom Fit to quickly fit all objects into the schematic. The following procedure shows you how to browse through design objects and find an object at any level of the design hierarchy. You can use the Hierarchy Browser or the Find command to do this. If you are familiar with the design hierarchy, the Hierarchy Browser can be the quickest method to locate an object. The Find command is best used to graphically browse and locate the object you want.

Browsing With the Hierarchy Browser

1. In the Hierarchy Browser, click the name of the net, port, or instance you want to select.

   The object is highlighted in the schematic.

2. To select a range of objects, you can press and hold the Shift key while clicking the selected objects in the range.

   The software selects and highlights all the objects in the range.

3. If the object is on a lower hierarchical level, do either of the following:
   - Expand the appropriate higher-level object by clicking the collapsed symbol next to it, and then select the object you want.
   - Push down into the higher-level object, and then select the object from the Hierarchy Browser.

   The selected object is highlighted in the schematic. However, you may have to filter the object to view it in the design hierarchy.

4. To select all objects of the same type, select them from the Hierarchy Browser. For example, you can find all the nets in your design.

Browsing With the Find Command

1. In a schematic, select the search icon ( ) or press Ctrl-f to open the Find dialog box.

2. Do the following in the dialog box:
   - Select the type of objects to find: instances, nets, ports, pins, and/or symbols.
   - Specify how you want the search to occur: for all hierarchies and/or allow * to search across the hierarchy separator.

3. Select on an object displayed in the dialog box below, then click the Select button.

   Click the Close button to end the Find search. Then, you can use the Filter command to display the object.

   When the search style options (Search all hierarchies and ".",can cross the hierarchy separator) are not enabled, the software searches for objects at the top level.

4. If you determine that the search is taking too long to run, notice that the Find button changes to Stop. Click Stop.

© 2016 Synopsys, Inc.

The multi-threaded Find command can be interrupted and canceled once the search term has been identified. If you let the search complete, you will see Finishing and Done appearing under the display window.

Using Wildcards with the Find Command

Use the following wildcards when you search the schematics:

*	The asterisk matches any sequence of characters.
?	The question mark matches any single character, but not the hierarchy separator by default.
.	The dot explicitly matches a hierarchy separator, so type one dot for each level of hierarchy. To use the dot as a pattern and not a hierarchy separator, type a backslash before the dot: \.

Crossprobing (Beta)

Crossprobing is the process of selecting an object in one view and having the object or the corresponding logic automatically highlighted in other views. Crossprobing helps you visualize where coding changes or timing constraints might help to reduce area or improve performance.

This section describes how to crossprobe from different views. It includes the following:

- Crossprobing within a View, on page 265
- Crossprobing from an HDL Analyst View, on page 266
- Crossprobing from the Text Editor Window, on page 268

Crossprobing within a View

Selecting an object name in the Hierarchy Browser highlights the object in the schematic, and vice versa.

Selected Object	Highlighted Object
Instance in schematic (single-click)	Module icon in Hierarchy Browser
Net in schematic	Net icon in Hierarchy Browser
Port in schematic	Port icon in Hierarchy Browser
Logic icon in Hierarchy Browser	Instance in schematic
Net icon in Hierarchy Browser	Net in schematic
Port icon in Hierarchy Browser	Port in schematic

In this example, when you select the DECODE module in the Hierarchy Browser, the DECODE module is automatically selected in the view.

© 2016 Synopsys, Inc.

Crossprobing from an HDL Analyst View

To crossprobe from the schematic to other open views or the source code files, select the object by clicking on it.

The software automatically highlights the object in all open views. If the open view is a schematic, the software highlights the object in the Hierarchy Browser on the left as well as in the schematic. If the highlighted object is in another hierarchy of a schematic, the view does not automatically track to the page. You may have to filter the schematic.

To crossprobe from the schematic to a source file when the source file is not open, the instance names must be the same. Notice that when you hover over an instance name in the schematic, it turns blue. You can click on this link, to automatically open the editor window of the source code file and highlight the appropriate code as shown below. A message is generated if a match cannot be found.

© 2016 Synopsys, Inc.

Crossprobing from the Text Editor Window

To crossprobe source in the text editor window or from the log file to a schematic, use this procedure. You can use this method to crossprobe from any text file with objects that have the same instance names as in the synthesis software.

1. Open the schematic to which you want to crossprobe.
2. Select the appropriate portion of text in the Text Editor window. In some cases, it may be necessary to select an entire block of text to crossprobe.
3. You can choose either to:
   - Highlight the objects in the path, right-click and select Filter in Analyst from the drop-down menu. The tool automatically filters the schematic so that you see just the selected objects in the view.
   - Highlight the objects in the path, right-click and select Select in Analyst from the drop-down menu. You might have to filter the selected objects to see them displayed in the schematic.

For example:

Text Editor

© 2016 Synopsys, Inc.

Analyzing With the HDL Analyst Tool (Beta)

The HDL Analyst tool is a graphical productivity tool that helps you visualize your synthesis results. It displays schematics of the design at different stages, allowing you to graphically view and analyze your design. At an early design stage, the schematic displays high-level structures like RAM, ROM, operators, and FSM as abstractions. Later in the cycle, these structures are converted to gates and mapped to technology-specific resources.

To analyze information or compare views with the log file, the FSM view, and the source code, you can use techniques like crossprobing, flattening, and filtering. See the following for more information about analysis techniques.

- Viewing Design Hierarchy and Context, on page 270
- Filtering Schematics, on page 274
- Expanding Pin and Net Logic, on page 276
- Dissolving and Partial Dissolving of Buses and Pins, on page 282
- Flattening Schematic Hierarchy, on page 283
- Using the FSM Viewer, on page 286

For additional information about navigating the HDL Analyst views or using other techniques like crossprobing, see the following:

- Working in the Schematic (Beta), on page 242
- Exploring Design Hierarchy (Beta), on page 255
- Finding Objects (Beta), on page 262
- Crossprobing (Beta), on page 265

Viewing Design Hierarchy and Context

Most large designs are hierarchical, so the software provides tools that help you view hierarchy details or put the details in context. Alternatively, you can browse and navigate hierarchy with the Push/Pop command, or flatten the design to view internal hierarchy.

This section describes how to use interactive hierarchical viewing operations to better analyze your design. Automatic hierarchy viewing operations that are built into other commands are described in the context in which they appear.

1. To view the internal logic of instances in your design, do either of the following:
   - To view the logic of an individual instance, push into it. This generates a new schematic with the internal details. Click the Back icon to return to the previous view.
   - To view the logic of all instances in the design, select all the required instances and right-click then select Peek. This command lets you see internal logic in content, by adding the internal details to the current schematic. If the view is too cluttered with this option on, filter the view (see *Filtering Schematics, on page 274*) or push into the primitive. Click the Back icon to return to the previous view after filtering or pushing into the object.

   The following figure compares these two methods:

Result of pushing into a primitive (new view of lower-level logic)

Result of enabling Peek (same view with internal logic)

© 2016 Synopsys, Inc.

Note: When more than 500 instances are contained in a block, its content will not automatically be shown in the schematic. To display the contents of the selected instances, right-click and select Show Contents.

2. Suppose you just used the peek option to see the internal logic of an instance. To return back to the schematic state before using peek and while the peek objects are still highlighted, right-click and select Hide Contents from the drop-down menu.

3. To view the internal logic of a hierarchical instance, you can push into the instance, dissolve the selected instance with the Dissolve command, or flatten the design.

Pushing into an instance	Generates a view that shows only the internal logic. You do not see the internal hierarchy in context. To return to the previous view, click Back. See *Exploring Object Hierarchy with Push/Pop Commands*, on page 257 for details.
Flattening the entire design	Opens a view where the entire design is flattened. Large flattened designs can be overwhelming. See *Flattening Schematic Hierarchy*, on page 283 for details about flattening designs.
Flattening an instance by dissolving	Generates a view where the hierarchy of the selected instances is flattened, but the rest of the design is unaffected. This provides context. See *Flattening Schematic Hierarchy*, on page 283 for details about dissolving instances.

The following schematic shows an instance that has been dissolved in the view.

4. The software automatically traces a critical path through different hierarchical levels using hollow boxes with nested internal logic (transparent instances) to indicate levels in hierarchical instances.

© 2016 Synopsys, Inc.

Filtering Schematics

Filtering is a useful first step in analysis, because it focuses analysis on the relevant parts of the design. Some commands, like the Expand commands, automatically generate filtered views; this procedure only discusses manual filtering, where you use the Filter command to isolate selected objects.

This table lists the advantages of using filtering over flattening:

Filter Schematic Command	Flatten Commands
Loads part of the design; better memory usage	Loads entire design
Combine filtering with the Push/Pop command, and history buttons (Back and Forward) to move freely between hierarchical levels	You can use the Back arrow or Show Top View icon to return to previous view that has been flattened.

1. Select the objects that you want to isolate. For example, you can select two connected objects.
2. Select the Filter command, using one of these methods:
   - Right-click and select Filter from the popup menu.
   - Click the Filter icon (buffer gate) ( ).

The software filters the design and displays the selected objects in a filtered view. These objects are isolated in the schematic displayed. Select Unfilter to take you back to the view where objects are at the same level.

You can now analyze the problem, and do operations like the following:

Trace paths, build up logic	See *Expanding Pin and Net Logic*, on page 276
Filter further	Select objects and filter again
Find objects	See *Finding Objects (Beta)*, on page 262
Flatten	See *Flattening Schematic Hierarchy*, on page 283. You can hide transparent or opaque instances.
Crossprobe from filtered view	See *Crossprobing from an HDL Analyst View*, on page 266

3. To return to the previous schematic, click the Back arrow. If you flattened the hierarchy, right-click and select the Back arrow or the Show Top View icon to return to the top-level unflattened view.

For additional information about filtering schematics, see *Filtering Schematics*, on page 274 and *Flattening Schematic Hierarchy*, on page 283.

© 2016 Synopsys, Inc.

Expanding Pin and Net Logic

When you are working in a filtered view, you might need to include more logic in your selected set to debug your design.

Use the Expand commands with the Filter and Flatten commands to isolate just the logic that you want to examine. Filtering isolates logic, flattening removes hierarchy. See *Filtering Schematics*, on page 274 and *Flattening Schematic Hierarchy*, on page 283 for details.

1. To expand logic from a pin hierarchically across boundaries, use the following commands.

To ...	Do this ...
See the first-level cells connected to a pin in the same hierarchy	Select a pin and select Expand. See *Expanding Filtered Logic Example*, on page 277.
See the first-level cells connected to a pin at any level of hierarchy	Select a pin and select Hierarchical Expand Nets.
See all cells until a register or port is connected to the selected pin at the same level of hierarchy	Select a pin and select Expand to Reg/Port.
See all cells until a register or port is connected to the selected pin at any level of hierarchy	Select a pin and select Hierarchical Expand to Reg/Port.
See internal cells connected to a pin	Select a pin and select Expand Inwards. The software filters the schematic and displays the internal cells closest to the port. See *Expanding Inwards Example*, on page 277.

Expanding Filtered Logic Example

Expanding Inwards Example

© 2016 Synopsys, Inc.

Expanding Hierarchically

2. To expand logic from a net, do the following:
   - Use the commands shown in the following table.
   - Select a net, then right-click and select the command from the right-click options.

To ...	Do this ...
See all instances connected to the selected net being filtered	Select a net and select Filter by Nets.
Select the instances in the same hierarchy connected to the selected net	Select a net and select Expand Nets.
Select only one object connected to a port or pin	Select a pin or port and select Expand to One Object.
Select and show instances connected to the selected net at any level of hierarchy. The instance that drives the net and the instance which is driven by the net are shown.	Select a net and select Hierarchical Expand Nets.

The following figures illustrate this.

Filter by Nets

© 2016 Synopsys, Inc.

Expand Nets

Expand to One Object

Hierarchical Expand Nets

3. You can also isolate the paths to generate a schematic for a path between objects. To display connections to and from the selected instance, highlight it then right-click and select Isolate Paths from the drop-down menu.

© 2016 Synopsys, Inc.

Dissolving and Partial Dissolving of Buses and Pins

The HDL Analyst tool has options for handling buses and pins in the display that can help you analyze your design easier. You can expand logic from a bus port or specific bits of a port.

1. To expand logic from a bus port to its bit ports:
   - Select a net.
   - Right-click and select Dissolve from the drop-down menu.

To expand logic from specific bits of the port:

- Right-click and select Partial Dissolve from the drop-down menu.
- Select the pin (pc[10]) to be expanded.
- Click OK.

Flattening Schematic Hierarchy

Flattening removes hierarchy so you can view the logic without hierarchical levels. In most cases, you do not have to flatten your hierarchical schematic to debug and analyze your design, because you can use a combination of filtering and expanding to view logic at different levels. However, if you must flatten the design use the following techniques, which include flattening and dissolving instances.

1. To flatten any level of hierarchy to logic cells below the current level, right-click and select Flatten Schematic from the drop-down menu.

   The software flattens the design hierarchy and displays it in the window. To return to the previous level, select the Back arrow.

2. To selectively flatten some hierarchical instances in your design by dissolving them, do the following:
   - Select the instances to be flattened.
   - Right-click and select Dissolve.

The results differ slightly, depending on the kind of view from which you dissolve instances.

© 2016 Synopsys, Inc.

Starting View	Software Generates a ...
Filtered	Filtered view with the internal logic of dissolved instances displayed within hollow bounding boxes (transparent instances), and the hierarchy of the rest of the design unchanged. If the transparent instance does not display internal logic, use one of the alternatives described in step 4 of *Viewing Design Hierarchy and Context*, on page 270. Use the Back button to return to the undissolved view.
Unfiltered	New, flattened view with the dissolved instances flattened in place (no nesting) to Boolean logic, and the hierarchy of the rest of the design unchanged. You can use the Back button to return to previous or the top-level views.

The following figure illustrates this.

Dissolved logic for prgmcntr shown filtered when started from filtered view

Dissolved logic for prgmcntr shown flattened in context when you start from an unfiltered view

Use this technique if you only want to flatten part of your design while retaining the hierarchical context. If you want to flatten most of the design, use the technique described in the previous step. Instead of dissolving instances, you can use a combination of the filtering commands and the Push/Pop command.

© 2016 Synopsys, Inc.

Using the FSM Viewer

The FSM viewer displays state transition bubble diagrams for FSMs in the design, along with additional information about the FSM. You can use this viewer to view state machines.

1. To start the FSM viewer, open the compiled view and highlight the FSM instance, click the right mouse button and select View State Machine from the popup menu.

   The FSM viewer opens. The viewer consists of a transition bubble diagram and a table for the encodings and transitions. If you used Verilog to define the FSMs, the viewer displays binary values for the state machines if you defined them with the ‘define keyword, and actual names if you used the parameter keyword.

2. The following table summarizes basic viewing operations.

To view ...	Do ...
from and to states, and conditions for each transition	Click the Transitions tab at the bottom of the table.
the correspondence between the states and the FSM registers in the RTL view	Click the RTL Encoding tab.
the correspondence between the states and the registers in the Technology View	Click the Mapped Encodings tab (available after synthesis).
only the transition diagram without the table	Select View->FSM table or click the FSM Table icon. You might have to scroll to the right to see it.

This figure shows you the mapping information for a state machine. The Transitions tab shows you simple equations for conditions for each state. The RTL Encodings tab has a State column that shows the state names in the source code, and a Registers column for the corresponding RTL encoding. The Mapped Encoding tab shows the state names in the code mapped to actual values.

© 2016 Synopsys, Inc.

	From State	To State	Condition
1	LapDisplay	LapDisplay	Laplevel&!Reset
2	Zero	LapDisplay	Lap&!Start&!Reset
3	StopSecond	StopSecond	!Lap&!Start&!Reset
4	StopFirst	StopSecond	Lap&!Start&!Reset
5	CountCont	StopSecond	Start&!Reset
6	StopSecond	StopDiff	Lap&!Start&!Reset
7	StopDiff	StopDiff	!Lap&!Start&!Reset
8	StopFirst	StopFirst	!Lap&!Start&!Reset
9	CountFirst	StopFirst	Start&!Reset
10	StopSecond	CountCont	Start&!Reset

States and Conditions

State	cur_state[6]	cur_state[5]	cur_state[4]	cur_state[3]	cur_state[2]	cur_state[1]	cur_state[0]
Zero	0	0	0	0	0	0	1
CountFirst	0	0	0	0	0	1	0
CountCont	0	0	0	0	1	0	0
StopFirst	0	0	0	1	0	0	0
StopDiff	0	0	1	0	0	0	0
StopSecond	0	1	0	0	0	0	0
LapDisplay	1	0	0	0	0	0	0

Mapped Encoding

State	Register
Zero	cur_state[0]
CountFirst	cur_state[1]
CountCont	cur_state[2]
StopFirst	cur_state[3]
StopDiff	cur_state[4]
StopSecond	cur_state[5]
LapDisplay	cur_state[6]

RTL Encoding

3. To view just one selected state,
   - Select the state by clicking on its bubble. The state is highlighted.
   - Click the right mouse button and select the filtering criteria from the popup menu: output, input, or any transition.

   The transition diagram now shows only the filtered states you set. The following figure shows filtered views for output and input transitions for one state.

CountCont state filtered by output transitions

CountCont state filtered by input transitions

Similarly, you can check the relationship between two or more states by selecting the states, filtering them, and checking their properties.

4. To view the properties for a state,

© 2016 Synopsys, Inc.

Net Properties

Pin Properties

– Select the state.

– Click the right mouse button and select Properties from the popup menu. A form shows you the properties for that state.

To view the properties for the entire state machine like encoding style, number of states, and total number of transitions between states, deselect any selected states, click the right mouse button outside the diagram area, and select Properties from the popup menu.

Working in the Standard Schematic

The HDL Analyst includes the RTL and Technology views, which are schematic used to graphically analyze your design. In the Synplify product, these views are part of the optional HDL Analyst package. The RTL view is available after a design is compiled; the Technology view is available after a designed has been synthesized and contains technology-specific primitives. In the Synplify Premier product, a RTL Design Plan view is available after a floorplan has been created with physical constraint regions and synthesized for the device.

The HDL Analyst includes the RTL and Technology views, which are schematics used to graphically analyze your design. In the Synplify product, these views are part of the optional HDL Analyst package. The RTL view is available after a design is compiled; the Technology view is available after a designed has been synthesized and contains technology-specific primitives.

For detailed descriptions of these views, see the *HDL Analyst Tool* section of the *Reference Manual*. This section describes basic procedures you use in the RTL and Technology views. The information is organized into these topics:

- Opening the Views, on page 243
- Opening the Views, on page 243
- Viewing Object Properties, on page 246
- Selecting Objects in the RTL/Technology Views, on page 299
- Working with Multisheet Schematics, on page 300
- Moving Between Views in a Schematic Window, on page 302
- Setting Schematic Preferences, on page 303
- Managing Windows, on page 304

For information on specific tasks like analyzing critical paths, see the following sections:

- Exploring Object Hierarchy by Pushing/Popping, on page 307
- Exploring Object Hierarchy of Transparent Instances, on page 313
- Browsing to Find Objects in HDL Analyst Views, on page 314
- Crossprobing, on page 327

© 2016 Synopsys, Inc.

- Analyzing With the HDL Analyst Tool, on page 335

Differentiating Between the HDL Analyst Views

RTL View	Technology View
Generated after compilation.	Generated after mapping.
Technology-independent components at a high level of abstraction, like adders, registers, large muxes, and state machines.	Technology-specific primitives like look-up tables, cascade and carry chains, muxes and flip-flops.
srs database (Synopsys proprietary).	srm database (Synopsys proprietary).

Opening the Views

The procedure for opening an RTL or Technology view is similar; the main difference is the design stage at which these views are available.

1. Start at the appropriate design stage:

RTL view	Start with a compiled design.
Technology view	Start with a mapped (synthesized) design.

2. Open the view as described in this table:

Hierarchical RTL or Technology view	Use one of these methods: • Select HDL Analyst->RTL->Hierarchical View. • Click the RTL View icon ( ) (a plus sign inside a circle). • Double-click the `srs` file in the Implementation Results view. To open a flattened RTL view, select HDL Analyst->RTL->Flattened View.
Hierarchical Technology view	Use one of these methods: • Select HDL Analyst ->Technology->Hierarchical View. • Click the Technology View icon (NAND gate icon). • Double-click the `srm` file in the Implementation Results view.
Flattened RTL or Technology view	Select HDL Analyst->RTL->Flattened View or HDL Analyst->Technology->Flattened View

All RTL and Technology views have the schematic on the right and a pane on the left that contains a hierarchical list of the objects in the design. This pane is called the Hierarchy Browser. The bar at the top of contains additional information. See Hierarchy Browser, on page 58 in the *Reference Manual* for a description of the Hierarchy Browser.

© 2016 Synopsys, Inc.

RTL View

Technology View

Viewing Object Properties

There are a few ways in which you can view the properties of objects.

1. To temporarily display the properties of a particular object, hold the cursor over the object. A tooltip temporarily displays the information. at the cursor and in the status bar at the bottom of the tool window.

2. Select the object, right-click, and select Properties. The properties and their values are displayed in a table.

If you select an instance, you can view the properties of the associated pins by selecting the pin from the list. Similarly, if you select a port, you can view the properties on individual bits.

3. To flag objects by property, follow these steps:
   - Open an RTL or Technology view.
   - Select Options->HDL Analyst Options->Visual Properties, and select the properties you want to view from the pull-down list. Some properties are only available in certain views.

© 2016 Synopsys, Inc.

– Close the HDL Analyst Options dialog box.

– Enable View->Visual Properties. If you do not enable this, the software does not display the property flags in the schematics. The tool uses a rectangular flag with the property name and value to annotate all objects in the current view that have the specified property. Different properties use different colors, so you can enable and view many properties at the same time.

Example: Slow and New Properties

The slow property is useful for analyzing your critical path, because it denotes objects that do not meet the timing criteria. The following figure shows a filtered view of a critical path, with slow instances flagged in blue.

The New property helps with debugging because it quickly identifies objects that have been added to the current schematic with commands like Expand. You can step through successive filtered views to determine what was added at each step.

The next figure expands one of the pins from the previous filtered view. The new instance added to the view has two flags: new and slow.

© 2016 Synopsys, Inc.

Using the orig_inst_of Property for Parameterized Modules

The compiler automatically uniquifies parameterized modules or instances. Properties are available to identify the RTL names of both uniquified and original modules or instances.

- inst_of property – identifies module or instance by uniquified name
- orig_inst_of property – identifies module or instance by its original name before it was uniquified

In the following example, the top-level module (top) instantiates the module sub multiple times using different parameter values. The compiler uniquifies the module sub as sub_3s, sub_1s, and sub_4s.

Top.v

```
module top (input clk, [7:0] din, output [7:0] dout);
   sub #(.W(3)) UUT1 (.clk, .din(din[2:0]), .dout(dout[2:0]));
   sub #(.W(1)) UUT2 (.clk, .din(din[3]), .dout(dout[3]));
   sub #(.W(4)) UUT3 (.clk, .din(din[7:4]), .dout(dout[7:4]));
endmodule

module sub #(parameter W = 0) (
   input clk,
   input [W-1:0] din,
   output logic [W-1:0] dout);

always@(posedge clk)
   begin
      dout <= din;
   end
endmodule
```

RTL View

TCL Command Example

Use the get_prop command with the orig_inst_of property to identify the original RTL name for the module:

```
% get_prop -prop orig_inst_of {v:sub_3s}
sub

% get_prop -prop orig_inst_of {i:UUT3}
sub
```

Selecting Objects in the RTL/Technology Views

For mouse selection, standard object selection rules apply: In selection mode, the pointer is shaped like a crosshair.

To select ...	Do this ...
Single objects	Click on the object in the RTL or Technology schematic, or click the object name in the Hierarchy Browser.

© 2016 Synopsys, Inc.

To select ...	Do this ...
Multiple objects	Use one of these methods: • Draw a rectangle around the objects. • Select an object, press Ctrl, and click other objects you want to select. • Select multiple objects in the Hierarchy Browser. See *Browsing With the Hierarchy Browser*, on page 314. • Use Find to select the objects you want. See *Using Find for Hierarchical and Restricted Searches*, on page 316.
Objects by type (instances, ports, nets)	Use Edit->Find to select the objects (see *Browsing With the Find Command*, on page 315), or use the Hierarchy Browser, which lists objects by type.
All objects of a certain type (instances, ports, nets)	To select all objects of a certain type, do either of the following: • Right-click and choose the appropriate command from the Select All Schematic/Current Sheet popup menus. • Select the objects in the Hierarchy Browser.
No objects (deselect all currently selected objects)	Click the left mouse button in a blank area of the schematic or click the right mouse button to bring up the pop-up menu and choose Unselect All. Deselected objects are no longer highlighted.

The HDL Analyst view highlights selected objects in red. If the object you select is on another sheet of the schematic, the schematic tracks to the appropriate sheet. If you have other windows open, the selected object is highlighted in the other windows as well (crossprobing), but the other windows do not track to the correct sheet. Selected nets that span different hierarchical levels are highlighted on all the levels. See *Crossprobing*, on page 327 for more information about crossprobing.

Some commands affect selection by adding to the selected set of objects: the Expand commands, the Select All commands, and the Select Net Driver and Select Net Instances commands.

Working with Multisheet Schematics

The title bar of the RTL or Technology view indicates the number of sheets in that schematic. In a multisheet schematic, nets that span multiple sheets are indicated by sheet connector symbols, which you can use for navigation.

1. To reduce the number of sheets in a schematic, select Options->HDL Analyst Options and increase the values set for Sheet Size Options - Instances and Sheet Size Options - Filtered Instances. To display fewer objects per sheet (increase the number of sheets), increase the values.

   These options set a limit on the number of objects displayed on an unfiltered and filtered schematic sheet, respectively. A low Filtered Instances value can cause lower-level logic inside a transparent instance to be displayed on a separate sheet. The sheet numbers are indicated inside the empty transparent instance.

2. To navigate through a multisheet schematic, refer to this table. It summarizes common operations and ways to navigate.

To view ...	Use one of these methods ...
Next sheet or previous sheet	Select View->Next/Previous Sheet. Press the right mouse button and draw a horizontal mouse stroke (left to right for next sheet, right to left for previous sheet). Click the icons: Next Sheet ( ) or Previous Sheet ( ) Press Shift-right arrow (Next Sheet) or Shift-left arrow (Previous sheet). Navigate with View->Back and View ->Forward if the next/previous sheets are part of the display history.
A specific sheet number	Select View->View Sheets and select the sheet. Click the right mouse button, select View Sheets from the popup menu, and then select the sheet you want. Press Ctrl-g and select the sheet you want.
Lower-level logic of a transparent instance on separate sheets	Check the sheet numbers indicated inside the empty transparent instance. Use the sheet navigation commands like Next Sheet or View Sheets to move to the sheet you need.
All objects of a certain type	To highlight all the objects of the same type in the schematic, right-click and select the appropriate command from the Select All Schematic popup menu. To highlight all the objects of the same type on the current sheet, right-click and select the appropriate command from the Select All Sheet popup menu.
Selected items only	Filter the schematic as described in *Filtering Schematics*, on page 339.

© 2016 Synopsys, Inc.

To view ...	Use one of these methods ...
A net across sheets	If there are no sheet numbers displayed in a hexagon at the end of the sheet connector, select Options ->HDL Analyst Options and enable Show Sheet Connector Index. Right-click the sheet connector and select the sheet number from the popup as shown in the following figure.

Sheet Connector Symbol

Connected sheet numbers

Sheet connector with multisheet popup menu

Moving Between Views in a Schematic Window

When you filter or expand your design, you move through a number of different design views in the same schematic window. For example, you might start with a view of the entire design, zoom in on an area, then filter an object, and finally expand a connection in the filtered view, for a total of four views.

1. To move back to the previous view, click the Back icon or draw the appropriate mouse stroke.

   The software displays the last view, including the zoom factor. This does not work in a newly generated view (for example, after flattening) because there is no history.

2. To move forward again, click the Forward icon or draw the appropriate mouse stroke.

   The software displays the next view in the display history.

Setting Schematic Preferences

You can set various preferences for the RTL and Technology views from the user interface.

1. Select Options->HDL Analyst Options. For a description of all the options on this form, see New HDL Analyst Options Command, on page 327 in the *Reference Manual*.

2. The following table details some common operations:

To ...	Do this ...
Display the Hierarchy Browser	Enable Show Hierarchy Browser (General tab).
Control crossprobing from an object to a P&R text file	Enable Enhanced Text Crossprobing. (General tab)
Determine the number of objects displayed on a sheet.	Set the value with Maximum Instances on the Sheet Size tab. Increase the value to display more objects per sheet.
Determine the number of objects displayed on a sheet in a filtered view.	Set the value with Maximum Filtered Instances on the Sheet Size tab. Increase the number to display more objects per sheet. You cannot set this option to a value less than the Maximum Instances value.

Some of these options do not take effect in the current view, but are visible in the next schematic you open.

3. To view hierarchy within a cell, enable the General->Show Cell Interiors option.

Show Cell Interior off

Show Cell Interior on

© 2016 Synopsys, Inc.

4. To control the display of labels, first enable the Text->Show Text option, and then enable the Label Options you want. The following figure illustrates the label that each option controls.

For a more detailed information about some of these options, see Schematic Display, on page 103 in the *Reference Manual.*

5. Click OK on the HDL Analyst Options form.

The software writes the preferences you set to the `ini` file, and they remain in effect until you change them.

Managing Windows

As you work on a project, you open different windows. For example, you might have two Technology views, an RTL view, and a source code window open. The following guidelines help you manage the different windows you have open. For information about cycling through the display history in a single schematic, see *Moving Between Views in a Schematic Window*, on page 302.

1. Toggle on View->Workbook Mode.

Below the Project view, you see tabs like the following for each open view. The tab for the current view is on top. The symbols in front of the view name on the tab help identify the kind of view.

2. To bring an open view to the front, if the window is not visible, click its tab. If part of the window is visible, click in any part of the window.

   If you previously minimized the view, it will be in minimized form. Double-click the minimized view to open it.

3. To bring the next view to the front, click Ctrl-F6 in that window.

4. Order the display of open views with the commands from the Window menu. You can cascade the views (stack them, slightly offset), or tile them horizontally or vertically.

5. To close a view, press Ctrl-F4 in that window or select File->Close.

© 2016 Synopsys, Inc.

Exploring Design Hierarchy

Schematics generally have a certain amount of design hierarchy. You can move between hierarchical levels using the Hierarchy Browser or Push/Pop mode. For additional information, see *Analyzing With the HDL Analyst Tool*, on page 335. The topics include:

- Traversing Design Hierarchy with the Hierarchy Browser, on page 306
- Exploring Object Hierarchy by Pushing/Popping, on page 307
- Exploring Object Hierarchy of Transparent Instances, on page 313

Traversing Design Hierarchy with the Hierarchy Browser

The Hierarchy Browser is the list of objects on the left side of the RTL and Technology views. It is best used to get an overview, or when you need to browse and find an object. If you want to move between design levels of a particular object, Push/Pop mode is more direct. Refer to *Exploring Object Hierarchy by Pushing/Popping*, on page 307 for details.

The hierarchy browser allows you to traverse and select the following:

- Instances and submodules
- Ports
- Internal nets
- Clock trees (in an RTL view)

The browser lists the objects by type. A plus sign in a square icon indicates that there is hierarchy under that object and a minus sign indicates that the design hierarchy has been expanded. To see lower-level hierarchy, click on the plus sign for the object. To ascend the hierarchy, click on the minus sign.

Refer to Hierarchy Browser Symbols, on page 59 in the *Reference Manual* for an explanation of the symbols.

Exploring Object Hierarchy by Pushing/Popping

To view the internal hierarchy of a specific object, it is best to use Push/Pop mode or examine transparent instances, instead of using the Hierarchy Browser described in *Traversing Design Hierarchy with the Hierarchy Browser*, on page 306. You can access Push/Pop mode with the Push/Pop Hierarchy icon, the Push/Pop Hierarchy command, or mouse strokes.

When combined with other commands like filtering and expansion commands, Push/Pop mode can be a very powerful tool for isolating and analyzing logic. See *Filtering Schematics*, on page 339, *Expanding Pin and Net Logic*, on page 341, and *Expanding and Viewing Connections*, on page 345 for details about filtering and expansion. See the following sections for information about pushing down and popping up in hierarchical design objects:

- Pushing into Objects, on page 308, next
- Popping up a Hierarchical Level, on page 311

© 2016 Synopsys, Inc.

Pushing into Objects

In the schematic, you can push into objects and view the lower-level hierarchy. You can use a mouse stroke, the command, or the icon to push into objects:

1. To move down a level (push into an object) with a mouse stroke, put your cursor near the top of the object, hold down the right mouse button, and draw a vertical stroke from top to bottom. You can push into the following objects; see step 3 for examples of pushing into different types of objects.
   - Hierarchical instances. They can be displayed as pale yellow boxes (opaque instances) or hollow boxes with internal logic displayed (transparent instances). You cannot push into a hierarchical instance that is hidden with the Hide Instance command (internal logic is hidden).

   - Technology-specific primitives. The primitives are listed in the Hierarchy Browser in the Technology view, under Instances - Primitives.
   - Inferred ROMs and state machines.

   The remaining steps show you how to use the icon or command to push into an object.

2. Enable Push/Pop mode by doing one of the following:
   - Select View->Push/Pop Hierarchy.
   - Right-click in the Technology view and select Push/Pop Hierarchy from the popup menu.
   - Click the Push/Pop Hierarchy icon ( ) in the toolbar (two arrows pointing up and down).
   - Press F2.

   The cursor changes to an arrow. The direction of the arrow indicates the underlying hierarchy, as shown in the following figure. The status bar at the bottom of the window reports information about the objects over which you move your cursor.

3. To push (descend) into an object, click on the hierarchical object. For a transparent instance, you must click on the pale yellow border. The following figure shows the result of pushing into a ROM.

   When you descend into a ROM, you can push into it one more time to see the ROM data table. The information is in a view-only text file called rom.info.

© 2016 Synopsys, Inc.

Similarly, you can push into a state machine. When you push into an FSM from the RTL view, you open the FSM viewer where you can graphically view the transitions. For more information, see *Using the FSM Viewer*, on page 352. If you push into a state machine from the Technology view, you see the underlying logic.

Popping up a Hierarchical Level

1. To move up a level (pop up a level), put your cursor anywhere in the design, hold down the right mouse button, and draw a vertical mouse stroke, moving from the bottom upwards.

© 2016 Synopsys, Inc.

The software moves up a level, and displays the next level of hierarchy.

2. To pop (ascend) a level using the commands or icon, do the following:
   - Select the command or icon if you are not already in Push/Pop mode. See *Pushing into Objects*, on page 308for details.
   - Move your cursor to a blank area and click.
3. To exit Push/Pop mode, do one of the following:
   - Click the right mouse button in a blank area of the view.
   - Deselect View->Push/Pop Hierarchy.
   - Deselect the Push/Pop Hierarchy icon.
   - Press F2.

Exploring Object Hierarchy of Transparent Instances

Examining a transparent instance is one way of exploring the design hierarchy of an object. The following table compares this method with pushing (described in *Exploring Object Hierarchy by Pushing/Popping*, on page 307).

	Pushing	Transparent Instance
User control	You initiate the operation through the command or icon.	You have no direct control; the transparent instance is automatically generated by some commands that result in a filtered view.
Design context	Context lost; the lower-level logic is shown in a separate view	Context maintained; lower-level logic is displayed inside a hollow yellow box at the hierarchical level of the parent.

© 2016 Synopsys, Inc.

Finding Objects

In the schematic, you can use the Hierarchy Browser or the Find command to find objects, as explained in these sections:

- Browsing to Find Objects in HDL Analyst Views, on page 314
- Using Find for Hierarchical and Restricted Searches, on page 316
- Using Wildcards with the Find Command, on page 319
- Using Find to Search the Output Netlist, on page 324

For information about the Tcl Find command, which you use to locate objects, and create collections, see find, on page 104 in the *Command Reference Manual*.

Browsing to Find Objects in HDL Analyst Views

You can always zoom in to find an object in the RTL and Technology schematics. The following procedure shows you how to browse through design objects and find an object at any level of the design hierarchy. You can use the Hierarchy Browser or the Find command to do this. If you are familiar with the design hierarchy, the Hierarchy Browser can be the quickest method to locate an object. The Find command is best used to graphically browse and locate the object you want.

Browsing With the Hierarchy Browser

1. In the Hierarchy Browser, click the name of the net, port, or instance you want to select.

   The object is highlighted in the schematic.

2. To select a range of objects, select the first object in the range. Then, scroll to display the last object in the range. Press and hold the Shift key while clicking the last object in the range.

   The software selects and highlights all the objects in the range.

3. If the object is on a lower hierarchical level, do either of the following:
   - Expand the appropriate higher-level object by clicking the plus symbol next to it, and then select the object you want.

– Push down into the higher-level object, and then select the object from the Hierarchy Browser.

The selected object is highlighted in the schematic. The following example shows how moving down the object hierarchy and selecting an object causes the schematic to move to the sheet and level that contains the selected object.

4. To select all objects of the same type, select them from the Hierarchy Browser. For example, you can find all the nets in your design.

Browsing With the Find Command

1. In a schematic, select HDL Analyst->Find or press Ctrl-f to open the Object Query dialog box.

2. Do the following in the dialog box:

   – Select objects in the selection box on the left. You can select all the objects or a smaller set of objects to browse. If length makes it hard to read a name, click the name in the list to cause the software to display the entire name in the field at the bottom of the dialog box.

© 2016 Synopsys, Inc.

– Click the arrow to move the selected objects over to the box on the right.

The software highlights the selected objects.

3. In the Object Query dialog box, click on an object in the box on the right.

   The software tracks to the schematic page with that object.

Using Find for Hierarchical and Restricted Searches

You can always zoom in to find an object in the RTL and Technology schematics or use the Hierarchy Browser (see *Browsing to Find Objects in HDL Analyst Views*, on page 314). This procedure shows you how to use the Find command to do hierarchical object searches or restrict the search to the current level or the current level and its underlying hierarchy.

Note that Find only adds to the current selection; it does not deselect anything that is already selected. you can use successive searches to build up exactly the selection you need, before filtering.

1. If needed, restrict the range of the search by filtering the view.

   See *Viewing Design Hierarchy and Context*, on page 336 and *Filtering Schematics*, on page 339 for details. With a filtered view, the software only searches the filtered instances, unless you set the scope of the search to Entire Design, as described below, in which case Find searches the entire design.

   You can use the filtering technique to restrict your search to just one schematic sheet. Select all the objects on one sheet and filter the view. Continue with the procedure.

2. To further restrict the range of the search, hide instances you do not need.

   You can do this in addition to filtering the view, or instead of filtering the view. Hidden instances and their hierarchy are excluded from the search. When you have finished the search, use the Unhide Instances command to make the hierarchy visible again.

3. Open the Object Query dialog box.

– Do one of the following: right click in the RTL or Technology view and select Find from the popup menu, press Ctrl-f, or click the Find icon ( ).
– Reposition the dialog box so you can see both your schematic and the dialog box.

4. Select the tab for the type of object. The Unhighlighted box on the left lists all objects of that type (instances, symbols, nets, or ports).

   For fastest results, search by Instances rather than Nets. When you select Nets, the software loads the whole design, which could take some time.

5. Click one of these buttons to set the hierarchical range for the search: Entire Design, Current Level & Below, or Current Level Only, depending on the hierarchical level of the design to which you want to restrict your search.

   The range setting is especially important when you use wildcards. See *Effect of Hierarchy and Range on Wildcard Searches*, on page 319 for details. Current Level Only or Current Level & Below are useful for searching filtered schematics or critical path schematics.

   The lower-level details of a transparent instance appear at the current level and are included in the search when you set it to Current Level Only. To exclude them, temporarily hide the transparent instances, as described in step 2.

© 2016 Synopsys, Inc.

Use Entire Design to hierarchically search the whole design. For large hierarchical designs, reduce the scope of the search by using the techniques described in the first step.

The Unhighlighted box shows available objects within the scope you set. Objects are listed in alphabetical order, not hierarchical order.

6. To search for objects in the mapped database or the output netlist, set the Name Space option.

   The name of an object might be changed because of synthesis optimizations or to match the place-and-route tool conventions, so that the object name may no longer match the name in the original netlist. Setting the Name Space option ensures that the Find command searches the correct database for the object. For example, if you set this option to Tech View, the tool searches the mapped database (`srm`) for the object name you specify. For information about using this feature to find objects from an output netlist, see *Using Find to Search the Output Netlist*, on page 324.

7. Do the following to select objects from the list. To use wildcards in the selection, see the next step.
   - Click on the objects you want from the list. If length makes it hard to read a name, click the name in the list to cause the software to display the entire name in the field at the bottom of the dialog box.
   - Click Find 200 or Find All. The former finds the first 200 matches, and then you can click the button again to find the next 200.
   - Click the right arrow to move the objects into the box on the right, or double-click individual names.

   The schematic displays highlighted objects in red.

8. Do the following to select objects using patterns or wildcards.
   - Type a pattern in the Highlight Wildcard field. See *Using Wildcards with the Find Command*, on page 319 for a detailed discussion of wildcards.

     The Unhighlighted list shows the objects that match the wildcard criteria. If length makes it hard to read a name, click the name in the list to cause the software to display the entire name in the field at the bottom of the form.

– Click the right arrow to move the selections to the box on the right, or double-click individual names. The schematic displays highlighted objects in red.

You can use wildcards to avoid typing long pathnames. Start with a general pattern, and then make it more specific. The following example browses and uses wildcards successively to narrow the search.

Find all instances three levels down	`*.*.*`
Narrow search to find instances that begin with i_	`i_*.*.*`
Narrow search to find instances that begin with un2 after the second hierarchy separator	`i_*.*.un2*`

Note that there are some differences when you specify the find command in the RTL view, Technology view, or the constraint file.

9. You can leave the dialog box open to do successive Find operations. Click OK or Cancel to close the dialog box when you are done.

For detailed information about the Find command and the Object Query dialog box, see Find Command (HDL Analyst), on page 185 of the *Reference Manual*.

Using Wildcards with the Find Command

Use the following wildcards when you search the schematics:

*	The asterisk matches any sequence of characters.
?	The question mark matches any single character.
.	The dot explicitly matches a hierarchy separator, so type one dot for each level of hierarchy. To use the dot as a pattern and not a hierarchy separator, type a backslash before the dot: \.

Effect of Hierarchy and Range on Wildcard Searches

The asterisk and question mark wildcards do not cross hierarchical boundaries, but search each level of hierarchy individually with the search pattern. This default is affected by the following:

- Hierarchical separators

© 2016 Synopsys, Inc.

Dots match hierarchy separators, unless you use the backslash escape character in front of the dot (\.). Hierarchical search patterns with a dot (l*.*) are repeated at each level included in the scope. If you use the `*.*` pattern with Current Level, the software matches non-hierarchical names at the current level that include a dot.

- Search range

  The scope of the search determines the starting point for the searches. Some times the starting point might make it appear as if the wildcards cross hierarchical boundaries. If you are at 2A in the following figure and the scope of the search is set to Current Level and Below, separate searches start at 2A, 3A1, and 3A2. Each search does not cross hierarchical boundaries. If the scope of the search is Entire Design, the wildcard searches run from each hierarchical point (1, 2A, 2B, 3A1, 3A2, 3B1, 3B2, and 3B3). The result of an asterisk search (*) with Entire Design is a list of all matches in the design, regardless of the current level.

See *Wildcard Search Examples*, on page 321 for examples.

How a Wildcard Search Works

1. The starting point of a wildcard search depends on the range set for the search.

Entire Design	Starts at top level and uses the pattern to search from that level. It then moves to any child levels below the top level and searches them. The software repeats the search pattern at each hierarchical point in the design until it searches the entire design.
Current Level	Starts at the current hierarchical level and searches that level only. A search started at 2A only covers 2A.
Current Level and Below	Starts at the current hierarchical level and searches that level. It then moves to any child levels below the starting point and conducts separate searches from each of these starting points.

2. The software applies the wildcard pattern to all applicable objects within the range. For Current Level and Current Level and Below, the current level determines the starting point.

   Dots match hierarchy separators, unless you use the backslash escape character in front of the dot (\.). Hierarchical search patterns with a dot (l*.*) are repeated at each level included in the scope. See *Effect of Hierarchy and Range on Wildcard Searches*, on page 319 and *Wildcard Search Examples*, on page 321 for details and examples, respectively. If you use the `*.*` pattern with Current Level, the software matches non-hierarchical names at the current level that include a dot.

Wildcard Search Examples

The figure shows a design with three hierarchical levels, and the table shows the results of some searches on this design.

© 2016 Synopsys, Inc.

Scope	Pattern	Starting Point	Finds Matches in ...
Entire Design	*	3A1	1, 2A, 2B, 3A1, 3A2, 3B1, 3B2, and 3B3 (* at all levels)
	.	2B	2A and 2B (*.* from 1) 3A1, 3A2, 3B1, 3B2, and 3B3 (*.* from 2A and 2B) No matches in 1 (because of the hierarchical dot), unless a name includes a non-hierarchical dot.
Current Level	*	1	1 only (no hierarchical boundary crossing)
	.	2B	2B only. No search of lower levels even though the dot is specified, because the scope is Current Level. No matches, unless a 2B name includes a non-hierarchical dot.
Current Level and Below	*	2A	2A only (no hierarchical boundary crossing)
	.	1	2A and 2B (*.* from 1) 3A1, 3A2, 3B1, 3B2, and 3B3 (*.* from 2A and 2B) No matches from 1, because the dot is specified.
	.	2B	3B1, 3B2, and 3B3 (*.* from 2B)
	.	3A2	No matches (no hierarchy below 3A2)
	..*	1	3A1, 3A2, 3B1, 3B2, and 3B3 (*.*.* from 1) Search ends because there is no hierarchy two levels below 2A and 2B.

Difference from Tcl Search

The FPGA synthesis tool confines the simple search to within one level of hierarchy. The following command searches each level of hierarchy individually for the specified pattern:

```
find -hier *abc*addr_reg[*]
```

If you want to go through the hierarchy, you must add the hierarchy separators to the search pattern:

```
find {*.*.abc.*.*.addr_reg[*]}
```

Find Command Differences in HDL Analyst Views and Constraint File

There are some slight differences when you use the Find command in the RTL view, Technology view, and the constraint files:

- You cannot use find to search for bit registers of a bit array in the RTL or Technology views, but you can specify it in a constraint file, where the following command will work:

  ```
  find -seq {i:modulex_inst.qb[7]}
  ```

  In a HDL Analyst view, you cannot find {i:modulex_inst.qb[7]}, but you can specify and find {i:modulex_inst.qb[7:0]}.

- By default, the following Tcl command does not find objects in the RTL view, although it does find objects in the Technology view:

  ```
  -hier -seq * -filter @clock == clk75
  ```

  To make this work in an RTL view, you must turn on Annotated Properties for Analyst in the Device tab of the Implementation Options dialog box, recompile the design, and then open a new RTL view.

Combining Find with Filtering to Refine Searches

You can combine the Find command with the filtering commands to better effect. Depending on what you want to do, use the Find command first, or a filtering command.

1. To limit the range of a search, do the following:
   - Filter the design.
   - Use the Find command on the filtered view, but set the search range to Current Level Only.
2. Select objects for a filtered view.
   - Use the Find command to browse and select objects.
   - Filter the objects to display them.

© 2016 Synopsys, Inc.

Using Find to Search the Output Netlist

When the synthesis tool creates an output netlist file, some names are optimized for use in the P&R tool. When you debug your design for place and route looking for a particular object, use the Name Space option in the Object Query dialog box to locate the optimized names in the output netlist. The following procedure shows you how to locate an object, highlight and filter it in the Technology view, and crossprobe to the source code for editing.

1. Select the output netlist file option in the Implementations Results tab of the Implementation Options dialog box.

2. After you synthesize your design, open your output netlist file and select the name of the object you want to find.

3. Copy the name and open a Technology view.

4. In the Technology view, press Ctrl-f or select Edit->Find to open the Object Query dialog box and do the following:
   - Paste the object name you copied into the Highlight Search field.
   - Set the Name Space option to Netlist and click Find All.

Search by Tech View

Search by Netlist

If you leave the Name Space option set to the default of Tech View, the tool does not find the name because it is searching the mapped database instead of the output netlist.

– Double click the name to move it into the Highlighted field and close the dialog box.

In the Technology view, the name is highlighted in the schematic.

5. Select HDL Analyst->Filter Schematic to view only the highlighted portion of the schematic.

Filtered View

The tooltip shows the equivalent name in the Technology view.

© 2016 Synopsys, Inc.

6. Double click on the filtered schematic to crossprobe to the corresponding code in the HDL file.

Crossprobing

Crossprobing is the process of selecting an object in one view and having the object or the corresponding logic automatically highlighted in other views. Highlighting a line of text, for example, highlights the corresponding logic in the schematic. Crossprobing helps you visualize where coding changes or timing constraints might help to reduce area or improve performance.

You can crossprobe between the RTL view, Technology view, the FSM Viewer, the log file, the source files, and some external text files from place-and-route tools. However, not all objects or source code crossprobe to other views, because some source code and RTL view logic is optimized away during the compilation or mapping processes.

This section describes how to crossprobe from different views. It includes the following:

- Crossprobing within an RTL/Technology View, on page 327
- Crossprobing from the RTL/Technology View, on page 328
- Crossprobing from the Text Editor Window, on page 330
- Crossprobing from the Tcl Script Window, on page 333
- Crossprobing from the FSM Viewer, on page 333

Crossprobing within an RTL/Technology View

Selecting an object name in the Hierarchy Browser highlights the object in the schematic, and vice versa.

Selected Object	Highlighted Object
Instance in schematic (single-click)	Module icon in Hierarchy Browser
Net in schematic	Net icon in Hierarchy Browser
Port in schematic	Port icon in Hierarchy Browser
Logic icon in Hierarchy Browser	Instance in schematic
Net icon in Hierarchy Browser	Net in schematic
Port icon in Hierarchy Browser	Port in schematic

© 2016 Synopsys, Inc.

In this example, when you select the DECODE module in the Hierarchy Browser, the DECODE module is automatically selected in the RTL view.

Crossprobing from the RTL/Technology View

1. To crossprobe from an RTL or Technology views to other open views, select the object by clicking on it.

   The software automatically highlights the object in all open views. If the open view is a schematic, the software highlights the object in the Hierarchy Browser on the left as well as in the schematic. If the highlighted object is on another sheet of a multi-sheet schematic, the view does not automatically track to the page. If the crossprobed object is inside a hidden instance, the hidden instance is highlighted in the schematic.

   If the open view is a source file, the software tracks to the appropriate code and highlights it. The following figure shows crossprobing between the RTL, Technology, and Text Editor (source code) views.

2. To crossprobe from the RTL or Technology view to the source file when the source file is not open, double-click on the object in the RTL or Technology view.

   Double-clicking automatically opens the appropriate source code file and highlights the appropriate code. For example, if you double-click an object in a Technology view, the HDL Analyst tool automatically opens an editor window with the source code and highlights the code that contains the selected register.

The following table summarizes the crossprobing capability from the RTL or Technology view.

© 2016 Synopsys, Inc.

From	To	Procedure
RTL	Source code	Double-click an object. If the source code file is not open, the software opens the Text Editor window to the appropriate section of code. If the source file is already open, the software scrolls to the correct section of the code and highlights it.
RTL	Technology	The Technology view must be open. Click the object to highlight and crossprobe.
RTL	FSM Viewer	The FSM view must be open. The state machine must be coded with a onehot encoding style. Click the FSM to highlight and crossprobe.
Technology	Source code	If the source code file is already, open, the software scrolls to the correct section of the code and highlights it. If the source code file is not open, double-click an object in the Technology view to open the source code file.
Technology	RTL	The RTL view must be open. Click the object to highlight and crossprobe.

Crossprobing from the Text Editor Window

To crossprobe from a source code window or from the log file to an RTL, Technology, or FSM view, use this procedure. You can use this method to crossprobe from any text file with objects that have the same instance names as in the synthesis software. For example, you can crossprobe from place-and-route files. See *Example of Crossprobing a Path from a Text File*, on page 331 for a practical example of how to use crossprobing.

1. Open the RTL, FSM, or Technology view to which you want to crossprobe.
2. To crossprobe from an error, warning, or note in the html log file, click on the file name to open the corresponding source code in another Text Editor window; to crossprobe from a text log file, double-click on the text of the error, warning, or note.
3. To crossprobe from a third-party text file (not source code or a log file), select Options->HDL Analyst Options->General, and enable Enhanced text crossprobing.

4. Select the appropriate portion of text in the Text Editor window. In some cases, it may be necessary to select an entire block of text to crossprobe.

   The software highlights the objects corresponding to the selected code in all the open windows. For example, if you select a state name in the code, it highlights the state in the FSM viewer. If an object is on another schematic sheet or on another hierarchical level, the highlighting might not be obvious. If you filter the RTL or schematic view (right-click in the source code window with the selected text and select Filter Schematic from the popup menu), you can isolate the highlighted objects for easy viewing.

Example of Crossprobing a Path from a Text File

This example selects a path in a log file and crossprobes it in the Technology view. You can use the same technique to crossprobe from other text files like place-and-route files, as long as the instance names in the text file match the instance names in the synthesis tool.

1. Open the log file, the RTL, and Technology views.

2. Select the path objects in the log file.

   – Select the column by pressing Alt and dragging the cursor to the end of the column. On the Linux platform, use the key to which the Alt function is mapped; this is usually the Ctrl-Alt key combination.

   – To select all the objects in the path, right-click and choose Select in Analyst from the popup menu. Alternatively, you can select certain objects only, as described next.

   The software selects the objects in the column, and highlights the path in the open RTL and Technology views.

© 2016 Synopsys, Inc.

Text Editor

Technology View

– To further filter the objects in the path, right-click and choose Select From from the popup menu. On the form, check the objects you want, and click OK. Only the corresponding objects are highlighted.

3. To isolate and view only the selected objects, do this in the Technology view: press F12, or right-click and select the Filter Schematic command from the popup menu.

   You see just the selected objects.

Crossprobing from the Tcl Script Window

Crossprobing from the Tcl script window is useful for debugging error messages.

To crossprobe from the Tcl Script window to the source code, double-click a line in the Tcl window. To crossprobe a warning or error, first click the Messages tab and then double-click the warning or error. The software opens the relevant source code file and highlights the corresponding code.

Crossprobing from the FSM Viewer

You can crossprobe to the FSM Viewer if you have the FSM view open. You can crossprobe from an RTL, Technology, or source code window.

To crossprobe from the FSM Viewer, do the following:

1. Open the view to which you want to crossprobe: RTL/Technology view, or the source code file.

© 2016 Synopsys, Inc.

2. Do the following in the open FSM view:
   - For FSMs with a onehot encoding style, click the state bubbles in the bubble diagram or the states in the FSM transition table.
   - For all other FSMs, click the states in the bubble diagram. You cannot use the transition table because with these encoding styles, the number of registers in the RTL or Technology views do not match the number of registers in the FSM Viewer.

   The software highlights the corresponding code or object in the open views. You can only crossprobe from a state in the FSM table if you used a onehot encoding style.

Analyzing With the HDL Analyst Tool

The HDL Analyst tool is a graphical productivity tool that helps you visualize your synthesis results. It consists of RTL-level and technology-primitive level schematics that let you graphically view and analyze your design.

- RTL View
  Using BEST® (Behavior Extracting Synthesis Technology) in the RTL view, the software keeps a high-level of abstraction and makes the RTL view easy to view and debug. High-level structures like RAMs, ROMs, operators, and FSMs are kept as abstractions in this view instead of being converted to gates. You can examine the high-level structure, or push into a component and view the gate-level structure.
- Technology View
  The software uses module generators to implement the high-level structures from the RTL view, and maps them to technology-specific resources.

To analyze information, compare the current view with the information in the RTL/Technology view, the log file, the FSM view, and the source code, you can use techniques like crossprobing, flattening, and filtering. See the following for more information about analysis techniques.

- Viewing Design Hierarchy and Context, on page 336
- Filtering Schematics, on page 339
- Expanding Pin and Net Logic, on page 341
- Expanding and Viewing Connections, on page 345
- Flattening Schematic Hierarchy, on page 346
- Minimizing Memory Usage While Analyzing Designs, on page 351

For additional information about navigating the HDL Analyst views or using other techniques like crossprobing, see the following:

- Working in the Schematic (Beta), on page 242
- Exploring Design Hierarchy, on page 306
- Finding Objects, on page 314
- Crossprobing, on page 327

© 2016 Synopsys, Inc.

Viewing Design Hierarchy and Context

Most large designs are hierarchical, so the synthesis software provides tools that help you view hierarchy details or put the details in context. Alternatively, you can browse and navigate hierarchy with Push/Pop mode, or flatten the design to view internal hierarchy.

This section describes how to use interactive hierarchical viewing operations to better analyze your design. Automatic hierarchy viewing operations that are built into other commands are described in the context in which they appear. For example, Viewing Critical Paths, on page 361 describes how the software automatically traces a critical path through different hierarchical levels using hollow boxes with nested internal logic (transparent instances) to indicate levels in hierarchical instances.

1. To view the internal logic of primitives in your design, do either of the following:
   - To view the logic of an individual primitive, push into it. This generates a new schematic with the internal details. Click the Back icon to return to the previous view.
   - To view the logic of all primitives in the design, select Options->HDL Analyst Options->General, and enable Show Cell Interior. This command lets you see internal logic in context, by adding the internal details to the current schematic and all subsequent views. If the view is too cluttered with this option on, filter the view (see *Filtering Schematics, on page 339*) or push into the primitive. Click the Back icon to return to the previous view after filtering or pushing into the object.

The following figure compares these two methods:

Result of pushing into a primitive (new view of lower-level logic)

Result of enabling Show Cell Interior option (same view with internal logic)

2. To hide selected hierarchy, select the instance whose hierarchy you want to exclude, and then select Hide Instances from the HDL Analyst menu or the right-click popup menu in the schematic.

   You can hide opaque (solid yellow) or transparent (hollow) instances. The software marks hidden instances with an H in the lower left. Hidden instances are like black boxes; their hierarchy is excluded from filtering, expanding, dissolving, or searching in the current window, although they can be crossprobed. An instance is only hidden in the current view window; other view windows are not affected. Temporarily hiding unnecessary hierarchy focuses analysis and saves time in large designs.

   Before you save a design with hidden instances, select Unhide Instances from the HDL Analyst menu or the right-click popup menu and make the hidden internal hierarchy accessible again. Otherwise, the hidden instances are saved as black boxes, without their internal logic. Conversely, you can use this feature to reduce the scope of analysis in a large design by hiding instances you do not need, saving the reduced design to a new name, and then analyzing it.

3. To view the internal logic of a hierarchical instance, you can push into the instance, dissolve the selected instance with the Dissolve Instances command, or flatten the design. You cannot use these methods to view the internal logic of a hidden instance.

© 2016 Synopsys, Inc.

Pushing into an instance	Generates a view that shows only the internal logic. You do not see the internal hierarchy in context. To return to the previous view, click Back. See *Exploring Object Hierarchy by Pushing/Popping*, on page 307 for details.
Flattening the entire design	Opens a new view where the entire design is flattened, except for hidden hierarchy. Large flattened designs can be overwhelming. See *Flattening Schematic Hierarchy*, on page 346 for details about flattening designs. Because this is a new view, you cannot use Back to return to the previous view. To return to the top-level unflattened schematic, right-click in the view and select Unflatten Schematic.
Flattening an instance by dissolving	Generates a view where the hierarchy of the selected instances is flattened, but the rest of the design is unaffected. This provides context. See *Flattening Schematic Hierarchy*, on page 346 for details about dissolving instances.

4. If the result of filtering or dissolving is a hollow box with no internal logic, try either of the following, as appropriate, to view the internal hierarchy:
   - Select Options->HDL Analyst Options->Sheet Size and increase the value of Maximum Filtered Instances. Use this option if the view is not too cluttered.
   - Use the sheet navigation commands to go to the sheets indicated in the hollow box.

   If there is too much internal logic to display in the current view, the software puts the internal hierarchy on separate schematic sheets. It displays a hollow box with no internal logic and indicates the schematic sheets that contain the internal logic.

5. To view the design context of an instance in a filtered view, select the instance, right-click, and select Show Context from the popup menu.

   The software displays an unfiltered view of the hierarchical level that contains the selected object, with the instance highlighted. This is useful when you have to go back and forth between different views during analysis. The context differs from the Expand commands, which show connections. To return to the original filtered view, click Back.

Filtering Schematics

Filtering is a useful first step in analysis, because it focuses analysis on the relevant parts of the design. Some commands, like the Expand commands, automatically generate filtered views; this procedure only discusses manual filtering, where you use the Filter Schematic command to isolate selected objects. See Chapter 3 of the *Reference Manual* for details about these commands.

This table lists the advantages of using filtering over flattening:

Filter Schematic Command	Flatten Commands
Loads part of the design; better memory usage	Loads entire design
Combine filtering with Push/Pop mode, and history buttons (Back and Forward) to move freely between hierarchical levels	Must use Unflatten Schematic to return to top level, and flatten the design again to see lower levels. Cannot return to previous view if the previous view is not the top-level view.

1. Select the objects that you want to isolate. For example, you can select two connected objects.

   If you filter a hidden instance, the software does not display its internal hierarchy when you filter the design. The following example illustrates this.

2. Select the Filter Schematic command, using one of these methods:
   - Select Filter Schematic from the HDL Analyst menu or the right-click popup menu.
   - Click the Filter Schematic icon (buffer gate) ( ).

© 2016 Synopsys, Inc.

– Press F12.

– Press the right mouse button and draw a narrow V-shaped mouse stroke in the schematic window. See Help->Mouse Stroke Tutor for details.

The software filters the design and displays the selected objects in a filtered view. The title bar indicates that it is a filtered view. Hidden instances have an H in the lower left. The view displays other hierarchical instances as hollow boxes with nested internal logic (transparent instances). For descriptions of filtered views and transparent instances, see Filtered and Unfiltered Schematic Views, on page 96 and Transparent and Opaque Display of Hierarchical Instances, on page 101in the *Reference Manual.* If the transparent instance does not display internal logic, use one of the alternatives described in *Viewing Design Hierarchy and Context*, on page 336, step 4.

3. If the filtered view does not display the pin names of technology primitives and transparent instances that you want to see, do the following:

   – Select Options->HDL Analyst Options->Text and enable Show Pin Name.

   – To temporarily display a pin name, move the cursor over the pin. The name is displayed as long as the cursor remains over the pin. Alternatively, select a pin. The software displays the pin name until you make another selection. Either of these options can be applied to

individual pins. Use them to view just the pin names you need and keep design clutter to a minimum.

– To see all the hierarchical pins, select the instance, right-click, and select Show All Hier Pins.

You can now analyze the problem, and do operations like the following:

Trace paths, build up logic	See *Expanding Pin and Net Logic*, on page 341 and *Expanding and Viewing Connections*, on page 345
Filter further	Select objects and filter again
Find objects	See *Finding Objects*, on page 314
Flatten, or hide and flatten	See *Flattening Schematic Hierarchy*, on page 346. You can hide transparent or opaque instances.
Crossprobe from filtered view	See *Crossprobing from the RTL/Technology View*, on page 328

4. To return to the previous schematic, click the Back icon. If you flattened the hierarchy, right-click and select Unflatten Schematic to return to the top-level unflattened view.

For additional information about filtering schematics, see *Filtering Schematics*, on page 339 and *Flattening Schematic Hierarchy*, on page 346.

Expanding Pin and Net Logic

When you are working in a filtered view, you might need to include more logic in your selected set to debug your design. This section describes commands that expand logic fanning out from pins or nets; to expand paths, see *Expanding and Viewing Connections*, on page 345.

Use the Expand commands with the Filter Schematic, Hide Instances, and Flatten commands to isolate just the logic that you want to examine. Filtering isolates logic, flattening removes hierarchy, and hiding instances prevents their internal hierarchy from being expanded. See *Filtering Schematics*, on page 339 and *Flattening Schematic Hierarchy*, on page 346 for details.

1. To expand logic from a pin hierarchically across boundaries, use the following commands.

© 2016 Synopsys, Inc.

To ...	Do this (HDL Analyst->Hierarchical/Popup menu) ...
See all cells connected to a pin	Select a pin and select Expand. See *Expanding Filtered Logic Example*, on page 343.
See all cells that are connected to a pin, up to the next register	Select a pin and select Expand to Register/Port. See *Expanding Filtered Logic to Register/ Port Example*, on page 344.
See internal cells connected to a pin	Select a pin and select Expand Inwards. The software filters the schematic and displays the internal cells closest to the port. See *Expanding Inwards Example*, on page 344.

The software expands the logic as specified, working on the current level and below or working up the hierarchy, crossing hierarchical boundaries as needed. Hierarchical levels are shown nested in hollow bounding boxes. The internal hierarchy of hidden instances is not displayed.

For descriptions of the Expand commands, see HDL Analyst Menu, on page 302 of the *Reference Manual*.

2. To expand logic from a pin at the current level only, do the following:
   - Select a pin, and go to the HDL Analyst->Current Level menu or the right-click popup menu->Current Level.
   - Select Expand or Expand to Register/Ports. The commands work as described in the previous step, but they do not cross hierarchical boundaries.
3. To expand logic from a net, use the commands shown in the following table.
   - To expand at the current level and below, select the commands from the HDL Analyst->Hierarchical menu or the right-click popup menu.
   - To expand at the current level only, select the commands from the HDL Analyst->Current Level menu or the right-click popup menu->Current Level.

To ...	Do this ...
Select the driver of a net	Select a net and select Select Net Driver. The result is a filtered view with the net driver selected (*Selecting the Net Driver Example*, on page 345).
Trace the driver, across sheets if needed	Select a net and select Go to Net Driver. The software shows a view that includes the net driver.
Select all instances on a net	Select a net and select Select Net Instances. You see a filtered view of all instances connected to the selected net.

Expanding Filtered Logic Example

© 2016 Synopsys, Inc.

Expanding Filtered Logic to Register/Port Example

Expanding Inwards Example

Selecting the Net Driver Example

Expanding and Viewing Connections

This section describes commands that expand logic between two or more objects; to expand logic out from a net or pin, see *Expanding Pin and Net Logic*, on page 341. You can also isolate the critical path or use the Timing Analyst to generate a schematic for a path between objects, as described in Analyzing Timing in Schematic Views, on page 358.

Use the following path commands with the Filter Schematic and Hide Instances commands to isolate just the logic that you want to examine. The two techniques described here differ: Expand Paths expands connections between selected objects, while Isolate Paths pares down the current view to only display connections to and from the selected instance.

For detailed descriptions of the commands mentioned here, see Commands That Result in Filtered Schematics, on page 123 in the *Reference Manual*.

1. To expand and view connections between selected objects, do the following:
   - Select two or more points.
   - To expand the logic at the current level only, select HDL Analyst-> Current Level->Expand Paths or popup menu->Current Level Expand Paths.
   - To expand the logic at the current level and below, select HDL Analyst-> Hierarchical->Expand Paths or popup menu->Expand Paths.

© 2016 Synopsys, Inc.

2. To view connections from all pins of a selected instance, right-click and select Isolate Paths from the popup menu.

Starting Point	The Filtered View Traces Paths (Forward and Back) From All Pins of the Selected Instance...
Filtered view	Traces through all sheets of the filtered view, up to the next port, register, hierarchical instance, or black box.
Unfiltered view	Traces paths on the current schematic sheet only, up to the next port, register, hierarchical instance, or black box.

Unlike the Expand Paths command, the connections are based on the schematic used as the starting point; the software does not add any objects that were not in the starting schematic.

Flattening Schematic Hierarchy

Flattening removes hierarchy so you can view the logic without hierarchical levels. In most cases, you do not have to flatten your hierarchical schematic to debug and analyze your design, because you can use a combination of

filtering, Push/Pop mode, and expanding to view logic at different levels. However, if you must flatten the design, use the following techniques, which include flattening, dissolving, and hiding instances.

1. To flatten an entire design down to logic cells, use one of the following commands:
   - For an RTL view, select HDL Analyst->RTL->Flattened View. This flattens the design to generic logic cells.
   - For a Technology view, select Flattened View or Flattened to Gates View from the HDL Analyst->Technology menu. Use the former command to flatten the design to the technology primitive level, and the latter command to flatten it further to the equivalent Boolean logic.

   The software flattens the top-level design and displays it in a new window. To return to the top-level design, right-click and select Unflatten Schematic.

   Unless you really require the entire design to be flattened, use Push/Pop mode and the filtering commands (*Filtering Schematics*, on page 339) to view the hierarchy. Alternatively, you can use one of the selective flattening techniques described in subsequent steps.

2. To selectively flatten transparent instances when you analyze critical paths or use the Expand commands, select Flatten Current Schematic from the HDL Analyst menu, or select Flatten Schematic from the right-click popup menu.

   The software generates a new view of the current schematic in the same window, with all transparent instances at the current level and below flattened. RTL schematics are flattened down to generic logic cells and Technology views down to technology primitives. To control the number of hierarchical levels that are flattened, use the Dissolve Instances command described in step 4.

   If your view only contains hidden hierarchical instances or pale yellow (opaque) hierarchical instances, nothing is flattened. If you flatten an unfiltered (usually the top-level design) view, the software flattens all hierarchical instances (transparent and opaque) at the current level and below. The following figure shows flattened transparent instances.

© 2016 Synopsys, Inc.

Because the flattened view is a new view, you cannot use Back to return to the unflattened view or the views before it. Use Unflatten Schematic to return to the unflattened top-level view.

3. To selectively flatten the design by hiding instances, select hierarchical instances whose hierarchy you do not want to flatten, right-click, and select Hide Instances. Then flatten the hierarchy using one of the Flatten commands described above.

   Use this technique if you want to flatten most of your design. If you want to flatten only part of your design, use the approach described in the next step.

   When you hide instances, the software generates a new view where the hidden instances are not flattened, but marked with an H in the lower left corner. The rest of the design is flattened. If unhidden hierarchical instances are not flattened by this procedure, use the Flattened View or Flattened to Gates View commands described in step 1 instead of the Flatten

Current Schematic command described in step 2, which only flattens transparent instances in filtered views.

You can select the hidden instances, right-click, and select Unhide Instances to make their hierarchy accessible again. To return to the unflattened top-level view, right-click in the schematic and select Unflatten Schematic.

4. To selectively flatten some hierarchical instances in your design by dissolving them, do the following:
   - If you want to flatten more than one level, select Options->HDL Analyst Options and change the value of Dissolve Levels. If you want to flatten just one level, leave the default setting.
   - Select the instances to be flattened.
   - Right-click and select Dissolve Instances.

   The results differ slightly, depending on the kind of view from which you dissolve instances.

Starting View	Software Generates a ...
Filtered	Filtered view with the internal logic of dissolved instances displayed within hollow bounding boxes (transparent instances), and the hierarchy of the rest of the design unchanged. If the transparent instance does not display internal logic, use one of the alternatives described in step 4 of *Viewing Design Hierarchy and Context*, on page 336. Use the Back button to return to the undissolved view.
Unfiltered	New, flattened view with the dissolved instances flattened in place (no nesting) to Boolean logic, and the hierarchy of the rest of the design unchanged. Select Unflatten Schematic to return to the top-level unflattened view. You cannot use the Back button to return to previous views because this is a new view.

© 2016 Synopsys, Inc.

The following figure illustrates this.

Dissolved logic for prgmcntr shown nested when started from filtered view

Dissolve

Use this technique if you only want to flatten part of your design while retaining the hierarchical context. If you want to flatten most of the design, use the technique described in the previous step. Instead of dissolving instances, you can use a combination of the filtering commands and Push/Pop mode.

Minimizing Memory Usage While Analyzing Designs

When working with large hierarchical designs, use the following techniques to use memory resources efficiently.

- Before you do any analysis operations such as searching, flattening, expanding, or pushing/popping, hide (HDL Analyst->Hide Instances) the hierarchical instances you do not need. This saves memory resources, because the software does not load the hierarchy of the hidden instances.
- Temporarily divide your design into smaller working files. Before you do any analysis, hide the instances you do not need. Save the design. The `srs` and `srm` files generated are smaller because the software does not save the hidden hierarchy. Close any open HDL Analyst windows to free all memory from the large design. In the Implementation Results view, double-click one of the smaller files to open the RTL or Technology schematic. Analyze the design using the smaller, working schematics.
- Filter your design instead of flattening it. If you must flatten your design, hide the instances whose hierarchy you do not need before flattening, or use the Dissolve Instances command. See *Flattening Schematic Hierarchy, on page 346* for details. For more information on the Expand Paths and Isolate Paths commands, see RTL and Technology Views Popup Menus, on page 371 of the *Reference Manual*.
- When searching your design, search by instance rather than by net. Searching by net loads the entire design, which uses memory.
- Limit the scope of a search by hiding instances you do not need to analyze. You can limit the scope further by filtering the schematic in addition to hiding the instances you do not want to search.

© 2016 Synopsys, Inc.

Using the FSM Viewer

The FSM viewer displays state transition bubble diagrams for FSMs in the design, along with additional information about the FSM. You can use this viewer to view state machines.

1. To start the FSM viewer, open the RTL view and either
   - Select the FSM instance, click the right mouse button and select View FSM from the popup menu.
   - Push down into the FSM instance (Push/Pop icon).

   The FSM viewer opens. The viewer consists of a transition bubble diagram and a table for the encodings and transitions. If you used Verilog to define the FSMs, the viewer displays binary values for the state machines if you defined them with the ‘define keyword, and actual names if you used the parameter keyword.

2. The following table summarizes basic viewing operations.

To view ...	Do ...
from and to states, and conditions for each transition	Click the Transitions tab at the bottom of the table.
the correspondence between the states and the FSM registers in the RTL view	Click the RTL Encoding tab.
the correspondence between the states and the registers in the Technology View	Click the Mapped Encodings tab (available after synthesis).
only the transition diagram without the table	Select View->FSM table or click the FSM Table icon. You might have to scroll to the right to see it.

This figure shows you the mapping information for a state machine. The Transitions tab shows you simple equations for conditions for each state. The RTL Encodings tab has a State column that shows the state names in the source code, and a Registers column for the corresponding RTL encoding. The Mapped Encoding tab shows the state names in the code mapped to actual values.

© 2016 Synopsys, Inc.

	From State	To State	Condition
1	LapDisplay	LapDisplay	Laplevel&!Reset
2	Zero	LapDisplay	Lap&!Start&!Reset
3	StopSecond	StopSecond	!Lap&!Start&!Reset
4	StopFirst	StopSecond	Lap&!Start&!Reset
5	CountCont	StopSecond	Start&!Reset
6	StopSecond	StopDiff	Lap&!Start&!Reset
7	StopDiff	StopDiff	!Lap&!Start&!Reset
8	StopFirst	StopFirst	!Lap&!Start&!Reset
9	CountFirst	StopFirst	Start&!Reset
10	StopSecond	CountCont	Start&!Reset

States and Conditions

State	cur_state[6]	cur_state[5]	cur_state[4]	cur_state[3]	cur_state[2]	cur_state[1]	cur_state[0]
Zero	0	0	0	0	0	0	1
CountFirst	0	0	0	0	0	1	0
CountCont	0	0	0	0	1	0	0
StopFirst	0	0	0	1	0	0	0
StopDiff	0	0	1	0	0	0	0
StopSecond	0	1	0	0	0	0	0
LapDisplay	1	0	0	0	0	0	0

Mapped Encoding

State	Register
Zero	cur_state[0]
CountFirst	cur_state[1]
CountCont	cur_state[2]
StopFirst	cur_state[3]
StopDiff	cur_state[4]
StopSecond	cur_state[5]
LapDisplay	cur_state[6]

RTL Encoding

3. To view just one selected state,
   - Select the state by clicking on its bubble. The state is highlighted.
   - Click the right mouse button and select the filtering criteria from the popup menu: output, input, or any transition.

   The transition diagram now shows only the filtered states you set. The following figure shows filtered views for output and input transitions for one state.

Similarly, you can check the relationship between two or more states by selecting the states, filtering them, and checking their properties.

4. To view the properties for a state,
   - Select the state.
   - Click the right mouse button and select Properties from the popup menu. A form shows you the properties for that state.

   To view the properties for the entire state machine like encoding style, number of states, and total number of transitions between states, deselect any selected states, click the right mouse button outside the diagram area, and select Properties from the popup menu.

5. To view the FSM description in text format, select the state machine in the RTL view and View FSM Info File from the right mouse popup. This is an example of the FSM Info File, *statemachine*.info.

```
State Machine: work.Control(verilog)-cur_state[6:0]
No selected encoding - Synplify will choose
Number of states: 7
Number of inputs: 4
Inputs:
   0: Laplevel
   1: Lap
   2: Start
   3: Reset
   Clock: Clk
```


© 2016 Synopsys, Inc.

```
Transitions: (input, start state, destination state)
   -100 S0 S6
   --10 S0 S2
   ---1 S0 S0
   -00- S0 S0
   --10 S1 S3
   -100 S1 S2
   -000 S1 S1
   ---1 S1 S0
   --10 S2 S5
   -000 S2 S2
   -100 S2 S1
   ---1 S2 S0
   -100 S3 S5
   -000 S3 S3
   --10 S3 S1
   ---1 S3 S0
   -000 S4 S4
   --1- S4 S0
   -1-- S4 S0
   ---1 S4 S0
   -000 S5 S5
   -100 S5 S4
   --10 S5 S2
   ---1 S5 S0
   1--0 S6 S6
   ---1 S6 S0
   0--- S6 S0
```

CHAPTER 8

Analyzing Timing

This chapter describes typical analysis tasks. It describes graphical analysis with the HDL Analyst tool as well as interpretation of the text log file. It covers the following:

- Analyzing Timing in Schematic Views, on page 358
- Generating Custom Timing Reports with STA, on page 365
- Using Analysis Design Constraints, on page 368
- Using Auto Constraints, on page 375

© 2016 Synopsys, Inc.

Analyzing Timing in Schematic Views

You can use the HDL Analyst and Timing Analyst functionality to analyze timing. This section describes the following:

- Viewing Timing Information, on page 358
- Annotating Timing Information in the Schematic Views, on page 359
- Analyzing Clock Trees in the RTL View, on page 361
- Viewing Critical Paths, on page 361
- Handling Negative Slack, on page 364
- Generating Custom Timing Reports with STA, on page 365

Viewing Timing Information

Some commands, like Show Critical Path, Hierarchical Critical Path, Flattened Critical Path, automatically enable Show Timing Information and display the timing information. The following procedure shows you how to do so manually.

1. To analyze timing, enable HDL Analyst->Show Timing Information.

   This displays the timing numbers for all instances in a Technology view. It shows the following:

Delay	This is the first number displayed. • Combinational logic This first number is the cumulative path delay to the output of the instance, which includes the net delay of the output. • Flip-flops This first number is the path delay attributed to the flip-flop. The delay can be associated with either the input or output path, whichever is worse, because the flip-flop is the end of one path and the start of another.
Slack Time	This is the second number, and it is the slack time of the worst path that goes through the instance. A negative value indicates that timing constraints can not be met.

Annotating Timing Information in the Schematic Views

You can annotate the schematic views with timing information for the components in the design. Once the design is annotated, you can search for these properties and their associated instances.

1. On the Device tab of the Implementation Options dialog box, enable Annotated Properties for Analyst. This feature is supported on Lattice iCE devices.

For each synthesis implementation and each place-and-route implementation, the tool generates properties and stores them in two files located in the project folder:

`.sap`	Synplify Annotated Properties Contains the annotated design properties generated after compilation, like clock pins.
`.tap`	Timing Annotated Properties Contains the annotated timing properties generated after compilation.

2. To view the annotated timing, open an RTL or Technology view.

3. To view the timing information from another associated implementation, do the following:
   - Open an RTL or Technology view. It displays the timing information for that implementation.
   - Select HDL Analyst->Select Timing, and select another implementation from the list. The list contains the main implementation and all

© 2016 Synopsys, Inc.

associated place-and-route implementations. The timing numbers in the current Analyst view change to reflect the numbers from the selected implementation.

In the following example, an RTL View shows timing data from the test implementation and the test/pr_1 (place and route) implementation.

4. Once you have annotated your design, you can filter searches using these properties with the find command.
   - Use the find -filter {@*propName*>=*propValue*} command for the searches. See Find Filter Properties, on page 119 in the *Command Reference Manual* for a list of properties. For information about the find command, see find, on page 104 in the *Command Reference Manual*.
   - Precede the property name with the @ symbol.

   For example to find fanouts larger than 60, specify find -filter {@fanout>=60}.

Analyzing Clock Trees in the RTL View

To analyze clock trees in the RTL view, do the following:

1. In the Hierarchy Browser, expand Clock Tree, select all the clocks, and filter the design.

   The Hierarchy Browser lists all clocks and the instances that drive them under Clock Tree. The filtered view shows the selected objects.

2. If necessary, use the filter and expand commands to trace clock connections back to the ports and check them.

   For details about the commands for filtering and expanding paths, see *Filtering Schematics*, on page 339, *Expanding Pin and Net Logic*, on page 341 and *Expanding and Viewing Connections*, on page 345.

3. Check that your defined clock constraints cover the objects in the design.

   If you do not define your clock constraints accurately, you might not get the best possible synthesis optimizations.

Viewing Critical Paths

The HDL Analyst tool makes it simple to find and examine critical paths and the relevant source code. The following procedure shows you how to filter and analyze a critical path. You can also use the procedure described in *Generating Custom Timing Reports with STA*, on page 365 to view this and other paths.

1. If needed, set the slack time for your design.
   - Select HDL Analyst->Set Slack Margin.
   - To view only instances with the worst-case slack time, enter a zero.
   - To set a slack margin range, type a value for the slack margin, and click OK. The software gets a range by subtracting this number from the slack time, and the Technology view displays instances within this range. For example, if your slack time is -10 ns, and you set a slack margin of 4 ns, the command displays all instances with slack times between -6 ns and -10 ns. If your slack margin is 6 ns, you see all instances with slack times between -4 ns and -10 ns.

© 2016 Synopsys, Inc.

2. Display the critical path using one of the following methods. The Technology view displays a hierarchical view that highlights the instances and nets in the most critical path of your design.
   - To generate a hierarchical view of the critical path, click the Show Critical Path icon (stopwatch icon ( ), select HDL Analyst->Technology->Hierarchical Critical Path, or select the command from the popup menu. This is a filtered view in the same window, with hierarchical logic shown in transparent instances. History commands apply, so you can return to the previous view by clicking Back.
   - To flatten the hierarchical critical path described above, right-click and select Flatten Schematic. The software generates a new view in the current window, and flattens only the transparent instances needed to show the critical path; the rest of the design remains hierarchical. Click Back to go the top-level design.
   - To generate a flattened critical path in a new window, select HDL Analyst->Technology->Flattened Critical Path. This command uses more memory because it flattens the entire design and generates a new view for the flattened critical path in a new window. Click Back in this window to go to the flattened top-level design or to return to the previous window.

3. Use the timing numbers displayed above each instance to analyze the path. If no numbers are displayed, enable HDL Analyst->Show Timing Information. Interpret the numbers as follows:

   Delay
   For combinational logic, it is the cumulative delay to the output of the instance, including the net delay of the output. For flip-flops, it is the portion of the path delay attributed to the flip-flop. The delay can be associated with either the input path or output path, whichever is worse, because the flip-flop is the end of one path and the start of another.

   Slack time
   Slack of the worst path that goes through the instance. A negative value indicates that timing has not been met.

   8.8, 1.2

4. View instances in the critical path that have less than the worst-case slack time. For additional information on handling slack times, see *Handling Negative Slack*, on page 364.

   If necessary change the slack margin and regenerate the critical path.

5. Crossprobe and check the RTL view and source code. Analyze the code and the schematic to determine how to address the problem. You can add more constraints or make code changes.

6. Click the Back icon to return to the previous view. If you flattened your design during analysis, select Unflatten Schematic to return to the top-level design.

   There is no need to regenerate the critical path, unless you flattened your design during analysis or changed the slack margin. When you flatten your design, the view is regenerated so the history commands do not apply and you must click the Critical Path icon again to see the critical path view.

7. Rerun synthesis, and check your results.

   If you have fixed the path, the window displays the next most critical path when you click the icon.

   Repeat this procedure and fix the design for the remaining critical paths. When you are within 5-10 percent of your desired results, place and route your design to see if you meet your goal. If so, you are done. If your vendor provides timing-driven place and route, you might improve your results further by adding timing constraints to place and route.

© 2016 Synopsys, Inc.

Handling Negative Slack

Positive slack time values (greater than or equal to 0 ns) are good, while negative slack time values (less than 0 ns) indicate the design has not met timing requirements. The negative slack value indicates the amount by which the timing is off because of delays in the critical paths of your design.

The following procedure shows you how to add constraints to correct negative slack values. Timing constraints can improve your design by 10 to 20 percent.

1. Display the critical path in a filtered Technology view.
   - For a hierarchical critical path, either click the Critical Path icon, select HDL Analyst->Show Critical Path, or select HDL Analyst->Technology->Hierarchical Critical Path.
   - For a flat path, select HDL Analyst->Technology->Flattened Critical Path.
2. Analyze the critical path.
   - Check the end points of the path. The start point can be a primary input or a flip-flop. The end point can be a primary output or a flip-flop.
   - Examine the instances. Use the commands described in *Expanding Pin and Net Logic*, on page 341 and *Expanding and Viewing Connections*, on page 345. For more information on filtering schematics, see *Filtering Schematics*, on page 339.
3. Determine whether there is a timing exception, like a false or multicycle path. If this is the cause of the negative slack, set the appropriate timing constraint.

   If there are fewer start points, pick a start point to add the constraint. If there are fewer end points, add the constraint to an end point.
4. If your design does not meet timing by 20 percent or more, you may need to make structural changes. You could do this by doing either of the following:
   - Enabling options like pipelining (Pipelining, on page 418), retiming (Retiming, on page 422), or resource sharing (Sharing Resources, on page 438).
   - Modifying the source code.
5. Rerun synthesis and check your results.

Generating Custom Timing Reports with STA

The log file generated after synthesis includes a timing report and default timing information. Use the stand-alone timing analyst (STA) when you need to generate a customized timing report (`ta`) for the following situations:

- You need more details about a specific path
- You want results for paths other than the top five timing paths (log file default)
- You want to modify constraints and analyze, without resynthesizing. See *Using Analysis Design Constraints*, on page 368 for details.

The following procedure shows you how to generate a custom report:

1. Select Analysis->Timing Analyst or click on the Timing Analyst icon( ).
2. Fill in the parameters.
   - You can type in the from/to or through points, or you can cut and paste or drag and drop valid objects from the Technology view (not the RTL view) into the fields. See Timing Report Generation Parameters, on page 291 in the *Reference Manual* for details on timing analysis parameters and how they can be filtered.
   - Set options for clock reports as needed.
   - Specify a name for the output timing report (`ta`).

© 2016 Synopsys, Inc.

3. Click Generate to run the report.

   The software generates a custom report file called *projectName*.ta, located in the implementation directory (the directory you specified for synthesis results). The software also generates a corresponding output netlist file, with an `srm` extension.

4. Analyze results.
   - View the report (Open Report) in the Text Editor. The following figure is a sample report showing analysis results based on maximum delay for the worst paths.

```
##### START OF TIMING REPORT #####[
# Timing Report written on Mon Jun 05 11:05:33 2006
#


Top view:               sw
Requested Frequency:    10.0 MHz
Wire load mode:         top
Paths requested:        8
from:                   i:swmult2o[11]
Constraint File(s):


Worst From-To Path Information
******************************


Path information for path number 1:
    Requested Period:                       100.000
    - Setup time:                           0.245
    = Required time:                        99.755

    - Propagation time:                     2.215
    = Slack   :                             97.540

    Number of logic level(s):               12
    Starting point:                         swmult2o[11] / regout
    Ending point:                           o_o[12] / datain
    The start point is clocked by           clk_i [rising] on pin clk
    The end   point is clocked by           clk_i [rising] on pin clk

Instance / Net                               Pin        Pin                Arrival     No. of
Name                 Type                    Name       Dir     Delay      Time        Fan Out(s)
-----------------------------------------------------------------------------------------------------
swmult2o[11]         stratixii_lcell_ff      regout     Out     0.094      0.094       -
swmult2o[11]         Net                     -          -       0.311      -           2
swmult1_carry_9      stratixii_lcell_comb    dataf      In      -          0.405       -
```

– View the netlist (View Critical Path) in a Technology view. This Technology view, labeled Timing View in the title bar, shows only the paths you specified in the Timing Analyst dialog box. Note that the Timing Analyst and Show Critical Path commands (and equivalent icons and shortcuts) are disabled whenever the Timing View is active.

© 2016 Synopsys, Inc.

Using Analysis Design Constraints

Besides generating custom timing reports (see *Generating Custom Timing Reports with STA*, on page 365), you can also use the Stand-alone Timing Analyst to create constraints in an adc file. You can use these constraints to experiment with different timing values, or to add or modify timing constraints.

The advantage to using analysis design constraints (ADC) is that you do not have to resynthesize the whole design. This reduces debugging time because you can get a quick estimate, or try out different values. The Standalone Timing Analyst (STA) puts these constraints in an Analysis Design Constraints file (adc). The process for using this file is summarized in the following flow diagram:

See the following for details:

- Scenarios for Using Analysis Design Constraints, on page 369
- Creating an ADC File, on page 370
- Using Object Names Correctly in the adc File, on page 374

Scenarios for Using Analysis Design Constraints

The following describe situations where you can effectively use `adc` constraints to debug, explore options or modify constraints. For details about creating these constraints, see *Creating an ADC File*, on page 370.

- What-if analysis of design performance
  If your design meets the target frequency, you can use `adc` constraints to analyze higher target frequencies, or analyze performance of a module in a different design/technology/target device.
- Constraints on enable registers
  Similarly, you can apply syn_reference_clock on enable registers to analyze if the enables have a regular pattern like clock, or if they operate on a frequency other than clock. For example:

FDC	`create_clock {clk} -name {clk} -freq 100 -clockgroup clk_grp_0`
ADC	`define_attribute {n:en} syn_reference_clock {clk2}` `create_clock {clk2} -name {clk2} -freq 50 -clockgroup clk_grp_1`

- Adding additional timing exceptions
  When you analyze the results of the first synthesis run, you often find functional or clock-to-clock timing exceptions, and you can handle these with `adc` constraints. For example:
  - Applying false paths on synchronization circuitry
  - Adding false paths between clocks belonging to different clock groups

  You must add these constraints to see more critical paths in the design. The `adc` constraints let you add these constraints on the fly, and helps you debug designs faster.
- Modifying timing exceptions that were previously applied
  For example you might want to set a multicycle path constraint for a path that was defined as a false path in the constraint file or vice versa. To modify the timing exception, you must first ignore or reset the timing exception that was set in the constraint file, as described in *Using Analysis Design Constraints*, on page 368, step 3.

© 2016 Synopsys, Inc.

Creating an ADC File

The following procedure explains how to create an `adc` file.

1. Select File->New.
2. Do the following in the dialog box that opens:
   - Select Analysis Constraint File.

   - Type a name and location for the file. The tool automatically assigns the `adc` extension to the filename.
   - Enable Add to Project, and click OK. This opens the text editor where you can specify the new constraints.
3. Type in the constraints you want and save the file. Remember the following when you enter the constraints:
   - Keep in mind that the original `fdc` file has already been applied to the design. Any timing exception constraints in this file must not conflict

with constraints that are already in effect. For example, if there is a conflict when multiple timing exceptions (false path, path delay, and multicycle timing constraints) are applied to the same path, the tool uses this order to resolve conflicts: false path, multicycle path, max delay. See Conflict Resolution for Timing Exceptions, on page 197 for details about how the tool prioritizes timing exceptions.

– The object names must be mapped object names, so use names from the Technology view, not names from the RTL view. Unlike the constraint file (RTL view), the `adc` constraints apply to the mapped database because the database is not remapped with this flow. For more information, see *Using Object Names Correctly in the adc File*, on page 374.

– If you want to modify an existing constraint for a timing exception, you must first reset the original `fdc` constraint, and then apply the new constraint. In the following example the multicycle path constraint was changed to 3:

Original FDC	set_multicycle_path –to [get_cells{a_reg*}] 2
ADC	reset_path –to {get_cells{a_reg*}] set_multicycle_path –to [get_cells{a_reg*}] 3

– When you are done, save and close the file. This adds the file to your project.

© 2016 Synopsys, Inc.

– You can create multiple `adc` files for different purposes. For example, you might want to keep timing exception constraints, I/O constraints, and clock constraints in separate files. If you have an existing `adc` file, use the Add File command to add this file to your project. Select Analysis Design Constraint Files (*.adc) as the file type.

4. Run timing analysis.

   – Select Analysis->Timing Analyst or click the Timing Analyst icon ( ). The Timing Analyst window will look like the example below, with pointers to the `srm` file, the original `fdc` and the new `adc` files you created.

– If you have multiple `adc` files, enable the ones you want.

– If you have a previous run and want to save that report, type a new name for the output `ta` file. If you do not specify a name, the tool overwrites the previous report.

– Fill in other parameters as appropriate, and click Generate.

The tool runs static timing analysis in the same implementation directory as the original implementation. The tool applies the `adc` constraints on top of the `fdc` constraints. Therefore, `adc` constraints affect timing results only if there are no conflicts with `fdc` constraints.

The tool generates a timing report called *_adc.ta and an *_adc.srm file by default. It does not change any synthesis outputs, like the output netlist or timing constraints for place and route.

© 2016 Synopsys, Inc.

5. Analyze the results in the timing report and *_adc.srm file.
6. If you need to resynthesize after analysis, add the adc constraints as an fdc file to the project and rerun synthesis.

Using Object Names Correctly in the adc File

Constraints and collections applied in the constraint file reference the RTL-level database. Synthesis optimizations such as retiming and replication can change object names during mapping because objects may be merged.

The standalone timing analyst does not map objects. It just reads the gate-level object names from the post-mapping database; this is reflected in the Technology view. Therefore, you must define objects either explicitly or with collections from the Technology view when you enter constraints into the adc file. Do not use RTL names when you create these constraints (see *Creating an ADC File*, on page 370 for details of that process).

Example

Assume that register en_reg is replicated during mapping to reduce fanout. Further, registers en_reg and en_reg_rep2 connect to register dataout[31:0]. In this case, if you define the following false path constraint in the adc file, then the standalone timing analyzer does not automatically treat paths from the replicated register en_reg_rep2 as false paths.

```
set_false_path -from {{i:en_reg}} -to {{i:dataout[31:0]}}
```

Unlike constraints in the fdc file, you must specify this replicated register explicitly or as a collection. Only then are all paths properly treated as false paths. So in this example, you must define the following constraints in the adc file:

```
set_false_path  -from {{i:en_reg}}  -to {{i:dataout[31:0]}}

set_false_path  -from {{i:en_reg_rep2}}
   -to {{i:dataout[31:0]}}
```

or

```
define_scope_collection  en_regs {find -seq {i:en_reg*}
   -filter (@name == en_reg || @name == en_reg_rep2)}

set_false_path  -from {{$en_regs}}  -to {{i:dataout[31:0]}}
```

Using Auto Constraints

Auto constraining lets you synthesize with automatic constraints as a first step to get an idea of what you can achieve. Automatic constraints generate the fastest design implementation, so they force the timing engine to work harder. Based on the results from auto-constraining, you can refine the constraints manually later. For an explanation of how auto constraints work, see *Results of Auto Constraints*, on page 377

1. To automatically constrain your design, first do the following:
   - Set your device to a technology that supports auto-constraining. With supported technologies, the Auto Constrain button under Frequency in the Project view is available.

   - Do not define any clocks. If you define clocks using the SCOPE window or a constraint file, or set the frequency in the Project view, the software uses the user-defined create_clock constraints instead of auto constraints.
   - Make sure any multi-cycle or false path constraints are specified on registers.
2. Enable the Auto Constrain button on the left side of the Project view. Alternatively, select Project->Implementation Options->Constraints, and enable the Auto Constrain option there.

© 2016 Synopsys, Inc.

3. If you want to auto constrain I/O paths, select Project->Implementation Options->Constraints and enable Use Clock Period for Unconstrained IO.

   If you do not enable this option, the software only auto constrains flop-to-flop paths. Even when the software auto constrains the I/O paths, it does not generate these constraints for forward-annotation.

4. Synthesize the design.

   The software puts each clock in a separate clock group and adjusts the timing of each clock individually. At different points during synthesis it adjusts the clock period of each clock to be a target percentage of the current clock period, usually 15% - 25%.

   After the clocks, the timing engine constrains I/O paths by setting the default combinational path delay for each I/O path to be one clock period.

   The software writes out the generated constraints in a file called AutoConstraint_*designName*.sdc in the run directory. It also forward-annotates these constraints to the place-and-route tools.

5. Check the results in AutoConstraint_*designName*.sdc and the log file. To open the constraint file as a text file, right-click on the file in the Implementation Results view and select Open as Text.

The flop-to-flop constraints use syntax like the following:

```
create_clock -name {c:leon|clk} -period 13.327 -clockgroup
Autoconstr_clkgroup_0 -rise 0.000 -fall 6.664 -route 0.000
```

6. You can now add this generated constraint file to the project and rerun synthesis with these constraints.

Results of Auto Constraints

This section contains information about the following:

- Stages of the Auto Constrain Algorithm, on page 377
- I/O Constraints and Timing Exceptions, on page 377
- Reports and Forward-annotation, on page 378
- Repeatability of Results, on page 379

Stages of the Auto Constrain Algorithm

To auto constrain, do not define any clocks. When you enable the Auto Constrain option, the synthesis software goes through these stages:

1. It infers every clock in the design.
2. It puts each clock in its own clock group.
3. It invokes mapper optimizations in stages and generates the best possible synthesis results. You should only use Auto Constrain early in the synthesis process to get a general idea of how fast your design runs. This option is not meant to be a substitute for declaring clocks.
4. For each clock, including the system clock, the software maintains a negative slack of between 15 and 25 percent of the requested frequency.

I/O Constraints and Timing Exceptions

The auto constrain algorithm infers all the clocks, because none are defined. It handles the following timing situations as described below:

- I/O constraints

  You can auto constrain I/O paths as well as flop-to-flop paths by selecting Project->Implementation Options->Constraints and enabling Use Clock

© 2016 Synopsys, Inc.

Period for Unconstrained IO. The software does not write out these I/O constraints.

- Timing exceptions like multicycle and false paths

  The auto constraint algorithm honors SCOPE multicycle and false path constraints that are specified as constraints on registers.

Auto Constrain Limitations

The Auto Constrain feature has the following limitations:

- Does not respect the vendor-provided maximum frequency constraints for clock generators.
- Over constrains designs with output critical paths.

Reports and Forward-annotation

In the log file, the software reports the Requested and Estimated Frequency or Requested and Estimated Period and the negative slack for each clock it infers. The log file contains all the details.

The software also generates a constraint file in the run directory called AutoConstraint_*designName*.sdc, which contains the auto constraints generated. The following is an example of an auto constraint file:

```
#Begin clock constraint

create_clock -name {c:leon|clk} -period 13.327 -rise 0.000 -fall
6.664

#End clock constraint
```

The software forward-annotates the create_clock constraints, writing out the appropriate file for the place-and-route tool.

Repeatability of Results

If you use the requested frequency resulting from the Auto constrain option as the requested frequency for a regular synthesis run, you might not get the same results as you did with auto constraints. This is because the software invokes the mapper optimizations in stages when it auto constrains. The results from a previous stage are used to drive the next stage. As the interim optimization results vary, there is no guarantee that the final results will stay the same.

© 2016 Synopsys, Inc.

CHAPTER 9

Inferring High-Level Objects

This chapter contains guidelines on how to structure your code or attach attributes so that the synthesis tools can automatically infer high-level objects like RAMs. See the following for more information:

- Defining Black Boxes for Synthesis, on page 382
- Defining State Machines for Synthesis, on page 391
- Specifying Safe FSMs, on page 396
- Automatic RAM Inference, on page 398
- Initializing RAMs, on page 407

© 2016 Synopsys, Inc.

Defining Black Boxes for Synthesis

Black boxes are predefined components for which the interface is specified, but whose internal architectural statements are ignored. They are used as place holders for IP blocks, legacy designs, or a design under development.

This section discusses the following topics:

- Instantiating Black Boxes and I/Os in Verilog, on page 382
- Instantiating Black Boxes and I/Os in VHDL, on page 384
- Adding Black Box Timing Constraints, on page 386
- Adding Other Black Box Attributes, on page 390

For information about using black boxes with the Clock Conversion option, see *Working with Gated Clocks*, on page 574.

Instantiating Black Boxes and I/Os in Verilog

Verilog black boxes for macros and I/Os come from two sources: commonly-used or vendor-specific components that are predefined in Verilog macro libraries, or black boxes that are defined in another input source like a schematic. For information about instantiating black boxes in VHDL, see Instantiating Black Boxes and I/Os in VHDL, on page 384. Additional information about black boxes can be found in *Working with Gated Clocks*, on page 574.

The following process shows you how to instantiate both types as black boxes. Refer to the *installDirectory*/examples directory for examples of instantiations of low-level resources.

1. To instantiate a predefined Verilog module as a black box:
   - Select the library file with the macro you need from the *installDirectory*/lib/*technology* directory. Files are named *technology*.v. Most vendor architectures provide macro libraries that predefine the black boxes for primitives and macros.
   - Make sure the library macro file is the first file in the source file list for your project.

2. To instantiate a module that has been defined in another input source as a black box:
   - Create an empty macro that only contains ports and port directions.
   - Put the syn_black_box synthesis directive just before the semicolon in the module declaration.

```
module myram (out, in, addr, we) /* synthesis syn_black_box */;
   output [15:0] out;
   input [15:0] in;
   input [4:0] addr;
   input we;
endmodule
```

   - Make an instance of the stub in your design.
   - Compile the stub along with the module containing the instantiation of the stub.
   - To simulate with a Verilog simulator, you must have a functional description of the black box. To make sure the synthesis software ignores the functional description and treats it as a black box, use the translate_off and translate_on constructs. For example:

```
module adder8(cout, sum, a, b, cin);
// Code that you want to synthesize
/* synthesis translate_off */
// Functional description.
/* synthesis translate_on */
// Other code that you want to synthesize.
endmodule
```

3. To instantiate a vendor-specific (black box) I/O that has been defined in another input source:
   - Create an empty macro that only contains ports and port directions.
   - Put the syn_black_box synthesis directive just before the semicolon in the module declaration.
   - Specify the external pad pin with the black_box_pad_pin directive, as in this example:

```
module BBDLHS(D,E,GIN,GOUT,PAD,Q)
      /* synthesis syn_black_box black_box_pad_pin="PAD"
```

   - Make an instance of the stub in your design.

© 2016 Synopsys, Inc.

- Compile the stub along with the module containing the instantiation of the stub.

4. Add timing constraints and attributes as needed. See *Adding Black Box Timing Constraints*, on page 386 and *Adding Other Black Box Attributes*, on page 390.

5. After synthesis, merge the black box netlist and the synthesis results file using the method specified by your vendor.

Instantiating Black Boxes and I/Os in VHDL

VHDL black boxes for macros and I/Os come from two sources: commonly-used or vendor-specific components that are predefined in VHDL macro libraries, or black boxes that are defined in another input source like a schematic. For information about instantiating black boxes in VHDL, see *Instantiating Black Boxes and I/Os in Verilog*, on page 382.

The following process shows you how to instantiate both types as black boxes. Refer to the *installDirectory*/examples directory for examples of instantiations of low-level resources.

1. To instantiate a predefined VHDL macro (for a component or an I/O),
   - Select the library file with the macro you need from the *installDirectory*/lib/*vendor* directory. Files are named *family*.vhd. Most vendor architectures provide macro libraries that predefine the black boxes for primitives and macros.
   - Add the appropriate library and use clauses to the beginning of your design units that instantiate the macros.

     ```
     library family;
     use family.components.all;
     ```

2. To create a black box for a component from another input source:
   - Create a component declaration for the black box.
   - Declare the syn_black_box attribute as a boolean attribute.
   - Set the attribute to true.

```
library synplify;
use synplify.attributes.all;
entity top is
   port (clk, rst, en, data: in bit; q: out bit);
end top;

architecture structural of top is
component bbox
   port(Q: out bit; D, C, CLR: in bit);
end component;

attribute syn_black_box of bbox: component is true;
...
```

– Instantiate the black box and connect the ports.

```
begin
my_bbox: bbox port map (
   Q => q,
   D => data,
   C => clk,
   CLR => rst);
```

– To simulate with a VHDL simulator, you must have the functional description of a black box. To make sure the synthesis software ignores the functional description and treats it as a black box, use the translate_off and translate_on constructs. For example:

```
architecture behave of ram4 is
begin
   -- synthesis translate_off
      stimulus: process (clk, a, b)
   -- Functional description
end process;
   -- synthesis translate_on

-- Other source code you WANT synthesized
```

3. To create a vendor-specific (black box) I/O for an I/O defined in another input source:
   – Create a component declaration for the I/O.
   – Declare the black_box_pad_pin attribute as a string attribute.
   – Set the attribute value on the component to be the external pin name for the pad.

© 2016 Synopsys, Inc.

```
library synplify;
use synplify.attributes.all;
...

component mybuf
   port(O: out bit; I: in bit);
end component;
attribute black_box_pad_pin of mybuf: component is "I";
```

– Instantiate the pad and connect the signals.

```
begin
data_pad: mybuf port map (
   O => data_core,
   I => data);
```

4. Add timing constraints and attributes. See Adding Black Box Timing Constraints, on page 386, *Using Gated Clocks for Black Boxes*, on page 597, and Adding Other Black Box Attributes, on page 390.

Adding Black Box Timing Constraints

A black box does not provide the software with any information about internal timing characteristics. You must characterize black box timing accurately, because it can critically affect the overall timing of the design. To do this, you add constraints in the source code or in the SCOPE interface.

You attach black box timing constraints to instances that have been defined as black boxes. There are three black box timing constraints, syn_tpd, syn_tsu, and syn_tco. There are additional attributes for black box pins and black boxes with gated clocks; see Adding Other Black Box Attributes, on page 390 and *Using Gated Clocks for Black Boxes*, on page 597.

1. Define the instance as a black box, as described in *Instantiating Black Boxes and I/Os in Verilog*, on page 382 or *Instantiating Black Boxes and I/Os in VHDL*, on page 384.

2. Determine the kind of constraint for the information you want to specify:

To define ...	Use ...
Propagation delay through the black box	syn_tpd
Setup delay (relative to the clock) for input pins	syn_tsu
Clock-to-output delay through the black box	syn_tco

3. In VHDL, use the following syntax for the constraints.
   – Use the predefined attributes package by adding this syntax

```
library synplify;
use synplify.attributes.all;
```

   In VHDL, you must use the predefined attributes package. For each directive, there are ten predeclared constraints in the attributes package, from *directive_name1* to *directive_name10*. If you need more constraints, declare the additional constraints using integers greater than 10. For example:

```
attribute syn_tco11 : string;
attribute syn_tco12 : string;
```

   – Define the constraints in either of these ways:

VHDL syntax	`attribute attributeName<n> : "att_value"`
Verilog-style notation	`attribute attributeName<n> of bbox_name : component is "att_value"`

   The following table shows the appropriate syntax for att_value. See the *Attribute Reference Manual* for complete syntax information.

© 2016 Synopsys, Inc.

Attribute	Value Syntax
syn_tsu<*n*>	*bundle* -> [!]*clock* = *value*
syn_tco<*n*>	[!]*clock* -> *bundle* = *value*
syn_tpd<*n*>	*bundle* -> *bundle* = *value*

- <*n*> is a numerical suffix.
- *bundle* is a comma-separated list of buses and scalar signals, with no intervening spaces. For example, A,B,C.
- ! indicates (optionally) a negative edge for a clock.
- *value* is in ns.

The following is an example of black box attributes, using VHDL signal notation:

```
architecture top of top is
component rcf16x4z port(
   ad0, ad1, ad2, ad3 : in std_logic;
   di0, di1, di2, di3 : in std_logic;
   wren, wpe : in std_logic;
   tri : in std_logic;
   do0, do1, do2 do3 : out std_logic;
end component

attribute syn_tpd1 of rcf16x4z : component is
   "ad0,ad1,ad2,ad3 -> do0,do1,do2,do3 = 2.1";
attribute syn_tpd2 of rcf16x4z : component is
   "tri -> do0,do1,do2,do3 = 2.0";
attribute syn_tsu1 of rcf16x4z : component is
   "ad0,ad1,ad2,ad3 -> ck = 1.2";
attribute syn_tsu2 of rcf16x4z : component is
   "wren,wpe,do0,do1,do2,do3 -> ck = 0.0";
```

4. In Verilog, add the directives as comments, as shown in the following example. For explanations about the syntax, see the table in the previous step or the *Attribute Reference Manual.*

```
module ram32x4 (z, d, addr, we, clk)
   /* synthesis syn_black_box
   syn_tpd1="addr[3:0]->z[3:0]=8.0"
   syn_tsu1="addr[3:0]->clk=2.0"
   syn_tsu2="we->clk=3.0" */;
output [3:0] z;
```

```
input [3:0] d;
input [3:0] addr;
input we;
input clk;
endmodule
```

5. To add black box attributes through the SCOPE interface, do the following:
   - Open the SCOPE spreadsheet and select the Attributes panel.
   - In the Object column, select the name of the black-box module or component declaration from the pull-down list. Manually prefix the black box name with **v:** to apply the constraint to the view.
   - In the Attribute column, type the name of the timing attribute, followed by the numerical suffix, as shown in the following table. You cannot select timing attributes from the pull-down list.
   - In the Value column, type the appropriate value syntax, as shown in the table in step 3.
   - Save the constraint file, and add it to the project.

   The resulting constraint file contains syntax like this:

   define_attribute v:{*blackboxModule***}** *attribute<n>* **{***attributeValue***}**

6. Synthesize the design and check black box timing.

© 2016 Synopsys, Inc.

Adding Other Black Box Attributes

Besides black box timing constraints, you can also add other attributes to define pin types on the black box or define gated clocks. You cannot use the attributes for all technologies. Check the *Attribute Reference Manual* for details about which technologies are supported. For information about black boxes with gated clocks, see *Using Gated Clocks for Black Boxes*, on page 597.

1. To specify that a clock pin on the black box has access to global clock routing resources, use syn_isclock.
2. To specify that the software need not insert a pad for a black box pin, use black_box_pad_pin. Use this for technologies that automatically insert pad buffers for the I/Os.
3. To define a tristate pin so that you do not get a mixed driver error when there is another tristate buffer driving the same net, use black_box_tri_pins.

Defining State Machines for Synthesis

A finite state machine (FSM) is a piece of hardware that advances from state to state at a clock edge. The synthesis software recognizes and extracts the state machines from the HDL source code. For guidelines on setting up the source code, see the following:

- Defining State Machines in Verilog, on page 391
- Defining State Machines in VHDL, on page 392
- Specifying FSMs with Attributes and Directives, on page 393

For information about the attributes used to define state machines, see Running the FSM Compiler, on page 442. For information about implementing safe FSMs, see *Specifying Safe FSMs*, on page 396.

Defining State Machines in Verilog

The synthesis software recognizes and automatically extracts state machines from the Verilog source code if you follow the coding guidelines listed below. The software attaches the syn_state_machine attribute to each extracted FSM.

For alternative ways to define state machines, see *Defining State Machines for Synthesis*, on page 391.

Follow these Verilog coding guidelines:

- In Verilog, model the state machine with case, casex, or casez statements in always blocks. Check the current state to advance to the next state and then set output values. Do not use if statements.
- Always use a default assignment as the last assignment in the case statement, and set the state variable to ‘bx. This is a “don’t care” statement and ensures that the software can remove unnecessary decoding and gates.
- Make sure the state machines have a synchronous or asynchronous reset to set the hardware to a valid state after power-up, or to reset the hardware when you are operating.

© 2016 Synopsys, Inc.

- Specify explicit state values for states with parameter or ‘define statements. This is an example of a parameter statement that sets the current state to 2’h2:

  ```
  parameter state1 = 2’h1, state2 = 2’h2;
  ...
  current_state = state2;
  ```

  This example shows how to set the current state value with `define statements:

  ```
  ‘define state1 2’h1
  ‘define state2 2’h2
  ...
  current_state = ‘state2;
  ```

- Make state assignments using parameter with symbolic state names. Use parameter over `define, because `define is applied globally while parameter definitions are local. Local definitions make it easier to reuse common state names in multiple FSM designs, like RESET, IDLE, READY, READ, WRITE, ERROR, and DONE.

  If you use `define to assign the names, you cannot reuse a state name because it has already been used in the global name space. To reuse the same names in this scenario, you have to use `undef and `define statements between modules to redefine the names. This method makes it difficult to probe the internal values of FSM state buses from a testbench and compare them to the state names.

Defining State Machines in VHDL

The synthesis software recognizes and automatically extracts state machines from the VHDL source code if you follow the coding guidelines below. For alternative ways to define state machines, see *Defining State Machines for Synthesis*, on page 391.

The following are VHDL guidelines for coding. The software attaches the syn_state_machine attribute to each extracted FSM.

- Use case statements to check the current state at the clock edge, advance to the next state, and set output values. You can also use if-then-else statements, but case statements are preferable.

- If you do not cover all possible cases explicitly, include a when others assignment as the last assignment of the case statement, and set the state vector to some valid state.
- If you create implicit state machines with multiple `WAIT` statements, the software does not recognize them as state machines.
- Make sure the state machines have a synchronous or asynchronous reset to set the hardware to a valid state after power-up, or to reset the hardware when you are operating.
- To choose an encoding style, attach the syn_encoding attribute to the enumerated type. The software automatically encodes your state machine with the style you specified.

Specifying FSMs with Attributes and Directives

If your design has state machines, the software can extract them automatically with the FSM Compiler, or you can manually attach attributes to state registers to define them as state machines. See Optimizing State Machines, on page 441 for information about automatic FSM extraction, and *Defining State Machines for Synthesis*, on page 391 for other ways to specify FSMs.

The following steps show you how to manually attach attributes to define FSMs for extraction.

1. To determine how state machines are extracted, set attributes in the source code as shown in the following table:

To ...	Attribute
Specify a state machine for extraction and optimization	syn_state_machine=1
Prevent state machines from being extracted and optimized	syn_state_machine=0
Prevent the state machine from being optimized away	syn_preserve=1

For information about how to add attributes, see *Specifying Attributes and Directives*, on page 99.

© 2016 Synopsys, Inc.

2. To determine the encoding style for the state machine, set the syn_encoding attribute in the source code or in the SCOPE window. For VHDL users there are alternative methods, described in the next step.

   The FSM Compiler honors the syn_encoding setting. The different values for this attribute are briefly described here; refer to the *Attributes Reference* manual for complete details.

Situation: If ...	syn_encoding Value	Explanation
Area is important	sequential	One of the smallest encoding styles.
Speed is important	onehot	Usually the fastest style and suited to most FPGA styles.
Recovery from an invalid state is important	safe, with another style. For example: `/* synthesis syn_encoding = "safe, onehot" */`	Forces the state machine to reset in certain situations. For example, if an alpha particle hit in a hostile operating environment causes a spontaneous register change, you can use safe to reset the state machine. For further information, see *Specifying Safe FSMs*, on page 396.
There are <5 states	sequential	Default encoding.
A large output decoder follows the FSM	sequential | gray	Could be faster than onehot, even though the value must be decoded to determine the state. For sequential, more than one bit can change at a time; for gray, only one bit changes at a time, but more than one bit can be hot.
There are a large number of flip-flops	onehot	Fastest style, because each state variable has one bit set, and only one bit of the state register changes at a time.

3. If you are using VHDL, you have two choices for defining encoding:
   – Use syn_encoding as described above and enable the FSM compiler.

– Use syn_enum_encoding to define the states (sequential, onehot, gray, and safe) and disable the FSM compiler. If you do not disable the FSM compiler, the syn_enum_encoding values are not implemented. This is because the FSM compiler, a mapper operation, overrides syn_enum_encoding, which is a compiler directive. Use the syn_enum_encoding method for user-defined FSM encoding. For example:

```
attribute syn_enum_encoding of state_type : type is "001 010 101";
```


© 2016 Synopsys, Inc.

Specifying Safe FSMs

Typically, unspecified or unreachable FSM states are optimized away during synthesis, so if an SEU causes a bit to be inverted, the FSM can be put into an undefined, invalid state, and lock up the circuit. An SEU fault is a change of state caused by ions or electro-magnetic radiation that affects sequential elements. The basic principle of a safe FSM is to prevent the state machine from getting stuck in an unknown state because of an SEU.

Safe FSMs are primarily required by commercial or military-aerospace users, especially those who want to ensure that their FSMs are tolerant of single event upset (SEU) faults and continue to function correctly.

Implementing Safe Case FSMs, on page 396 describes ways to ensure high reliability and fault tolerance for FSMs:

Vendor Support for Safe FSMs

The following Lattice technologies support the specification of safe case FSMs:

Lattice	ECP3, MachXO2

Implementing Safe Case FSMs

To implement safe case FSMs, follow this procedure:

1. Select a supported device in the synthesis tool.

   See *Vendor Support for Safe FSMs*, on page 396 for a list.

2. To globally implement safe case FSMs, go to the Implementation Options-> High Reliability tab, and enable the Preserve and Decode Unreachable States (FSM, Counters, Sequential Logic) option.

The high reliability safe case option turns off sequential optimizations that would otherwise optimize away some FSM states.

3. To apply safe case on an individual module or architecture, set the syn_safe_case directive on a module.

   This is a Verilog example: *module /** **syn_safe_case =1***/*

   For details about this directive, see syn_safe_case, on page 223 in the *Attribute Reference Manual*.

4. To choose an encoding style that can be implemented when building the recovery logic for the FSM, use the syn_encoding attribute. The software honors the values you specify for the encoding style. You can apply the attribute globally on a module or architecture in the Verilog/VHDL source code, as well as, the FDC constraint file. For example:

   ```
   module /*synthesis syn_encoding="encodingStyles";*/
   ```

   ```
   attribute syn_encoding of architecture : signal is
      "encodingStyles";
   ```

   You can also define a SCOPE collection in the constraint file, then apply the attribute to the collection:

   ```
   define_scope_collection sm {find -hier -inst * -filter
      inst_of==statemachine}
   define_attribute {$sm} {syn_encoding} {encodingStyles}
   ```

   For details about this attribute, see syn_encoding, on page 67.

   You must enable the FSM Compiler option to ensure that the syn_encoding attribute takes affect. This overrides the default FSM compiler encoding for the state machine.

© 2016 Synopsys, Inc.

Automatic RAM Inference

Instead of instantiating synchronous RAMs, you can let the synthesis tools automatically infer them directly from the HDL source code and map them to the appropriate technology-specific RAM resources on the FPGA. This approach lets you maintain portability.

Here are some of the advantages offered by the inference approach:

- The tool automatically infers the RAMs from the HDL code, which is technology-independent. This means that the design is portable from one technology to another without rework.
- RAM inference is the best method for prototyping.
- The tool automatically adds the extra glue logic needed to ensure that the logic is correct.
- The software automatically runs timing-driven synthesis for inferred RAMs.

For further details about RAM inference, see *Inferring Block RAM*, on page 401.

Block RAM

The synthesis software can implement the block RAM it infers using different types of block RAM and different block RAM modes.

Types of Block RAM

The synthesis software can infer different kinds of block RAM, according to how the code is set up. For details about block RAM inference, see *Inferring Block RAM*, on page 401 and *RAM Attributes*, on page 399. For inference examples, and see Block RAM Examples, on page 291.

The synthesis tool can infer the following kinds of block RAM:

- Single-port RAM
- Dual-port RAM

Based on how the read and write ports are used, dual-port RAM can be further classified as follows:

- Simple dual-port
- Dual-port
- True dual-port

Supported Block RAM Modes

Block RAM supports three operating modes, which determine the output of the RAM when write enable is active. The synthesis tools infer the mode from the RTL you provide. It is best to explicitly describe the RAM behavior in the code, so as to correctly infer the operating mode you want. Refer to the examples for recommended coding styles.

The block RAM operating modes are described in the following table:

Mode	When write enable (WE) is active ...
WRITE_FIRST	This is a transparent mode, and the input data is simultaneously written into memory and stored in the RAM data output (DO). DO uses the value of the RAM data input (DI). See WRITE_FIRST Mode Example, on page 291 for an example.
READ_FIRST	This mode is read before write. The data previously stored at the write address appears at the RAM data output (DO) first, and then the RAM input data is stored in memory. DO uses the value of the memory content. See READ_FIRST Mode Example, on page 293 for an example.
NO_CHANGE	RAM data output (DO) remains the same during a write operation, with DO containing the last read data. See NO_CHANGE Mode Example, on page 294 for an example.

RAM Attributes

In addition to the automatic inference by the tool, you can specify RAM inference with the syn_ramstyle and syn_rw_conflict_logic attributes. The syn_ramstyle attribute explicitly specifies the kind of RAM you want, while the syn_rw_conflict_logic attribute specifies that you want to infer a RAM, but leave it to the synthesis tools to select the kind of RAM, as appropriate.

© 2016 Synopsys, Inc.

Attribute-Based Inference of Block RAM

For block RAM, the syn_ramstyle attribute has a number of valid values, all of which are extensively described in the documentation. This section confines itself to the following values, which are most relevant to the discussion:

syn_ramstyle Value	Description
block_ram	Enforces the inference and implementation of a technology-specific RAM.
registers	Prevents inference of a RAM, and maps the RAM to flip-flops and logic.
no_rw_check	Does not create overhead logic to account for read-write conflicts.

If you specify the syn_rw_conflict_logic attribute, the synthesis tools can infer block RAM, depending on the design. If the tool does infer block RAM, it does not insert bypass logic around the block RAM to account for read-write conflicts and prevent simulation mismatches. In this way its functionality is the same as syn_ramstyle with no_rw_check, which does not insert bypass logic either.

Specifying the Attributes

You set the attribute in the HDL source code, through the SCOPE interface or in an FPGA constraint file.

HDL Source Code

Set the attribute on the Verilog register or VHDL signal that holds the output values of the RAM. The following syntax shows how to specify the attribute in Verilog and VHDL code:

Verilog
```
reg [7:0] ram_dout [127:0]
/*synthesis syn_ramstyle = "block_ram"*/;
reg [d_width-1:0] mem [mem_depth-1:0]
/*synthesis syn_rw_conflict_logic = 0*/;
```

VHDL
```
attribute syn_ramstyle of ram_dout : signal is "block_ram";
```

SCOPE

For the syn_ramstyle attribute, set the attribute on the RAM register memory signal, mem, as shown below. For the syn_rw_conflict_logic attribute, set it on the instance or set it globally. The attributes are written out to a constraints file using the syntax described in the next section.

	Enabled	Object Type	Object	Attribute	Value	Val Type
1	☑	<any>	i:mem[7:0]	syn_ramstyle	block_ram	string

Constraints File

In the fdc Tcl constraints file written out from the SCOPE interface, the syn_ramstyle attribute is attached to the register mem signal of the RAM, and the syn_rw_conflict_logic attribute is attached to the view, as shown below:

```
define_attribute {i:mem[7:0]} {syn_ramstyle} {block_ram}

define_attribute {v:mem[0:7]} syn_rw_conflict_logic {0}
```

For the syn_rw_conflict_logic attribute, you can also specify it globally, as well as on individual modules and instances:

```
define_global_attribute syn_rw_conflict_logic {0}
```

Inferring Block RAM

Based on the design and how you code it, the tool can infer the following kinds of block RAM: single-port, simple dual-port, dual-port, and true dual-port. The details about RAM inference and setup guidelines are described here:

- Setting up the RTL and Inferring Block RAM, on page 402
- Simple Dual-Port Block RAM Inference, on page 403
- Dual-Port RAM Inference, on page 405
- True Dual-Port RAM Inference, on page 405

© 2016 Synopsys, Inc.

Setting up the RTL and Inferring Block RAM

To ensure that the tool infers the kind of block RAM you want, do the following:

1. Set up the RAM HDL code in accordance with the following guidelines:
   - The RAM must be synchronous. It must not have any asynchronous control signals connected. The synthesis tools do not infer asynchronous block RAM.
   - You must register either the read address or the output.
   - The RAMs must not be too small, as the tool does not infer block RAM for small-sized RAMs. The size threshold varies with the target technology.
2. Set up the clocks and read and write ports to infer the kind of RAM you want. The following table summarizes how to set up the RAM in the RTL:

RAM	Clock	Read Ports	Write Ports
Single-port	Single clock	One; same as write	One; same as read
Simple dual-port	Single or dual clock	One dedicated read	One dedicated write
Dual-port	Single or dual clock	Two independent reads	One dedicated write
True dual-port	Single or dual clock	Two independent reads	Two independent writes

See *Dual-Port RAM Inference*, on page 405 and *True Dual-Port RAM Inference*, on page 405 for additional information.

   For illustrative code examples, see the single-port and dual-port examples listed in Block RAM Examples, on page 291.

3. If needed, guide automatic inference with the syn_ramstyle attribute:
   - To force the inference of block RAM, specify syn_ramstyle=blockram.
   - To prevent a block RAM from being inferred or if your resources are limited, use syn_ramstyle=registers.
   - If you know your design does not read and write to the same address simultaneously, specify syn_ramstyle=no_rw_check to ensure that the

synthesis tool does not unnecessarily create bypass logic for resolving conflicts.

4. Synthesize the design.

   The tool first compiles the design and infers the RAMs, which it represents as abstract technology-independent primitives like RAM1 and RAM2. You can view these RAMs in the RTL view, which is a graphic, technology-independent representation of your design after compilation:

   It is important that the compiler first infers the RAM, because the tool only maps the inferred RAM primitives to technology-specific block RAM. Any RAM that is not inferred is mapped to registers. You can view the mapped RAMs in the Technology view, which is a graphic representation of your design after synthesis, and shows the design mapped to technology-specific resources.

Simple Dual-Port Block RAM Inference

Simple dual-port RAMs (SDP) are block RAMs with one port dedicated to read operations and one port dedicated to write operations. SDP RAMs offer the unique advantage of combining ports and using them to pack double the data width and address width.

The synthesis tools map SDP RAMs to RAM primitives in the architecture. A unique set of addresses, clocks, and enable signals are used for each port. The synthesis tool might also set the RAM_MODE property on the RAM to indicate the RAM mode.

The inference of simple dual-port RAM is dependent on the size of the address and data. The RAM must follow the coding guidelines listed below to be inferred.

- The read and write addresses must be different

© 2016 Synopsys, Inc.

- The read and write clocks can be different
- The enable signals can be different

Here is an example where the tool infers SDP RAM:

```
module Read_First_RAM (
   read_clk,
   read_address,
   data_in,
   write_clk,
   rd_en,
   wr_en,
   reg_en,
   write_address,
   data_out);

parameter address_width = 8;
parameter data_width = 32;
parameter depth = 256;
input read_clk, write_clk;
input rd_en;
input wr_en;
input reg_en;
input [address_width-1:0] read_address, write_address;
input [data_width-1:0] data_in;
output [data_width-1:0] data_out;
//wire [data_width-1:0] data_out;
reg [data_width-1:0] mem [depth -1 : 0]/* synthesis
syn_ramstyle="no_rw_check"
   */;
reg [data_width-1:0] data_out;

always @(posedge write_clk)
if(wr_en)
   mem[write_address] <= data_in;

always @(posedge read_clk)
if(rd_en)
   data_out <= mem[read_address];

endmodule
```

Dual-Port RAM Inference

Dual-port RAM is configured to have read and/or write operations from both ports of the RAM. One such configuration is a RAM with one port for both read and write operations and another dedicated read-only port. A unique set of addresses, clocks, and enable signals are used for each port. The synthesis tool sets properties on the RAM to indicate the RAM mode.

To infer dual-port block RAM, the RAM must follow the coding rules described below.

- The read and write addresses must be different
- The read and write clocks can be different
- The enable signals can be different

True Dual-Port RAM Inference

True dual-port RAMs (TDP) are block RAMs with two write ports and two read ports. The compiler extracts a RAM2 primitive for RAMs with two write ports or two read ports and the tool maps this primitive to TDP RAM. The ports operate independently, with different clocks, addresses and enables.

The synthesis tool also sets the RAM_MODE property on the RAM to indicate the RAM mode.

The compiler infers TDP block RAM based on the write processes. The implementation depends on whether the write enables use one process or multiple processes:

- When all the writes are made in one process, there are no address conflicts, and the compiler generates an nram that is later mapped to either true dual-port block RAM. The following coding results in an nram

© 2016 Synopsys, Inc.

with two write ports, one with write address waddr0 and the other with write address waddr1:

```
always @(posedge clk)
begin
   if(we1) mem[waddr0] <= data1;
   if(we2) mem[waddr1] <= data2;
end
```

- When the writes are made in multiple processes, the software does not infer a multiport RAM unless you explicitly specify the syn_ramstyle attribute with a value that indicates the kind of RAM to implement, or with the no_rw_check value. If the attribute is not specified as such, the software does not infer an nram, but infers a RAM with multiple write ports. You get a warning about simulation mismatches when the two addresses are the same.

  In the following case, the compiler infers an nram with two write ports because the syn_ramstyle attribute is specified. The writes associated with waddr0 and waddr1 are we1 and we2, respectively.

```
reg [1:0] mem [7:0] /* synthesis syn_ramstyle="no_rw_check" */;
always @(posedge clk1)
begin
   if(we1) mem[waddr0] <= data1;
end

always @(posedge clk2)
begin
   if(we2) mem[waddr1] <= data2;
end
```

Initializing RAMs

You can specify startup values for RAMs and pass them on to the place-and-route tools. See the following topics for ways to set the initial values:

- Initializing RAMs in Verilog, on page 407
- Initializing RAMs in VHDL, on page 408
- Initializing RAMs with $readmemb and $readmemh, on page 411

Initializing RAMs in Verilog

In Verilog, you specify startup values using initial statements, which are procedural `assign` statements guaranteed by the language to be executed by the simulator at the start of simulation. This means that any assignment to a variable within the body of the initial statement is treated as if the variable was initialized with the corresponding LHS value. You can initialize memories using the built-in load memory system tasks $readmemb (binary) and $readmemh (hex).

The following procedure is the recommended method for specifying initial values:

1. Create a data file with an initial value for every address in the memory array. This file can be a binary file or a hex file. See Initialization Data File, on page 307in the *Reference Manual* for details of the formats for these files.

2. Do the following in the Verilog file to define the module:
   - Include the appropriate task enable statement, $readmemb or $readmemh, in the initial statement for the module:

     $readmemb ("*fileName***", *memoryName* [, *startAddress* [, *stopAddress*]]);**

     $readmemh ("*fileName***", *memoryName* [, *startAddress* [, *stopAddress*]]);**

     Use $readmemb for a binary file and use $readmemh for a hex file. For descriptions of the syntax, see Initial Values for RAMs, on page 304in the *Reference Manual*.
   - Make sure the array declaration matches the order in the initial value data file you specified. As the file is read, each number encountered is assigned to a successive word element of the memory. The software

© 2016 Synopsys, Inc.

starts with the left-hand address in the memory declaration, and loads consecutive words until the memory is full or the data file has been completely read. The loading order is the order in the declaration. For example, with the following memory definition, the first line in the data file corresponds to address 0:

```
reg [7:0] mem_up [0:63]
```

With this next definition, the first line in the data file applies to address 63:

```
reg [7:0] mem_down [63:0]
```

3. To forward-annotate initial values, use the $readmemb or $readmemh statements, as described in *Initializing RAMs with $readmemb and $readmemh*, on page 411.

   See Example 1: RAM Initialization, on page 305 in the *Reference Manual* for an example of a Verilog single-port RAM.

Initializing RAMs in VHDL

There are two ways to initialize the RAM in the VHDL code: with signal declarations or with variable declarations.

Initializing VHDL Rams with Signal Declarations

The following example shows a single-port RAM that is initialized with signal initialization statements. For alternative methods, see *Initializing VHDL Rams with Variable Declarations*, on page 410.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity w_r2048x28 is
port (
   clk : in  std_logic;
   adr : in  std_logic_vector(10 downto 0);
   di : in  std_logic_vector(26 downto 0);
   we : in  std_logic;
   dout : out std_logic_vector(26 downto 0));
end;
```

```
architecture arch of w_r2048x28 is

-- Signal Declaration --

type MEM is array(0 to 2047) of std_logic_vector (26 downto 0);
signal memory : MEM := (
"1111111111111111000000000000"
,"1111100110111010100111110001"
,"1110011110001111001011001111"
,"1100101100111011110011110001"
,"1010011110001111111100110111"
,"1000000000000011111111111111"
,"0101100001110011111100110111"
,"0011010011000111110011110001"
,"0001100001110011001011001111"
,"0000011001000110100111110001"
,"0000000000000010000000000000"
,"0000011001000101011000011110"
,"0001100001110000110100111000"
,"0011010011000100011000011110"
,"0101100001110000000110011000"
,"0111111111111110000000000000"
,"1010011110001100000110011000"
,"1100101100111000011000011110"
,"1110011110001100110100111000"
,"1111100110111001011000011110"
,"1111111111111110111111111111"
,"1111100110111010100111110001"
,"1110011110001111001011001111"
,"1100101100111011110011110001"
,"1010011110001111111100110111"
,"1000000000000011111111111111"
,others => (others => '0'));

begin
process(clk)

begin
   if rising_edge(clk) then
      if (we = '1') then
         memory(conv_integer(adr)) <= di;
      end if;
         dout <= memory(conv_integer(adr));
   end if;
end process;

end arch;
```


© 2016 Synopsys, Inc.

Initializing VHDL Rams with Variable Declarations

The following example shows a RAM that is initialized with variable declarations. For alternative methods, see *Initializing VHDL Rams with Signal Declarations*, on page 408 and *Initializing RAMs with $readmemb and $readmemh*, on page 411.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity one is
generic (data_width : integer := 6;
   address_width :integer  := 3
);

port (data_a :in std_logic_vector(data_width-1 downto 0);
   raddr1    :in unsigned(address_width-2 downto 0);
      waddr1 :in unsigned(address_width-1 downto 0);
      we1    :in std_logic;
      clk :in std_logic;
      out1 :out std_logic_vector(data_width-1 downto 0));
end;

architecture rtl of one is
   type mem_array is array(0 to 2**(address_width) -1) of
      std_logic_vector(data_width-1 downto 0);
begin

WRITE1_RAM :  process (clk)
   variable mem : mem_array := (1 => "111101", others => (1=>'1',
      others => '0'));
begin
   if rising_edge(clk) then
      out1  <= mem(to_integer(raddr1));
      if (we1 = '1') then
         mem(to_integer(waddr1)) := data_a;
      end if;
   end if;
end process WRITE1_RAM;
end rtl;
```

Initializing RAMs with $readmemb and $readmemh

1. Create a data file with an initial value for every address in the memory array. This file can be a binary file or a hex file. See Initialization Data File, on page 307 in the *Reference Manual* for details.

2. Include one of the task enable statements, $readmemb or $readmemh, in the initial statement for the module:

   $readmemb ("*fileName***", *memoryName* [, *startAddress* [, *stopAddress*]]);**
   $readmemh ("*fileName***", *memoryName* [, *startAddress* [, *stopAddress*]]);**

   Use $readmemb for a binary file and $readmemh for a hex file. For details about the syntax, see Initial Values for RAMs, on page 304 in the *Reference Manual*.

© 2016 Synopsys, Inc.

CHAPTER 10

Specifying Design-Level Optimizations

This chapter covers techniques for optimizing your design using built-in tools or attributes. For vendor-specific optimizations, see Chapter 16, *Optimizing for Lattice Designs*. It describes the following:

- Tips for Optimization, on page 414
- Pipelining, on page 418
- Retiming, on page 422
- Preserving Objects from Being Optimized Away, on page 429
- Optimizing Fanout, on page 434
- Sharing Resources, on page 438
- Inserting I/Os, on page 439
- Optimizing State Machines, on page 441
- Inserting Probes, on page 446

© 2016 Synopsys, Inc.

Tips for Optimization

The software automatically makes efficient trade-offs to achieve the best results. However, you can optimize your results by using the appropriate control parameters. This section describes general design guidelines for optimization. The topics have been categorized as follows:

- General Optimization Tips, on page 414
- Optimizing for Area, on page 415
- Optimizing for Timing, on page 416

General Optimization Tips

This section contains general optimization tips that are not directly area or timing-related. For area optimization tips, see *Optimizing for Area*, on page 415. For timing optimization, see *Optimizing for Timing*, on page 416.

- In your source code, remove any unnecessary priority structures in timing-critical designs. For example, use `CASE` statements instead of nested `IF-THEN-ELSE` statements for priority-independent logic.
- If your design includes safe state machines, use the syn_encoding attribute with a value of safe. This ensures that the synthesized state machines never lock in an illegal state.
- For FSMs coded in VHDL using enumerated types, use the same encoding style (syn_enum_encoding attribute value) on both the state machine enumerated type and the state signal. This ensures that there are no discrepancies in the type of encoding to negatively affect the final circuit.
- Make sure that the source code supports inferencing or instantiation by using architecture-specific resources like memory blocks.
- Some designs benefit from hierarchical optimization techniques. To enable hierarchical optimization on your design, set the syn_hier attribute to firm.
- For accurate results with timing-driven synthesis, explicitly define clock frequencies with a constraint, instead of using a global clock frequency.

Optimizing for Area

This section contains information on optimizing to reduce area. Optimizing for area often means larger delays, and you will have to weigh your performance needs against your area needs to determine what works best for your design. For tips on optimizing for performance, see *Optimizing for Timing*, on page 416. General optimization tips are in *General Optimization Tips*, on page 414.

- Increase the fanout limit when you set the implementation options. A higher limit means less replicated logic and fewer buffers inserted during synthesis, and a consequently smaller area. In addition, as P&R tools typically buffer high fanout nets, there is no need for excessive buffering during synthesis. See *Setting Fanout Limits*, on page 434 for more information.
- Enable the Resource Sharing option when you set implementation options. With this option checked, the software shares hardware resources like adders, multipliers, and counters wherever possible, and minimizes area. This is a global setting, but you can also specify resource sharing on an individual basis for lower-level modules. See *Sharing Resources*, on page 438 for details.
- For designs with large FSMs, use the gray or sequential encoding styles, because they typically use the least area. For details, see *Specifying FSMs with Attributes and Directives*, on page 393.
- If you are mapping into a CPLD and do not meet area requirements, set the default encoding style for FSMs to sequential instead of onehot. For details, see *Specifying FSMs with Attributes and Directives*, on page 393.
- For small CPLD designs (less than 20K gates), you might improve area by using the syn_hier attribute with a value of flatten. When specified, the software optimizes across hierarchical boundaries and creates smaller designs.

© 2016 Synopsys, Inc.

Optimizing for Timing

This section contains information on optimizing to meet timing requirements. Optimizing for timing is often at the expense of area, and you will have to balance the two to determine what works best for your design. For tips on optimizing for area, see *Optimizing for Area*, on page 415. General optimization tips are in *General Optimization Tips*, on page 414.

- Use realistic design constraints, about 10 to 15 percent of the real goal. Over-constraining your design can be counter-productive because you can get poor implementations. Typically, you set timing constraints like clock frequency, clock-to-clock delay paths, I/O delays, register I/O delays and other miscellaneous path delays. Use clock, false path, and multi-cycle path constraints to make the constraints realistic.
- Enable the Retiming option. This optimization moves registers into I/O buffers if this is permitted by the technology and the design. However, it may add extra registers when clouds of logic are balanced across more than one register-to-register timing path. Extra registers are only added in parallel within the timing path and only if no extra latency is added by the additional registers. For example, if registers are moved across a 2x1 multiplexer, the tool adds two new registers to accommodate the select and data paths.

  You can set this option globally or on specific registers. When it is enabled, it automatically enables pipelining as well. See *Retiming*, on page 422 for details.
- Enable the Pipelining option. With this optimization enabled, the tool moves existing registers into a ROM or multiplier. Unlike retiming, it does not add any new logic. Pipelining reduces routing and delay and the extra instance delay of the external register by moving it into the ROM or multiplier and making it a built-in register.
- Select a balanced fanout constraint. A large constraint creates nets with large fanouts, and a low fanout constraint results in replicated logic. See *Setting Fanout Limits*, on page 434 for information about setting limits and using the syn_maxfan attribute. You can use this in conjunction with the syn_replicate attribute that controls register duplication and buffering.
- Control register duplication and buffering criteria with the syn_replicate attribute. The tool automatically replicates registers during optimization, and you can use this attribute globally or locally on a specific register to turn off register duplication. See *Controlling Buffering and Replication,*

on page 436 for a description. Use syn_replicate in conjunction with the syn_maxfan attribute that controls fanout.

- If the critical path goes through arithmetic components, try disabling Resource Sharing. You can get faster times at the expense of increased area, but use this technique carefully. Adding too many resources can cause longer delays and defeat your purpose.
- If the P&R and synthesis tools report different critical paths, use a timing constraint with the -route option. With this option, the software adds route delay to its calculations when trying to meet the clock frequency goal. Use realistic values for the constraints.
- For FSMs, use the onehot encoding style, because it is often the fastest implementation. If a large output decoder follows an FSM, gray or sequential encoding could be faster.
- For designs with black boxes, characterize the timing models accurately, using the syn_tpd, syn_tco, and syn_tso directives.
- If you see warnings about feedback muxes being created for signals when you compile your source code, make sure to assign set/resets for the signals. This improves performance by eliminating the extra mux delay on the input of the register.
- Make sure that you pass your timing constraints to the place-and-route tools, so that they can use the constraints to optimize timing.

© 2016 Synopsys, Inc.

Pipelining

Pipelining is the process of splitting logic into stages so that the first stage can begin processing new inputs while the last stage is finishing the previous inputs. This ensures better throughput and faster circuit performance. If you are using selected technologies which use pipelining, you can also use the related technique of retiming to improve performance. See *Retiming*, on page 422 for details.

For pipelining, the software splits the logic by moving registers into the multiplier or ROM:

This section discusses the following pipelining topics:

- Prerequisites for Pipelining, on page 418
- Pipelining the Design, on page 418

Prerequisites for Pipelining

Pipelining is only supported for certain Lattice technologies. ROMs to be pipelined must be at least 512 words. Anything below this limit is too small.

Pipelining the Design

The following procedure shows you techniques for pipelining.

1. Make sure the design meets the criteria described in *Prerequisites for Pipelining*, on page 418.
2. To enable pipelining for the whole design, check the Pipelining check box. from the button panel in the Project window, or with the Project->Implementation Options command (Options tab). The option is only available in the appropriate technologies.

Use this approach as a first pass to get a feel for which modules you can pipeline. If you know exactly which registers you want to pipeline, add the attribute to the registers in the source code or interactively using the SCOPE interface.

3. To check whether individual registers are suitable for pipelining, do the following:
   - Open the RTL view of the design.
   - Select the register and press F12 to filter the schematic view.

   - In the new schematic view, select the output and type e (or select Expand from the popup menu. Check that the register is suitable for pipelining.

© 2016 Synopsys, Inc.

4. To enable pipelining on selected registers, use either of the following techniques:
   - Check the Pipelining checkbox and attach the syn_pipeline attribute with a value of 0 or false to any registers you do not want the software to move. This attribute specifies that the register cannot be moved for pipelining.
   - Do not check the Pipelining checkbox. Attach the syn_pipeline attribute with a value of 1 or true to any registers you want the software to consider for retiming. This attribute marks the register as one that can be moved during retiming, but does not necessarily force it to be moved during retiming.

   The following are examples of the attribute:

   SCOPE Interface:

	Enabled	Object Type	Object	Attribute	Value	Val Type	Descr
1	✔	register	res[15:0]	syn_pipeline	1	boolean	Controls pipelining c

   Verilog Example:

```
reg [`lefta:0] a_aux;
reg [`leftb:0] b_aux;
reg [`lefta+`leftb+1:0] res /* synthesis syn_pipeline=1 */;
reg [`lefta+`leftb+1:0] res1;
```

   VHDL Example:

```
architecture beh of onereg is
signal temp1, temp2, temp3,
   std_logic_vector(31 downto 0);
attribute syn_pipeline : boolean;
attribute syn_pipeline of temp1 : signal is true;
attribute syn_pipeline of temp2 : signal is true;
attribute syn_pipeline of temp3 : signal is true;
```

5. Click Run.

   The software looks for registers where all the flip-flops of the same row have the same clock, no control signal, or the same unique control signal, and pushes them inside the module. It attaches the syn_pipeline attribute to all these registers. If there already is a syn_pipeline attribute on a register, the software implements it.

6. Check the log file (`*.srr`). You can use the Find command for occurrences of the word pipelining to find out which modules got pipelined.

   The log file entries look like this:

```
@N:|Pipelining module res_out1
@N:|res_i is level 1 of the pipelined module res_out1
@N:|r is level 2 of the pipelined module res_out1
```


© 2016 Synopsys, Inc.

Retiming

Retiming improves the timing performance of sequential circuits without modifying the source code. It automatically moves registers (register balancing) across combinational gates or LUTs to improve timing while maintaining the original behavior as seen from the primary inputs and outputs of the design. Retiming moves registers across gates or LUTs, but does not change the number of registers in a cycle or path from a primary input to a primary output. However, it can change the total number of registers in a design.

The retiming algorithm retimes only edge-triggered registers. It does not retime level-sensitive latches. Note that registers associated with RAMS, DSPs, and the mapping for generated clocks may be moved, regardless of the Retiming option setting. The Retiming option is not available if it does not apply to the family you are using.

These sections contain details about using retiming.

- Controlling Retiming, on page 422
- Retiming Example, on page 424
- Retiming Report, on page 425
- How Retiming Works, on page 426

Controlling Retiming

The following procedure shows you how to use retiming.

1. To enable retiming for the whole design, check the Retiming check box.

   You can set the Retiming option from the button panel in the Project window, or with the Project->Implementation Options command (Options tab). The option is only available in certain technologies.

For Lattice designs, retiming is a superset of pipelining, so when you select Retiming, you automatically select Pipelining. See *Pipelining*, on page 418 for more information.

Retiming works globally on the design, and moves edge-triggered registers as needed to balance timing.

2. To enable retiming on selected registers, use either of the following techniques:
   - Check the Retiming checkbox and attach the syn_allow_retiming attribute with a value of 0 or false to any registers you do not want the software to move. This attribute specifies that the register cannot be moved for retiming. Refer to *How Retiming Works*, on page 426 for a list of the components the retiming algorithm will move.
   - Do not check the Retiming checkbox. Attach the syn_allow_retiming attribute with a value of 1 or true to any registers you want the software to consider for retiming. You can do this in the SCOPE interface or in the source code. This attribute marks the register as one that can be moved during retiming, but does not necessarily force it to be moved during retiming. If you apply the attribute to an FSM, RAM or SRL that is decomposed into flip-flops and logic, the software applies the attribute to all the resulting flip-flops

© 2016 Synopsys, Inc.

Retiming is a superset of pipelining; therefore adding syn_allow_retiming=1 on any registers implies that syn_pipeline =1.

3. You can also fine-tune retiming using attributes:
   - To preserve the power-on state of flip-flops without sets or resets (FD or FDE) during retiming, set syn_preserve=1 or syn_allow_retiming=0 on these flip-flops.
   - To force flip-flops to be packed in I/O pads, set syn_useioff=1 as a global attribute. This will prevent the flip-flops from being moved during retiming.
4. Set other options for the run. Retiming might affect some constraints and attributes. See *How Retiming Works*, on page 426 for details.
5. Click Run to start synthesis.

   After the LUTs are mapped, the software moves registers to optimize timing. See *Retiming Example*, on page 424 for an example. The software honors other attributes you set, like syn_preserve, syn_useioff, and syn_ram-style. See *How Retiming Works*, on page 426 for details.

   Note that the tool might retime registers associated with RAMs, DSPs, and generated clocks, regardless of whether the Retiming option is on or off.

   The log file includes a retiming report that you can analyze to understand the retiming changes. It contains a list of all the registers added or removed because of retiming. Retimed registers have a _ret suffix added to their names. See *Retiming Report*, on page 425 for more information about the report.

Retiming Example

The following example shows a design with retiming disabled and enabled.

The top figure shows two levels of logic between the registers and the output, and no levels of logic between the inputs and the registers.

The bottom figure shows the results of retiming the three registers at the input of the OR gate. The levels of logic from the register to the output are reduced from two to one. The retimed circuit has better performance than the original circuit. Timing is improved by transferring one level of logic from the critical part of the path (register to output) to the non-critical part (input to register).

Retiming Report

The retiming report is part of the log file, and includes the following:

- The number of registers added, removed, or untouched by retiming.
- Names of the original registers that were moved by retiming and which no longer exist in the Technology view.
- Names of the registers created as a result of retiming, and which did not exist in the RTL view. The added registers have a _ret suffix.

© 2016 Synopsys, Inc.

How Retiming Works

This section describes how retiming works when it moves sequential components (flip-flops). Registers associated with RAMs, DSPs, and the mapping for fixing generated clocks might be moved, whether Retiming is enabled or not. Here are some implications and results of retiming:

- Flip-flops with no control signals (resets, presets, and clock enables) are moved. Flip-flops with minimal control logic can also be retimed. Multiple flip-flops with reset, set or enable signals that need to be retimed together are only retimed if they have exactly the same control logic.
- The software does not retime the following combinational sequential elements: flip-flops with both set and reset, flip-flops with attributes like syn_preserve, flip-flops packed in I/O pads, level-sensitive latches, registers that are instantiated in the code, SRLs, and RAMs. If a RAM with combinational logic has syn_ramstyle set to registers, the registers can be retimed into the combinational logic.
- Retimed flip-flops are only moved through combinational logic. The software does not move flip-flops across the following objects: black boxes, sequential components, tristates, I/O pads, instantiated components, carry and cascade chains, and keepbufs.
- You might not be able to crossprobe retimed registers between the RTL and the Technology view, because there may not be a one-to-one correspondence between the registers in these two views after retiming. A single register in the RTL view might now correspond to multiple registers in the Technology view.
- Retiming affects or is affected by, these attributes and constraints:

Attribute/Constraint	Effect
False path constraint	Does not retime flip-flops with different false path constraints. Retimed registers affect timing constraints.
Multicycle constraint	Does not retime flip-flops with different multicycle constraints. Retimed registers affect timing constraints.
Register constraint	Does not maintain set_reg_input_delay and set_reg_output_delay constraints. Retimed registers affect timing constraints.

Attribute/Constraint	Effect
from/to timing exceptions	If you set a timing constraint using a from/to specification on a register, it is not retimed. The exception is when using a max_delay constraint. In this case, retiming is performed but the constraint is not forward annotated. (The max_delay value would no longer be valid.)
syn_hier=macro	Does not retime registers in a macro with this attribute.
syn_keep	Does not retime across keepbufs generated because of this attribute.
syn_hier=macro	Does not retime registers in a macro with this attribute.
syn_pipeline	Automatically enabled if retiming is enabled.
syn_preserve	Does not retime flip-flops with this attribute set.
syn_probe	Does not retime net drivers with this attribute. If the net driver is a LUT or gate, no flip-flops are retimed across it.
syn_reference_clock	On a critical path, does not retime registers with different syn_reference_clock values together, because the path effectively has two different clock domains.
syn_useioff	Does not override attribute-specified packing of registers in I/O pads. If the attribute value is false, the registers can be retimed. If the attribute is not specified, the timing engine determines whether the register is packed into the I/O block.
syn_allow_retiming	Registers are not retimed if the value is 0.

- Retiming does not change the simulation behavior (as observed from primary inputs and outputs) of your design, However if you are monitoring (probing) values on individual registers inside the design, you might need to modify your test bench if the probe registers are retimed.

© 2016 Synopsys, Inc.

- Beginning with the C-2009.09-SP1 release, the behavior for retiming unconstrained I/O pads has changed. If retiming is enabled, registers connected to unconstrained I/O pins are not retimed by default. If you want to revert back to how retiming I/O paths was previously implemented, you can:
  - Globally turn on the Use clock period for unconstrained IO switch from the Constraints tab of the Implementation Options panel.
  - Add constraints to all input/output ports.
  - Separately constrain each I/O pin as required.

Preserving Objects from Being Optimized Away

Synthesis can collapse or remove nets during optimization. If you want to retain a net for simulation, probing, or for a different synthesis implementation, you must specify this with an attribute. Similarly, the software removes duplicate registers or instances with unused output. If you want to preserve this logic for simulation or analysis, you must use an attribute. The following table lists the attributes to use in each situation. For details about the attributes and their syntax, see the *Attributes Reference Manual*.

To Preserve ...	Use ...	Result
Nets	syn_keep on wire or reg (Verilog), or signal (VHDL).	Keeps net for simulation, a different synthesis implementation, or for passing to the place-and-route tool.
Nets for probing	syn_probe on wire or reg (Verilog), or signal (VHDL)	Preserves internal net for probing.
Shared registers	syn_keep on input wire or signal of shared registers	Preserves duplicate driver cells, prevents sharing. See *Using syn_keep for Preservation or Replication*, on page 430 for details on the effects of applying syn_keep to different objects.
Sequential components	syn_preserve on reg or module (Verilog), signal or architecture (VHDL)	Preserves logic of constant-driven registers, keeps registers for simulation, prevents sharing
FSMs	syn_preserve on reg or module (Verilog), signal (VHDL)	Prevents the output port or internal signal that holds the value of the state register from being optimized
Instantiated components	syn_noprune on module or component (Verilog), architecture or instance (VHDL)	Keeps instance for analysis, preserves instances with unused outputs

See the following for more information:

- Using syn_keep for Preservation or Replication, on page 430
- Controlling Hierarchy Flattening, on page 432
- Preserving Hierarchy, on page 433

© 2016 Synopsys, Inc.

Using syn_keep for Preservation or Replication

By default the tool considers replicated logic redundant, and optimizes it away. If you want to maintain the redundant logic, use syn_keep to preserve the logic that would otherwise be optimized away.

The following Verilog code specifies a replicated AND gate:

```
module redundant1(ina,inb,out1);
input  ina,inb;
output out1,out2;
wire out1;
wire out2;

assign out1 = ina & inb;
assign out2 = ina & inb;
endmodule
```

The compiler implements the AND function by replicating the outputs out1 and out2, but optimizes away the second AND gate because it is redundant.

To replicate the AND gate in the previous example, apply syn_keep to the input wires, as shown below:

```
module redundant1d(ina,inb,out1,out2);
input ina,inb;
output out1,out2;
wire out1;
wire out2;

wire in1a /*synthesis syn_keep = 1*/;
wire in1b /*synthesis syn_keep = 1*/;
wire in2a /*synthesis syn_keep = 1*/;
wire in2b /*synthesis syn_keep = 1 */;
```

```
assign in1a = ina;
assign in1b = inb;
assign in2a = ina;
assign in2b = inb;
assign out1 = in1a & in1b;
assign out2 = in2a & in2b;
endmodule
```

Setting syn_keep on the input wires ensures that the second AND gate is preserved:

You must set syn_keep on the input wires of an instance if you want to preserve the logic, as in the replication of this AND gate. If you set it on the outputs, the instance is not replicated, because syn_keep preserves the nets but not the function driving the net. If you set syn_keep on the outputs in the example, you get only one AND gate, as shown in the next figure.

© 2016 Synopsys, Inc.

Controlling Hierarchy Flattening

Optimization flattens hierarchy. To control the flattening, use the syn_hier attribute as described here. You can also use the attribute to prevent flattening, as described in *Preserving Hierarchy*, on page 433.

1. Attach the syn_hier attribute with the value you want to the module or architecture you want to preserve.

To ...	Value ...
Flatten all levels below, but not the current level	flatten
Remove the current level of hierarchy without affecting the lower levels	remove
Remove the current level of hierarchy and the lower levels	flatten, remove
Flatten the current level (if needed for optimization)	soft

   You can also add the attribute in SCOPE instead of the HDL code. If you use SCOPE to enter the attribute, make sure to use the v: syntax. For details, see syn_hier, on page 104 in the *Attribute Reference Manual*.

   The software flattens the design as directed. If there is a lower-level syn_hier attribute, it takes precedence over a higher-level one.

2. If you want to flatten the entire design, use the syn_netlist_hierarchy attribute set to false, instead of the syn_hier attribute.

This flattens the entire netlist and does not preserve any hierarchical boundaries. See syn_netlist_hierarchy, on page 142 in the *Attribute Reference Manual* for the syntax.

Preserving Hierarchy

The synthesis process includes cross-boundary optimizations that can flatten hierarchy. To override these optimizations, use the syn_hier attribute as described here. You can also use this attribute to direct the flattening process as described in *Controlling Hierarchy Flattening*, on page 432.

1. Attach the syn_hier attribute to the module or architecture you want to preserve. You can also add the attribute in SCOPE. If you use SCOPE to enter the attribute, make sure to use the v: syntax.

2. Set the attribute value:

To ...	Value ...
Preserve the interface but allow cell packing across the boundary	firm
Preserve the interface and contents with no exceptions	macro
Flatten lower levels but preserve the interface of the specified design unit	flatten, firm

The software flattens the design as directed. If there is a lower-level syn_hier attribute, it takes precedence over a higher-level one.

© 2016 Synopsys, Inc.

Optimizing Fanout

You can optimize your results with attributes and directives, some of which are specific to the technology you are using. Similarly, you can use specify objects or hierarchy that you want to preserve during synthesis. For a complete list of all the directives and attributes, see the *Attribute Reference Manual*. This section describes the following:

- Setting Fanout Limits, on page 434
- Controlling Buffering and Replication, on page 436

Setting Fanout Limits

Optimization affects net fanout. If your design has critical nets with high fanout, you can set fanout limits. You can only do this in certain technologies. For details specific to individual technologies, see the *Reference Manual.*

1. To set a global fanout limit for the whole design, do either of the following:
   - Select Project-> Implementation Options->Device and type a value for the Fanout Guide option.
   - Apply the syn_maxfan attribute to the top-level view or module.

   The value sets the number of fanouts for a given driver, and affects all the nets in the design. The defaults vary, depending on the technology. Select a balanced fanout value. A large constraint creates nets with large fanouts, and a low fanout constraint results in replicated or buffered logic. Both extremes affect routing and design performance. The right value depends on your design. The same value of 32 might result in fanouts of 11 or 12 and large delays on the critical path in one design or in excessive replication in another design.

   The software uses the value as a soft limit, or a guide. It traverses the inverters and buffers to identify the fanout, and tries to ensure that all fanouts are under the limit by replicating or buffering where needed (see *Controlling Buffering and Replication*, on page 436 for details). However, the synthesis tool does not respect the fanout limit absolutely; it ignores the limit if the limit imposes constraints that interfere with optimization.

2. To override the global fanout guideline and set a soft fanout limit at a lower level, set the syn_maxfan attribute on modules, views, or non-primitive instances.

   These limits override the more global limits for that object. However, these limits still function as soft limits, and are replicated or buffered, as described in *Controlling Buffering and Replication*, on page 436.

Attribute specified on ...	Effect
Module or view	Soft limit for the module; overrides the global setting.
Non-primitive instance	Soft limit; overrides global and module settings
Clock nets or asynchronous control nets	Soft limit.

3. To set a hard or absolute limit, set the syn_maxfan attribute on a port, net, register, or primitive instance.

   Fanouts that exceed the hard limit are buffered or replicated, as described in *Controlling Buffering and Replication*, on page 436.

4. To preserve net drivers from being optimized, attach the syn_keep or syn_preserve attributes.

   For example, the software does not traverse a syn_keep buffer (inserted as a result of the attribute), and does not optimize it. However, the software can optimize implicit buffers created as a result of other operations; for example, it does not respect an implicit buffer created as a result of syn_direct_enable.

5. Check the results of buffering and replication in the following:
   - The log file (click View Log). The log file reports the number of buffered and replicated objects and the number of segments created for the net.
   - The HDL Analyst views. The software might not follow DRC rules when buffering or replicating objects, or when obeying hard fanout limits.

© 2016 Synopsys, Inc.

Controlling Buffering and Replication

To honor fanout limits (see *Setting Fanout Limits*, on page 434) and reduce fanout, the software either replicates components or adds buffers. The tool uses buffering to reduce fanout on input ports, and uses replication to reduce fanout on nets driven by registers or combinational logic. The software first tries replication, replicating the net driver and splitting the net into segments. This increases the number of register bits in the design. When replication is not possible, the software buffers the signals. Buffering is more expensive in terms of intrinsic delay and resource consumption. The following table summarizes the behavior.

Replicates When ...	Creates Buffers When ...
syn_replicate is 1	syn_replicate is 0. The syn_replicate attribute is used only to turn off the replication.
	syn_maxfan is set on a port/net that is driven by a port or I/O pad
	The net driver has a syn_keep or syn_preserve attribute
	The net driver is not a primitive gate or register

You can control whether high fanout nets are buffered or replicated, using the techniques described here:

- To use buffering instead of replication, set syn_replicate with a value of 0 globally, or on modules or registers. The syn_replicate attribute prevents replication, so that the software uses buffering to satisfy the fanout limit. For example, you can prevent replication between clock boundaries for a register that is clocked by clk1 but whose fanin cone is driven by clk2, even though clk2 is an unrelated clock in another clock group.
- To specify that high-fanout clock ports should not be buffered, set syn_noclockbuf globally, or on individual input ports. Use this if you want to save clock buffer resources for nets with lower fanouts but tighter constraints.
- Inverters merged with fanout loads increase fanout on the driver during placement and routing. A distinction is made between a keep buffer created as the result of the syn_keep attribute being applied by the user (explicit keep buffer) and a keep buffer that exists as the result of another attribute (implicit keep buffer). For example, the syn_direct_enable

attribute inserts a keep buffer. When a syn_maxfan attribute is applied to the output of an explicit keep buffer, the signal is buffered (the keep buffer is not traversed so that the driver is not replicated). When the syn_maxfan attribute is applied to the output of an implicit keep buffer, the keep buffer is traversed and the driver is replicated.

- Turn off buffering and replication entirely, by setting syn_maxfan to a very high number, like 1000.

© 2016 Synopsys, Inc.

Sharing Resources

One of the ways to optimize area is to use resource sharing in the compiler. With resource sharing, the software uses the same arithmetic operators for mutually exclusive statements; for example, with the branches of a case statement. Conversely, you can improve timing by disabling resource sharing, but at the expense of increased area.

Compiler resource sharing is on by default. You can set it globally and then override the global setting on individual modules

1. To disable resource sharing globally for the whole design, use one of the methods below.

   Leave the default setting to improve area; disable the option to improve timing.

   – Select Project->Implementation Options->Options, disable Resource Sharing. Alternatively, disable the Resource Sharing button on the left side of the Project view.

   – Apply the syn_sharing directive to the top-level module or architecture in the source code. See syn_sharing, on page 229 of the *Attribute Reference Manual* for syntax and examples.

Verilog
```
module top(out, in, clk_in) /* synthesis syn_sharing = "off" */;
```

VHDL
```
architecture rtl of top is
  attribute syn_sharing : string;
  attribute syn_sharing of rtl : architecture is "false";
```

   – Edit your project file and include the following command: set_option -resource_sharing 0

   When you save the project file, it includes the Tcl `set_option -resource_sharing` command.

   You cannot specify syn_sharing from the SCOPE interface, because it is a compiler directive, and works during the compilation stage of synthesis. The resource sharing setting does not affect the mapper, so even if resource sharing is disabled, the tool can share resources during the mapping phase to optimize the design and improve results.

2. To specify resource sharing on an individual basis or override the global setting, specify the syn_sharing attribute for the lower-level module/architecture.

Inserting I/Os

You can control I/O insertion globally, or on a port-by-port basis.

1. To control the insertion of I/O pads at the top level of the design, use the Disable I/O Insertion option as follows:
   - Select Project->Implementation Options and click the Device panel.
   - Enable the option (checkbox on) if you want to do a preliminary run and check the area taken up by logic blocks, before synthesizing the entire design.

     Do this if you want to check the area your blocks of logic take up, before you synthesize an entire FPGA. If you disable automatic I/O insertion, you do not get *any* I/O pads in your design, unless you manually instantiate them.
   - Leave the Disable I/O Insertion checkbox empty (disabled) if you want to automatically insert I/O pads for all the inputs, outputs and bidirectionals.

     When this option is set, the software inserts I/O pads for inputs, outputs, and bidirectionals in the output netlist. Once inserted, you can override the I/O pad inserted by directly instantiating another I/O pad.
   - For the most control, enable the option and then manually instantiate the I/O pads for specific pins, as needed.

   For more information about using the Disable I/O Insertion option with Lattice iCE devices, see syn_insert_pad, on page 117.

2. For Lattice designs, you can force I/O pads to be inserted for input ports that do not drive logic with the syn_force_pads attribute:
   - To force I/O pad insertion at the module level, set the syn_force_pads attribute on the module. Set the attribute value to 1. To disable I/O pad insertion at the module level, set the syn_force_pads attribute for the module to 0.

© 2016 Synopsys, Inc.

– To force I/O pad insertion on an individual port, set the syn_force_pads attribute on the port with a value to 1. To disable I/O insertion for a port, set the attribute on the port with a value of 0.

Enable this attribute to preserve user-instantiated pads, insert pads on unconnected ports, insert bi-directional pads on bi-directional ports instead of converting them to input ports, or insert output pads on unconnected outputs.

If you do not set the syn_force_pads attribute, the synthesis design optimizes any unconnected I/O buffers away.

Optimizing State Machines

You can optimize state machines with the symbolic FSM Compiler tool. The Symbolic FSM Compiler is an advanced state machine optimizer, it automatically recognizes state machines in your design and optimizes them. Unlike other synthesis tools that treat state machines as regular logic, the FSM Compiler extracts the state machines as symbolic graphs, and then optimizes them by re-encoding the state representations and generating a better logic optimization starting point for the state machines.

For more information, see the following:

- Deciding when to Optimize State Machines, on page 441
- Running the FSM Compiler, on page 442

Deciding when to Optimize State Machines

The FSM Compiler is an automatic tool for encoding state machines, but you can also specify FSMs manually with attributes. For more information about using attributes, see *Specifying FSMs with Attributes and Directives*, on page 393.

Here are the main reasons to use the FSM Compiler:

- To generate better results for your state machines

  The software uses optimization techniques that are specifically tuned for FSMs, like reachability analysis for example. The FSM Compiler also lets you convert an encoded state machine to another encoding style (to improve speed and area utilization) without changing the source. For example, you can use a onehot style to improve results.

- To debug the state machines

  State machine description errors result in unreachable states, so if you have errors, you will have fewer states. You can check whether your source code describes your state machines correctly. You can also use the FSM Viewer to see a high-level bubble diagram and crossprobe from there. For information about the FSM Viewer, see *Using the FSM Viewer*, on page 352.

© 2016 Synopsys, Inc.

Running the FSM Compiler

You can run the FSM Compiler tool on the whole design or on individual FSMs. See the following:

- Running the FSM Compiler on the Whole Design, on page 442
- Running the FSM Compiler on Individual FSMs, on page 443

Running the FSM Compiler on the Whole Design

1. Enable the compiler by checking the Symbolic FSM Compiler box in one of these places:
   - The main panel on the left side of the project window
   - The Options tab of the dialog box that comes up when you click the Add Implementation/New Impl or Implementation Options buttons
2. To set a specific encoding style for a state machine, define the style with the syn_encoding attribute, as described in *Specifying FSMs with Attributes and Directives*, on page 393.

   If you do not specify a style, the FSM Compiler picks an encoding style based on the number of states.
3. Click Run to run synthesis.

   The software automatically recognizes and extracts the state machines in your design, and instantiates a state machine primitive in the netlist for each FSM it extracts. It then optimizes all the state machines in the design, using techniques like reachability analysis, next state logic optimization, state machine re-encoding and proprietary optimization algorithms. Unless you specified an encoding style, the tool automatically selects the encoding style. If you did specify a style, the tool uses that style.

   In the log file, the FSM Compiler writes a report that includes a description of each state machine extracted and the set of reachable states for each state machine.
4. Select View->View Log File and check the log file for descriptions of the state machines and the set of reachable states for each one. You see text like the following:

```
Extracted state machine for register cur_state
State machine has 7 reachable states with original encodings of:
   0000001
   0000010
   0000100
   0001000
   0010000
   0100000
   1000000
....
original code -> new code
   0000001 -> 0000001
   0000010 -> 0000010
   0000100 -> 0000100
   0001000 -> 0001000
   0010000 -> 0010000
   0100000 -> 0100000
   1000000 -> 1000000
```

5. Check the state machine implementation in the RTL and Technology views and in the FSM viewer.
   - In the RTL view you see the FSM primitive with one output for each state.
   - In the Technology view, you see a level of hierarchy that contains the FSM, with the registers and logic that implement the final encoding.
   - In the FSM viewer you see a bubble diagram and mapping information. For information about the FSM viewer, see *Using the FSM Viewer*, on page 352.
   - In the `statemachine.info` text file, you see the state transition information.

Running the FSM Compiler on Individual FSMs

If you have state machines that you do not want automatically optimized by the FSM Compiler, you can use one of these techniques, depending on the number of FSMs to be optimized. You might want to exclude state machines from automatic optimization because you want them implemented with a specific encoding or because you do not want them extracted as state machines. The following procedure shows you how to work with both cases.

© 2016 Synopsys, Inc.

1. If you have just a few state machines you do not want to optimize, do the following:
   - Enable the FSM Compiler by checking the box in the button panel of the Project window.
   - If you do not want to optimize the state machine, add the syn_state_machine directive to the registers in the Verilog or VHDL code. Set the value to 0. When synthesized, these registers are not extracted as state machines.

Verilog	`reg [3:0] curstate /* synthesis syn_state_machine=0 */;`

VHDL	`signal curstate : state_type;`
	`attribute syn_state_machine : boolean;`
	`attribute syn_state_machine of curstate : signal is false;v`

   - If you want to specify a particular encoding style for a state machine, use the syn_encoding attribute, as described in *Specifying FSMs with Attributes and Directives*, on page 393. When synthesized, these registers have the specified encoding style.
   - Run synthesis.

   The software automatically recognizes and extracts all the state machines, except the ones you marked. It optimizes the FSMs it extracted from the design, honoring the syn_encoding attribute. It writes out a log file that contains a description of each state machine extracted, and the set of reachable states for each FSM.

2. If you have many state machines you do not want optimized, do this:
   - Disable the compiler by disabling the Symbolic FSM Compiler box in one of these places: the main panel on the left side of the project window or the Options tab of the dialog box that comes up when you click the Add Implementation or Implementation Options buttons. This disables the compiler from optimizing any state machine in the design. You can now selectively turn on the FSM compiler for individual FSMs.
   - For state machines you want the FSM Compiler to optimize automatically, add the syn_state_machine directive to the individual state registers in the VHDL or Verilog code. Set the value to 1. When synthesized, the FSM Compiler extracts these registers with the default encoding styles according to the number of states.

Verilog	`reg [3:0] curstate /* synthesis syn_state_machine=1 */;`

VHDL	`signal curstate : state_type;`
	`attribute syn_state_machine : boolean;`
	`attribute syn_state_machine of curstate : signal is true;`

– For state machines with specific encoding styles, set the encoding style with the syn_encoding attribute, as described in *Specifying FSMs with Attributes and Directives*, on page 393. When synthesized, these registers have the specified encoding style.

– Run synthesis.

The software automatically recognizes and extracts only the state machines you marked. It automatically assigns encoding styles to the state machines with the syn_state_machine attribute, and honors the encoding styles set with the syn_encoding attribute. It writes out a log file that contains a description of each state machine extracted, and the set of reachable states for each state machine.

3. Check the state machine in the log file, the RTL and technology views, and the FSM viewer. For information about the FSM viewer, see *Using the FSM Viewer*, on page 352.

© 2016 Synopsys, Inc.

Inserting Probes

Probes are extra wires that you insert into the design for debugging. When you insert a probe, the signal is represented as an output port at the top level. You can specify probes in the source code or by interactively attaching an attribute.

Specifying Probes in the Source Code

To specify probes in the source code, you must add the syn_probe attribute to the net. You can also add probes interactively, using the procedure described in *Adding Probe Attributes Interactively*, on page 447.

1. Open the source code file.
2. For Verilog source code, attach the syn_probe attribute as a comment on any internal signal declaration:

   ```
   module alu(out, opcode, a, b, sel);
      output [7:0] out;
      input [2:0] opcode;
      input [7:0 a, b;
      input sel;
      reg [7:0] alu_tmp /* synthesis syn_probe=1 */;
      reg [7:0] out;
   //Other code
   ```

   The value 1 indicates that probe insertion is turned on. For detailed information about Verilog attributes and examples of the files, see the *Attribute Reference Manual*.

   To define probes for part of a bus, specify where you want to attach the probes; for example, if you specify reg [1:0] in the previous code, the software only inserts two probes.

3. For VHDL source code, add the syn_probe attribute as follows:

   ```
   architecture rtl of alu is
      signal alu_tmp : std_logic_vector(7 downto 0);
      attribute syn_probe : boolean;
      attribute syn_probe of alu_tmp : signal is true;
      --other code;
   ```

For detailed information about VHDL attributes and sample files, see the *Attribute Reference Manual*.

4. Run synthesis.

   The software looks for nets with the syn_probe attribute and creates probes and I/O pads for them.

5. Check the probes in the log file (`*.srr`) and the Technology view.

   This figure shows some probes and probe entries in the log file.

```
Adding property syn_probe, value 1, to net pc[0]
Adding property syn_probe, value 1, to net pc[1]
Adding property syn_probe, value 1, to net pc[2]
Adding property syn_probe, value 1, to net pc[3]
....
@N|Added probe pc_keep_probe_1[0] on pc_keep[0] in
eight_bit_uc
@N|Also padding probe pc_keep_probe_1[0]
@N|Added probe pc_keep_probe_2[1] on pc_keep[1] in
eight_bit_uc
@N|Also padding probe pc_keep_probe_2[1]
@N|Added probe pc_keep_probe_3[2] on pc_keep[2] in
eight_bit_uc
```

Adding Probe Attributes Interactively

The following procedure shows you how to insert probes by adding the syn_probe attribute through the SCOPE interface. Alternatively, you can add the attribute in the source code, as described in *Specifying Probes in the Source Code*, on page 446.

1. Open the SCOPE window and click Attributes.

2. Push down as necessary in an RTL view, and select the net for which you want to insert a probe point.

   Do not insert probes for output or bidirectional signals. If you do, you see warning messages in the log file.

3. Do the following to add the attribute:

   – Drag the net into a SCOPE cell.

© 2016 Synopsys, Inc.

- Add the prefix n: to the net name in the SCOPE window. If you are adding a probe to a lower-level module, the name is created by concatenating the names of the hierarchical instances.
- If you want to attach probes to part but not all of a bus, make the change in the Object column. For example, if you enter n:UC_ALU.longq[4:0] instead of n:UC_ALU.longq[8:0], the software only inserts probes where specified.
- Select syn_probe in the Attribute column, and type 1 in the Value column.
- Add the constraint file to the project list.

4. Rerun synthesis.
5. Open a Technology view and check the probe wires that have been inserted. You can use the Ports tab of the Find form to locate the probes.

   The software adds I/O pads for the probes. The following figure shows some of the pads in the Technology view and the log file entries.

```
Adding property syn_probe, value 1, to net pc[0]
Adding property syn_probe, value 1, to net pc[1]
Adding property syn_probe, value 1, to net pc[2]
Adding property syn_probe, value 1, to net pc[3]
....
@N|Added probe pc_keep_probe_1[0] on pc_keep[0] in
eight_bit_uc
@N|Also padding probe pc_keep_probe_1[0]
@N|Added probe pc_keep_probe_2[1] on pc_keep[1] in
eight_bit_uc
@N|Also padding probe pc_keep_probe_2[1]
@N|Added probe pc_keep_probe_3[2] on pc_keep[2] in
eight_bit_uc
```

CHAPTER 11

Working with Compile Points

The following sections describe compile points and how to use them in logic synthesis iterative flows:

- Compile Point Basics, on page 450
- Compile Point Synthesis Basics, on page 458
- Synthesizing Compile Points, on page 468
- Resynthesizing Incrementally, on page 479

© 2016 Synopsys, Inc.

Compile Point Basics

Compile points are RTL partitions of the design that you define before synthesizing the design. Compile points can be defined manually, or the tool can generate them automatically. The software treats each compile point as a block, and can synthesize, optimize, place, and route the compile points independently. Compile points can be nested.

See the following topics for some details about compile points:

- Advantages of Compile Point Design, on page 450
- Manual Compile Points, on page 452
- Nested Compile Points, on page 453
- Compile Point Types, on page 453

Advantages of Compile Point Design

Designing with compile points makes it more efficient to work with the increasingly larger designs of today and the corresponding team approach to design. They offer several advantages, which are described here:

- Compile Points and Design Flows, next
- Runtime Savings, on page 451
- Design Preservation, on page 451

Compile Points and Design Flows

Compile points improve the efficacy of both top-down and bottom-up design flows:

- In a traditional bottom-up design flow, compile points make it possible to easily divide up the design effort between designers or design teams. The compile points can be worked on separately and individually. The compile point synthesis flow eliminates the need to maintain the complex error-prone scripts for stitching, modeling, and ordering required by the traditional bottom-up design flow.
- From a top-down design flow perspective, compile points make it easier to work on the top-level design. You can mark compile points that are still being developed as black boxes, and synthesize the top level with

what you have. You can also customize the compile point type settings for individual compile points to take advantage of cross-boundary optimizations.

You can also synthesize incrementally, because the tool does not resynthesize compile points that are unchanged when you resynthesize the design. This saves runtime and also preserves parts of the design that are done while the rest of the design is completed.

See *Compile Point Synthesis*, on page 464 for a description of the synthesis process with compile points.

Runtime Savings

Compile points are the required foundation for multiprocessing and incremental synthesis, both of which translate directly to runtime savings:

- Multiprocessing runs synthesis as multiple parallel processes, using the compile points as the partitions that are synthesized in parallel on different processors.
- Incremental synthesis uses compile points to determine which portions of the design to resynthesize, only resynthesizing the compile points that have been modified. See *Resynthesizing Compile Points Incrementally*, on page 479.

Design Preservation

Using compile points addresses the need to maintain the overall stability of a design while portions of the design evolve. When you use compile points to partition the design, you can isolate one part from another. This lets you preserve some compile points, and only resynthesize those that need to be rerun. These scenarios describe some design situations where compile points can be used to isolate parts of the design and run incremental synthesis:

- During the initial design phase, design modules are still being designed. Use compile points to preserve unchanged design modules and evaluate the effects of modifications to parts of the design that are still changing.
- During design integration, use compile points to preserve the main design modules and only allow the glue logic to be remapped.
- If your design contains IP, synthesize the IP, and use compile points to preserve them while you run incremental synthesis on the rest of the design.

© 2016 Synopsys, Inc.

- In the final stages of the design, use compile points to preserve design modules that do not need to be updated while you work through minor RTL changes in some other part of the design.

Manual Compile Points

You can create compile points manually. Manual compile points require setup, but provide more control because they let you define the partition boundaries and constraints instead of the tool.

- Manual compile points (MCP)

  Manual compile points provide more control. You can specify boundary constraints for each compile point individually. You can separate completed parts of the design from parts that are still being designed, or fine-tune the compile points to take advantage of as many cross-boundary optimizations as possible. For example, you can ensure that a critical path does not cross a compile point boundary, thus ensuring synthesis results with optimal performance.

Guidelines for Using Manual Compile Points

The table lists some guidelines:

Use Manual Compile Points...
When you know the design in detail. Create manual compile points to get better QoR. Good candidates for manual compile points include the following: • Completed modules with registered interfaces, where you want to preserve the design • Modules created to include an entire critical path, so as to get the best performance. • Modules that are less likely to be affected by cross boundary optimizations like constant propagation and register absorption.
When you do not want further optimizations to a completed compile point.
When you want more control to determine cross-boundary optimizations on an individual basis.

Nested Compile Points

A design can have any number of compile points, and compile points can be nested inside other compile points. In the following figure, compile point CP6 is nested inside compile point CP5, which is nested inside compile point CP4.

To simplify things, the term *child* is used to refer to a compile point that is contained inside another compile point; the term *parent* is used to refer to a container compile point that contains a child. These terms are not used in their strict sense of direct, immediate containment: If a compile point A is nested in B, which is nested in C, then A and B are both considered children of C, and C is a parent of both A and B. The top level is considered the parent of all compile points. In the figure above, both CP5 and CP6 are children of CP4; both CP4 and CP5 are parents of CP6; CP5 is an immediate child of CP4 and an immediate parent of CP6.

Compile Point Types

Compile point designs do not have as good QoR as designs without them because the boundaries limit optimizations. Cross-boundary optimizations typically improve area and timing, at the expense of runtime. The compile

© 2016 Synopsys, Inc.

point type determines whether boundary optimizations are allowed. For manual compile points, you define the type. See *Defining the Compile Point Type*, on page 474 for details.

These are descriptions of the soft, hard, and locked compile types:

- Soft

  Compile point boundaries can be reoptimized during top-level mapping. Timing optimizations like sizing, buffering, and DRC logic optimizations can modify boundary instances of the compile point and combine them with functions from the next higher level of the design. The compile point interface can also be modified. Multiple instances are uniquified. Any optimization changes can propagate both ways: into the compile point and from the compile point to its parent.

  Using soft mode usually yields the best quality of results, because the software can utilize boundary optimizations. On the other hand, soft compile points can take a longer time to run than the same design with hard or locked compile points. Unless they are at the leaf level, soft compile points are not processed in parallel. Upper levels that contain soft compile points cannot be processed until the lower level has been mapped, with the top level processed last.

  The following figure shows the soft compile point with a dotted boundary to show that logic can be moved in or out of the compile point.

- Hard

  For hard compile points, the compile point boundary can be reoptimized during top-level mapping and instances on both sides of the boundary can be modified by timing and DRC optimizations using top-level constraints. However, the boundary is not modified. Any changes can propagate in either direction while the compile point boundary (port/interface) remains unchanged. Multiple instances are uniquified. For performance improvements, constant propagation and removal of unused logic optimizations are performed across hard compile points.

  In the following figure, the solid boundary on the hard compile point indicates that no logic can be moved in or out of the compile point.

Optimization on both sides

  The hard compile point type allows for optimizations on both sides of the boundary without changing the boundary. There is a trade-off in quality of results to keep the boundaries. Using hard also allows for hierarchical equivalence checking for the compile point module.

- Locked

  This is the default compile point type for manual compile points. With a locked compile point, the tool does not make any interface changes or reoptimize the compile point during top-level mapping. An interface logic model (ILM) of the compile point is created (see *Interface Logic Models, on page 460*) and included for the top-level mapping. The ILM remains unchanged during top-level mapping.

  The locked value indicates that all instances of the same compile point are identical and unaffected by top-level constraints or critical paths. As a result, multiple instances of the compile point module remain identical even though the compile point is uniquified. The Technology view (srm

© 2016 Synopsys, Inc.

file) shows unique names for the multiple instances, but in the final Verilog netlist (`vma` file) the original module names for the multiple instances are restored.

Timing optimization can only modify instances outside the compile point. Although the compile point is used to time the top-level netlist, changes do not propagate into or out of a locked compile point. The following figure shows a solid boundary for the locked compile point to indicate that no logic is moved in or out of the compile point during top-level mapping.

This mode has the largest trade-off in terms of QoR, because there are no boundary optimizations. So, it is very important to provide accurate constraints for locked compile points. The following table lists some advantages and limitations with the locked compile point:

Advantages	Limitations
Consumes smallest amount of memory. Used for large designs because of this memory advantage.	Interface timing
Provides most runtime advantage compared to other compile point types.	Constant propagation

Advantages	Limitations
Allows for obtaining stable results for a completed part of the design.	BUFG insertion
Allows for hierarchical place and route with multiple output netlists for each compile point and the top-level output netlist.	GSR hookup
Allows for hierarchical simulation.	IO pads, like IBUFs and OBUFs, should not be instantiated within compile points

Compile Point Type Summary

The following table summarizes how the tool handles different compile points during synthesis:

Features	**Compile Point Type**		
	Soft	**Hard**	**Locked**
Boundary optimizations	Yes	Limited	No
Uniquification of multiple instance modules	Yes	Yes	Limited
Compile point interface (port definitions)	Modified	Not modified	Not modified
Hierarchical simulation	No	no	Yes
Hierarchical equivalence checking	No	Yes	Yes
Interface Logic Model (created/used)	No	No	Yes

© 2016 Synopsys, Inc.

Compile Point Synthesis Basics

This section describes the compile point constraint files and timing models, and describes the steps the tool goes through to synthesize compile points. See the following for details:

- Compile Point Constraint Files, on page 458
- Interface Logic Models, on page 460
- Interface Timing for Compile Points, on page 461
- Compile Point Synthesis, on page 464
- Incremental Compile Point Synthesis, on page 465
- Forward-annotation of Compile Point Timing Constraints, on page 466

For step-by-step information about how to use compile points, see *Synthesizing Compile Points*, on page 468.

Compile Point Constraint Files

A compile point design can contain two levels of constraint files, as described below:

- The constraint file at the top level

  This is a required file, and contains constraints that apply to the entire design. This file also contains the definitions of the compile points in the design. The define_compile_point command is automatically written to the top-level constraint file for each compile point you define.

  The following figure shows that this design has one locked compile point, pgrm_cntr. It uses the following syntax to define the compile point:

  ```
  define_compile_point {v:work.prgm_cntr} -type {locked}
  ```

```
13
14 ###==== BEGIN Collections - (Populated from tab in SCOPE, do not edit)
15 ###==== END Collections
16
17 ###==== BEGIN Clocks - (Populated from tab in SCOPE, do not edit)
18 create_clock  {p:clock} -period {10}
19
20 ###==== END Clocks
21
22 ###==== BEGIN "Generated Clocks" - (Populated from tab in SCOPE, do not edit)
23 ###==== END "Generated Clocks"
24
25 ###==== BEGIN Inputs/Outputs - (Populated from tab in SCOPE, do not edit)
26 ###==== END Inputs/Outputs
27
28 ###==== BEGIN "Delay Paths" - (Populated from tab in SCOPE, do not edit)
29 ###==== END "Delay Paths"
30
31 ###==== BEGIN Attributes - (Populated from tab in SCOPE, do not edit)
32 ###==== END Attributes
33
34 ###==== BEGIN "I/O Standards" - (Populated from tab in SCOPE, do not edit)
35 ###==== END "I/O Standards"
36
37 ###==== BEGIN "Compile Points" - (Populated from tab in SCOPE, do not edit)
38 define_compile_point  {v:work.prgm_cntr} -type {locked}
39 ###==== END "Compile Points"
40
```

- Constraints files at the compile point level

  These constraint files are optional, and are used for better control over manual compile points.

  The compile point constraints are specific to the compile point and only apply within it. If your design has manual compile points, you can define corresponding compile point constraint files for them. See *Setting Constraints at the Compile Point Level*, on page 475 for a step-by-step procedure.

  When compile point constraints are defined, the tool uses them to synthesize the compile point, not automatic interface timing. Note that depending on the compile point type, the tool might further optimize the compile points during top-down synthesis of the top level to improve timing performance and overall design results, but the compile point itself is synthesized with the defined compile point constraints.

  The first command in a compile point constraint file is define_current_design, and it specifies the compile point module for the contained constraints. This command sets the context for the constraint file. The

© 2016 Synopsys, Inc.

remainder of the file is similar to the top-level constraint file. For example:

```
define_current_design {work.pgrm_cntr}
```

```
12 define_current_design {work.prgm_cntr}
13 ###==== END Header
14
15 ###==== BEGIN Collections - (Populated from tab in SCOPE, do not edit)
16 ###==== END Collections
17
18 ###==== BEGIN Clocks - (Populated from tab in SCOPE, do not edit)
19 create_clock  {p:clock} -period {11.1111}
20
21 ###==== END Clocks
22
23 ###==== BEGIN "Generated Clocks" - (Populated from tab in SCOPE, do not edit)
24 ###==== END "Generated Clocks"
25
26 ###==== BEGIN Inputs/Outputs - (Populated from tab in SCOPE, do not edit)
27 ###==== END Inputs/Outputs
28
29 ###==== BEGIN "Delay Paths" - (Populated from tab in SCOPE, do not edit)
30 ###==== END "Delay Paths"
31
32 ###==== BEGIN Attributes - (Populated from tab in SCOPE, do not edit)
33 ###==== END Attributes
34
35 ###==== BEGIN "I/O Standards" - (Populated from tab in SCOPE, do not edit)
36 ###==== END "I/O Standards"
37
38 ###==== BEGIN "Compile Points" - (Populated from tab in SCOPE, do not edit)
39 ###==== END "Compile Points"
```

If your design has some compile points with their own constraint files and others without them, the tool uses the defined compile point constraints when it synthesizes those compile points. For the other compile points without defined constraints, it uses automatic interface timing, as described in *Interface Timing for Compile Points*, on page 461.

Interface Logic Models

The interface logic model (ILM) of a locked or hard compile point is a timing model that contains only the interface logic necessary for accurate timing. An ILM is a partial gate-level netlist that represents the original design accurately while requiring less memory during mapping. Using ILMs improves the runtime for static timing analysis without compromising timing accuracy.

The tool does not do any timing optimizations on an ILM. The interface logic is preserved with no modifications. All logic required to recreate timing at the top level is included in the ILM. ILM logic includes any paths from an input/inout port to an internal register, an internal register to an output/inout port, and an input/inout port to an output/inout port.

The tool removes internal register-to-register paths, as shown in this example. In this design, and_a is not included in the ILM because the timing path that goes through and_a is an internal register-to-register path.

Interface Timing for Compile Points

By default, the synthesis tool automatically infers timing constraints for all compile points from the top-level constraints. However, if a compile point has its own constraint file, the tool applies those compile point-specific constraints to synthesize the compile point.

- For automatic interface timing, the tool derives constraints from the top level and uses them to synthesize the compile point. The top level is synthesized at the same time as the other compile points.
- When there are compile point constraint files, the tool first synthesizes the compile point using the constraints in the compile point constraints file and then synthesizes the top level using the top-level constraints.

When it synthesizes a compile point, the tool considers all other compile points as black boxes and only uses their interface timing information. In the following figure, when the tool is synthesizing compile point A, it applies relevant timing information to the boundary registers of B and C, because it treats them as black boxes.

© 2016 Synopsys, Inc.

Interface Timing Example

The design below shows how the interface timing works on compile points.

Contents of level1 Module

Interface Timing Off

Interface timing is off for a compile point when you define constraints for it in a compile point constraints file. In this example, the following frequencies are defined for the level1 compile point shown above:

Clock	Period	Constraints File
Top-level clock	10 ns	Top-level constraint file
Compile point-level clock	20 ns	Compile point constraint file

When interface timing is off, the compile point log file (`srr`) reports the clock period for the compile point as 20 ns, which is the compile point period.

Interface Timing On

For automatic interface timing to run on a compile point (interface timing on), there must not be a compile-point level constraints file. When interface timing is on, the compile point log file (`srr`) reports the clock period for the top-level design, which is 10 ns:

© 2016 Synopsys, Inc.

Compile Point Synthesis

During synthesis, the tool first synthesizes the compile points and then maps the top level. The rest of this section describes the process that the tool goes through to synthesize compile points; for step-by-step information about what you need to do to use compile points, see *Synthesizing Compile Points, on page 468*.

Manual Compile Point Synthesis

The following headings describe the process of manual compile point synthesis:

Stage 1: Bottom-up Compile Point Synthesis

The tool synthesizes compile points individually from the bottom up. If you have enabled multiprocessing, it synthesizes the compile points in parallel using multiple processing jobs. For nested compile points, it starts with the compile point at the lowest level of hierarchy and works up the hierarchy.

A compile point stands on its own, and is optimized separately from its parent environment (the compile point container or the top level). This means that critical paths from a higher level do not propagate downwards, and they are unaffected by them.

If you have specified compile point-level constraints, the tool uses them to synthesize the compile point; if not, it uses automatic interface timing propagated from the top level. For compile point synthesis, the tool assumes that all other compile points are black boxes, and only uses the interface information.

When defined, compile point constraints apply within the compile point. For manual compile points, it is recommended that you set constraints on locked compile points, but setting constraints is optional for soft and hard compile points.

By default, synthesis stops if the tool encounters an error while synthesizing a compile point.

Stage 2: Top-Level Synthesis

Once all the compile points have been synthesized, the tool synthesizes the entire design from the top down, using the model information generated for each compile point and constraints defined in the top-level constraints file. You do not need to duplicate compile point constraints at a higher level, because the tool takes the compile point timing models into account when it synthesizes a higher level. Note that if you run standalone timing analysis on a compile point, the timing report reflects the top-level constraints and not the compile point constraints, although the tool used compile point level constraints to synthesize the compile point.

The software writes out a single output netlist and one constraint file for the entire design. See *Forward-annotation of Compile Point Timing Constraints*, on page 466 for a description of the constraints that are forward-annotated.

Incremental Compile Point Synthesis

The tool treats compile points as blocks for incremental synthesis. On subsequent synthesis runs, the tool runs incrementally and only resynthesizes those compile points that have changed, and the top level. The synthesis tool automatically detects design changes and resynthesizes compile points only if necessary. For example, it does not resynthesize a compile point if you only add or change a source code comment, because this change does not really affect the design functionality.

The tool resynthesizes a compile point that has already been synthesized, in any of these cases:

© 2016 Synopsys, Inc.

- The HDL source code defining the compile point is changed in such a way that the design logic is changed.
- The constraints applied to the compile point are changed.
- Any of the options on the Device panel of the Implementation Options dialog box, except Update Compile Point Timing Data, are changed. In this case the entire design is resynthesized, including all compile points.
- You intentionally force the resynthesis of your entire design, including all compile points, with the Run -> Resynthesize All command.
- The Update Compile Point Timing Data device mapping option is enabled and at least one child of the compile point (at any level) has been remapped. The option requires that the parent compile point be resynthesized using the updated timing model of the child. This includes the possibility that the child was remapped earlier, while the option was disabled. The newly enabled option requires that the updated timing model of the child be taken into account, by resynthesizing the parent.

For each compile point, the software creates a subdirectory named for the compile point, in which it stores intermediate files that contain hierarchical interface timing and resource information that is used to synthesize the next level. Once generated, the model file is not updated unless there is an interface design change or you explicitly specify it. If you happen to delete these files, the associated compile point will be resynthesized and the files regenerated.

Forward-annotation of Compile Point Timing Constraints

In addition to a top-level constraint file, each compile point can have its own constraint file. Constraints are forward-annotated to placement and routing from the top-level as well as the compile point-level files. However, not all compile point constraints are forward-annotated, as explained below. For example, constraints on top-level ports are always forward annotated, but compile point port constraints are not forward annotated.

- Top-level constraints are forward-annotated.
- Constraints applied to the interface (ports and bit ports) of the compile point are not forward-annotated.
  These include input_delays, output_delays, and clock definitions on the ports. Such constraints are only used to map the compile point itself,

not its parents. They are not used in the final timing report, and they are not forward-annotated.

- Constraints applied to instances inside the compile point are forward-annotated
  Constraints like timing exceptions and internal clocks are used to map the compile point and its parents. They are used in the final timing report, and they are forward-annotated.

© 2016 Synopsys, Inc.

Synthesizing Compile Points

This section describes the synthesis process with manual compile points in your design:

- The Manual Compile Point Flow, on page 468
- Creating a Top-Level Constraints File for Compile Points, on page 471
- Defining Manual Compile Points, on page 472
- Setting Constraints at the Compile Point Level, on page 475
- Analyzing Compile Point Results, on page 477

The Manual Compile Point Flow

Using manual compile points is most advantageous in the following situations, where you

- Have to work with a large design
- Experience long runtimes, or need to reduce synthesis runtime
- Require the maximum QoR from logic synthesis
- Can adjust design methodology to get the best results from the tools

The following figure summarizes the process for using manual compile points in your design.

This procedure describes the steps in more detail:

1. Set up the project.
   - Create the project and add RTL and IP files to the project, as usual.
   - Target a device and technology for which compile points are supported.
   - Set other options as usual.
2. Compile the design (F7) to initialize the constraints file.
3. Do the following in the top-level constraint file:
   - Define compile points in the top-level constraint file. See *Creating a Top-Level Constraints File for Compile Points*, on page 471. Note that by default, the tool automatically calculates the interface timing for all compile points.

© 2016 Synopsys, Inc.

– Set timing constraints and attributes in the top-level constraint file:

Constraint	Apply to ...	Example
Clock	All clocks in the design.	create_clock {p:clk} -name clk -period 100 -clockgroup cg1
I/O constraints	All top-level port constraints. Register the compile point I/O boundaries to improve timing.	set_input_delay {p:a} {1} -clock {clk:r}
Timing exceptions	All timing exceptions that are outside the compile point module, or that might be partially in the compile point modules.	set_false_path -from {i:reg1} -to {i:reg2}
Attributes	All attributes that are applicable to the rest of the design, not within the compile points.	define_attribute {i:statemachine_1} syn_encoding {sequential}

4. Set compile point-specific constraints as needed in a separate, compile point-level constraint file.

   See *Setting Constraints at the Compile Point Level*, on page 475 for a step-by-step procedure. After setting the compile point constraints, add the compile point constraint file to the project.

5. Synthesize the design.

   The tool synthesizes the compile points separately and then synthesizes the top level. See *Compile Point Synthesis*, on page 464 for details about the process.

   – The first time it runs synthesis, the tool maps the entire design.

   – For subsequent synthesis runs, the tool only maps compile points that were modified since the last run. It preserves unchanged compile points.

   You can also run synthesis on individual compile points, without synthesizing the whole design.

6. Analyze the synthesis results using the top-level srr log file.

   See *Analyzing Compile Point Results*, on page 477 for details.

7. If you do not meet your design goals, make necessary changes to the RTL, constraints, or synthesis controls, and re-synthesize the design.

   The tool runs incremental synthesis on the modified parts of the design, as described in *Incremental Compile Point Synthesis*, on page 465. See *Resynthesizing Compile Points Incrementally*, on page 479 for a detailed procedure.

Creating a Top-Level Constraints File for Compile Points

All compile points require a top-level constraints file. If you have manual compile points, define them in this file. The top-level file also contains design-level constraints. The following procedure describes how to create a top-level constraints file for a compile point design.

1. Create the top-level constraints file.
   - To define compile points in an existing top-level constraint file, open a SCOPE window by double-clicking the file in the Project view.
   - To define compile points in a new top-level constraint file, click the SCOPE icon. Click on the FPGA Constraints (SCOPE) button.

   The SCOPE window opens. It includes a Current Design field, where you can specify constraints for the top-level design from the drop-down menu and define manual compile points.

2. Set top-level constraints like input/output delays, clock frequencies or multicycle paths.

   You do not have to redefine compile point constraints at the top level as the tool uses them to synthesize the compile points.

3. Define manual compile points if needed.

© 2016 Synopsys, Inc.

See *Defining Manual Compile Points*, on page 472 for details.

4. Save the top-level constraints file and add it to the project.

Defining Manual Compile Points

Compile points and constraints are both saved in a constraint file, so this step can be combined with the setting of constraints, as convenient. This procedure only describes how to define compile points. You define compile points in a top-level constraint file. You can add the compile point definitions to an existing top-level constraint file or create a new file.

1. From the Current Design field, select the module for which you want to create the compile point.

2. Click the Compile Points tab in the top-level constraints file.

   See *Creating a Top-Level Constraints File for Compile Points*, on page 471 if you need information about creating this file.

3. Set the module you want as a compile point.

   Do this by either selecting a module from the drop-down list in the View column, or dragging the instance from the HDL Analyst RTL view to the View column. The equivalent Tcl command is define_compile_point, as shown in this example:

```
define_compile_point {v:work.m3} -type {black_box}
```

You can get a list of all the modules from which you can select and designate compile points with the Tcl find command, as shown here:

```
c_print [find -hier -view {*} -filter ((!@is_black_box) && (@is_verilog == 1 ||
@is_vhdl == 1))] -file view.txt
```

4. Set the Type to locked, hard, or soft, according to your design goals. See *Defining the Compile Point Type*, on page 474 for details.

   This tags the module as a compile point. The following figure shows the prgm_cntr module set as a locked compile point:

5. Save the top-level constraint file.

You can now open the compile point constraint file and define constraints for the compile point, as needed for manual compile points. See *Setting Constraints at the Compile Point Level*, on page 475 for details.

© 2016 Synopsys, Inc.

Defining the Compile Point Type

The compile point type you select depends on your design goals. For descriptions of the various compile point types, see *Compile Point Types*, on page 453. This procedure shows you how to set the compile point type in the top-level constraint file when you define the compile points:

1. When runtime is the main objective and QoR is not a primary concern, set the compile point type as follows on the SCOPE Compile Points tab:

Situation	Compile Point Type
RTL is almost ready	locked
RTL is still being built but the module interface is ready	black_box If you define black box compile points, you must update the value and rerun synthesis after you have completed the RTL for the compile point.

The following example shows the Tcl command and the equivalent version in the in the SCOPE GUI:

```
define_compile_point  {v:work.user_top} -type {locked}
```

	Enable	View	Type
1	☑	v:work.user_top	locked

2. When runtime and QoR are both important, do the following to ensure the best performance while still saving runtime:
   - Register the I/O boundaries for the compile points.
   - As far as possible, put the entire critical path into the same compile point.
   - Set each compile point type individually, using these compile point types:

Situation	Compile Point Type
Need boundary optimizations	soft
Do not need boundary optimizations	locked

3. If your goal is design preservation, set the compile point you want to preserve to locked.

Setting Constraints at the Compile Point Level

You can specify constraints for each compile point in individual constraint files. (See *Compile Point Constraint Files*, on page 458 for a description of the files.) It is recommended that you specify constraints for each locked manual compile point, but you do not need to set them for soft and hard compile points.

When you specify compile point constraints, the tool synthesizes the compile point using the compile point timing models instead of automatic interface timing from the top level. This procedure explains how to create a (compile point constraint file, and set constraints for the compile point:

1. In an open project, click the SCOPE icon ( ). Click on the FPGA Constraints (SCOPE) button. The New Constraints File dialog box opens.

2. From the Current Design field, select the module for which you want to create the compile point.

3. Check that you are in the right file.

   A default name for the compile point file appears in the banner of the SCOPE window. Unlike the top-level constraint file, the Compile Point tab in the SCOPE UI is greyed out when the constraint file is for a compile point.

© 2016 Synopsys, Inc.

4. Set constraints for the compile point. In particular, do the following:
   - Define clocks for the compile point.
   - Specify I/O delay constraints for non-registered I/O paths that may be critical or near critical.
   - Set port constraints for the compile point that are needed for top-level mapping.

   The tool uses the compile point constraints you define to synthesize the compile point. Compile point port constraints are not used at the parent level, because compile point ports do not exist at that level.

   You can specify SCOPE attributes for the compile point as usual. See *Using Attributes with Compile Points*, on page 476 for some exceptions.

5. Save the file and add it to the project. When prompted, click Yes to add the constraint file to the top-level design project.

   Otherwise, use Save As to write a file such as, *moduleName*.fdc to the current directory. The hierarchical paths for compile point modules in the constraint file are specified at the compile point level; not the top-level design.

Using Attributes with Compile Points

You can use attributes as usual when you set constraints for compile points. The following sections describe some caveats and exceptions:

- syn_hier

  When you use syn_hier on a compile point, the only valid value is flatten. All other values of this attribute are ignored for compile points. The

syn_hier attribute behaves normally for all other module boundaries that are not defined as compile points.

- syn_allowed_resources

  Apply the syn_allowed_resources attribute globally or to a compile point to specify its allowed resources. When a compile point is synthesized, the resources of its siblings and parents cannot be taken into account because it stands alone as an independent synthesis unit. This attribute limits dedicated resources such as block RAMs or DSPs that the compile point can use, so that there are adequate resources available during the top-down flow.

Analyzing Compile Point Results

The software writes all timing and area results to a single log file in the implementation directory. You can check this file and the RTL and Technology views to determine if your design has met the goals for area and performance. You can also view and isolate the critical paths, search for and highlight design objects and crossprobe between the schematics and source files.

1. Check that the design meets the target frequency for the design. Use the Watch window or check the log file.

2. Open the log file and check the following:
   - Check top-level and compile point boundary timing. You can also check this visually using the RTL and Technology view schematics. If you find negative slack, check the critical path. If the critical path crosses the compile point boundary, you might need to improve the compile point constraints.
   - If the design was resynthesized, check the Summary of Compile Points section to see if compile points were preserved or remapped.

```
Summary of Compile Points :
***************************
Name            Status      Reason                  Start Time                  End Time                    Realtime      CPU Time
-------------------------------------------------------------------------------------------------------------------------------------
user_top        Remapped    Mapping options changed Mon Mar 14 04:43:42 2011    Mon Mar 14 04:44:27 2011    0h:00m:44s    0h:00m:33s
```

   - Review all warnings and determine which should be addressed and which can be ignored.

© 2016 Synopsys, Inc.

- Review the area report in the log file and determine if the cell usage is acceptable for your design.
- Check all DRC information.

3. Check other files:
   - Check the individual compile point module log files. The tool creates a separate directory for each compile point module under the implementation directory. Check the compile point log file in this directory for synthesis information about the compile point synthesis run.
   - Check the compile point timing report. This report is located in the compile point results directory of the implementation directory for each compile point.

4. Check the RTL and Technology view schematics for a graphic view of the design logic. Even though instantiations of compile points do not have unique names in the output netlist, they have unique names in the Technology view. This is to facilitate timing analysis and the viewing of critical paths.

   Note: Compile points of type {hard} are easily located in the Technology view with the color green.

5. Fix any errors.

   Remember that the mapper reports an error if synthesis at a parent level requires that interface changes be made to a locked compile point. The software does not change the compile point interface, even if changes are required to fix DRC violations.

Resynthesizing Incrementally

Incremental synthesis can significantly reduce runtime on subsequent runs. It can also help with design stabilization and preservation. The following describe the incremental synthesis process, and how compile points are used in incremental synthesis within the tool and with other tools:

- Incremental Compile Point Synthesis, on page 465
- Resynthesizing Compile Points Incrementally, on page 479

Resynthesizing Compile Points Incrementally

The following figure illustrates how compile points (CP) are used in incremental synthesis.

1. To synthesize a design incrementally, make the changes you need to fix errors or improve your design.

© 2016 Synopsys, Inc.

- Define new compile point constraints or modify existing constraints in the existing constraint file or in a new constraint file for the compile point. Save the file.
- If necessary, reset implementation options. Click Implementation Options and modify the settings (operating conditions, optimization switches, and global frequency).

To obtain the best results, define any required constraints and set the proper implementation options for the compile point before resynthesizing.

2. Click Run to resynthesize the design.

   When a design is resynthesized, compile points are not resynthesized unless source code logic, implementation options, or constraints have been modified. If there are no compile point interface changes, the software synthesizes the immediate parent using the previously generated model file for the compile point. See *Incremental Compile Point Synthesis*, on page 465 for details.

3. Check the log file for changes.

   The following figure illustrates incremental synthesis by comparing compile point summaries. After the first run, a syntax change was made in the mult module, and a logic change in the comb_logic module. The figure shows that incremental synthesis resynthesizes comb_logic (logic change), but does not resynthesize mult because the logic did not change even though there was a syntax change. Incremental synthesis re-uses the mapped file generated from the previous run to incrementally synthesize the top level.

4. To force the software to generate a new model file for the compile point, click Implementation Options on the Device tab and enable Update Compile Point Timing Data. Click Run.

   The software regenerates the model file for each compile point when it synthesizes the compile points. The new model file is used to synthesize the parent. The option remains in effect until you disable it.

5. To override incremental synthesis and force the software to resynthesize all compile points whether or not there have been changes made, use the Run->Resynthesize All command.

   You might want to force resynthesis to propagate changes from a locked compile point to its environment, or resynthesize compile points one last time before tape out. When you use this option, incremental synthesis is disabled for the current run only. The Resynthesize All command does not regenerate model files for the compile points unless there are interface changes. If you enable Update Compile Point Timing Data and select Resynthesize All, you can resynthesize the entire design and regenerate the compile point model files, but synthesis will take longer than an incremental synthesis run.

© 2016 Synopsys, Inc.

CHAPTER 12

Working with IP Input

This chapter describes how to work with IP from different sources. It describes the following:

- Generating IP with SYNCore, on page 484
- The Synopsys FPGA IP Encryption Flow, on page 524
- Working with Encrypted IP, on page 529
- Using Hyper Source, on page 543
- Working with Lattice IP, on page 548

© 2016 Synopsys, Inc.

Generating IP with SYNCore

You can use the SYNCore IP wizard to generate FIFO, RAM, ROM, adder/subtractor, and counter implementations. See the following for more information.

- Specifying FIFOs with SYNCore, on page 484
- Specifying RAMs with SYNCore, on page 490
- Specifying Byte-Enable RAMs with SYNCore, on page 497
- Specifying ROMs with SYNCore, on page 504
- Specifying Adder/Subtractors with SYNCore, on page 509
- Specifying Counters with SYNCore, on page 516

Specifying FIFOs with SYNCore

The SYNCore IP Wizard helps you generate Verilog code for your FIFO implementations. The following procedure shows you how to generate Verilog code for a FIFO using the SYNCore IP wizard.

Note: The SYNCore FIFO model uses Verilog 2001. When adding a FIFO model to a Verilog-95 design, be sure to enable the Verilog 2001 check box on the Verilog tab of the Implementation Options dialog box or include a set_option -vlog_std v2001 statement in your project file to prevent a syntax error.

---

1. Start the wizard.
   - From the synthesis tool GUI, select Run->Launch SYNCore or click the Launch SYNCore icon to start the SYNCore IP wizard.

– In the window that opens, select sfifo_model and click Ok. This opens the first screen of the wizard.

2. Specify the parameters you need in the five pages of the wizard. For details, refer to *Specifying SYNCore FIFO Parameters*, on page 488.

   The FIFO symbol on the left reflects the parameters you set.

© 2016 Synopsys, Inc.

3. After you have specified all the parameters you need, click the Generate button (lower left).

   The tool displays a confirmation message (TCL execution successful!) and writes the required files to the directory you specified in the parameters. The HDL code is in Verilog.

   The FIFO generated is a synchronous FIFO with symmetric ports and with the same clock controlling both the read and write operations. Data is written or read on the rising edge of the clock. All resets are synchronous with the clock. All edges (clock, enable, and reset) are considered positive.

   SYNCore also generates a testbench for the FIFO that you can use for simulation. The testbench covers a limited set of vectors for testing.

   You can now close the SYNCore wizard.

4. Add the FIFO you generated to your design.
   - Use the Add File command to add the Verilog design file that was generated and the syncore_sfifo.v file to your project. These files are in the directory for output files that you specified on page 1 of the wizard.
   - Use a text editor to open the instantiation_file.vin template file, which is located in the same directory. Copy the lines that define the memory, and paste them into your top-level module. The following shows a template file (in red text) inserted into a top-level module.

```
module top (
input Clk,
input [15:0] DataIn,
input WrEn,
input RdEn,
output Full,
output Empty,
output [15:0] DataOut
);

fifo_a32 <instanceName>(
.Clock(Clock)
,.Din(Din)
,.Write_enable(Write_enable)
,.Read_enable(Read_enable)
,.Dout(Dout)
,.Full(Full)
,.Empty(Empty)
)

endmodule
```

template

5. Edit the template port connections so that they agree with the port definitions in your top-level module as shown in the example below (the updated connection names are shown in red). You can also assign a unique name to each instantiation.

© 2016 Synopsys, Inc.

```
module top (
input Clk,
input [15:0] DataIn,
input WrEn,
input RdEn,
output Full,
output Empty,
output [15:0] DataOut
);

fifo_a32 busfifo(
.Clock(Clk)
,.Din(DataIn)
,.Write_enable(WrEn)
,.Read_enable(RdEn)
,.Dout(DataOut)
,.Full(Full)
,.Empty(Empty)
)
endmodule
```

Note that currently, the FIFO models will not be implemented with the dedicated FIFO blocks available in certain technologies.

Specifying SYNCore FIFO Parameters

The following elaborates on the parameter settings for SYNCore FIFOs. The status, handshaking, and programmable flags are optional. For descriptions of the parameters, see SYNCore FIFO Wizard, on page 253 in the *Reference Manual*. For timing diagrams, see SYNCore FIFO Compiler, on page 316in the *Reference Manual*.

1. Start the SYNCore wizard, as described in *Specifying FIFOs with SYNCore*, on page 484.

2. Do the following on page 1 of the FIFO wizard:
   - In Component Name, specify a name for the FIFO. Do not use spaces.
   - In Directory, specify a directory where you want the output files to be written. Do not use spaces.

- In Filename, specify a name for the Verilog output file with the FIFO specifications. Do not use spaces.
- Click Next. The wizard opens another page where you can set parameters.

3. For a FIFO with no status, handshaking, or programmable flags, use the default settings. You can generate the FIFO, as described in *Specifying FIFOs with SYNCore*, on page 484.

4. To set an almost full status flag, do the following on page 2 of the FIFO wizard:
   - Enable Almost Full.
   - Set associated handshaking flags for the signal as desired, with the Overflow Flag and Write Acknowledge options.
   - Click Next when you are done.

5. To set an almost empty status flag, do the following on page 3:
   - Enable Almost Empty.
   - Set associated handshaking flags for the signal as desired, with the Underflow Flag and Read Acknowledge options.
   - Click Next when you are done.

6. To set a programmable full flag, do the following:
   - Make sure you have enabled Full on page 2 of the wizard and set any handshaking flags you require.
   - Go to page 4 and enable Programmable Full.
   - Select one of the four mutually exclusive configurations for Programmable Full on page 4. See Programmable Full, on page 327 in the *Reference Manual* for details.
   - Click Next when you are done.

7. To set a programmable empty flag, do the following:
   - Make sure you have enabled Empty on page 3 of the wizard and set any handshaking flags you require.
   - Go to page 5 and enable Programmable Empty.
   - Select one of the four mutually exclusive configurations for Programmable Empty on page 5. See Programmable Empty, on page 330 in the *Reference Manual* for details.

© 2016 Synopsys, Inc.

You can now generate the FIFO and add it to the design, as described in *Specifying FIFOs with SYNCore*, on page 484.

Specifying RAMs with SYNCore

The SYNCore IP wizard helps you generate Verilog code for your RAM implementation requirements. The following procedure shows you how to generate Verilog code for a RAM using the SYNCore IP wizard.

Note: The SYNCore RAM model uses Verilog 2001. When adding a RAM model to a Verilog-95 design, be sure to enable the Verilog 2001 check box on the Verilog tab of the Implementation Options dialog box or include a set_option -vlog_std v2001 statement in your project file to prevent a syntax error.

---

1. Start the wizard.
   - From the synthesis tool GUI, select Run->Launch SYNCore or click the Launch SYNCore icon to start the SYNCore IP wizard.

   - In the window that opens, select ram_model and click Ok. This opens the first screen of the wizard.

2. Specify the parameters you need in the wizard.
   - For details about the parameters for a single-port RAM, see *Specifying Parameters for Single-Port RAM*, on page 493.

– For details about the parameters for a dual-port RAM, see *Specifying Parameters for Dual-Port RAM*, on page 494. Note that dual-port implementations are only supported for some technologies.

The RAM symbol on the left reflects the parameters you set.

The default settings for the tool implement a block RAM with synchronous resets, and where all edges (clock, enable, and reset) are considered positive.

3. After you have specified all the parameters you need, click the Generate button in the lower left corner.

   The tool displays a confirmation message is displayed (TCL execution successful!) and writes the required files to the directory you specified in the parameters. The HDL code is in Verilog.

   SYNCore also generates a testbench for the RAM. The testbench covers a limited set of vectors.

   You can now close the SYNCore Memory Compiler.

4. Edit the RAM files if necessary.

   – The default RAM has a no_rw_check attribute enabled. If you do not want this, edit syncore_ram.v and comment out the `define SYN_MULTI_PORT_RAM statement, or use `undef SYN_MULTI_PORT_RAM.

   – If you want to use the synchronous RAMs available in the target technology, make sure to register either the read address or the outputs.

5. Add the RAM you generated to your design.

   – Use the Add File command to add the Verilog design file that was generated and the syncore_ram.v file to your project. These files are in the directory for output files that you specified on page 1 of the wizard.

   – Use a text editor to open the instantiation_file.vin template file, which is located in the same directory. Copy the lines that define the memory, and paste them into your top-level module. The following figure shows a template file (in red text) inserted into a top-level module.

© 2016 Synopsys, Inc.

```
module top (

input ClkA,
input [7:0] AddrA,
input [15:0] DataInA,
input WrEnA,

output [15:0] DataOutA

);

myram2 <InstanceName> (
.PortAClk(PortAClk)
, .PortAAddr(PortAAddr)
, .PortADataIn(PortADataIn)
, .PortAWriteEnable(PortAWriteEnable)
, .PortADataOut(PortADataOut)
);

endmodule
```

template

6. Edit the template port connections so that they agree with the port definitions in your top-level module as shown in the example below (the updated connection names are shown in red). You can also assign a unique name to each instantiation.

```
module top (

input ClkA,
input [7:0] AddrA,
input [15:0] DataInA,
input WrEnA,

output [15:0] DataOutA

);


myram2 decoderram(
.PortAClk(ClkA)
, .PortAAddr(AddrA)
, .PortADataIn(DataInA)
, .PortAWriteEnable(WrEnA)
, .PortADataOut(DataOutA)
);

endmodule
```

Specifying Parameters for Single-Port RAM

To create a single-port RAM with the SYNCore Memory Compiler, you need to specify a single read/write address (single port) and a single clock. You only need to configure Port A. The following procedure lists what you need to specify. For descriptions of each parameter, refer to SYNCore RAM Wizard, on page 263 in the *Reference Manual*.

1. Start the SYNCore RAM wizard, as described in *Specifying RAMs with SYNCore*, on page 490.
2. Do the following on page 1 of the RAM wizard:
   - In Component Name, specify a name for the memory. Do not use spaces.
   - In Directory, specify a directory where you want the output files to be written. Do not use spaces.

© 2016 Synopsys, Inc.

- In Filename, specify a name for the Verilog file that will be generated with the RAM specifications. Do not use spaces.
- Enter data and address widths.
- Enable Single Port, to specify that you want to generate a single-port RAM. This automatically enables Single Clock.
- Click Next. The wizard opens another page where you can set parameters for Port A.

The RAM symbol dynamically updates to reflect the parameters you set.

3. Do the following on page 2 of the RAM wizard:
   - Set Use Write Enable to the setting you want.
   - Set Register Read Address to the setting you want.
   - Set Synchronous Reset to the setting you want. Register Outputs is always enabled
   - Specify the read access you require for the RAM.

   You can now generate the RAM by clicking Generate, as described in *Specifying RAMs with SYNCore*, on page 490. You do not need to specify any parameters on page 3, as this is a single-port RAM and you do not need to specify Port B. All output files are in the directory you specified on the first page of the wizard.

   For details about setting dual-port RAM parameters, see *Specifying Parameters for Dual-Port RAM*, on page 494. For read/write timing diagrams, see Read/Write Timing Sequences, on page 341 of the *Reference Manual*.

Specifying Parameters for Dual-Port RAM

The following procedure shows you how to set parameters for dual-port memory in the SYNCore wizard. Dual-port RAMs are only supported for some technologies. For information about generating single-port RAMs, see *Specifying Parameters for Single-Port RAM*, on page 493. It shows you how to generate these common RAM configurations:

- One read access and one write access
- Two read accesses and one write access
- Two read accesses and two write accesses

For the corresponding read/write timing diagrams, see Read/Write Timing Sequences, on page 341 of the *Reference Manual*.

1. Start the SYNCore RAM wizard, as described in *Generating IP with SYNCore*, on page 484.

2. Do the following on page 1 of the RAM wizard:
   - In Component Name, specify a name for the memory. Do not use spaces.
   - In Directory, specify a directory where you want the output files to be written. Do not use spaces.
   - In Filename, specify a name for the Verilog file that will be generated with the RAM specifications. Do not use spaces.
   - Enter data and address widths.
   - Enable Dual Port, to specify that you want to generate a dual-port RAM.
   - Specify the clocks.

For a single clock ...	Enable Single Clock.
For separate clocks for each of the ports ...	Enable Separate Clocks For Each Port.

   - Click Next. The wizard opens another page where you can set parameters for Port A.

3. Do the following on page 2 of the RAM wizard to specify settings for Port A:
   - Set parameters according to the kind of memory you want to generate:

One read & one write	Enable Read Only Access.
Two reads & one write	Enable Read and Write Access. Specify a setting for Use Write Enable.
Two reads & two writes	Enable Read and Write Access. Specify a setting for Use Write Enable. Specify a read access option for Port A.

   - Specify a setting for Register Read Address.

© 2016 Synopsys, Inc.

- Set Synchronous Reset to the setting you want. Register Outputs is always enabled.
- Click Next. The wizard opens another page where you can set parameters for Port B. The page and the parameters are identical to the previous page, except that the settings are for Port B instead of Port A.

4. Specify the settings for Port B on page 3 of the wizard according to the kind of memory you want to generate:

One read & one write	Enable Write Only Access. Set Use Write Enable to the setting you want.
Two reads & one write	Enable Read Only Access. Specify a setting for Register Read Address.
Two reads & two writes	Enable Read and Write Access. Specify a setting for Use Write Enable. Specify a setting for Register Read Address. Set Synchronous Reset to the setting you want. Note that Register Outputs is always enabled. Select a read access option for Port B.

The RAM symbol on the left reflects the parameters you set. All output files are written to the directory you specified on the first page of the wizard.

You can now generate the RAM by clicking Generate, as described in *Generating IP with SYNCore*, on page 484, and add it to your design.

Specifying Byte-Enable RAMs with SYNCore

The SYNCore IP wizard helps you generate SystemVerilog code for your byte-enable RAM implementation requirements. The following procedure shows you how to generate SystemVerilog code for a byte-enable RAM using the SYNCore IP wizard.

Note: The SYNCore byte-enable RAM model uses SystemVerilog. When adding a byte-enable RAM to your design, be sure to enable the System Verilog check box on the Verilog tab of the Implementation Options dialog box or include a set_option -vlog_std sysv statement in your project file to prevent a syntax error.

---

1. Start the wizard.
   - From the FPGA synthesis tool GUI, select Run->Launch SYNCore or click the Launch SYNCore icon to start the SYNCore IP wizard.

   - In the window that opens, select byte_en_ram_model and click Ok to open the first page (page1) of the wizard.

© 2016 Synopsys, Inc.

2. Specify the parameters you need in the wizard. For details about the parameters, see *Specifying Byte-Enable RAM Parameters*, on page 501. The BYTE ENABLE RAM symbol on the left reflects any parameters you set.

3. After you have specified all the parameters you need, click the Generate button in the lower left corner. The tool displays a confirmation message (TCL execution successful!) and writes the required files to the directory you specified on page 1 of the wizard. The HDL code is in SystemVerilog.

   SYNCore also generates a test bench for the byte-enable RAM component. The test bench covers a limited set of vectors. You can now close the SYNCore byte-enable RAM compiler.

4. Edit the generated files for the byte-enable RAM component if necessary.

5. Add the byte-enable RAM that you generated to your design.
   - On the Verilog tab of the Implementation Options dialog box, make sure that SystemVerilog is enabled.
   - Use the Add File command to add the Verilog design file that was generated (the filename entered on page 1 of the wizard) and the

syncore_*.v file to your project. These files are in the directory for output files that you specified on page 1 of the wizard.

– Use a text editor to open the instantiation_file.vin template file. This file is located in the same output files directory. Copy the lines that define the byte-enable RAM and paste them into your top-level module.

– Edit the template port connections so that they agree with the port definitions in the top-level module; also change the instantiation name to agree with the component name entered on page 1. The following figure shows a template file inserted into a top-level module with the updated component name and port connections in red.

© 2016 Synopsys, Inc.

```
module top
   (input ClockA,
    input [3:0] AddA
    input [31:0] DataIn
    input WrEnA,
    input Reset
    output [31:0] DataOut
   )

INST_TAG

SP_RAM #
   (.ADD_WIDTH(4),
    .WE_WIDTH(2),
    .RADDR_LTNCY_A(1), // 0 - No Latency, 1 - 1 Cycle Latency
    .RDATA_LTNCY_A(1), // 0 - No Latency, 1 - 1 Cycle Latency
    .RST_TYPE_A(1), // 0 - No Reset, 1 synchronous
    .RST_RDATA_A({32{1'b1}}),
    .DATA_WIDTH(32)
   )

4x32spram
   (// Output Ports
    .RdDataA(DataIn),
    // Input Ports
    .WrDataA(DataOut),
    .WenA(WeEnA),
    .AddrA(AddA),
    .ResetA(Reset),
    .ClkA(ClockA)
    ;
```

Port List

Port A interface signals are applicable for both single-port and dual-port configurations; Port B signals are applicable for dual-port configuration only.

Name	Type	Description
ClkA	Input	Clock input for Port A
WenA	Input	Write enable for Port A; present when Port A is in write mode
AddrA	Input	Memory access address for Port A
ResetA	Input	Reset for memory and all registers in core; present with registered read data when Reset is enabled; active low (cannot be changed)
WrDataA	Input	Write data to memory for Port A; present when Port A is in write mode
RdDataA	Output	Read data output for Port A; present when Port A is in read or read/write mode
ClkB	Input	Clock input for Port B; present in dual-port mode
WenB	Input	Write enable for Port B; present in dual-port mode when Port B is in write mode
AddrB	Input	Memory access address for Port B; present in dual-port mode
ResetB	Input	Reset for memory and all registers in core; present in dual-port mode when read data is registered and Reset is enabled; active low (cannot be changed)
WrDataB	Input	Write data to memory for Port B; present in dual-port mode when Port B is in write mode
RdDataB	Output	Read data output for Port B; present in dual-port mode when Port B is in read or read/write mode

Specifying Byte-Enable RAM Parameters

When creating a single-port, byte-enable RAM with the SYNCore IP wizard, you must specify a single read address and a single clock; you only need to configure the Port A parameters on page 2 of the wizard.

© 2016 Synopsys, Inc.

When creating a dual-port, byte-enable RAM, you must additionally configure the Port B parameters on page 3 of the wizard.

The following procedure lists the parameters you need to specify. For descriptions of each parameter, refer to Parameter List, on page 348 in the Reference Manual.

1. Start the SYNCore byte-enable RAM wizard as described in *Specifying Byte-Enable RAMs with SYNCore*, on page 497.

2. Do the following on page 1 of the byte-enable RAM wizard:
   - Specify a name for the memory in the Component Name field; do not use spaces.
   - Specify a directory name in the Directory field where you want the output files to be written; do not use spaces.
   - Specify a name in the File Name field for the SystemVerilog file to be generated with the byte-enable RAM specifications; do not use spaces.
   - Enter a value for the address width of the byte-enable RAM; the maximum depth of memory is limited to 2^256.
   - Enter a value for the data width for the byte-enable RAM; data width values range from 2 to 256.
   - Enter a value for the write enable width; write-enable width values range from 1 to 4.
   - Select Single Port to generate a single-port, byte-enable RAM or select Dual Port to generate a dual-port, byte-enable RAM.
   - Click Next to open page 2 of the wizard.

   The Byte Enable RAM symbol dynamically updates to reflect the parameters that you set.

3. Do the following on page 2 (configuring Port A) of the wizard:
   - Select the Port A configuration. Only Read and Write Access mode is valid for single-port configurations; this mode is selected by default.
   - Set Pipelining Address Bus and Output Data according to your application. By default, read data is registered; you can register both the address and data registers.
   - Set the Configure Reset Options. Enabling the checkbox enables the synchronous reset for read data. This option is enabled only when the read data is registered. Reset is active low and cannot be changed.
   - Configure output reset data value options under Specify output data on reset; reset data can be set to default value of all '1' s or to a user-defined decimal value. Reset data value options are disabled when the reset is not enabled for Port A.
   - Set Write Enable for Port A value; default for the write-enable level is active high.
4. If you are generating a dual-port, byte-enable RAM, set the Port B parameters on page 3 (note that the Port B parameters are only enabled when Dual Port is selected on page 1).

   The Port B parameters are identical to the Port A parameters on page 2. When using the dual-port configuration, when one port is configured for read access, the other port can only be configured for read/write access or write access.
5. Generate the byte-enable RAM by clicking Generate. Add the file to your project and edit the template file as described in *Specifying Byte-Enable RAMs with SYNCore*, on page 497. For read/write timing diagrams, see Read/Write Timing Sequences, on page 345 of the reference manual.

© 2016 Synopsys, Inc.

Specifying ROMs with SYNCore

The SYNCore IP wizard helps you generate Verilog code for your ROM implementation requirements. The following procedure shows you how to generate Verilog code for a ROM using the SYNCore IP wizard.

Note: The SYNCore ROM model uses Verilog 2001. When adding a ROM model to a Verilog-95 design, be sure to enable the Verilog 2001 check box on the Verilog tab of the Implementation Options dialog box or include a set_option -vlog_std v2001 statement in your project file to prevent a syntax error.

---

1. Start the wizard.
   - From the FPGA synthesis tool GUI, select Run->Launch SYNCore or click the Launch SYNCore icon to start the SYNCore IP wizard.

   - In the window that opens, select rom_model and click Ok to open page 1 of the wizard.

2. Specify the parameters you need in the wizard. For details about the parameters, see *Specifying ROM Parameters*, on page 508. The ROM symbol on the left reflects any parameters you set.

3. After you have specified all the parameters you need, click the Generate button in the lower left corner. The tool displays a confirmation message (TCL execution successful!) and writes the required files to the directory you specified on page 1 of the wizard. The HDL code is in Verilog.

   SYNCore also generates a testbench for the ROM. The testbench covers a limited set of vectors.

   You can now close the SYNCore ROM Compiler.

4. Edit the ROM files if necessary. If you want to use the synchronous ROMs available in the target technology, make sure to register either the read address or the outputs.

5. Add the ROM you generated to your design.
   – Use the Add File command to add the Verilog design file that was generated and the syncore_rom.v file to your project. These files are in

© 2016 Synopsys, Inc.

the directory for output files that you specified on page 1 of the wizard.

– Use a text editor to open the instantiation_file.vin template file. This file is located in the same output files directory. Copy the lines that define the ROM, and paste them into your top-level module. The following figure shows a template file (in red text) inserted into a top-level module.

```
module test_rom_style(z,a,clk,en,rst);
input clk,en,rst;
output reg [3:0] z;
input [6:0] a;

my1stROM   <InstanceName> (
   // Output Ports
   .DataA(DataA),

   // Input  Ports
   .ClkA(ClkA),
   .EnA(EnA),
   .ResetA(ResetA),
   .AddrA(AddrA)
);
```

template

6. Edit the template port connections so that they agree with the port definitions in the top-level module as shown in the example below. You can also assign a unique name to each instantiation.

```
module test_rom_style(z,a,clk,en,rst);
input clk,en,rst;
output reg [3:0] z;
input [6:0] a;

my1stROM decode_rom(
   // Output Ports
   .DataA(z),

   // Input  Ports
   .ClkA(clk),
   .EnA(en),
   .ResetA(rst),
   .AddrA(a)
);
```

Port List

PortA interface signals are applicable for both single-port and dual-port configurations; PortB signals are applicable for dual-port configuration only.

Name	Type	Description
ClkA	Input	Clock input for Port A
EnA	Input	Enable input for Port A
AddrA	Input	Read address for Port A
ResetA	Input	Reset or interface disable pin for Port A
DataA	Output	Read data output for Port A
ClkB	Input	Clock input for Port B
EnB	Input	Enable input for Port B
AddrB	Input	Read address for Port B
ResetB	Input	Reset or interface disable pin for Port B
DataB	Output	Read data output for Port B

© 2016 Synopsys, Inc.

Specifying ROM Parameters

If you are creating a single-port ROM with the SYNCore IP wizard, you need to specify a single read address and a single clock, and you only need to configure the Port A parameters on page 2. If you are creating a dual-port ROM, you must additionally configure the Port B parameters on page 3. The following procedure lists what you need to specify. For descriptions of each parameter, refer to SYNCore RAM Wizard, on page 263 in the *Reference Manual*.

1. Start the SYNCore ROM wizard, as described in *Specifying ROMs with SYNCore*, on page 504.

2. Do the following on page 1 of the ROM wizard:
   - In Component Name, specify a name for the memory. Do not use spaces.
   - In Directory, specify a directory where you want the output files to be written. Do not use spaces.
   - In Filename, specify a name for the Verilog file that will be generated with the ROM specifications. Do not use spaces.
   - Enter values for Read Data width and ROM address width (minimum depth value is 2; maximum depth of the memory is limited to 2^256).
   - Select Single Port Rom to indicate that you want to generate a single-port ROM or select Dual Port Rom to generate a dual-port ROM.
   - Click Next. The wizard opens page 2 where you set parameters for Port A.

   The ROM symbol dynamically updates to reflect any parameters you set.

3. Do the following on page 2 (Configuring Port A) of the RAM wizard:
   - For synchronous ROMs, select Register address bus AddrA and/or Register output data bus DataA to register the read address and/or the outputs. Selecting either checkbox enables the Enable for Port A checkbox which is used to select the Enable level.
   - Set the Configure Reset Options. Enabling the checkbox enables the type of reset (asynchronous or synchronous) and allows an output data pattern (all 1's or a specified pattern) to be defined on page 4.

4. If you are generating a dual-port ROM, set the port B parameters on page 3 (the page 3 parameters are only enabled when Dual Port Rom is selected on page 1).

5. On page 4, specify the location of the ROM initialization file and the data format (Hexadecimal or Binary). ROM initialization is supported using memory-coefficient files. The data format is either binary or hexadecimal with each data entry on a new line in the memory-coefficient file (specified by parameter INIT_FILE). Supported file types are `txt`, `mem`, `dat`, and `init` (recommended).

6. Generate the ROM by clicking Generate, as described in *Specifying ROMs with SYNCore*, on page 504 and add it to your design. All output files are in the directory you specified on page 1 of the wizard.

   For read/write timing diagrams, see Read/Write Timing Sequences, on page 341 of the *Reference Manual*.

Specifying Adder/Subtractors with SYNCore

The SYNCore IP wizard helps you generate Verilog code for your adder/subtractor implementation requirements. The following procedure shows you how to generate Verilog code for an adder/subtractor using the SYNCore IP wizard.

Note: The SYNCore adder/subtractor models use Verilog 2001. When adding an adder/subtractor model to a Verilog-95 design, be sure to enable the Verilog 2001 check box on the Verilog tab of the Implementation Options dialog box or include a set_option -vlog_std v2001 statement in your project file to prevent a syntax error.

---

1. Start the wizard.
   – From the FPGA synthesis tool GUI, select Run->Launch SYNCore or click the Launch SYNCore icon to start the SYNCore IP wizard.

© 2016 Synopsys, Inc.

– In the window that opens, select addnsub_model and click Ok to open page1 of the wizard.

2. Specify the parameters you need in the wizard. For details about the parameters, see *Specifying Adder/Subtractor Parameters*, on page 515. The ADDnSUB symbol on the left reflects any parameters you set.

3. After you have specified all the parameters you need, click the Generate button in the lower left corner.

   The tool displays a confirmation message (TCL execution successful!) and writes the required files to the directory you specified on page 1 of the wizard. The HDL code is in Verilog.

   The SYNCore wizard also generates a testbench for your adder/subtractor. The testbench covers a limited set of vectors. You can now close the wizard.

4. Add the adder/subtractor you generated to your design.
   - Edit the adder/subtractor files if necessary.
   - Use the Add File command to add the Verilog design file that was generated and the syncore_addnsub.v file to your project. These files are in the directory for output files that you specified on page 1 of the wizard.
   - Use a text editor to open the instantiation_file.v template file. This file is located in the same output files directory. Copy the lines that define the adder/subtractor and paste them into your top-level module. The following figure shows a template file (in red text) inserted into a top-level module.

© 2016 Synopsys, Inc.

```
module top (
    output [15 : 0] Out,
    input Clk,
    input [15 : 0] A,
    input CEA,
    input RSTA,
    input [15 : 0] B,
    input CEB,
    input RSTB,
    input CEOut,
    input RSTOut,
    input ADDnSUB,
    input CarryIn );

My_ADDnSUB <InstanceName> (
// Output Ports
    .PortOut(PortOut),
// Input  Ports
    .PortClk(PortClk),
    .PortA(PortA),
    .PortCEA(PortCEA),
    .PortRSTA(PortRSTA),
    .PortB(PortB),
    .PortCEB(PortCEB),
    .PortRSTB(PortRSTB),
    .PortCEOut(PortCEOut),
    .PortRSTOut(PortRSTOut),
    .PortADDnSUB(PortADDnSUB),
    .PortCarryIn(PortCarryIn) );

endmodule
```

template

5. Edit the template port connections so that they agree with the port definitions in your top-level module as shown in the example below (the updated connection names are shown in red). You can also assign a unique name to each instantiation.

```
module top (
   output [15 : 0] Out,
   input Clk,
   input [15 : 0] A,
   input CEA,
   input RSTA,
```

```
   input [15 : 0] B,
   input CEB,
   input RSTB,
   input CEOut,
   input RSTOut,
   input ADDnSUB,
   input CarryIn);

My_ADDnSUB ADDnSUB_inst(
// Output Ports
   .PortOut(Out),
// Input  Ports
   .PortClk(Clk),
   .PortA(A),
   .PortCEA(CEA),
   .PortRSTA(RSTA),
   .PortB(B),
   .PortCEB(CEB),
   .PortRSTB(RSTB),
   .PortCEOut(CEOut),
   .PortRSTOut(RSTOut),
   .PortADDnSUB(ADDnSUB),
   .PortCarryIn(CarryIn));
endmodule
```


© 2016 Synopsys, Inc.

Port List

The following table lists the port assignments for all possible configurations; the third column specifies the conditions under which the port is available.

Port Name	Description	Required/Optional
PortA	Data input for adder/subtractor Parameterized width and pipeline stages	Always present
PortB	Data input for adder/subtractor Parameterized width and pipeline stages	Not present if adder/subtractor is configured as a constant adder/subtractor
PortClk	Primary clock input; clocks all registers in the unit	Always present
PortRstA	Reset input for port A pipeline registers (active high)	Not present if pipeline stage for port A is 0
PortRstB	Reset input for port B pipeline registers (active high)	Not present if pipeline stage for port B is 0 or for constant adder/subtractor
PortADDnSUB	Selection port for dynamic operation	Not present if adder/subtractor configured as standalone adder or subtractor
PortRstOut	Reset input for output register (active high)	Not present if output pipeline stage is 0
PortCEA	Clock enable for port A pipeline registers (active high)	Not present if pipeline stage for port A is 0
PortCEB	Clock enable for port B pipeline registers (active high)	Not present if pipeline stage for port B is 0 or for constant adder/subtractor
PortCarryIn	Carry input for adder/subtractor	Always present
PortCEOut	Clock enable for output register (active high)	Not present if output pipeline stage is 0
PortOut	Data output	Always present

Specifying Adder/Subtractor Parameters

The SYNCore adder/subtractor can be configured as any of the following:

- Adder
- Subtractor
- Dynamic Adder/Subtractor

If you are creating a constant input adder, subtractor, or a dynamic adder/subtractor with the SYNCore IP wizard, you must select Constant Value Input and specify a value for port B in the Constant Value/Port B Width field on page 2 of the parameters. The following procedure lists the parameters you need to define when generating an adder/subtractor. For descriptions of each parameter, see *SYNCore Adder/Subtractor Wizard, on page 274* in the *Reference Manual*.

1. Start the SYNCore adder/subtractor wizard as described in *Specifying Adder/Subtractors with SYNCore*, on page 509.
2. Enter the following on page 1 of the wizard:
   - In the Component Name field, specify a name for your adder/subtractor. Do not use spaces.
   - In the Directory field, specify a directory where you want the output files to be written. Do not use spaces.
   - In the Filename field, specify a name for the Verilog file that will be generated with the adder/subtractor definitions. Do not use spaces.
   - Select the appropriate configuration in Configure the Mode of Operation.
3. Click Next. The wizard opens page 2 where you set parameters for port A and port B.
4. Configure Port A and B.
   - In the Configure Port A section, enter a value in the Port A Width field.
   - If you are defining a synchronous adder/subtractor, check Register Input A and then check Clock Enable for Register A and/or Reset for Register A.
   - To configure port B as a constant port, go to the Configure Port B section and check Constant Value Input. Enter the constant value in the Constant Value/Port B Width field.

© 2016 Synopsys, Inc.

- To configure port B as a dynamic port, go to the Configure Port B section and check Enable Port B and enter the port width in the Constant Value/Port B Width field.
- To define a synchronous adder/subtractor, check Register Input B and then check Clock Enable for Register B and/or Reset for Register B.

5. In the Configure Output Port section:
   - Enter a value in the Output port Width field.
   - If you are registering the output port, check Register output Port.
   - If you are defining a synchronous adder/subtractor check Clock Enable for Register PortOut and/or Reset for Register PortOut.
6. In the Configure Reset type for all Reset Signal section, click Synchronous Reset or Asynchronous Reset as appropriate.

   As you enter the page 2 parameters, the ADDnSUB symbol dynamically updates to reflect the parameters you set.
7. Generate the adder/subtractor by clicking the Generate button as described in *Specifying Adder/Subtractors with SYNCore*, on page 509 and add it to your design. All output files are in the directory you specified on page 1 of the wizard.

Specifying Counters with SYNCore

The SYNCore IP wizard helps you generate Verilog code for your counter implementation requirements. The following procedure shows you how to generate Verilog code for a counter using the SYNCore IP wizard.

Note: The SYNCore counter model uses Verilog 2001. When adding a counter model to a Verilog-95 design, be sure to enable the Verilog 2001 check box on the Verilog tab of the Implementation Options dialog box or include a set_option -vlog_std v2001 statement in your project file to prevent a syntax error.

---

1. Start the wizard.
   - From the FPGA synthesis tool GUI, select Run->Launch SYNCore or click the Launch SYNCore icon to start the SYNCore IP wizard.

– In the window that opens, select counter_model and click Ok to open page1 of the wizard.

2. Specify the parameters you need in the wizard. For details about the parameters, see *Specifying Counter Parameters*, on page 522. The COUNTER symbol on the left reflects any parameters you set.

© 2016 Synopsys, Inc.

3. After you have specified all the parameters you need, click the Generate button in the lower left corner.

   The tool displays a confirmation message (TCL execution successful!) and writes the required files to the directory you specified on page 1 of the wizard. The HDL code is in Verilog.

   The SYNCore wizard also generates a testbench for your counter. The testbench covers a limited set of vectors. You can now close the wizard.

4. Add the counter you generated to your design.
   - Edit the counter files if necessary.
   - Use the Add File command to add the Verilog design file that was generated and the syncore_addnsub.v file to your project. These files are in the directory for output files that you specified on page 1 of the wizard.
   - Use a text editor to open the instantiation_file.v template file. This file is located in the same output files directory. Copy the lines that define the counter and paste them into your top-level module. The following figure shows a template file (in red text) inserted into a top-level module.

```
module counter #(
    parameter COUNT_WIDTH = 5,
    parameter STEP = 2,
    parameter RESET_TYPE = 0,
    parameter LOAD = 2,
    parameter MODE = "Dynamic" )

(
// Output Ports
       output wire [WIDTH-1:0] Count,
// Input Ports
       input wire Clock,
       input wire Reset,
       input wire Up_Down,
       input wire Load,
       input wire [WIDTH-1:0] LoadValue,
       input wire Enable );

SynCoreCounter #(
    .COUNT_WIDTH(COUNT_WIDTH),
    .STEP(STEP),
    .RESET_TYPE(RESET_TYPE),
    .LOAD(LOAD),
    .MODE(MODE) )

SynCoreCounter_ins1 (
    .PortCount(PortCount),
    .PortClk(PortClk),
    .PortRST(PortRST),
    .PortUp_nDown(PortUp_nDown),
    .PortLoad(PortLoad),
    .PortLoadValue(PortLoadValue),
    .PortCE(PortCE) );

endmodule
```

template

© 2016 Synopsys, Inc.

```
module counter #(
   parameter COUNT_WIDTH = 5,
   parameter STEP = 2,
   parameter RESET_TYPE = 0,
   parameter LOAD = 2,
   parameter MODE = "Dynamic")

(
// Output Ports
      output wire [WIDTH-1:0] Count,
// Input Ports
      input wire Clock,
      input wire Reset,
      input wire Up_Down,
      input wire Load,
      input wire [WIDTH-1:0] LoadValue,
      input wire Enable);

SynCoreCounter #(
   .COUNT_WIDTH(COUNT_WIDTH),
   .STEP(STEP),
   .RESET_TYPE(RESET_TYPE),
   .LOAD(LOAD),
   .MODE(MODE))

SynCoreCounter_ins1 (
   .PortCount(PortCount),
   .PortClk(Clock),
   .PortRST(Reset),
   .PortUp_nDown(Up_Down),
   .PortLoad(Load),
   .PortLoadValue(LoadValue),
   .PortCE(Enable));

endmodule
```

template

5. Edit the template port connections so that they agree with the port definitions in your top-level module as shown in the example below (the updated connection names are shown in red). You can also assign a unique name to each instantiation. You can also assign a unique name to each instantiation.

```
module counter #(
   parameter COUNT_WIDTH = 5,
   parameter STEP = 2,
   parameter RESET_TYPE = 0,
   parameter LOAD = 2,
   parameter MODE = "Dynamic")

(
// Output Ports
      output wire [WIDTH-1:0] Count,
// Input Ports
      input wire Clock,
      input wire Reset,
      input wire Up_Down,
      input wire Load,
      input wire [WIDTH-1:0] LoadValue,
      input wire Enable);

SynCoreCounter #(
   .COUNT_WIDTH(COUNT_WIDTH),
   .STEP(STEP),
   .RESET_TYPE(RESET_TYPE),
   .LOAD(LOAD),
   .MODE(MODE))

SynCoreCounter_ins1 (
   .PortCount(Count),
   .PortClk(Clock),
   .PortRST(Reset),
   .PortUp_nDown(Up_Down),
   .PortLoad(Load),
   .PortLoadValue(LoadValue),
   .PortCE(Enable));

endmodule
```

Port List

The following table lists the port assignments for all possible configurations; the third column specifies the conditions under which the port is available.

© 2016 Synopsys, Inc.

Port Name	Description	Required/Optional
PortCE	Count Enable input pin with size one (active high)	Always present
PortClk	Primary clock input	Always present
PortLoad	Load Enable input which loads the counter (active high).	Not present for parameter LOAD=0
PortLoadValue	Load value primary input (active high)	Not present for parameter LOAD=0 and LOAD=1
PortRST	Reset input which resets the counter (active high)	Always present
PortUp_nDown	Primary input which determines the counter mode. 0 = Up counter 1 = Down counter	Present only for MODE="Dynamic"
PortCount	Counter primary output	Always present

Specifying Counter Parameters

The SYNCore counter can be configured for any of the following functions:

- Up Counter
- Down Counter
- Dynamic Up/Down Counter

The counter core can have a constant or variable input load or no load value. If you are creating a constant-load counter, you will need to select Enable Load and Load Constant Value on page 2 of the wizard. If you are creating a variable-load counter, you will need to select Enable Load and Use Variable Port Load on page 2. The following procedure lists the parameters you need to define when generating a counter. For descriptions of each parameter, see SYNCore Counter Wizard, on page 278 of the *Reference Manual*.

1. Start the SYNCore counter wizard, as described in *Specifying Counters with SYNCore*, on page 516.
2. Enter the following on page 1 of the wizard:
   - In the Component Name field, specify a name for your counter. Do not use spaces.
   - In the Directory field, specify a directory where you want the output files to be written. Do not use spaces.
   - In the Filename field, specify a name for the Verilog file that will be generated with the counter definitions. Do not use spaces.
   - Enter the width and depth of the counter in the Configure the Counter Parameters section.
   - Select the appropriate configuration in the Configure the Mode of Counter section.
3. Click Next. The wizard opens page 2 where you set parameters for PortLoad and PortLoadValue.
   - Select Enable Load option and the required load option in Configure Load Value section.
   - Select the required reset type in the Configure Reset type section.

   The COUNTER symbol dynamically updates to reflect the parameters you set.
4. Generate the counter core by clicking Generate button. All output files are written to the directory you specified on page1 of the wizard.

© 2016 Synopsys, Inc.

The Synopsys FPGA IP Encryption Flow

The Synopsys FPGA IP encryption flow is a design flow that encourages interoperability while protecting IP implementations using encryption/decryption technologies. This flow offers the following advantages: interoperability, protection of IP, reuse of IP, and a standard flow for IP encryption. Currently, Synopsys FPGA synthesis products support the following encryption technologies:

- IEEE 1735-2014 with key-block (Version 1)
- OpenIP
- Synenc-encrypted IP

See the following encryption technologies:

- Encrypting IP with the encryptP1735.pl Script, on page 531
- Encrypting IP with the encryptIP Script, on page 536

Overview of the Synopsys FPGA IP Flow

The complete flow for protecting IP requires a partnership between the IP vendor, Synopsys, and the silicon vendor as illustrated in the following figure. However, depending on the level of agreement between Synopsys and the silicon vendor downstream, the re-encryption of IP following synthesis can vary from the ideal flow shown in the figure.

For further details of the hand-offs between vendors and how encryption and decryption are handled, see *Encryption and Decryption*, on page 525.

Encryption and Decryption

There are two major classes of encryption/decryption algorithms: symmetric, and asymmetric (see Encryption and Decryption Methodologies, on page 373 in the *Reference Manual* for details). Each has its own advantages and disadvantages. The approach for the Synopsys FPGA IP flow is a hybrid scheme that uses both asymmetric and symmetric encryption to leverage the strengths of each scheme. The methodology described here can also be used for other design hand-offs. For example, for a handoff from synthesis to place-and-route, the synthesis tool would be in the upstream position occupied by the IP vendor in this flow, and the FPGA vendor would be in the downstream position occupied by the synthesis tool.

© 2016 Synopsys, Inc.

The following figure illustrates the steps in this encryption/decryption methodology, showing the handoff from an IP vendor to a Synopsys FPGA synthesis tool.

The following describes each of the phases shown in the figure. Note that Synopsys provides the following scripts to simplify and automate the process of encrypting data for the IP vendor.

- IEEE 1735-2014
-
- OpenIP

Data Encryption - Step 1

The IP vendor encrypts the IP data using their own symmetric key. This key is called the *data key*. The result of encoding is a *data block*. Using symmetric encryption offers two advantages to the IP vendor: fast data encryption because it is symmetric encryption, and freedom to use any symmetric scheme they choose.

Symmetric Encryption/Decryption with One Key

Data Key Encryption - Step 2

Next, the IP vendor encrypts the data key used to encode the IP block, and generates a *key block*. For this operation, the vendor uses RSA asymmetric encryption and the public key provided by Synopsys.

Asymmetric Encryption/Decryption with Public and Private Keys

Asymmetric encryption offers the following advantages:

- Although asymmetric encryption is compute-intensive, the data key itself is small, so this is not time-intensive.
- The IP vendor can use public keys from different vendors to encrypt the same block for different EDA vendors. This capability ensures that IP consistency is maintained, because there is no need for multiple copies.
- Only the public key from the downstream vendor needs to be passed to the IP vendor.

© 2016 Synopsys, Inc.

Bundling of Encrypted Data Block and Data Key - Step 3

The IP vendor bundles the encrypted data block with the key block into one file for handoff to the EDA vendor. Note that this methodology allows the IP vendor to create just one version of the IP which includes the key blocks for all the downstream vendors it supports; for example, a synthesis tool and a simulation tool. Also, this approach eliminates the need to securely transmit the symmetric key, because this is included in the file. Security is maintained because both the key and the data are encrypted.

In the figure, this is the point at which the IP vendor hands off the IP to the synthesis tool.

Data Key Decryption - Step 4

Decryption is a two-stage process. The first step is to decrypt the symmetric data key from the IP vendor, which was encrypted using the asymmetric public key provided. To decode this key, use the private key counterpart to the public key and extract the data key.

Data Decryption - Step 5

The second step is to use the extracted data key to access the IP data. As the data key is the original symmetric key used to encode the IP, the process is quick. The synthesis tools can now synthesize the unencrypted IP.

After synthesis, the IP can be re-encrypted if the vendor has adopted one of the Synopsys methodologies. See Output Methods for encryptIP, on page 36 in the *Reference Manual* for a description of the choices available.

Re-Encryption in the Synopsys FPGA IP Flow

Re-encryption of the synthesized IP for FPGA vendors downstream requires that the FPGA vendor supply Synopsys with a public key. When the input file includes a downstream key block, the re-encrypted data is accessible to the downstream tool. If such an agreement is not in place, the IP is treated as a black box. Accordingly, you can have an IP flow that outputs black boxes in the netlists, plaintext netlists, or encrypted netlists.

© 2016 Synopsys, Inc.

Working with Encrypted IP

The Synopsys FPGA IP encryption schemes available include:

- IEEE 1735-2014
-
- OpenIP

With either of these approaches, the IP vendor can encrypt and control distribution of the IP from their own website. The synthesis user will have access from the synthesis tool to the IP that the vendor makes available for download and evaluation within a synthesis design.

The following sections describe how to encrypt and package your IP for evaluation if you are an IP vendor, and how to access and evaluate available IP, if you are an end-user.

- Encrypting Your IP, on page 529
- Preparing the IP Package, on page 540

Encrypting Your IP

IP vendors can use either of the supported Synopsys FPGA IP schemes to provide IP for synthesis users to evaluate and use. Both schemes uses a two-stage encryption process:

- First, encrypt your IP files using a symmetric encryption algorithm and your own session or data key to create an encrypted data block.
- Next, encrypt the session key for the encrypted data block using an asymmetric algorithm and the Synopsys public key. The Synplify tool currently supports RSA encryption.

Synopsys provides scripts to simplify this process. See the following procedures for details on script usage.

© 2016 Synopsys, Inc.

Preparing and Encrypting Your IP

To prepare and encrypt your IP, do the following:

1. Gather your RTL files.

   You only encrypt the RTL. You can encrypt any number of Verilog and VHDL (or mixed) RTL files to form your encrypted IP, and each file can be encrypted in its entirety.

2. Determine your file setup for each IP.
   - Create a single set of files for the IP (for use with all supported FPGAs) if your IP has no vendor-specific or vendor-optimized content and if the output method is supported by all intended consumers (blackbox or plaintext).
   - Create multiple versions of your protected IP if you have specific FPGA vendors or specific FPGA vendor families, if you are using FPGA device-family specific RTL like architecture-specific instantiations, or if you optimized your RTL or constraints for use with a specific FPGA vendor device family or FPGA vendor.
   - Encrypt the files with the appropriate encryption script as described in one of the following subsections:*Encrypting IP with the encryptP1735.pl Script*, on page 531
   - *Encrypting IP with the encryptIP Script*, on page 536.

3. Package your IP, as described in *Preparing the IP Package*, on page 540.

4. Verify that your IP works with the synthesis tools by going through the procedure that the user would use.
   - Start the synthesis tool and load the IP with the Import IP->Import IP Package command. You can load your IP into an existing Synplify project.
   - For system-level IP, run it through the System Designer™ tool and ensure bus-model compatibility between your IP and any other IP to which it interfaces. See the System Designer documentation for details on using this tool.
   - Run synthesis.

Encrypting IP with the encryptP1735.pl Script

The encryptP1735.pl script supports the IEEE 1735-2014 standard with limited interoperability. The encryptP1735.pl script accepts inputs from three sources: command line arguments, the RTL input file, and a file containing the public keys. The RTL input file, depending on the use model, may contain encryption attributes.

A keys.txt file, which contains the public key for consumption by Synopsys FPGA tools, is included with the script. Add other public keys to this file when the IP is to be consumed by additional EDA tools.

The following procedure shows you how to encrypt your data with the encryptP1735.pl script. This script automates the two-stage encryption process described in the Synopsys FPGA IP scheme (*The Synopsys FPGA IP Encryption Flow*, on page 524). The encryptP1735.pl script:

- First encrypts your IP files using a symmetric encryption algorithm and your own session or data key to create an encrypted data block.
- Next encrypts the session key for the encrypted data block using an asymmetric algorithm and the Synopsys public key. The tool currently supports RSA encryption.

The encryptP1735.pl script is located in the *installDir*/lib directory and requires the installation of Perl on your machine. The following examples show typical script applications. For more information on the script and the command line arguments, see encryptP1735, on page 38 in the Command Reference Manual. The encryptP1735 encryption script supports the following three use models for encrypting RTL files:

- Full-File Use Model
- Partial File with All Pragmas Use Model
- Partial File with Minimal Pragmas Use Model

Full-File Use Model

With a full-file use model, the RTL contains no encryption-related pragmas, and the entire RTL file is encrypted by encryptP1735.pl to create the decryption envelope. This use model is intended to be used with complete RTL files that do not require the addition of encryption attributes; the encryptP1735.pl script automatically adds these attributes to create the decryption envelope.

© 2016 Synopsys, Inc.

To illustrate the full-file use model, consider a single, Verilog file (tb_encrypt.v) to be encrypted without pragmas. This file contains a single module named secret.

```
module secret (a, b, clk);
input a, clk;
output b;
reg b=0;

always @(posedge clk) begin
   b = a;
end
endmodule
```

With no encryption-related pragmas in the RTL file, a decryption envelope is created with the command:

```
perl encryptP1735.pl -list mylist -log encryptP1735.log
```

In the above command, the list file (mylist) contains the single Verilog file tb_encrypt.v. The command uses the default keys.txt file from the directory *installLocation*/lib as the public keys file to create the decryption envelope file tb_encrypt.vp. The resulting messages are written in the encryptP1735.log file.

Partial File with All Pragmas Use Model

With the partial file with all pragmas use model, the RTL contains all of the encryption-related pragmas which are used by the encryptP1735.pl script with the default keys.txt file to create the decryption envelope.

When there are overlapping pragmas in the RTL and the keys.txt file, the RTL pragma takes precedence over the corresponding pragma in the keys.txt file. For example, if the data_method pragma contains des-cbc in the RTL and aes128-cbc in the keys.txt file, the following pragma is copied to the decryption envelope:

```
data_method="des-cbc"
```

Verilog Example

To illustrate the partial file with all pragmas use model, consider a single, Verilog file (tb_encrypt.v) to be encrypted. This file contains a module named secret and all the encryption-related pragmas with the exception of the key_public_key in the RTL itself.

```
module secret (a, b, clk);
input a, clk;
output b;

`pragma protect version=1
`pragma protect encoding=(enctype="base64")
`pragma protect author="author-a", author_info="author-a-details"
`pragma protect encrypt_agent="encryptP1735.pl", encrypt_agent_info="Synplify encryption scripts"
`pragma protect key_keyowner="Synopsys",key_keyname="SYNP05_001", key_method="rsa", key_block
`pragma protect data_keyowner="ip-vendor-a",data_keyname="fpga-ip", data_method="des-cbc"
`pragma protect begin

reg b=0;

always @(posedge clk) begin
    b = a;
end

`pragma protect end

endmodule
```

With the encryption-related pragmas in the RTL file, a decryption envelope is created with the command:

```
perl encryptP1735.pl -list mylist
```

In the above command, the list file (mylist) contains the single Verilog file tb_encrypt.v. The command uses the default keys.txt file from the directory *installLocation*/lib as the public keys file to create the decryption envelope file tb_encrypt.vp. Any messages from the run are not output to a log file.

VHDL Example

To encrypt a partial VHDL file with an all pragmas use model, consider the single, VHDL file (tb_encrypt.vhd). The file contains a single entity/architecture pair named secret with all the encryption-related pragmas with the exception of the key_public_key in the RTL itself.

Note: VHDL formatted pragmas do not include a “pragma” string preceding the keyword “protect.”

```
library IEEE;
use IEEE.std_logic_1164.all;

entity secret is
    port (clk : in std_logic;
            a : in std_logic;
            b : out std_logic);
end entity;

architecture rtl of secret is

`protect version=1
`protect author="author-a", author_info="author-a-details"
`protect encrypt_agent="encryptP1735.pl", encrypt_agent_info="Synplify encryption scripts"
`protect encoding=(enctype="base64")
`protect key_keyowner="Synopsys", key_keyname="SYNP05_001", key_method="rsa", key_block
```


© 2016 Synopsys, Inc.

```
`protect data_keyowner="ip-vendor-a", data_keyname="fpga-ip", data_method="des-cbc"
`protect begin

signal b_reg: std_logic;
begin
    process (clk) is
    begin
        if rising_edge(clk) then
            b_reg <= a;
        end if;
    end process;
    b <= b_reg;

`protect end

end architecture;
```

With the encryption-related pragmas in the RTL file, a decryption envelope is created with the command:

```
perl encryptP1735.pl -list mylist
```

In the above command, the list file (mylist) contains the single VHDL file tb_encrypt.vhd. The command uses the default keys.txt file from the directory *installLocation*/lib as the public keys file to create the decryption envelope file tb_encrypt.vhdp. Any messages from the run are not output to a log file.

Partial File with Minimal Pragmas Use Model

With the partial file with minimal pragmas use model, the RTL contains only the begin and end encryption-related pragmas to indicate the start and end points of the encryption. This use model is helpful when automatic insertion is used to reduce the amount of manually inserted pragmas. After defining the start- and end-point pragmas, the encryptP1735.pl script, using the default keys.txt file, creates the decryption envelope. To illustrate this use model, consider a single, Verilog file (tb_encrypt.v) to be encrypted with only begin and end pragmas. This file contains a single module named secret.

```
module secret (a, b, clk);
input a, clk;
output b;
`pragma protect begin
reg b=0;

always @(posedge clk) begin
   b = a;
end
`pragma protect end
endmodule
```

With the start and end pragmas in the RTL file, a decryption envelope is created with the command:

```
perl encryptP1735.pl -list mylist
```

In the above command, the list file (mylist) contains the single Verilog file tb_encrypt.v. The command uses the default keys.txt file from the directory *installLocation*/lib as the public keys file to create the decryption envelope file tb_encrypt.vp. The absence of a specified log file (-log option) results in no messages being written to the log file.

Encrypting Multiple RTL Files

The examples in the previous sections included a single Verilog or VHDL file. Multiple files can be similarly encrypted by adding their file names to the specified list file. Running the script produces encryption envelopes for each listed file. To create a list file for multiple files, add the name of each file to be encrypted on a separate line.

Public Keys Repository File

The encryptP1735.pl encryption script requires public keys from the file specified by the -public_keys (or -pk) option. This file includes public keys for each of the tools that require a key block in the encrypted file.

```
// Use verilog pragma syntax in this file

`pragma protect version=1
`pragma protect author="default"
`pragma protect author_info="default"

`pragma protect key_keyowner="Synopsys", key_keyname="SYNP05_001", key_method="rsa"
`pragma protect key_public_key

MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAybsQaMidiCHZyh14wbXn
UpP8lK+jJY5oLpGqDfSW5PMXBVp0WFd1d32onXEpRkwxEJLlK4RgS43d0FG2ZQ1l
irdimRKNnUtPxsrJzbMr74MQkwmG/X7SEe/lEqwK9Uk77cMEncLycI5yX4f/K9Q9
WS5nLD+Nh6BL7kwR0vSevfePC1fkOa1uC7b7Mwb1mcqCLBBRP9/eF0wUIoxVRzjA
+pJvORwhYtZEhnwvTblBJsnyneT1LfDi/D5WZoikTP/0KBiP87QHMSuVBydMA7J7
g6sxKB92hx2Dpv1ojds1Y5ywjxFxOAA93nFjmLsJq3i/P0lv5TmtnCYX3Wkryw4B
eQIDAQAB

// Add additional public keys below this line

// Add additional public keys above this line

`pragma protect data_keyowner="default-ip-vendor"
`pragma protect data_keyname="default-ip-key"
`pragma protect data_method="aes128-cbc"

// End of file
```


© 2016 Synopsys, Inc.

For the partial file with all pragmas use model, the following pragma attribute values must match the corresponding values in the key-block section of the encryption envelope:

```
`pragma protect key_keyowner="Synopsys", key_keyname="SYNP05_001", key_method="rsa"
```

Note: This encryptP1735.pl script is compatible only with versions I-2013.09 and later of the Synopsys FPGA synthesis tool.

For information on the pragmas supported by the encryptP1735.pl script, see Pragmas Used in the encryptIP Script, on page 381 of the Reference Manual.

Encrypting IP with the encryptIP Script

The following procedure shows you how to encrypt your data with the encryptIP (OpenIP) script. The encryptIP script automates the two-stage encryption process proposed in the Synopsys FPGA IP scheme (*The Synopsys FPGA IP Encryption Flow*, on page 524).

- First, it encrypts your IP files using a symmetric encryption algorithm and your own session or data key. This creates an encrypted data block.
- Next, it encrypts the session key for the encrypted data block using an asymmetric algorithm and the Synopsys public key. The Synplify tool currently supports RSA encryption.

1. Install the encryptIP Perl script.
   - Download the encryptIP Perl script.
   - Install Perl on your machine. You cannot run the script if you do not have Perl installed.
2. Make sure that the encryptIP script specifies the decryption key and the matching key length:
   - Specify the symmetric data decryption key with the -k option. Optionally, you can also specify a symmetric encryption key in hexadecimal format with the -kx option.
   - Make sure you specify the right key length for the encryption algorithm with the -c option. For example, TEST1234 becomes a 64-bit key, so you specify the des-cbc algorithm.

See Syntax, on page 34 in the *Reference Manual* for full details of the encryptip syntax.

3. Make sure you specify the appropriate output method (-om) when you run the script.

   This is important because the output method (-om) determines what is encrypted to the user. When the example above is synthesized, the user can view the output netlist because the output method specified is plain-text, which means that the synthesis output netlist includes the IP netlist in an unencrypted and readable form. See *Specifying the Script Output Method*, on page 537 for more information.

   The script encrypts the IP with the standard symmetric encryption algorithm you specified, and produces a data_block. The data key used for encrypting the HDL is then encrypted with an asymmetric algorithm and the Synopsys public key, and produces a key_block. The data_block and the key_block are combined with the appropriate pragmas for the flow being used, and the script creates an encrypted HDL file. For a detailed figure, see *Encryption and Decryption*, on page 525.

   All other output files from synthesis, including `srm`, `srd`, and `srs` files, are encrypted using the same encryption method specified for the input to synthesis. Output constraints are not encrypted.

4. Run the encryptIP script on each RTL file you want to encrypt.

   The following example encrypts the Verilog plain_ip.v file into an encrypted file called protected_ip.v, using AES128-cbc encryption. The session key is MY_AES_SAMPLEKEY. See Syntax, on page 34 in the *Reference Manual* for details about the syntax and required parameters.

```
perl encryptIP -in plain_ip.v -out protected_ip.v -c aes128-cbc
   -k MY_AES_SAMPLEKEY –bd 16OCT2007 -om plaintext -v
```

5. Check the encrypted RTL file to make sure that there is only one key block present.

Specifying the Script Output Method

You can control access to the IP by setting the appropriate output method. You specify the output method using the -om parameter, as described in Syntax, on page 34 *or* Syntax, on page 39in the *Reference Manual*.

© 2016 Synopsys, Inc.

The output method mainly affects the output netlist. The following are guidelines for setting the output method for the encryptIP script, and detail the effects of different settings:

1. When using the encrypyIP script, set -om to persistent_key in the following cases:
   - If you are working with a Lattice non-CPLD technology
   - If you have an agreement in place with Synopsys and want the output netlist to be encrypted
2. Set -om to plaintext in the following cases:
   - If you want to allow the IP to be incorporated in a logic synthesis design

     Setting the output method to plaintext allows the tool to synthesize, run gate-level simulations, place and route, and implement an FPGA (that includes the IP) on a board.
   - If you want the IP to be freely optimized by the synthesis tools

     Although IP cores are already optimized, the synthesis tools can effect additional optimizations based on the design context in which it will be used. When the synthesis tool is allowed to optimize the IP, it can prune away IP logic that is unused or unnecessary in the current design context. Or take the case where the output of an instantiated IP core is timing-critical because it drives hundreds of user loads. If the synthesis tool can freely optimize, it can replicate sources within the core and fix the problem.
3. To let the IP be incorporated in a logic synthesis design, set -om to plaintext or blackbox.

   Setting the output method to plaintext allows the tool to synthesize, run gate-level simulations, place and route, and implement an FPGA (that includes the IP) on a board. Setting the output method to blackbox does not allow the tool to run gate-level simulations or place and route the IP, because it only uses the port and connectivity information.
4. If you have set -om to plaintext and you want to specify individual cores as white boxes, set the syn_macro directive to 1 on the view for the IP.

   Note that you must set this on the view, not the instance. When this is set, the tool treats the IP as a white box and only uses the timing and

connection information from the IP. The synthesis tool maintains the IP boundary and only trims unused logic inside the IP.

5. During synthesis, the IP contents appear as a black box in the RTL and Physical Analyst views, irrespective of the output method selected. When the output method is set to plaintext, you can push down into the IP from the Technology view.

6. After synthesis, the output method affects the results in the following ways:
   - Output constraints for an IP are in the standard Synopsys format and are not encrypted.
   - The output method affects the contents of the output netlist and its format. This table summarizes the encryptIP or encryptP1735 behavior with different output methods.

Method (-om)	Output Netlist After Synthesis
blackbox	The output netlist contains the IP interface only, and no IP contents. It only includes IP ports and connections. The IPs are treated as black boxes, and there are no nets or instances shown inside the IP. This applies to all the netlist formats generated for different vendors, whether it is HDL (`vm` or `vhm`), EDIF (`edf` or `edn`), or `vqm`.
plaintext	The output netlist contains your unencrypted synthesized IP, which is completely readable (nothing is encrypted).
persistent_key (encryptIP only)	The output netlist includes encrypted versions of the IP. For Lattice designs, the tool generates one output netlist (EDIF or HDL) that includes the encrypted IP blocks. The netlist is readable, except for the IP block sections, which are encrypted.

© 2016 Synopsys, Inc.

Preparing the IP Package

Do the following to package your IP and make it accessible from the synthesis tools:

1. Collect the files for the package.
   - Encrypt the files you need, as described in *Encrypting Your IP*, on page 529.
   - Make sure your package includes the files listed in *IP Package File List*, on page 541.
   - Structure the files as described in *Suggested Directory Structure*, on page 541.
2. If your IP package is intended for synthesis only, without subsystem assembly, create a compressed package for download, using one of these methods:
   - Create a compressed tarball (.tar.gz), which is a tar archive compressed with the gzip utility, using one of these commands:

     ```
     tar cf -fileList | gzip -c > compressed-tarball
     ```

     ```
     gtar -cf compressed-tarball fileList
     ```

     Preserve the directory structure when you run gzip.
   - Create a zip file (`zip`) by running WinZip. WinZip archives and preserves your directory hierarchy.
3. Post the packaged IP on your website for downloading.

   The user generally untars or unzips the IP package into a top-level directory after downloading it. The synthesis tools can then read the contents of the directory.
4. Supply Synopsys with the following:
   - The URL for the download package.
   - Vendor and advertising information you wish to display on the Synopsys website. See *Supplying Vendor Information*, on page 541 for details.

IP Package File List

Your IP package should contain the following files:

Files	Description
ipinfo.txt	Text file that lists the name of the IP, the version, restrictions for use, support contact information, and an email alias to request a licence for the full RTL for your IP.
Documentation, preferably a PDF	Documents the IP, and includes detailed information about usage restrictions like vendor, device family, etc.
Readme	An optional text file that contains instructions on use of the IP for assembly and/or synthesis, and hints on how to use it correctly.
Encrypted HDL or EDIF	Protected RTL for the IP, created using the Synopsys encryptIP script. See the documentation for details.
SDC constraints	Unencrypted design constraints for the IP.

Suggested Directory Structure

Follow these recommendations when you structure the IP package:

- If you include a Synopsys `prj` project file or an `xml` file for use in System Designer, make sure to use relative paths from the `prj` or `xml` directory to refer to other files like the Verilog or VHDL files.
- Always use relative paths to reference a file.
- Always preserve directory structure when you run gzip.
- You can place IP-XACT `xml` files in the top-level directory or in a common subdirectory. You can have multiple files or a single file for the same component or variants of a component. However, it is preferred that you keep all IP-XACT components that are in one library at the same directory level, even if it is many levels deep in the directory hierarchy.

Supplying Vendor Information

To make your IP accessible for downloads and evaluation from the Synopsys synthesis tools, you must supply Synopsys with some vendor information as well as information for each of the cores or IPs to be used.

© 2016 Synopsys, Inc.

1. Supply Synopsys with the following general information to advertise your company and IP on the Synopsys website:

IP vendor name and logo	Your vendor name and logo for display.
Optional IP description	Short paragraph describing the IP and key features.
Email alias	Synopsys sends leads to this alias when evaluation cores are requested on the Synopsys IP website.
Website URL	Unique URL for accessing IP. After the user has filled out lead information on the website, the Synopsys tool directs the user to this URL to download the IP. The lead form on your website can be pre-filled by prior arrangement with Synopsys Marketing.

2. Supply Synopsys with the following information about each core or IP to be used:

IP name	Name of the IP.
IP short description	Sentence describing the IP, which is displayed in the summary view on the Synopsys website.
IP paragraph description	More detailed description of the IP, covering functional description and compatibility with other cores or peripherals.
Notes about usage	Any other information, like licensing requirements
Core datasheet (HTML or PDF)	Information about the characteristics, features, functions, and interfaces.
Supported FPGA vendors and devices	List of the targeted vendors and devices that the core supports.
IP-XACT compatibility information	List of the IP-XACT version number supported, the IP-XACT VLNV, and the IP-XACT VLNVs of all the bus definitions required for the core, along with a link to download each of these bus definitions.

© 2016 Synopsys, Inc.

Using Hyper Source

Hyper source is a useful feature that lets you prototype ASIC designs that use one or more FPGAs. You can also use it to validate and debug the RTL for IP designs. See the following for details:

- Using Hyper Source for Prototyping, on page 543
- Using Hyper Source for IP Designs, on page 543
- Threading Signals Through the Design Hierarchy of an IP, on page 544

Using Hyper Source for Prototyping

For prototyping, use Hyper Source to efficiently thread nets across multiple modules to the top-level design to support Time Domain Multiplexing (TDM).

You can also use it to easily replace an ASIC RAM with an FPGA RAM. Follow these guidelines to replace an ASIC RAM with an FPGA RAM:

1. Change the RTL for the RAM instantiation.
2. Add an extra clock signal to all the module interfaces.

   Hyper source reduces the number of modified RTL modules to two, one for the RAM and one for the top level.

Using Hyper Source for IP Designs

For IP designs, Hyper Source is useful for validating and debugging the RTL without directly modifying it. After the RTL has been fully tested with complete QoR results, use Hyper Source to debug, as described in the following cases:

- Add some instrumentation logic that is not part of the original design, such as a cache profiler that counts cache misses or bus monitor that might count statistics about bus contention. The cache or bus might be buried deep inside the RTL; accessing the cache or the bus means ports might need to be added through several levels of hierarchy in the RTL. The instrumentation logic can be included anywhere in the design, so you can use hyper source and hyper connect to easily thread the necessary connections during synthesis.

© 2016 Synopsys, Inc.

- Insert other hyper sourcing inside the IP to probe, monitor, and verify correct operation of known signals within the IP.

Threading Signals Through the Design Hierarchy of an IP

Use this mechanism to thread a signal through the design hierarchy of a user IP. This signal can be threaded to a top-level port or signal. This works even if the Verilog or VHDL is compiled separately. The tool automatically adds ports and signals between the source and the connection. Otherwise, these connections must be manually added to the RTL code.

The following procedure describes a method for using hyper source, using the example HDL shown in *Hyper Source Example*, on page 545.

1. Define how to connect to the signal source. The following apply to this example:
   - Signal syn_hyper_source (in1) module defines the source, with a width of 1.
   - The tag name "tag_name" is the global name for the hyper source.
2. Define how to access the hyper source which drives the local signal or port. The following apply to this example:
   - Signal syn_hyper_connect (out1) module defines the connection. The signal width of 1 matches the source.
   - Tag name can be the global name or the instance path to the hyper source.
3. In this hierarchical design, note the following about hyper source:
   - Applies to the module lower_module.
   - Signal syn_hyper_source my_source(din) module is defined for the source with a width of 8.
   - The tag name of "probe_sig" must match the name used in the hyper connect block to thread the signal properly.
4. In this hierarchical design, note the following about the hyper connect:
   - Applies to the top-level module `top`, but can be any level of hierarchy.
   - Signal syn_hyper_connect connect_block (probe) module is defined for the connection with a width of 8.

– Tag name of "probe_sig" must match the name used in the hyper source block to thread the signal properly.

5. After you run synthesis, the following message appears in the log file:

```
Available hyper_sources - for debug and ip models
HyperSrc label sub2_module.sub1_module.lower_module.probe_sig

Making connections to hyper_source modules
@N: : hyper_example.v(54) | Connected syn_hyper_connect hstdm_training_done_connect, label probe_sig
Finished RTL optimizations (Time elapsed 0h:00m:01s; Memory used current: 122MB peak: 129MB)
```

Hyper Source Example

```
/* Connect to a signal you want to export example : in1*/
module syn_hyper_source(in1) /*synthesis syn_black_box=1 syn_noprune=1 */;
parameter w = 1;
parameter tag = "tag_name"; /* global name of hyper_source */
input [w-1:0] in1;
endmodule

/* Use to access hyper_source and drive a local signal or port example
:out1 */
module syn_hyper_connect(out1) /* synthesis syn_black_box=1 syn_noprune=1
*/;
parameter w = 1; /* width must match source */
parameter tag = "tag_name"; /* global name or instance path to hyper_source
*/
parameter dflt = 0;
parameter mustconnect = 1'b1;
output [w-1:0] out1;
endmodule

/* Example hierarchical design which uses hyper_source */
module lower_module (clk, dout, din1, din2, we);
output reg [7:0] dout;
input clk, we;
input [7:0] din1, din2;
wire [7:0] din;

syn_hyper_source my_source(din);
defparam my_source.tag = "probe_sig"; /* to thread the signal this
tag_name must match to name used in the hyper connect block */
defparam my_source.w = 8;

always @(posedge clk)
```


© 2016 Synopsys, Inc.

```
if (we)
   dout <= din;
assign din = din1 & din2;
endmodule

module sub1_module (clk, dout, din1, din2, we);
output[7:0] dout;
input clk, we;
input [7:0] din1, din2;
lower_module lower_module (clk, dout, din1, din2, we);
endmodule

module sub2_module (clk, dout, din1, din2, we);
output [7:0] dout;
input clk, we;
input [7:0] din1, din2;
sub1_module sub1_module (clk, dout, din1, din2, we);
endmodule

module top (clk, dout, din1, din2, we, probe);
output[7:0] dout;
output [7:0] probe;
input clk, we;
input [7:0] din1, din2;

syn_hyper_connect connect_block(probe);
defparam connect_block.tag = "probe_sig"; /* to thread the signal this
tag_name must match to name used in the hyper connect block */
defparam connect_block.w = 8;

sub2_module sub2_module (clk, dout, din1, din2, we);

endmodule
```

The following figures show how the hyper source signal automatically gets connected through the hierarchy of the IP in the HDL Analyst views.

RTL View

Technology View

© 2016 Synopsys, Inc.

Working with Lattice IP

The Lattice IP encryption scheme uses the flow described in *Encryption and Decryption*, on page 525. The following procedure details how to incorporate Lattice encrypted IP in your design.

1. Create a synthesis project.
2. Obtain the IP from Lattice, and add the encrypted file to your project.

   The Lattice IP is encrypted using the persistent_key method described in *Specifying the Script Output Method*, on page 537.

3. Synthesize your design.

   The tool automatically decrypts the protected IP and synthesizes it with the other unencrypted files in the design. During synthesis, the IP is treated as a black box and you cannot view its contents in HDL Analyst views.

   After synthesis, the tool generates one output netlist. The IP is re-encrypted and included in this netlist. The rest of the netlist is not encrypted and can be read. The IP is not included in any simulation netlists that are written out; it is treated as a black box.

   See *Encryption and Decryption*, on page 525 for a flow diagram of this process.

CHAPTER 13

Optimizing Processes for Productivity

This chapter covers topics that can help the advanced user improve productivity and interoperability with other tools. It includes the following:

- Using Batch Mode, on page 550
- Working with Tcl Scripts and Commands, on page 556
- Automating Flows with synhooks.tcl, on page 563

© 2016 Synopsys, Inc.

Using Batch Mode

Batch mode is a command-line mode in which you run scripts from the command line. You might want to set up multiple synthesis runs with a batch script. You can run in batch mode if you have a floating license, but not with a node-locked license.

Batch scripts are in Tcl format. For more information about Tcl syntax and commands, see *Working with Tcl Scripts and Commands*, on page 556.

This section describes the following operations:

- Running Batch Mode on a Project File, on page 550
- Running Batch Mode with a Tcl Script, on page 552
- Queuing Licenses, on page 553

Running Batch Mode on a Project File

Use this procedure to run batch mode if you already have a project file set up. You can also run batch mode from a Tcl script, as described in *Running Batch Mode with a Tcl Script*, on page 552.

1. Make sure you have a project file (`prj`) set up with the implementation options. For more information about creating this Tcl file, see *Creating a Tcl Synthesis Script*, on page 558.

2. From a command prompt, go to the directory where the project files are located, and type one of the following, depending on which product you are using:

   ```
   synplify_pro -batch project_file_name.prj
   ```

   The software runs synthesis in batch mode. Use absolute path names or a variable instead of a relative path name.

   If the -tclcmd switch is used, synthesis will not automatically run. To make synthesis run, project -run must be added:

   ```
   synplify_pro -batch myproj.prj -tclcmd "project -run"
   ```

The -tclcmd switch specifies the tcl commands to be executed before the synthesis starts. To run a constraint check before synthesis:

```
synplify_pro -batch myproj.prj -tclcmd "project -run constraint_-
check"
```

The -tclcmd switch also allows the synthesis results path to be changed.

```
synplify_pro -batch "D:/tests/myproj.prj"  -tclcmd "set_option
-result_file \"./impl1/test.edif\"; project -run"
```

The software returns the following codes after the batch run:

0 - OK
2 - logical error
3 - startup failure
4 - licensing failure
5 - batch not available
6 - duplicate-user error
7 - project-load error
8 - command-line error
9 - Tcl-script error
20 - graphic-resource error
21 - Tcl-initialization error
22 - job-configuration error
23 - parts error
24 - product-configuration error
25 - multiple top levels

3. If there are errors in the source files, check the standard output for messages. On Linux systems, this is generally the monitor; on Windows systems, it is the stdout.log file.

4. After synthesis, check the *resultFile*.srr log file for error messages about the run.

© 2016 Synopsys, Inc.

Running Batch Mode with a Tcl Script

The following procedure shows you how to create a Tcl batch script for running synthesis. If you already have a project file set up, use the procedure described in *Running Batch Mode on a Project File*, on page 550.

1. Create a Tcl batch script. See *Creating a Tcl Synthesis Script*, on page 558 for details.

2. Save the file with a `tcl` extension to the directory that contains your source files and other project files.

3. From a command prompt, go to the directory with the files and type one of the following as appropriate:

   `synplify_pro -batch` *`Tcl_script`*`.tcl`

   The software runs synthesis in batch mode. The synthesis (compilation and mapping) status results and errors are written to the log file *resultFile*.srr for each implementation. The synthesis tool also reports success and failure return codes.

4. Check for errors.
   - For source file or Tcl script errors, check the standard output for messages. On Linux systems, this is generally the monitor in addition to the stdout.log file; on Windows systems, it is the stdout.log file.
   - For synthesis run errors, check the *resultFile*.srr log file. The software uses the following error codes:

     0 - OK
     2 - logical error
     3 - startup failure
     4 - licensing failure
     5 - batch not available
     6 - duplicate-user error
     7 - project-load error
     8 - command-line error
     9 - Tcl-script error
     20 - graphic-resource error
     21 - Tcl-initialization error
     22 - job-configuration error
     23 - parts error
     24 - product-configuration error
     25 - multiple top levels

Queuing Licenses

A common problem when running in batch mode is that the run fails because all of the available licenses are in use. License queuing allows a batch run to wait for the next available license when a license is on the server but not immediately available.

You can specify either blocking or non-blocking queuing. With blocking-style queuing, the tool waits until a license becomes available; with non-blocking-style queuing, the tool waits the specified length of time for a license to become available.

You can also queue DesignWare IP licenses, so that they can be used as they become available.

For details, see the following:

- Queuing Considerations, on page 553
- Queuing Licenses, on page 553

Queuing Considerations

Consider these points when using queuing:

- A blocking-style queuing is used; license checkout does not exit until a license becomes available.
- There is no maximum wait time; once initiated, the tool can wait indefinitely for a license.
- If the server shuts down while the tool is waiting, a checkout failure is reported.
- When two licenses are required, queuing waits only until the first license becomes available (and not the second) to avoid holding a license unnecessarily.

Queuing Licenses

The following procedure describes how to specify blocking-style or non-blocking style queuing for synthesis licenses. You can specify the licensed features for queuing in an environment variable or directly in batch mode.

© 2016 Synopsys, Inc.

1. Specify the list of licensed features you want to queue, using either of the following methods:
   - Set the *toolName*_LICENSE_TYPE environment variable to the features you want. For example:

     ```
     SYNPLIFYPRO_LICENSE_TYPE=synplifypro:synplifypro_lattice
     ```

   - Specify a list of features to wait for using the -batch, -licensetype and -license_wait options. For example:

     ```
     synplify_pro -batch -license_wait -licensetype
     synplifypro:synplifypro_lattice myProject.prj
     ```

     See synplify_pro, on page 98 in the *Command Reference* for syntax details.

2. To enable blocking-style queuing, do one of the following:
   - Set environment variable *toolName*_LICENSE_WAIT=1 (*toolName* is the name of the FPGA synthesis tool).
   - In batch mode, include a -license_wait command-line argument, as shown in the following examples:

     ```
     synplify_pro -batch -license_wait Tcl_script.tcl
     ```

   With blocking-style queuing enabled, the tool waits until the requested license becomes available. It generates the following message in the stdout.log or the Tcl window:

   ```
   Waiting for license: toolName
   ```

3. To enable non-blocking-style queuing, do either of the following:
   - Set environment variable *toolName*_LICENSE_WAIT=*waitTime* (*toolName* is the name of the FPGA synthesis tool and *waitTime* is the maximum wait time in seconds). For example:

     ```
     SYNPLIFYPRO_LICENSE_WAIT=180
     ```

     The *waitTime* value determines the maximum wait time, in seconds:

***WaitTime* Value**	**Queuing Behavior**
Undefined or 0	Queuing off
1	Queuing on; wait indefinitely
>1	Queuing on; wait up to the specified number of seconds

– Include a -license_wait *waitTime* command-line argument when launching batch mode as shown in the following examples:

```
synplify_pro -batch -license_wait waitTime Tcl_script.tcl
```

When non-blocking-style queuing is enabled, the tool waits up to the maximum time limit specified for the license to become available. The tool generates the following message in stdout.log or the Tcl window:

```
Waiting up to n seconds for license: toolName
```


© 2016 Synopsys, Inc.

Working with Tcl Scripts and Commands

The software uses extensions to the popular Tcl (Tool Command Language) scripting language to control synthesis and for constraint files. See the following for more information:

- Using Tcl Commands and Scripts, next
- Generating a Job Script, on page 557
- Setting Number of Parallel Jobs, on page 557
- Creating a Tcl Synthesis Script, on page 558
- Using Tcl Variables to Try Different Clock Frequencies, on page 560
- Using Tcl Variables to Try Several Target Technologies, on page 561
- Running Bottom-up Synthesis with a Script, on page 562

You can also use synhooks Tcl scripts, as described in *Automating Flows with synhooks.tcl*, on page 563.

Using Tcl Commands and Scripts

1. To get help on Tcl syntax, do any of the following:
   - Refer to the online help (Help->Tcl Help) for general information about Tcl syntax.
   - Refer to the *Command Reference Manual* for information about the synthesis commands.
   - Enter help * in the Tcl window for a list of all the Tcl synthesis commands.
   - Enter help *commandName* in the Tcl window to see the syntax for an individual command.
2. To run a Tcl script, do the following:
   - Create a Tcl script. Refer to *Generating a Job Script*, on page 557 and *Creating a Tcl Synthesis Script*, on page 558.
   - Run the Tcl script by either entering source *Tcl_scriptfile* in the Tcl script window, or by selecting File->Run Tcl Script, selecting the Tcl file, and clicking Open.

The software runs the selected script by executing each command in sequence. For more information about Tcl scripts, refer to the following sections.

Generating a Job Script

You can record Tcl commands from the interface and use it to generate job scripts.

1. In the Tcl script window, enter recording -file *logfile* to write out a Tcl log file.
2. Work through a synthesis session.

   The software saves the commands from this session into a Tcl file that you can use as a job script or as a starting point for creating other Tcl files.

For the command syntax, see recording, on page 65.

Setting Number of Parallel Jobs

You can set the maximum number of parallel jobs by setting a variable in the `ini` file, by defining a Tcl variable or specifying the maximum number in the GUI.

1. To set the maximum number of parallel jobs in the `ini` file, do the following:
   - Open the `ini` file for the synthesis tool. For example, synplify_pro.ini.
   - Add the MaxParallelJobs variable to the `ini` file, as follows:

     ```
     [JobSetting]
     MaxParallelJobs=<n>
     ```

   The tool uses the MaxParallelJobs value from the `ini` file as the default for both the UI (Project->Options) and batch mode. This value remains in effect until you reset it in the `ini` file or from the GUI, as described in the next step. To locate this configuration and initialization file (`ini`), see Input Files, on page 252.

© 2016 Synopsys, Inc.

2. To set or change the maximum number of parallel jobs from the GUI, do the following:
   - Select Project->Options->Configure Compile Point Process.
   - Set the value you want in the Maximum number of parallel synthesis jobs field, and click OK. This field shows the current `ini` value, but you can reset it, and it will remain in effect until you change it again. The value you set is saved to the `ini` file.
3. To set a Tcl variable for the maximum number of parallel jobs, do the following:
   - Determine where you are going to define the variable. You can do this in the project file, or a Tcl file, or you can type it in the Tcl window. If you specify it in a Tcl file, you must source the file. If you specify it in the Tcl window, the tool does not save the value and it will be lost when you end the current session.
   - Specify the max_parallel_jobs variable with the set_option Tcl command:

     ```
     set_option -max_parallel_jobs value
     ```

   The tool applies the max_parallel_jobs *value* specified to all project files and their respective implementations. This is a global option. The maximum number of parallel jobs remains in effect until you specify a new value. This new value takes effect immediately, going forward. However, when you set this option from the Tcl command window, the max_parallel_jobs value is not saved and will be lost when you exit the application.

Creating a Tcl Synthesis Script

Tcl scripts are text files with a `tcl` extension. You can use the graphic user interface to help you create a Tcl script. Interactive commands that you use actually execute Tcl commands, which are displayed in the Tcl window as they are run. You can copy the command text and paste it into a text file that you build to run as a Tcl script. For example:

```
add_file prep2.v
set_option -technology ECP3
set_option -part MG328C
set_option -package MG328C

project -run
```

The following procedure covers general guidelines for creating a synthesis script.

1. Use a text file editor or select File->New, click the Tcl Script option, and type a name for your Tcl script.

2. Start the script by specifying the project with the project -new command. For an existing project, use project -load *project*.prj.

3. Add files using the add_file command. The files are added to their appropriate directories based on their file name extensions (see add_file, on page 20). Make sure the top-level file is last in the file list:

   ```
   add_file statemach.vhd
   add_file rotate.vhd
   add_file memory.vhd
   add_file top_level.vhd
   add_file design.fdc
   ```

   For information on constraints and vendor-specific attributes, see *Using a Text Editor for Constraint Files (Legacy)*, on page 63 for details about constraint files.

4. Set the design synthesis controls and the output:
   - Use the set_option command for setting implementation options and vendor-specific controls as needed. See the appropriate vendor chapter in the *Command Reference Manual* for details.
   - Set the output file information with project -result_file and project -log_file.

5. Set the file and run options:
   - Save the project with a project -save command
   - Run the project with a project -run command
   - Open the RTL and Technology views:

     ```
     open_file -rtl_view
     open_file -technology_view
     ```

6. Check the syntax.
   - Check case (Tcl commands are case-sensitive).
   - Start all comments with a hash mark (#).
   - Always use a forward slash (/) in directory and pathnames, even on the Windows platform.

© 2016 Synopsys, Inc.

Using Tcl Variables to Try Different Clock Frequencies

To create a single script for multiple synthesis runs with different clock frequencies, you need to create a Tcl variable for the different settings you want to try. For example, you might want to try different target technologies.

1. To create a variable, use this syntax:

```
set variable_name {
   first_option_to_try
   second_option_to_try
   ...}
```

2. Create a foreach loop that runs through each option in the list, using the appropriate Tcl commands. The following example shows a variable set up to synthesize a design with different frequencies. It also creates a separate log file for each run.

The following code shows the complete script:

```
project -load design.prj
set try_these {
   20.0
   24.0
   28.0
   32.0
   36.0
   40.0
}
```

```
foreach frequency $try_these {
   set_option -frequency $frequency
   project -log_file $frequency.srr
   project -run
   open_file -edit_file $frequency.srr
}
```

Using Tcl Variables to Try Several Target Technologies

This technique used here to run multiple synthesis implementations with different target technologies is similar to the one described in *Using Tcl Variables to Try Different Clock Frequencies*, on page 560. As in that section, you use a variable to define the target technologies you want to try.

1. Create a variable called try_these with a list of the technologies.

```
set try_these {

      ECP2 ECP3 # list of technologies
}
```

2. Add a foreach loop that creates a new implementation for each technology and opens the RTL view for each implementation.

```
foreach technology $try_these {
   impl -add
   set_option -technology $technology
   project -run -fg
   open_file -rtl_view
}
```


© 2016 Synopsys, Inc.

The following code example shows the script:

```
# Open a new project, set frequency, and add files.
   project -new
   set_option -frequency 33.3
   add_file -verilog D:/test/simpletest/prep2_2.v

# Create the Tcl variable to try different target technologies.
   set try_these
      ECP2 ECP3 # list of technologies
   }

# Loop through synthesis for each target technology.
   foreach technology $try_these {
      impl -add
      set_option -technology $technology
      project -run -fg
      open_file -rtl_view
   }
```

Running Bottom-up Synthesis with a Script

To run bottom-up synthesis, you create Tcl scripts for individual logic blocks, and a script for the top level that reads the other Tcl scripts.

1. Create a Tcl script for each logic block. The Tcl script must synthesize the block. See *Creating a Tcl Synthesis Script*, on page 558 for details.
2. Create a top-level script that reads the block scripts. Create the script with the with the project -new command.
3. Add the top-level data:
   - Add source and constraint files with the add_file command.
   - Set the top-level options with the set_option command.
   - Set the output file information with project -result_file and project -log_file.
   - Save the project with a project -save command.
   - Run the project with a project -run command.
4. Save the top-level script, and then run it using this syntax:

   source *block_script*.**tcl**

When you run this command, the entire design is synthesized, beginning with the lower-level logic blocks specified in the sourced files, and then the top level.

Automating Flows with synhooks.tcl

This procedure provides the advanced user with callbacks that let you customize your design flow or integrate with other products. For example, you might use the callbacks to send yourself email when a job is done (see *Automating Message Filtering with a Tcl Script*, on page 232), or to automatically copy files to another location after mapping. You can use the callback functions to integrate with a version control system, or generate the files needed to run verification. The procedure is based on a file called synhooks.tcl, which contains the Tcl callbacks.

1. Copy the synhooks.tcl file from the *installDirectory*/examples directory to a new location.

   You must copy the file to a new location so that it does not get overwritten by subsequent product installations and you can maintain your customizations from version to version. For example, copy it to C:/work/synhooks.tcl.

2. Define an environment variable called SYN_TCL_HOOKS, and point it to the absolute path location of the synhooks.tcl file. For example:

   $SYN_TCL_HOOKS=/*remote*/*rel*/*projects*/*MyProj*/synhooks.tcl

3. Open the synhooks.tcl file in a text editor, and edit the file so that the commands reflect what you want to do. The default file contains examples of the callbacks, which provide you with hooks at various points of the design process.

   – Customize the file by deleting the ones you do not need and by adding your customized code to the callbacks you want to use. The following table summarizes the various design phases where you can use the callbacks and lists the corresponding functions. For details of the syntax, refer to Tcl Script Examples, on page 452 in the *Reference Manual*.

© 2016 Synopsys, Inc.

Design Phase	Tcl Callback Function
Project Setup Callbacks	
Settings defaults for projects	proc syn_on_set_project_template
Creating projects	proc syn_on_new_project
Opening projects	proc syn_on_open_project
Closing projects	proc syn_on_close_project
Application Callbacks	
Starting the application after opening a project	proc syn_on_start_application
Exiting the application	proc syn_on_exit_application
Run Callbacks	
Starting a run. See *Example: proc syn_on_start_run*, on page 565.	proc syn_on_start_run
Ending a run	proc syn_on_end_run
Key Assignment Callbacks	
Setting an operation for Ctrl-F8. See *Example: proc syn_on_press_ctrl_f8*, on page 565.	proc syn_on_press_ctrl_f8
Setting an operation for Ctrl-F9	proc syn_on_press_ctrl_f9
Setting an operation for Ctrl-F11	proc syn_on_press_ctrl_f11

– Save the file.

As you synthesize your design, the software automatically executes the function callbacks you defined at the appropriate points in the design flow.

Example: proc syn_on_start_run

The following code example gets selected files from the project browser at the start of a run:

```
proc syn_on_start_run {compile c:/work/prep2.prj rev_1} {
   set sel_files [get_selected_files -browser]
      while {[expr [llength $sel_files] > 0]} {
         set file_name [lindex $sel_files 0]
         puts $file_name
         set sel_files [lrange $sel_files 1 end]
      }
}
```

Example: proc syn_on_press_ctrl_f8

The following code example gets all the selected files from the project browser and project directory when the Ctrl-F8 key combination is pressed:

```
proc syn_on_press_ctrl_f8 {} {
   set sel_files [get_selected_files]
      while {[expr [llength $sel_files] > 0]} {
         set file_name [lindex $sel_files 0]
         puts $file_name
         set sel_files [lrange $sel_files 1 end]
      }
}
```


© 2016 Synopsys, Inc.

CHAPTER 14

Improving Runtime

The following sections describe how to use multiprocessing to run parallel synthesis jobs and improve runtime. See Multiprocessing With Compile Points, on page 568.

© 2016 Synopsys, Inc.

Multiprocessing With Compile Points

This procedure describes how to run multiprocessing on a design with compile points. For information about defining compile points, see Chapter 11, *Working with Compile Points*.

To run compile points with multiprocessing:

1. If required, specify additional license types to use for multiprocessing, as described in *Using Different License Types for Multiprocessing*, on page 570.

   The tool uses one license per job, so additional licenses increase the number of jobs that can be run in parallel. The actual number of licenses used depends on certain factors. See *Specifying Licenses for Multiprocessing*, on page 569 for an explanation.

2. Select Options->Configure Compile Point Process and set the maximum number of jobs to run in parallel.

   See *Setting Maximum Parallel Jobs*, on page 568 for other ways to set this value.

3. Synthesize the design as usual.

   The synthesis software runs multiple, independent compile point jobs simultaneously, providing additional runtime improvements for the logical compile point synthesis flows.

Setting Maximum Parallel Jobs

You can set maximum number of parallel jobs in the following ways:

- Setting the MaxParallelJobs Variable in the ini File, on page 569
- Setting the max_parallel_jobs Tcl Option

Setting the MaxParallelJobs Variable in the ini File

The maximum number of parallel jobs is set in the product `ini` file. The following commands are set in the *product*.ini file (for example, synplify_pro.ini):

```
[JobSetting]

MaxParallelJobs=<n>
```

The MaxParallelJobs value is used by the UI as well as in batch mode. This value is effective until you specify a new value. To change the number of parallel jobs you can run, use the Options->Configure Compile Point Process command from the Project view menu. On the Configure Compile Point Process dialog box, in the Maximum number of parallel synthesis jobs field you will see the current `ini` value. You can specify a new MaxParallelJobs value which is effective until you change it again. Once you click OK, the new value is saved in the `ini` file.

Setting the max_parallel_jobs Tcl Option

You can also manually set an override value for the maximum number of parallel jobs. To do this, use the Tcl command:

set_option -max_parallel_jobs *numberJobs*

You can choose to:

- Source the Tcl file containing this option.
- Add this option to the Project file.
- Set this option from the Tcl command window.

This max_parallel_jobs value is applied to all project files and their respective implementations. This is a global option. The maximum number of parallel jobs remains in effect until you specify a new value. This new value takes affect immediately going forward. However, when you set this option from the Tcl command window, the max_parallel_jobs value is not saved and will be lost when you exit the application.

Specifying Licenses for Multiprocessing

When you decide to run parallel synthesis jobs, additional licenses may be required for the compile point jobs. By default, four parallel jobs use one license. For example, if you set the Maximum number of parallel synthesis jobs to

© 2016 Synopsys, Inc.

12, the synthesis tool consumes one license to run 4 compile point jobs and can utilize the two additional licenses to run 8 more parallel jobs if they are available for your computing environment. Licenses are released as jobs complete, and then consumed by new jobs which need to run.

The actual number of licenses utilized depends on the following factors:

1. Synthesis software scheme for the compile point requirements used to determine the maximum number of parallel jobs or licenses a particular design tries to use.
2. Value set on the Configure Compile Point Process dialog box.
3. Number of licenses actually available. You can use Help->Preferred License Selection to check the number of available license. If you need to increase the number of available licenses, you can specify multiple license types. For more information, see *Using Different License Types for Multiprocessing*, on page 570.

Factors 1 and 3 can change during a single synthesis run. The number of jobs equals the number of licenses; which then equates to the lowest value of these three factors.

Using Different License Types for Multiprocessing

You can specify multiple license types to increase the total number of licenses available for multiprocessing. Use either of these methods to specify the license types:

- Use the -licensetype command line option when you execute your tool.

  For example, suppose you have two synplifypro licenses, two synplifypro_all-vendor licenses, and three synplifypro_lattice licenses. Type the following at the command line:

  ```
  synplify_pro.exe -licensetype
  "synplifypro:synplifypro_allvendor:synplifypro_lattice"
  ```

- Use one of the following environment variables specified with the license type:
  - SYNPLIFYPRO_LICENSE_TYPE

```
setenv SYNPLIFYPRO_LICENSE_TYPE=
"synplifypro:synplifypro_allvendor:synplifypro_lattice"
```

Multiprocessing can access any of these license types for additional licenses.

© 2016 Synopsys, Inc.

SYNOPSYS®
Silicon to Software™

CHAPTER 15

Clock Conversion

ASICs are often implemented in FPGAs for prototyping. The inherent differences between ASICs and FPGAs can make this conversion difficult. One particular source of problems can be attributed to the complex clocking circuitry of FPGAs that often includes a large number of gated and internally generated clocks. The Synopsys FPGA synthesis tools provide two features, *Gated Clock Conversion* and *Generated Clock Conversion*, to address the complex clocking schemes.

These features move the generated-clock and gated-clock logic from the clock pin of a sequential element to its enable pin. Relocating these clocks allows the sequential elements to be tied directly to the skew-free source clock and also reduces the number of clock sources in the design which frees up routing resources and expedites placement and routing. Dedicated FPGA clock trees are routed to every sequential device on the die and are designed with low skew to avoid hold-time violations. Using these global clock trees allows the programmable routing resources of the FPGA to be used primarily for logic interconnect and simplifies static timing analysis because checks for hold-time violations based on minimum delays are unnecessary.

For details, refer to the following topics:

- Working with Gated Clocks, on page 574
- Optimizing Generated Clocks, on page 604

© 2016 Synopsys, Inc.

Working with Gated Clocks

This section first describes the gated-clock solution. The information is organized into the following sections:

- Obstacles to Conversion, on page 576
- Prerequisites for Gated Clock Conversion, on page 577
- Defining Clocks Properly, on page 579
- Synthesizing a Gated-Clock Design, on page 584
- Accessing the Clock Conversion Report, on page 585
- Analyzing the Clock Conversion Report, on page 586
- Interpreting Gated Clock Error Messages, on page 588
- Disabling Individual Gated Clock Conversions, on page 592
- Integrated Clock Gating Cells, on page 592
- Using Gated Clocks for Black Boxes, on page 597
- OR Gates Driving Latches, on page 598
- Obstructions to Optimization, on page 599
- Unsupported Constructs, on page 600
- Other Potential Workarounds to Solve Clock-Conversion Issues, on page 601
- Restrictions on Using Gated Clocks, on page 601

The clock conversion solution separates the gating from the clock inputs, and combines individual clock trees on the dedicated FPGA global clock trees. The software logically separates the gating from the clock and routes the gating to the clock enables on the sequential devices, using the programmable routing resources of the FPGA. The software separates a clock net going through an AND, NAND, OR, or NOR gate by doing one of the following:

- Inserting a multiplexer in front of the input pin of the synchronous element and connecting the clock net directly to the clock pin
- Moving the gating from the clock input pin to the dedicated enable pin, when this pin is available.

The ungated or base clock is routed to the clock inputs of the sequential devices using the global FPGA clock resources. Typically, many gated clocks are derived from the same base clock, so separating the gating from the clock allows a single global clock tree to be used for all gated clocks that reference that base clock.

It is not always necessary to convert all clocks from an ASIC design – fitting the design into the FPGA device and meeting timing are the ultimate goals. However, some situations require unconverted clock structures to be modified such as:

- Clock structures with a large number of loads resulting in congestion issues
- Clock structures resulting in a large clock skew between the starting and ending clocks on a path that does not meet timing

The following figure illustrates several cases of how gated clocks are converted.1

© 2016 Synopsys, Inc.

Gated Clock

Fixed Gated Clock

Gated Clock

Fixed Gated Clock

Gated Clock

Fixed Gated Clock

Obstacles to Conversion

The following is a summary of issues to look for that can prevent a clock structure from being converted. More details about each condition are described later.

- Are the clocks defined properly?
- Is the Clock Conversion option enabled?
- Does the clock structure contain instantiated cells?

- Is something in the clock logic blocking optimization?
- Are there any unsupported constructs?

Prerequisites for Gated Clock Conversion

For a gated clock to be converted successfully, the design must meet these requirements:

Condition	Description
Combinational logic only	The gated clock logic must consist only of combinational logic. A derived clock that is the output of a register is not converted.
Single base clock	Identify only one input to the combinational logic for the gated clock as a base clock. To identify a net as a clock, specify a period or frequency constraint for either the gate or the clock in the constraint file. This example defines the clk input as the base clock: `create_clock -name {clk} -freq 10.000 -clockgroup default_clkgroup`
Supported primitives	The sequential primitive clocked by the gated clock must be a supported object. The tools support gated-clock conversion for most sequential primitives. Black-box modules driven by gated clocks can be converted if special synthesis directives are used to define the clock and clock enable inputs to the black box. See *Using Gated Clocks for Black Boxes*, on page 597.
Correct logic format	See *Correct Logic Format*, on page 577 for an example of the correct logic format.

Correct Logic Format

Specifically, the combinational logic for the gated clock must satisfy the following two conditions to have the correct format:

- For at least one set of gating input values, the value output for the gated clock must be constant and not change as the base clock changes.
- For at least one value of the base clock, changes in the gating input must not change the value output for the gated clock.

© 2016 Synopsys, Inc.

The correct logic format requirements are illustrated with the simple gates shown in the following figures. When the software synthesizes a design with the Gated Clocks option enabled, clock enables for the AND gate and OR gate are converted, but the exclusive-OR gate shown in the second figure is not converted. The following table explains.

AND gate `gclks[1]`	If either gate[1] or gate[2] is 0, then gclks[1] is 0, independent of the value of clk which satisfies the first condition. Also, if clk is 0, then gclks[1] is 0, independent of the values of gate[1] and gate[2] which satisfies the second condition. Because gclks[1] satisfies both conditions, it is successfully converted to the clock-enable format.
OR gate `gclks[2]`	If either gate[1] or gate[2] is 1, then gclks[2] is 1 independent of the value of clk which satisfies the first condition. Also, if clk is 1, then gclks[2] is 1 independent of the value of gate[1] or gate[2] which satisfies the second condition. Because gclks[2] satisfies both conditions, it is successfully converted to the clock-enable format.
Exclusive-OR gate `gclks[3]`	Irrespective of the value of gate[3], gclks[3] continues to toggle. The exclusive-OR function causes gclks[3] to fail both conditions which prevents gclks[3] from being converted.

Before Gated Clock Conversion

After Gated Clock Conversion

The clock enables for the AND and OR gates are converted, but the clock enable for the exclusive OR remains unchanged.

Defining Clocks Properly

The most common problems preventing clock conversion from occurring automatically are related to improperly defined clocks. When defining gated and generated clocks:

- Gated clocks should be defined at the nodes to be connected to the clock pins of the sequential cells.
- Use create_generated_clock constraints to define the relationship between generated clocks and their sources.
- If a clock circuit contains both a clock generator and a gating element, a create_generated_clock constraint is needed. Without this constraint, the tool does not know the relationship between the gated signal and the source clock.

© 2016 Synopsys, Inc.

Defining Clocks Example

Using the above example, the first constraint applied is:

```
create_clock -name clk [get_ports clk] -period 10
```

In the above constraint, get_ports identifies clk as a port. With only the source clock defined, the tool is unable to determine which input of the AND gate is the clock and which input is the enable. Accordingly, no conversion is performed as shown in the following schematic diagram.

Continuing with the example, a second constraint is added:

```
create_clock -name clk [get_ports clk] -period 10
create_clock -name divclk [get_nets {divclk}] -period 20
```

In the second constraint, get_nets identifies divclk as a net. With independent clock constraints defining the source and generated clocks, the tool now knows which input of the AND gate is the clock, but still does not know the relationship between the source and generated clocks. In this case, the clock gate and data register 'Z' are converted to an enable flip-flop (FDCE) which is connected to the generated clock as shown in the next schematic.

© 2016 Synopsys, Inc.

For the final example, the second clock constraint is replaced with a generated clock constraint.

```
create_clock -name clk [get_ports clk] -period 10

create_generated_clock -name divclk [get_nets {n:divclk}]
   -source [get_ports clk] -divide_by 2
```

The tool knows which input of the AND gate is the clock and the relationship between the source and generated clocks. For this case, the entire clock circuit is converted to an enable on the 'Z' register, and the 'Z' register clock pin is connected directly to the source clock as shown in the final schematic.

Internal Clocks

Applying internal clock constraints to output pins of inferred combinational logic (for example, an AND gate) is not supported. Apply clock constraints only to nets driven by logical output pins.

Only One Clock per Clock Pin Fan-in Cone

Every fan-in cone of the clock pin of a sequential device should have one defined clock. As long as one clock is defined, the clock can be traced forward through unate logic to the clock pins.

© 2016 Synopsys, Inc.

When no clock is defined:

- The clock source is derived by tracing the clock pin back to the first non-trivial (multiple input) gate
- Because the tool does not know which input of a multiple-input cell to trace, a clock is inferred at the output of that cell
- Since inferred clocks were not defined by the user explicitly or implicitly, no conversion occurs

When more than one clock is defined, the tool does not know which clock to convert, and no conversion occurs.

Synthesizing a Gated-Clock Design

To synthesize a gated-clock design:

1. Make sure that the gated clocks have the correct logic format and satisfy the prerequisites for conversion. See *Prerequisites for Gated Clock Conversion*, on page 577 for details.
2. If the gated clock drives a black box, specify the clock and the associated clock enable signal with using syn_force_seq_prim, syn_isclock, and syn_gatedclk_en directives. See *Using Gated Clocks for Black Boxes*, on page 597 for details.
3. Make sure that the clock net has a constraint specified in the constraint file for the current implementation. If you do not specify an explicit constraint on the clock net or if you set a global frequency constraint, enabling clock conversion as described in the next step will not have any effect.
4. Enable the Clock Conversion option.
   - Select Project->Implementation Options.
   - On the GCC tab, click the Clock Conversion check box.

5. Synthesize the design. For gated clocks, the option converts qualified flip-flops, counters, latches, synchronous memories, and instantiated technology primitives. The software logically separates the gating from the clock and routes the gating to the clock enables on the sequential devices, using the programmable routing resources of the FPGA. The ungated base clock is routed to the clock inputs of the sequential devices using the global clock resources. Because many gated clocks are normally derived from the same base clock, separating the gating from the clock allows a single global clock tree to be used for all gated clocks that reference the same base clock.

   See *Restrictions on Using Gated Clocks*, on page 601 for additional information.

6. Check the results in the START OF CLOCK OPTIMIZATION REPORT section of the log file. See *Analyzing the Clock Conversion Report*, on page 586 for an example of this report.

Accessing the Clock Conversion Report

The clock conversion report can be accessed from the log file or from the Project Status tab.

Log File Access

To access the clock conversion report directly in the log file:

1. Open the log file by clicking the View Log button.
2. In the browser, expand the Mapper Report entry by clicking the adjacent '+' sign and click on the Clock Conversion link to go directly to the clock conversion report. Note that the browser is only available when the View log file in HTML option is enabled.

© 2016 Synopsys, Inc.

Project Status Tab Access

To access the clock conversion report directly from the Project Status tab, click the more link in the Optimizations Summary section at the bottom of the display.

Analyzing the Clock Conversion Report

When clock conversion is enabled, a typical report resembles the following:

```
#### START OF CLOCK OPTIMIZATION REPORT #####[

8 non-gated/non-generated clock tree(s) driving 434 clock pin(s) of sequential element(s)
2 gated/generated clock tree(s) driving 16 clock pin(s) of sequential element(s)
10 instances converted, 16 sequential instances remain driven by gated/generated clocks

========================================= Non-Gated/Non-Generated Clocks =========================================
Clock Tree ID     Driving Element     Drive Element Type     Fanout     Sample Instance
-----------------------------------------------------------------------------------------------------------------
ClockId0003       wb_clk_i            port                   215        camera_wishbone_if.dest_addr_cnt[31]
ClockId0004       bufg_inst2          BUFG                   163        control_synchronization.hsync_active_hi_sync
ClockId0005       ibufgds_inst        IBUFG
ClockId0006       bufg_inst3          BUFG                              ronization.cam_int_sync
ClockId0007       bufg_inst1          BUFG
ClockId0008       clk1                port                              reg_file_u1.ff_out[7]
ClockId0009       clk2                port                   2          clk_top_inst.clk_tree2_c1.ff_clk_u1.clk_out
ClockId0010       clk3                port                   1          clk_top_inst.reg_ff_u4.ff_out
=================================================================================================================
============================================================================ Gated/Generated Clocks ============================
Clock Tree ID     Driving Element                                          Sample Instance
-----------------------------------------------------------------------------------------------------------------
ClockId0001       clk_top_inst.clk_tree2_c1.com                            clk_top_inst.reg_file_u2.ff_out[6]
ClockId0002       clk_top_inst.clk_tree3_c2.q                              clk_top_inst.reg_file_u3.ff_out[7]
=================================================================================================================

##### END OF CLOCK OPTIMIZATION REPORT ######]
```

3-line Summary

Non-gated/non-generated Clock Tree Table

Gated/generated Clock Tree Table

3-line Summary

The 3-line summary provides a quick snapshot of the design.

- The first line reports the number of *clean* clock trees (8 in the above report) and the number of driven clock pins (434).
- The next line reports the number of gated/generated clock trees (2 in the above report) and the number of driven clock pins (16).
- The last line reports the number of instances converted (10 in the above report) and the number of remaining instances driven by gated/generated clocks (16).

The following find command can be used to group the converted instances:

```
find -hier -inst * -filter @syn_gc_converted==1
```

Non-gated/non-generated Clocks Table

The non-gated/non-generated clocks section provides details of the *clean* clocks in the design and also includes any clock trees that never included gating logic. This section reports the following:

- Clock Tree ID – created for all instances in the clock tree that lead to the sample instance. Clicking on an ID opens a filtered technology view of the source clock.

© 2016 Synopsys, Inc.

- Driving Element – identifies the source of the driving clock.
- Drive Element Type – identifies the type of drive element such as a port or BUFG.
- Fanout – lists the clock fanout.
- Sample Instance – an instance within the clock tree.

The following find command can be used to identify all instances in the clock path:

- `find -hier -inst * -filter @syn_sample_clock_path==CKIDxxxx`

If a clock is defined within the cone-of-logic for the gated clock, it is reported in this table.

Gated/generated Clocks Table

The gated/generated clocks section lists a sample failure within each clock tree in the design with an explanation of why the conversion failed (see the following section *Interpreting Gated Clock Error Messages*, on page 588). The gated/generated clocks section reports the following:

- Clock Tree ID – created for all instances in the clock tree that lead to the failed sample instance. Clicking on an ID opens a filtered technology view of the source clock.
- Driving Element – identifies the source of the driving clock.
- Drive Element Type – identifies the type of drive element such as a LUT.
- Fanout – lists the clock fanout.
- Sample Instance – an instance within the clock tree.

In the table, failures with common gating-control signals are only reported once. The software traverses back from the clock pin of the sample instance until it reaches the driving element at the point of the failure.

Interpreting Gated Clock Error Messages

The following table describes the gated clock conversion error messages reported in the Gated/Generated Clocks section of the clock conversion report.

Error Message	Explanation
Asynchronous set/reset mismatch prevents generated clock conversion	Unshared asynchronous signals have been detected between a FF-derived clock circuit and its sequential load.
Can't determine input clock driver	Self-explanatory. Specify the driver.
Clock conversion disabled	Gated clocks have been detected, but clock conversion is not enabled.
Clock propagation blocked by fixed hierarchy	Clock property has been blocked by a fixed hierarchy which prevents clock conversion from propagating upstream.
Clock propagation blocked by hard hierarchy	Clock property has been blocked by a hard hierarchy which prevents clock conversion from propagating upstream.
Clock propagation blocked by locked hierarchy	Clock property has been blocked by a locked hierarchy which prevents clock conversion from propagating upstream.
Clock propagation blocked by syn_keep	Clock property has been blocked by a syn_keep property which prevents clock conversion from propagating upstream.
Clock source is constant	Self-explanatory.
Combinational loop in clock network	Self-explanatory. Fix the combination loop.
FF-derived clock conversion disabled	FF-derived clocks have been detected, but clock conversion is not enabled.
Gating structure creates improper gating logic	Self-explanatory. Analyze and fix logic.
Illegal instance on clock path	Self-explanatory. Analyze and fix logic.
Inferred clock from port	Self-explanatory notification.
Input clock depends on output	Self-explanatory.
Latch gated by OR originally on clock tree	Self-explanatory.

© 2016 Synopsys, Inc.

Error Message	Explanation
Multiple clock inputs on sequential instance	Self-explanatory. Analyze and fix logic.
Multiple clocks on generating sequential element	Self-explanatory. Analyze and fix logic.
Multiple clocks on instance	Multiple clocks found feeding an instance of a clock tree.
Need declared clock or clock from port to derive clock from ff	Self-explanatory. Specify the clock.
No clocks found on inputs	No clocks found feeding an instance of a clock tree.
No generated or derived clock directive on output of sequential instance	Self-explanatory. Specify the clock.
No hierarchical driver	Self-explanatory. Specify the driver.
Signal from port	Self-explanatory.
Unconverted clock gate	Self-explanatory.
Unable to determine clock driver on net	Self-explanatory. Specify the clock driver.
Unable to determine clock input on sequential instance	Self-explanatory. Analyze and fix logic.
Unable to follow clock across hierarchy	Self-explanatory. Analyze and fix logic.
Unable to use latch as gated clock generator	Self-explanatory. Analyze and fix logic.

Checking the Log file

Many clock constraint problems can be identified by reviewing the Clock Summary table in the pre-mapping report section and the Performance Summary section of the log file shown below:

- Clock conversion is attempted on declared, generated, and derived clocks, and is never performed on inferred clocks or on the system clock.

- Declared, generated, and derived clocks can still have issues such as unsupported structures in their clock logic or incorrect constraints.
- Inferred clocks and the system clock are created when no clocks are defined in the fan-in logic of the clock pin of a sequential cell.

```
Performance Summary
*******************

Worst slack in design: 8.300

                                                          Clock      Clock
Starting Clock                                   Slack    Type       Group
-----------------------------------------------------------------------------------------
camera_open_source|CLKOUT0_derived_clock_CLKIN1  997.947  derived    Inferred_clkgroup_1
camera_open_source|CLKOUT1_derived_clock_CLKIN1  997.919  derived    Inferred_clkgroup_1
camera_open_source|camera_clock_pad_i            998.073  inferred   Inferred_clkgroup_1
camera_open_source|wb_clk_i                      996.467  inferred   Inferred_clkgroup_0
clk1                                             8.447    declared   default_clkgroup
clk2                                             8.300    declared   default_clkgroup
clk3                                             NA       declared   default_clkgroup
en3_0                                            NA       declared   default_clkgroup
System                                           NA       system     system_clkgroup
=========================================================================================

=========================================================================================
```

Following compilation, the software issues warnings about inferred clocks or the system clock, that include the clock name and one of the registers driven by that clock to help the user locate the issue in the HDL Analyst. The following are examples of these warnings:

- @W: MT529: Found inferred clock *clkName* which controls *numClkPins* sequential elements including *instanceName*. This clock has no specified timing constraint which may prevent conversion of gated or generated clocks and may adversely impact design performance.
- @W: MT531: Found signal identified as System clock which controls *numClkPins* sequential elements including *instanceName*. Using this clock, which has no specified timing constraint, can prevent conversion of gated or generated clocks and can adversely impact design performance.

© 2016 Synopsys, Inc.

Disabling Individual Gated Clock Conversions

By default, enabling gated-clock conversion applies globally to a design. For critical paths, individual clocked elements can be expressly excluded from the gated-clock conversion by adding a syn_keep directive directly to the input clock net. The syn_keep attribute can be applied in the source code or in a constraint file. The following example shows a syn_keep attribute applied to the gclk net in a source file.

```
module and_gate (clk, en, a, z);
input clk, en, a;
output reg z;
wire gclk /* synthesis syn_keep = 1 */;
assign gclk = en & clk;

always @(posedge gclk) z <= a;
endmodule
```

The corresponding entry in a constraint file is:

```
define_attribute {n:gclk} {syn_keep} {1}
```

Integrated Clock Gating Cells

The Synopsys synthesis tools do not read in ASIC cell libraries. To support conversion of Integrated Clock Gating (IGC) cells, these cells must be modeled in RTL.

If there are ASIC-only pins on an ICG cell, such as a scan-enable input or an observation output, the pins must be removed to allow the clock conversion to occur.

The following example shows an ICG cell with ASIC-only pins and the corresponding Verilog source code.

```
module icg (clk, en, scan_en, obs, gclk);
input clk, en, scan_en;
output obs, gclk;
wire t0, t1;
assign t0 = (!clk) ? en : t0;
assign obs = t0;
assign t1 = t0 | scan_en;
assign gclk = t1 & clk;
endmodule
```

The presence of the scan control and observation point signals forces the tool to preserve intermediate signals in the clock-gating circuit that prevents it from being optimized. In this case, the data register is connected directly to the source clock, but the latch and OR gate for the ICG model remain as part of the enable logic as shown in the following schematic.

© 2016 Synopsys, Inc.

The following code simplifies the RTL model of the complex IGC cell and allows the desired gated-clock conversion to occur.

```
module icg (clk, en, scan_en, obs, gclk);
input clk, en, scan_en;
output obs, gclk;
wire t0, t1;
assign t0 = (!clk) ? en : t0;
assign obs = 1’b0 /* t0 */;
assign t1 = t0 | 1'b0 /* scan_en */;
assign gclk = t1 & clk;
endmodule
```

In the modified code, the scan_en input, which is generally not used in an FPGA implementation, is replaced with a constant ‘0’. The obs observation point output, which normally is not required in an FPGA implementation, is also tied to a constant ‘0’. These changes remove the intermediate points as shown in the following schematic. Note that if no other changes are made to the design, this model still “fits” because the ports are unchanged. Also, the ‘0’ tied to the obs output propagates to simplify downstream logic.

© 2016 Synopsys, Inc.

Incorrect Phase

When there is an error in the RTL model of an ICG cell such that an incorrect phase of the clock is connected to the latch:

- Only the AND gate is converted leaving the latch driving the enable pin of the data flip-flops.

- A warning is generated stating "Active phase of latch *latchName* prevents its use as a stability latch for clock gate *clockGateName*."

Using Gated Clocks for Black Boxes

To convert gated clocks that drive black boxes, you must identify the clock and clock enable signal inputs to the black box using the syn_force_seq_prim, syn_isclock, and syn_gatedclk_clock_en directives. Refer to the *Attributes Reference Manual* for information about these directives. You assume responsibility for the functionality of the black-box content.

The following are Verilog and VHDL examples of a black box with the required directives specified.

Verilog

```
module bbe (ena, clk, data_in, data_out)
/* synthesis syn_black_box */
/* synthesis syn_force_seq_prim="clk" */
;
input clk
/* synthesis syn_isclock = 1 */
/* synthesis syn_gatedclk_clock_en="ena" */;
input data_in,ena;
output data_out;
endmodule
```


© 2016 Synopsys, Inc.

VHDL

```
library synplify;
use synplify.attributes.all;

entity bbe is
   port (clk : in std_logic;
         en : in std_logic;
         data_in : in std_logic;
         data_out : out std_logic);
attribute syn_isclock : boolean;
attribute syn_isclock of clk : signal is true;
attribute syn_gatedclk_clock_en : string;
attribute syn_gatedclk_clock_en of clk : signal is "en";
end bbe;

architecture behave of bbe is
attribute syn_black_box : boolean;
attribute syn_force_seq_prim : string;
attribute syn_black_box of behave : architecture is true;
attribute syn_force_seq_prim of behave : architecture is "clk";
begin
end behave;
```

OR Gates Driving Latches

To prevent problems in timing analysis when a latch is being driven by the OR of a clock and an enable, gated-clock conversion is implemented using the logic shown in the following diagram.

As illustrated in the above diagram, the enable (EN) and clock (CLK) feed separate latches, and XOR gates are used to condition the inputs and outputs. The conversion masks the combinational path from the timing engine and duplicates the behavior of the original circuit. Conversion only occurs if the original latch is not driving a clock network.

Obstructions to Optimization

The following optimization obstructions in the clock structure prevent conversion:

Inferred Clocks can Prevent Clock Conversions

Gated Clock Conversions

Beginning with the K-2015.09-SP1 release, the presence of inferred clocks can prevent the conversion of gated clocks (as the example illustrates below) or generated clocks. As a result, rerunning a design from a previous release may perform fewer conversions than previously reported.

In the example, clk_b is a declared clock and In_1 and In_2 are not declared as clocks. Previously, FF A could be a candidate for gated clock conversion. Now, this conversion cannot occur since there is also an inferred clock driving AND gate G1 (that is, the output of the AND gate G2 fed by In_1 and In_2 drives the clock of FF B).

“Keep_buffers” in the Clock Structure

Keep buffers are components that are added to the RTL view netlist to preserve specific nets. If a net in the clock circuitry is preserved by a keep buffer, it cannot be optimized away and can prevent conversion of the associated clock circuit. Keep buffers are the result of syn_keep attributes and constraints placed on nets such as false path, multi-cycle path, and other clock constraints. The following Tcl find command can be used to locate the keep buffers:

```
find -hier -view keepbuf
```


© 2016 Synopsys, Inc.

"Hard" Hierarchy Defined on a Clock Logic Block

A "hard" hierarchy attribute on a block containing a clock structure prevents conversion by forcing the interface of this block to remain unchanged. More specifically, if a hard hierarchy is between the clock logic and the data registers, the conversion cannot occur because the new signals for the enables cannot be added to the block interface. However, if all the clock logic is contained within the hard hierarchy, the gated clock conversion can occur. The following Tcl find command can be used to locate the hard hierarchies:

```
find -view * -filter (@syn_hier==hard)
```

Loads on Intermediate Nodes in the Clock Logic

Intermediate nodes in the clock logic should not have extra loads. For conversion to occur when extra loads are present, the RTL must be modified to separate the loads that drive clock pins of data registers from the other loads.

"Disable Sequential Optimizations" Project Option Prevents Latches from Being Removed

For conversion to occur, the Disable Sequential Optimizations option cannot be enabled.

Unsupported Constructs

The following are examples of constructs that prevent conversion from occurring.

- If the clock circuitry contains logic such that asynchronous set/reset signals can produce an active edge on the gated-clock line (without any change in the source clock), the circuit is not converted. Any active edges on the gated-clock output must be controlled by the source clock.
- By default, conversion only occurs if the registers in the generated-clock logic have the same asynchronous set/reset signals as the data registers that they control.
- If the wrong phase of the clock is connected to a latch in a gated-clock circuit, the following warning is reported:

  Active phase of latch *latchName* prevents its use as a stability latch for clock gate *clockGateName*

Other Potential Workarounds to Solve Clock-Conversion Issues

If the previous sections do not help resolve your issue, there a few more things you can try:

- Use a syn_direct_enable attribute on one of the gating signals to force the conversion.
- Recode RTL to fit one of the cases allowed for automatic conversion.

Restrictions on Using Gated Clocks

The Clock Conversion option has the following restrictions for gated clocks:

- A syn_keep attribute attached to a net, which should preserve the net during optimization, is not honored by the Clock Conversion option.
- A global buffer, when instantiated on a gated clock, blocks gated-clock conversion from being performed (the buffer is viewed as the clock source by the downstream circuitry). In the circuit below, the clk input to test_inst remains gated.

Similarly, defining a clock constraint at the output of a gated clock prevents conversion from continuing upstream.

Issues with Gated/Generated Clock Reporting

Several gated/generated clock reporting issues have been identified:

- When both inputs for XOR gates are defined as clocks, the software incorrectly reports the explanation as “No clocks found on input”. The correct explanation is “Improper clock gating structure.” For HAPs compatible devices, the correct explanation is “Multiple clocks found on inputs.”

© 2016 Synopsys, Inc.

- When both inputs for MUXs are defined as clocks, the software incorrectly reports “Multiple clocks on instances”. The correct explanation is “Improper clock gating structure”.
- Gated clocks might not be converted for logic from a hierarchy that uses syn_hier=hard/fixed. In the following example, the comb1 hierarchy has this attribute. Although the combinational logic within the comb2 hierarchy is outside the scope of syn_hier=hard, the enable logic of register, out1 does not get converted.

- For cascaded ICGs driving posedge flip-flops that use either compile points or the syn_hier attribute, the tool might produce a circuit with unused logic that does not get pruned away. See the shaded logic in the following figure. Depending on how the compile point or syn_hier boundaries are defined, the tool can report unused latches as unconverted, with the explanation “Unable to use latch as gated clock generator.”

RTL:

Post Mapping:

- The explanation "Unconverted clock gate" message indicates a gated/generated clock case that is not documented.

Clock Conversion for Multiple Input Clocks to OR Gate Latch

If multiple clocks are defined as inputs to an OR gate driving a latch, then clock conversion does not occur and the following message is generated: Multiple clocks on instance. If this message occurs, make sure that only one clock is input to the OR gate driving the latch.

© 2016 Synopsys, Inc.

Optimizing Generated Clocks

When the Clock Conversion option is enabled, the generated-clock logic is replaced during synthesis with logic that uses the initial clock with an enable. With generated-clock optimization, the original circuit functionality is preserved while performance is improved by reducing clock skew. For more information, see the following topics:

- Enabling Generated-Clock Optimization, on page 604
- Conditions for Generated-Clock Optimization, on page 604
- Generated-Clock Optimization Examples, on page 605

Enabling Generated-Clock Optimization

Generated-clock optimization is enabled by checking Clock Conversion on the GCC tab of the Implementation Options dialog box.

Conditions for Generated-Clock Optimization

To perform generated-clock optimization, the following conditions must be met:

1. The combinational logic must be driven by flip-flops.
2. The input flip-flops cannot have an active set or reset (an active-low reset must be tied high, or an active-high set must be tied low). Similar rules apply to all the input flip-flops in the cone.
3. All input flip-flops must be driven by the same edge of the same clock.

With generated-clock optimization, you do not have to specify a primary clock.

Generated-Clock Optimization Examples

Example 1: Generated Clock from Combinational Logic

The following code segment illustrates generated-clock optimization:

```
module gen_clk(clk1,a,b,c,q);
input clk1, a, b, c;
output q;
reg ao,bo,q;
wire en;

always @(posedge clk1)
   begin
      ao <= a;
      bo <= b;
   end
assign en = ao & bo;

always @(posedge en)
   begin
      q <= c;
   end
endmodule
```

With generated-clock optimization disabled (Clock Conversion unchecked), the circuit in the following figure shows a flip-flop (q) driven by a generated clock that originates from the combinational logic driven by flip-flops ao and bo which, in turn, are driven by the initial clock (clk1).

© 2016 Synopsys, Inc.

When generated-clock optimization is enabled (Clock Conversion checked), flip-flop q is replaced with an enable flip-flop. This flip-flop is clocked by the initial clock (clk1) and is enabled by combinational logic based on the a and b inputs as shown in the following figure.

Example 2: Generated Clock Followed by a Gate Clock

A design can have a register that is clocked by a generated clock, then followed by a gated clock. If the generated-clock constraint also includes the enable, the Clock Conversion option may optimize away the register and replace it with a constant value. When this occurs, the software converts the generated and gated-clock logic into a clock and an enable; the enable may never go active.

Example 3: Datapath Latch Conversion

This example shows a latch on the datapath that is clocked by a generated clock.

GCC converts the circuit to a flip-flop and multiplexer as shown in the following figure.

In this example, there are asynchronous set and reset signals on the clock generated from the flip-flop, but the load does not have asynchronous set and reset signals. By default, the tool does not run GCC in this scenario.

© 2016 Synopsys, Inc.

CHAPTER 16

Optimizing for Lattice Designs

This chapter covers techniques for optimizing your design for Lattice and Lattice iCE40. The information in this chapter is intended to be used together with the information in Chapter 9, *Inferring High-Level Objects*.

This chapter describes the following:

- Optimizing Lattice Designs, on page 610
- Optimizing Lattice iCE40 Designs, on page 623

For additional information about working with Lattice designs, see Passing Information to the P&R Tools, on page 628 and Generating Vendor-Specific Output, on page 629.

© 2016 Synopsys, Inc.

Optimizing Lattice Designs

This section describes the following techniques for working with Lattice designs:

- Instantiating Lattice Macros, on page 610
- Using Lattice GSR Resources, on page 612
- Inferring Lattice PIC Latches, on page 612
- Controlling I/O Insertion in Lattice Designs, on page 620
- Forward-Annotating Lattice Constraints, on page 621

Instantiating Lattice Macros

You can instantiate Lattice macros that are predefined in the Lattice libraries that come with the tool, in the *installDirectory*/lib directory.

1. To use a Verilog macro library, add the appropriate library to your project, making sure that it is the first file in the source files list.

   The Verilog macro libraries are under the *installDirectory*/lib directory: Add the library appropriate to the technology and language (v or vhd) you are using

LatticeXP device families	*installDirectory*/lib/lucent/xp *installDirectory*/lib/lucent/xp2 *installDirectory*/lib/lucent/xp3
LatticeSC/SCM device families	*installDirectory*/lib/lucent/sc *installDirectory*/lib/lucent/scm
MachXO device families	*installDirectory*/lib/lucent/machxo *installDirectory*/lib/lucent/machxo2
ECP/EC device families	*installDirectory*/lib/lucent/ec *installDirectory*/lib/lucent/ecp *installDirectory*/lib/lucent/ecp2 *nstallDirectory*/lib/lucent/ecp3

2. To use a VHDL macro library, add the appropriate library and use clauses to your VHDL source code at the beginning of the design units that instantiate the macros.

   You only need the VHDL macro libraries for simulation, but it is good practice to add them to the code. The library names may vary, depending on the map file name, which is often user-defined. The simulator uses the map file names to point to a library.

LatticeXP device families	`library xp;`
	`use xp.components.all`
	`library xp2;`
	`use xp.components.all`
	`library xp3;`
	`use xp.components.all`

LatticeSC/SCM device families	`library sc;`
	`use sc.components.all`
	`library scm;`
	`use sc.components.all`

MachXO device families	`library machxo;`
	`use machxo.components.all`
	`library machxo2;`
	`use machxo.components.all`

ECP/EC device families	`library ec;`
	`use ec.components.all`
	`library ecp;`
	`use ecp.components.all;`
	`library ecp2;`
	`use ecp2.components.all;`
	`library ecp3;`
	`use ecp2.components.all;`

3. Instantiate the macros from the library as described in *Instantiating Black Boxes and I/Os in Verilog*, on page 382 and *Instantiating Black Boxes and I/Os in VHDL*, on page 384.

© 2016 Synopsys, Inc.

Using Lattice GSR Resources

The following procedure describes how to use GSR (global set/reset) resources and check resource usage. The GSR resource is a prerouted signal that connects to the reset input of every flip-flop, regardless of any other defined reset signals.

1. For the LatticeECP/ECP2/EC, LatticeXP2/XP, LatticeSC/SCM, and MachXO families, you can control the use of GSR resources as follows:
   - To improve routability and performance, use the dedicated GSR resource. Select Project ->Implementation Options and enable the Force GSR Usage option on the Device tab.

     When you set this option, the synthesis software creates a GSR instance to access the resource. It uses the GSR resource for reset signals, instead of general routing. All registers are reset. when the GSR is activated, even if some flip-flops do not have a reset.
   - If a global set/reset does not correctly initialize the design, turn off the option. Select Project ->Implementation Options and disable the Force GSR Usage option on the Device tab. When this option is off, the software does not use the GSR resource unless all flip-flops have resets, and all resets use the same signal.
2. To optimize area, set the Resource Sharing option, as described in *Sharing Resources*, on page 438.
3. To check resource usage, do the following:
   - Synthesize the design.
   - Select View Log and check the Resource Usage section.

Inferring Lattice PIC Latches

The following procedure shows you how to control the inference of programmable I/O cells (PICs) in Lattice designs.

1. For the software to automatically infer PICs, make sure of the following:
   - The latch must be at the input port.
   - The latch must be directly driven by the input FPGA pad.
   - The design has one of the supported input control schemes: no clear or reset controls, and asynchronous clear or asynchronous resets.

After synthesis, the tool implements the following primitives for the PICs:

IF1S1D	Latches with asynchronous clear Latches with GSR used for clear
IF1S1B	Latches with asynchronous reset Latches with GSR used for reset.

See *Examples of PIC Latches*, on page 614 for examples of inferred PIC latches.

2. If you do not want to infer a PIC, set syn_keep on the input data net for the latch.

   After synthesis, the tool implements the latch as either a core latch with the LATCH primitive or as a mux, depending on the Lattice technology you selected. The following figure shows an input latch with no reset or clear implemented as a mux and a core latch in different technologies.

© 2016 Synopsys, Inc.

Examples of PIC Latches

The following examples show how the tool infers PICs from the code:

- Positive Level Data Latch with No Resets or Clears, on page 615
- Negative Level Data Latch with No Resets or Clears, on page 616
- Positive Level Data Latch with Asynchronous Reset, on page 617
- Positive Level Data Latch with Asynchronous Clear, on page 619

Positive Level Data Latch with No Resets or Clears

VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity inlatch is port
(clk : in std_logic;
din : in std_logic;
dout: out std_logic);
end entity inlatch;

architecture bhve of inlatch is
begin
  process(clk,din)
  begin
    if clk='1' then
      dout <= din;
    end if;
  end process;

end bhve;
```

Verilog

```
module inlatch (clk,din,dout);
input clk;  input din;  output dout;

reg dout;

always @(clk)
begin
  if(clk)
     dout <= din;
end

endmodule
```

With this code, the tool implements the IFS1S1B latch primitive in the Technology view:

© 2016 Synopsys, Inc.

Negative Level Data Latch with No Resets or Clears

VHDL

```
library ieee;
use ieee.std_logic_1164.all;
entity inlatch is port
(clk : in std_logic;
din : in std_logic;
dout: out std_logic);
end entity inlatch;
architecture bhve of inlatch is
begin
  process(clk,din)
  begin
    if clk='0' then
      dout <= din;
    end if;
  end process;
end bhve;
```

Verilog

```
module inlatch (clk,din,dout);
input clk;  input din;  output dout;
reg dout;
always @(clk)
begin
  if(!clk)
     dout <= din;
end
endmodule
```

With this code, the tool implements the IFS1S1B latch primitive in the Technology view:

Positive Level Data Latch with Asynchronous Reset

The tool infers the IFS1S1B latch primitive in the Technology view with the code shown below:

© 2016 Synopsys, Inc.

VHDL

```
library ieee;
use ieee.std_logic_1164.all;
entity inlatch is port
(clk : in std_logic; aset: in std_logic;
din : in std_logic; dout: out std_logic);
end entity inlatch;
architecture bhve of inlatch is
begin
 process(clk,din,aset)
 begin
   if aset ='1' then
     dout <='1';
   elsif clk='1' then
     dout <= din;
   end if;
 end process;
end bhve;
```

Verilog

```
module inlatch(clk,din,aset,dout);
input  clk; input  din; input aset; output dout;
reg  dout;
always @(clk or aset)
begin
 if(aset)
   dout <= 1'b1;
 else if (clk)
   dout <= din;
end
endmodule
```

Positive Level Data Latch with Asynchronous Clear

VHDL

```
library ieee;
use ieee.std_logic_1164.all;
entity inlatch is port
(clk : in std_logic; aclr: in std_logic;
din : in std_logic; dout: out std_logic); }
end entity inlatch;
architecture bhve of inlatch is
begin
 process(clk,din,aclr)
 begin
   if aclr ='1' then
     dout <='0';
   elsif clk='1' then
     dout <= din;
   end if;
end process;
end bhve;
```

Verilog

```
module inlatch(clk,din,aclr,dout);
input  clk; input  din; input aclr; output dout;
reg  dout;
always @(clk or aclr)
begin
  if(aclr)
    dout <= 1'b0;
  else if (clk)
    dout <= din;
end
endmodule
```

With this code, the tool infers the IFS1S1D primitive in the Technology view:

© 2016 Synopsys, Inc.

Controlling I/O Insertion in Lattice Designs

You can control I/O insertion globally, or on a port-by-port basis.

1. To control the insertion of I/O pads at the top level of the design, use the Disable I/O Insertion option as follows:
   - Select Project->Implementation Options and click the Device panel.
   - If you do not want to insert any I/O pads in the design, enable Disable I/O Insertion

     Do this if you want to check the area your blocks of logic take up, before you synthesize an entire FPGA. If you disable automatic I/O insertion, you do not get *any* I/O pads in your design, unless you manually instantiate them.
   - If you want to insert I/O pads, disable the Disable I/O Insertion option.

     When this option is set, the software inserts I/O pads for inputs, outputs, and bidirectionals in the output netlist. Once inserted, you can override the I/O pad inserted by directly instantiating another I/O pad.
2. To force I/O pads to be inserted for input ports that do not drive logic, follow the steps below.
   - To force I/O pad insertion at the module level, set the syn_force_pads attribute on the module. Set the attribute value to 1. To disable I/O pad insertion at the module level, set the syn_force_pads attribute for the module to 0.

– To force I/O pad insertion on an individual port, set the syn_force_pads attribute on the port with a value to 1. To disable I/O insertion for a port, set the attribute on the port with a value of 0.

Enable this attribute to preserve user-instantiated pads, insert pads on unconnected ports, insert bi-directional pads on bi-directional ports instead of converting them to input ports, or insert output pads on unconnected outputs.

If you do not set the syn_force_pads attribute, the synthesis design optimizes any unconnected I/O buffers away.

Forward-Annotating Lattice Constraints

You can forward-annotate multicycle and false path constraints to Lattice place-and-route tool by following the procedure below. For additional information about forward-annotation, see *Generating Constraint Files for Forward Annotation (Legacy)*, on page 64.

1. To forward-annotate a from, to, or through multicycle constraint, open the SCOPE spreadsheet and do either of the following:
   – Click the Multi-Cycle Paths tab. Depending on the type of constraint you want to set, select or type the instance name under the To, From or Through column. Next, set the number of clock cycles under the Cycles column.

     When you set this constraint, the software runs timing-driven synthesis and then forward-annotates the constraint.
   – Click the Other tab. In the Command column, type define_multicycle_path. In the Arguments column, type -from and the source port or register name, and -to and the destination port or register name. For example: -from in0_int -to output 2.

     When you set this constraint from the Other tab, the software forward-annotates the constraint, but does not run timing-driven synthesis using this constraint.

© 2016 Synopsys, Inc.

2. To forward-annotate a false path constraint, open the SCOPE spreadsheet and do either of the following:
   - Click the False Paths panel. Depending on the type of constraint you want to set, select or type the instance name under the To, From or Through column. When you set this constraint, the software runs timing-driven synthesis and then forward-annotates the constraint.
   - Click the Other tab. In the Command column, type define_false_path. In the Arguments column, type -from and the source port or register name, and -to and the destination port or register name. For example:

Command	Arguments
set_false_path	-from in1_int -to output
set_false_path	-from in* -to out*

   When you set this constraint from the Other tab, the software forward-annotates the constraint, but does not run timing-driven synthesis using this constraint.

3. Select Project->Implementation Options and enable the Write Vendor Constraint File option on the Implementation Results tab.
4. Run your design. The synthesis tool creates the $DESIGN_synplify.lpf file in the same directory as your results files.
5. Start the Lattice ispLEVER place-and-route tool and run the Map stage (after importing the $DESIGN_synplify.lpf file).
6. Run the PAR and BIT stages in ispLEVER.

Optimizing Lattice iCE40 Designs

This section describes the following topics when working with Lattice iCE40 designs:

- Using Sequential Logic
- Using Combinational Logic
- Handling Tristates
- Handling I/Os and Buffers

Using Sequential Logic

The synthesis software infers or instantiates the following Lattice iCE40 sequential logic depending on the control signal.

Primitive	Description
SB_DFF	D flip-flop
SB_DFFE	D flip-flop with clock enable
SB_DFFSR	D flip-flop with synchronous reset
SB_DFFR	D flip-flop with asynchronous reset
SB_DFFSS	D flip-flop with synchronous set
SB_DFFS	D flip-flop with asynchronous set
SB_DFFESR	D flip-flop with clock enable and synchronous reset
SB_DFFER	D flip-flop with clock enable and asynchronous reset
SB_DFFESS	D flip-flop with clock enable and synchronous set
SB_DFFES	D flip-flop with clock enable and asynchronous set
SB_DFFN	D flip-flop - negative edge clock
SB_DFFNE	D flip-flop - negative edge clock and clock enable
SB_DFFNSR	D flip-flop - negative edge clock with synchronous reset
SB_DFFNR	D flip-flop - negative edge clock with asynchronous reset
SB_DFFSS	D flip-flop - negative edge clock with synchronous set

© 2016 Synopsys, Inc.

Primitive	Description
SB_DFFNS	D flip-flop - negative edge clock with asynchronous set
SB_DFFNESR	D flip-flop - negative edge clock, enable and synchronous reset
SB_DFFNER	D flip-flop - negative edge clock, enable and asynchronous reset
SB_DFFNESS	D flip-flop - negative edge clock, enable and synchronous set
SB_DFFNES	D flip-flop - negative edge clock, enable and asynchronous set

Limitations

The synthesis software does not support initial values on flip-flops for the Lattice iCE40 technology. The Power ON state for Lattice iCE40 flip-flops is 0.

Using Combinational Logic

The synthesis software handles combinational logic as follows:

- This logic is mapped to the 4-input LUT (SB_LUT4).
- All data path operators (adders, subtractors, comparators, or multipliers) is mapped to carry chain logic (SB_CARRY) and additional LUT4s (SB_LUT4).

Handling Tristates

The synthesis software handles tristates as follows:

- Internal tristates are not supported and are converted to MUXes whenever possible. An error is generated if an internal tristate cannot be converted to a MUX.
- Tristates connected to an output port are absorbed into the I/O pad.

Handling I/Os and Buffers

The synthesis software handles I/Os and buffers as follows:

- The SB_IO primitive is used for inserting I/O pads.
- Since DDR inferencing is not supported, only input, output, and enable registers are packed into the I/O pads when available.
- The synthesis software does not infer SB_IO_DS, but allows for its instantiation.
- SB_GB is used for global buffers, such as clocks.
- SB_GB_IO is used for external clocks and SB_GB is used for internal clocks.
- Instantiated buffers are retained as is.
- Undriven pins in instantiated instances are left floating as it is in the RTL. Input pins in the RTL should not be connected to a floating net.
- The SB_WARMBOOT primitive can be instantiated and is treated as a black box.

© 2016 Synopsys, Inc.

CHAPTER 17

Working with Synthesis Output

This chapter covers techniques for optimizing your design for various vendors. The information in this chapter is intended to be used together with the information in Chapter 9, *Inferring High-Level Objects*.

This chapter describes the following:

- Passing Information to the P&R Tools, on page 628
- Generating Vendor-Specific Output, on page 629
- Invoking Third-Party Vendor Tools, on page 630

© 2016 Synopsys, Inc.

Passing Information to the P&R Tools

The following procedures show you how to pass information to the place-and-route tool; this information generally has no impact on synthesis. Typically, you use attributes to pass this information to the place-and-route tools. This section describes the following:

- Specifying Pin Locations, on page 628
- Generating Vendor-Specific Output, on page 629

Specifying Pin Locations

In certain technologies you can specify pin locations that are forward-annotated to the corresponding place-and-route tool. The following procedure shows you how to specify the appropriate attributes.

1. Start with a Lattice design
2. Add the appropriate attribute to the port. For a bus, list all the bus pins, separated by commas.
   - To add the attribute from the SCOPE interface, click the Attributes tab and specify the appropriate attribute and value.
   - To add the attribute in the source files, use the appropriate attribute and syntax. See the *Attribute Reference Manual* for syntax details.

Family	Attribute and Value
Lattice	`loc {pin_number}`

Generating Vendor-Specific Output

The following topics describe generating vendor-specific output in the synthesis tools. See Targeting Output to Your Vendor, on page 629.

Targeting Output to Your Vendor

You can generate output targeted to your vendor.

1. To specify the output, click the Implementation Options button.
2. Click the Implementation Results tab, and check the output files you need.

   The following table summarizes the outputs to set for the different vendors, and shows the P&R tools for which the output is intended.

Vendor	Output Netlist	P&R Tool
Lattice	EDIF (`.edf`)	Diamond

3. To generate mapped Verilog/VHDL netlists and constraint files, check the appropriate boxes and click OK.

   See *Specifying Result Options*, on page 91 for details about setting the option. For more information about constraint file output formats and how constraints get forward-annotated, see *Generating Constraint Files for Forward Annotation (Legacy)*, on page 64.

© 2016 Synopsys, Inc.

Invoking Third-Party Vendor Tools

You can invoke third-party tools from within the Synopsys FPGA synthesis products, and configure the locations and common arguments for the tools. This capability lets you modify source files or libraries or debug problems from within the third-party tool, without leaving the synthesis environment.

You can invoke pre-configured tools or add your own. The process consists of two steps:

- Configuring Tool Tags, on page 630
- Invoking a Third-Party Tool, on page 631

Configuring Tool Tags

A tool tag is a configuration definition for a tool you want to invoke from the synthesis interface. You define a tool tag to set up the third-party tool you want to use. The synthesis software has some popular applications already configured for you to use when the synthesis software starts up. The following procedure shows you how to define your own tool tags, or add command arguments. Use this to specify other tools, other versions of a tool, or to run a tool with different arguments.

1. Select Options->Configure 3rd Party Tool Options from the Project view.

2. Define the application tag information for the tool you want to invoke.
   - Specify the application you want to invoke in Application Tag Name.
   - Specify how you want to invoke the application tool. If you want to run the tool directly from the UI, select Direct Execution. If your application is a Tcl procedure, select TCL Mode.
   - Specify the location of the application executable or Tcl procedure name in Application Name with Path or Tcl Procedure Name.
   - Specify any command arguments you want in the Command Argument if any field. You can use this to define a new tool tag or to add arguments to a tool tag that is already defined.

     For a list of predefined command arguments, click the + button and select them from the list. Otherwise, type the command arguments. For other internal Synopsys tools, you must select $SynCode from the Command Argument if any field.
   - Click Apply.
   - Click Close.

   The tool saves these settings in the FPGA synthesis tool `.ini` file and retrieves them for subsequent invocations. For information about invoking a third-party tool, see *Invoking a Third-Party Tool*, on page 631, next.

Invoking a Third-Party Tool

You can define tool tags globally and then use these tool tags to run the third-party tool from the Project view for the specified tool tag only. Some common tool tags are pre-configured and are read when the application starts up. You can add or modify existing tool tags or define your own Tcl procedures to invoke within the FPGA synthesis tools.

1. Define a tool tag for your application, as described in *Configuring Tool Tags*, on page 630.

2. Right-click in the Project view on a file or folder which is configured to run the vendor tool, and select Launch Tools->Run Vendor Tool from the popup menu.

   This dialog box automatically displays tool tag information associated with the file or folder. If no tool tag information is specified, look for the parent hierarchy and edit or change it, if necessary.

© 2016 Synopsys, Inc.

3. To associate a file or folder with a particular third-party tool, do the following:
   - Select the file or folder in the Project view. If you select a folder, the third-party tool is associated with all the files in the folder. If you associate a tool with a file, this setting overrides the folder setting.
   - Right-click a file or folder and select Launch Tools->Run Vendor Tool from the popup menu.

   - In the Vendor Tool Invocation dialog box, select the application in Application Tag Name.
   - Include any additional options you want to use with this file when you invoke the vendor tool. You can set command arguments now, if you did not configure them earlier.
   - Verify the command string in the dialog box.
   - Click Save, and Close. The third-party tool is associated with the file or folder and appears in the Launch Tools menu.

4. To invoke an associated third-party tool for a file or folder, do the following:
   - Right-click the file or folder in the Project view.
   - Select Launch Tools-><*Third-Party Tool*> from the popup menu. The synthesis tool automatically runs the tool or Tcl procedure as specified.

5. To invoke the tool at the same time that you associate a third-party tool with a file or folder, or to add additional arguments on the fly, do the following:
   - Right-click a file or folder and select Launch Tools->Run Vendor Tool from the popup menu.
   - In the Vendor Tool Invocation dialog box, select the application in Application Tag Name.
   - Include any additional options you want to use with this file when you invoke the vendor tool. You can set command arguments now, if you did not configure them earlier.
   - Verify the command string in the dialog box.
   - Click Save. The tool and arguments you specified is associated with the file or folder and appears in the Launch Tools menu.

     If you defined a new tool tag, the 3rd Party Tool Configuration dialog box appears. After saving the settings here, go back to the Vendor Tool Invocation dialog box. You are prompted to save this information to the project file before invoking the third-party tool.
   - Click the Run button in the Vendor Tool Invocation dialog box. The synthesis tool launches the third-party tool or runs the Tcl procedure with the arguments you specified.

   These settings are saved in the FPGA synthesis tool `ini` file, from where it can be retrieved for subsequent invocations.

© 2016 Synopsys, Inc.

CHAPTER 18

Running Post-Synthesis Operations

The following topics describe how to run post-synthesis operations, like place-and-route and verification, with compatible tools:

- Running P&R Automatically after Synthesis, on page 636
- Simulating with the VCS Tool, on page 638

© 2016 Synopsys, Inc.

Running P&R Automatically after Synthesis

You can run place-and-route automatically from within the tool or in batch mode for Lattice iCE devices.

Note: To run place and route successfully, first set the environment variable PATH for the place-and-route tool.

The following diagram illustrates the flow.

For detailed procedures, see the following:

- Integrating Synthesis and Place-and-Route in One Run, on page 636
- Releasing the Synthesis License During Place and Route, on page 636

Integrating Synthesis and Place-and-Route in One Run

You can run the place-and-route tool for your target technology automatically after synthesis.

1. Check the Release Notes and make sure that you are using the correct version of the P&R tool.
2. To automatically run the P&R tool after synthesis completes, do the following:
   - Click on the Add P&R Implementation button. In the dialog box, select the P&R implementation you want to run and enable Run Place & Route following synthesis.
   - Synthesize the design.

   The tool automatically runs P&R after synthesis.

Releasing the Synthesis License During Place and Route

When invoking a third-party place-and-route tool from the FPGA synthesis tool, you can choose to have place and route continue to run even after exiting the synthesis tool so that it does not consume an FPGA license. The software lets you release the license for the synthesis tool and run the place-and-route tool in batch mode.

To release the FPGA license, specify the following command:

```
toolName -batch -license_release
```

Where *toolName* can be the following keywords: `synplify_pro`.

In synthesis batch mode (synbatch), the -license_release option obtains all the synthesis licenses that are checked out for the session and checks them in immediately after the place-and-route job is launched.

When licenses are released, you see the following message is generated:

```
Exiting session due to -license_release option
```


© 2016 Synopsys, Inc.

Simulating with the VCS Tool

The Synopsys VCS® tool is a high-performance, high-capacity Verilog simulator that incorporates advanced, high-level abstraction verification technologies into a single, open, native platform. You can launch this simulation tool from the synthesis tools on Linux and Unix platforms by following the steps below. The VCS tool does not run under the Windows operating system.

1. Set up the tools.
   - Install the VCS software and set up the $VCS_HOME environment variable to define the location of the software.
   - Set up the place-and-route tool.
   - In the synthesis software, either select Run->Configure and Launch VCS Simulator, or click the icon.

   If you did not set up the $VCS_HOME environment variable, you are prompted to define it. The Run VCS Simulator dialog box opens. For descriptions of the options in this dialog box, see Configure and Launch VCS Simulator Command, on page 280 of the *Reference Manual*.

2. Choose the category Simulation Type in the dialog box to configure the simulation options.
   - Specify the kind of simulation you want to run.

RTL simulation	Enable Pre-Synthesis
Post-synthesis netlist simulation	Enable Post-Synthesis
Post-P&R netlist simulation	Enable Post P&R

– Choose the category VCS Options in the dialog box to set options such as the following VCS commands.

To set ...	Type the option in ...
VLOGAN command options for compiling and analyzing Verilog, like the -q option	Verilog Compile
VHDLAN options for compiling and analyzing VHDL	VHDL Compile
VCS command options	Elaboration
SIMV command options, like `-debug`	Simulation

The options you set are written out as VCS commands in the script. If you leave the default settings the VCS tool uses the FPGA version of VCS and opens with the debugger (DVE) GUI and the waveform viewer. See the VCS documentation for details of command options.

3. If your project has Verilog files with `include statements, you must use the +incdir+ *fileName* argument when you specify the vlogan command. You enter the +incdir+ in the Verilog Compile field in the VCS Options dialog box, as shown below:

© 2016 Synopsys, Inc.

Example Verilog File:

```
`include "component.v"
module Top (input a, output x);
...
endmodule
```

The syntax for the VCS commands must reflect the relative location of the Verilog files:

– If the Verilog files are in the same directory as the top.v file, specify:

```
- vlogan -work work Top.v +incdir+ ./
```

– If the Verilog files are in the a directory above the top.v file, specify:

```
- vlogan -work work Top.v +incdir+ ../include1 +incdir+
  ../ include2
```

– If the Verilog files are in directories below and above the top.v file, specify:

```
- vlogan -work work Top.v +incdir+ ./include_dir1
  +incdir../include_dir2
```

4. Specify the libraries and test bench files, if you are using them.

– To specify a library, click the green Add button, and specify the library in the dialog box that opens. Use the full path to the libraries. For pre-synthesis simulation, specifying libraries is optional.

– For post-synthesis and post-P&R synthesis, by default the dialog box displays the UNISIM and SIMPRIM libraries in the P&R tool path. You can add and delete libraries or edit them, using the buttons on the side. To restore the defaults, click the Verilog Defaults or VHDL Defaults button, according to the language you are using.

– If you have test bench files, choose the category Test Bench Files in the dialog box to specify them. Use the buttons on the side to add, delete, or edit the files.

5. Specify the top-level module and run directory.

   – Choose the category Top Level Module in the dialog box to specify the top-level module or modules for the simulation.

   – If necessary, choose the category Run Directory near the bottom of the dialog box to edit the default run directory listed in the field. The default location is in the implementation results directory.

6. Generate the VCS script.

   – To view the script before generating it, click the View Script button on the top right of the dialog box. A window opens with the specified VCS commands and options.

   – To generate the VCS script, click Save As, or run VCS by clicking the Run button in the upper right. The tool generates the XML script in the directory specified.

© 2016 Synopsys, Inc.

7. To run VCS from the synthesis tool interface, do the following:
   - If you do not already have it open, open the Run VCS Simulator dialog box by clicking the VCS icon.
   - To use an existing script, click the Load From button on the lower right and select the script in the dialog box that opens. Then click Run in the Run VCS Simulator dialog box.
   - If you do not have an existing script, specify the VCS options, as described in the previous five steps. Click Run.

   The tool invokes VCS from the synthesis interface, using the commands in the script.

Limitations

If Verilog include paths have been added to your project file, these paths are not automatically added to the VCS script. Add the Verilog include paths manually by using one of the following workarounds:

- From the Run VCS Simulator dialog box, add `+incdir+`*includePath* in the Verilog Compile options field.
- Modify the VCS script file, adding the `+incdir+`*includePath* to all or any relevant vlogan commands.

Index

Symbols

.adc file 370

.ini file
  parallel jobs 557

.lpf file 622

.scf file 65

Numerics

3rd party vendor tools
  invoking 630

A

adc constraints 370

adc file
  creating 370
  object names 374

adc file, using 368

adders
  SYNCore 509

Alt key
  column editing 48
  mapping 331

analysis design constraint file (.adc) 370

analysis design constraints
  design scenarios 369

analysis design constraints (adc) 368

analysis design constraints (adc), using with sdc 370

Analyst view
  traversing hierarchy 255

archive utility
  using 130

archiving projects 130

area, optimizing 415

asterisk wildcard
  Find command 264, 319

attributes
  adding 99
  adding in constraint files 64
  adding in SCOPE 102
  adding in Verilog 101
  adding in VHDL 100
  effects of retiming 426
  for FSMs 393
  handling properties 106
  inferring RAM 399
  pipelining 420
  syn_hier (on compile points) 476
  VHDL package 100

Attributes panel
  using SCOPE 152

audience for the document 21

auto constraints, using 375

AutoConstraint_design_name.sdc 378

B

B.E.S.T 335

backslash
  escaping dot wildcard in Find command 264, 319

batch mode 550
  using find and expand 176

Behavior Extracting Synthesis Technology. *See* B.E.S.T

black boxes 382
  adding constraints 386
  adding constraints in SCOPE 389
  adding constraints in Verilog 388
  adding constraints in VHDL 387
  gated clock attributes 597
  instantiating in Verilog 382
  instantiating in VHDL 384
  passing VHDL boolean generics 59
  passing VHDL integer generics 60
  pin attributes 390
  setting in hierarchical projects 122

© 2016 Synopsys, Inc.

block RAM
  inferring 401
  modes 398
  types 398
block-first hierarchical development flow 31
blocking-style license queuing 553
blocks
  defining for hierarchical projects 108
bookmarks
  in source files 48
  using in log files 216
bottom-up design flow
  compile point advantages 450
bottom-up hierarchical synthesis flow 34
browsers 306
buffering
  controlling 436
byte-enable RAMs
  SYNCore 497

C

c_diff command, examples 182
c_intersect command, examples 182
c_list command
  different from c_print 184
  example 186
  using 185
c_print command
  different from c_list 184
  using 185
c_symdiff command, examples 183
c_union command, examples 182
callback functions, customizing flow 563
case sensitivity
  Find command (Tcl) 169
clock and path constraints
  setting 153
clock constraints
  setting 153
  setting (Legacy) 193
clock conversion report
  accessing 585
  analyzing 586
clock groups
  effect on false path constraints 167
clock trees 361
clocks
  implicit false path 167
Clocks panel
  using SCOPE 151
collections
  adding objects 181
  concatenating 181
  constraints 179
  copying 185
  creating from common objects 181
  creating from other collections 179
  creating in SCOPE 178
  creating in Tcl 180
  crossprobing objects 179
  definition 177
  diffing 181
  highlighting in HDL Analyst views 184
  listing objects 185
  listing objects and properties 184
  listing objects in a file 185
  listing objects in columnar format 184
  listing objects with c_list 184
  special characters 183
  Tcl window and SCOPE comparison 177
  using Tcl expand command 173
  using Tcl find command 172
  viewing 183
Collections panel
  using SCOPE 151
column editing 48
combination hierarchical synthesis flow 38
comments
  source files 48
compile point types
  hard 455
  locked 455
compile points
  advantages 450
  analyzing results 477
  automatic timing budgeting 461
  child 453
  constraint files 458

constraints for forward-annotation 467
constraints, internal 466
creating constraint file 475
defined 450
defining in constraint files 472
feature summary 457
incremental synthesis 479
manual compile point flow 468
nested 453
optimization 464
order of synthesis 464
parent 453
preserving with syn_hier 477
resynthesis 465
setting constraints 475
setting type 474
syn_hier 476
synthesis process 464
synthesizing 468
types 453
using syn_allowed_resources attribute 477

compile points and hierarchical project management 30

Compile Points panel 152

compile-point synthesis
interface logic models 460

compile-point synthesis flow
defining compile points 472
setting constraints 475

compiler directives (Verilog)
specifying 95

constants
extracting from VHDL source code 97

constraint files
applying to a collection 179
compile point 458, 467
creating in a text editor 63
editing 159
effects of retiming 426
options 90
setting for compile points 475
vendor-specific 64

constraints
defining clocks (Legacy) 188
defining register delays (Legacy) 189
specifying through points 163
types 151
types (legacy) 190
using FDC template command 149

context
for object in filtered view 338

context help editor 44
SystemVerilog 44

counters
SYNCore 516

create_fdc_template
using 149

critical paths
delay 363
flat view 362
hierarchical view 362
negative slack on clock enables (Legacy) 200
slack time 363
using -route 417
viewing 361

crossprobing 327
and retiming 426
collection objects 179
filtering text objects for 332
from FSM viewer 333
from log file 216
from message viewer 229
from text files 330
Hierarchy Browser 265, 327
importance of encoding style 334
new HDL Analyst views 266
paths 331
RTL view 328
schematic views 265
Technology view 328
Text Editor view 328
text file example 331
to FSM Viewer 333
to place-and-route file 303
Verilog file 328
VHDL file 328
within RTL and Technology views 327

current level
expanding logic from net 342
expanding logic from pin 342
searching current level and below 316

custom folders
creating 77
hierarchy management 77

customization

© 2016 Synopsys, Inc.

callback functions 563

D

data block 527
data key 527
default enum encoding 97
define_attribute 106
Delay Paths panel
  using SCOPE 152
design flow
  customizing with callback functions 563
design flows
  hierarchical project management 30
  team design 30
design guidelines 414
design hierarchy
  viewing 270, 336
design size
  amount displayed on a sheet 303
design views
  moving between views 254, 302
device options
  See also implementation options
directives
  adding 99
  adding in Verilog 101
  adding in VHDL 100
  black box 387, 388
  for FSMs 393
  handling properties 106
  specifying for Verilog compiler 95
  syn_state_machine 444
  syn_tco 388
    adding black box constraints 387
  syn_tpd 388
    adding black box constraints 387
  syn_tsu 388
    adding black box constraints 387
dissolving instances for flattening hierarchy 349
dot wildcard
  Find command 264, 319
drivers
  preserving duplicates with syn_keep 429
  selecting 345
dual-port RAMs
  SYNCore parameters 494

E

Editing window 46
editor view
  context help 44
emacs text editor 51
encoding styles
  and crossprobing 334
  default VHDL 97
  FSM Compiler 442
encryption flow. *See* ReadyIP, encryptIP
encryptip output constraints 539
encryptip output method
  effect on output netlists 539
encryptIP script
  controlling output 537
  encrypting IP 536
  output methods 537
encryptP1735 script
  encrypting multiple files 535
  location 531
  public keys repository file 535
  use models 531
encryptP1735P script
  encrypting IP 531
environment variables
  SYN_TCL_HOOKS 563
error codes 551
error messages
  gated clock report 588
errors
  definition 46
  filtering 228
  sorting 228
  source files 46
  Verilog 46
  VHDL 46
expand
  batch mode 176
Expand command
  connection logic 345
  pin and net logic 276, 341

using 342
expand command (Tcl). *See* Tcl expand command
Expand Inwards command
using 276, 342
Expand Paths command
different from Isolate Paths 345
Expand to Register/Port command
using 342
expanding
connections 345
pin and net logic 276, 341

F

false paths
defining between clocks (Legacy) 204
I/O paths 167
impact of clock group assignments 167
impact of clock group assignments (Legacy) 204
ports 167
ports (Legacy) 204
registers 167
registers (Legacy) 204
setting constraints 167
setting constraints (Legacy) 204
fanouts
buffering vs replication 436
hard limits 435
soft global limit 434
soft module-level limit 435
using syn_keep for replication 430
using syn_maxfan 434
feature comparison
FPGA tools 17
FIFOs
compiling with SYNCore 484
files
.lpf 622
.prf file 230
.scf 65
filtered messages 232
fsm.info 443
log 213
message filter (prf) 230
output 629
rom.info 309
searching 127
statemachine.info 355
synhooks.tcl 563
Tcl 556
See also Tcl commands
Tcl batch script 552
Filter Schematic command, using 274, 339
Filter Schematic icon, using 274, 339
filtering 274, 339
advantages over flattening 274, 339
using to restrict search 316
Find command
316
browsing with 315
hierarchical search 317
long names 315
message viewer 228
reading long names 318
search scope, effect of 319
search scope, setting 317
searching the mapped database 318
searching the output netlist 324
setting limit for results 318
using in RTL and Technology views 316
using wildcards 264, 319
wildcard examples 321
find command
nuances and differences 323
find command (Tcl)
See Tcl find command
finding information
information organization 22
Fix Gated Clocks option. *See* gated clocks
Flatten Current Schematic command
transparent instances 347
using 347
Flatten Schematic command
using 347
flattening 283, 346
See also dissolving
compared to filtering 274, 339
dissolving instances 283, 349
hidden instances 348
transparent instances 347
using syn_hier 432
using syn_netlist_hierarchy 432

© 2016 Synopsys, Inc.

forward annotation
vendor-specific constraint files 64
forward-annotation
compile point constraints 466
constraints 621
FPGA Design Constraints Editor
using TCL View 157
frequency
clocks (Legacy) 197
defining for non-clock signals (Legacy) 197
internal clocks (Legacy) 197
setting global 89
from constraints 162
FSM Compiler
advantages 441
enabling 442
FSM encoding
user-defined 395
using syn_enum_encoding 395
FSM Explorer
when to use 441
FSM view
crossprobing from source file 330
FSM Viewer 352
crossprobing 333
fsm.info file 443
FSMs
See also FSM Compiler, FSM Explorer
attributes and directives 393
defining in Verilog 391
defining in VHDL 392
definition 391
optimizing with FSM Compiler 441
properties 355
safe. *See* safe FSMs
state encodings 354
transition diagram 352

G

gated clocks
attributes for black boxes 597
conversion example 578
conversion requirements 577
defining (Legacy) 202
error messages 588
examples 575
procedure for fixing 584
restrictions 601
gated-clock conversion
excluding elements 592
Generated Clocks panel
using SCOPE 151
generated-clock conversion 604
generics
extracting from VHDL source code 97
passing boolean 59
passing integer 60
global optimization options 87
GSR resources 612

H

HDL Analyst
See also RTL view, Technology view
critical paths 361
crossprobing 265, 327
filtering schematics 274, 339
Push/Pop mode 309, 312
traversing hierarchy with mouse strokes 307
traversing hierarchy with Push/Pop mode 258, 309
using 270, 335
HDL Analyst tool
deselecting objects 249, 300
selecting/deselecting objects 248, 299
HDL Analyst views
highlighting collections 184
HDL views, annotating timing information 359
help
information organization 22
hidden instances
consequences of saving 337
flattening 348
restricting search by hiding 316
specifying 337
status in other views 337
hierarchical design
expanding logic from nets 342
expanding logic from pins 276, 341
hierarchical instances
dissolving 283, 349

hiding. *See* hidden instances, Hide Instances command
multiple sheets for internal logic 338
pin name display 340
viewing internal logic 337

hierarchical objects
pushing into with mouse stroke 257, 308
traversing with Push/Pop mode 258, 309

hierarchical project management and compile points 30

hierarchical project management flows
bottom-up development flow 31
bottom-up synthesis flow 34
mixed block synthesis flow 38
top-down development flow 32
top-down synthesis flow 36

hierarchical projects 30
analyzing 125
configuring for synthesis 121
multiple implementations 117
synthesis options 121

hierarchical search 316

hierarchy
flattening 283, 347
traversing 255, 306

hierarchy browser
clock trees 361
controlling display 303
crossprobing from 265, 327
defined 255, 306
finding objects 262, 314
traversing hierarchy 306

hierarchy management (custom folders) 77

high reliability
safe FSMs 396
using safe FSM 396

hyper source
example 545
for IPs 543
for prototyping 543
IP design hierarchy 543
threading signals 544

I

I/O insertion 439, 620

I/O pads
specifying I/O standards 156

I/O paths
false path constraint 167

I/O standards
specifying 156

I/O Standards panel
using SCOPE 152

I/Os
auto-constraining 376
constraining 155
constraining (Legacy) 203
preserving 440, 621
Verilog black boxes 382
VHDL black boxes 384

implementation options 85
device 85
global frequency 89
global optimization 87
part selection 85
specifying results 91

implementations
copying 84
deleting 84
multiple. *See* multiple implementations.
overwriting 84
renaming 84

incremental synthesis
compile points 479

initializing 407

initializing RAM 407

Input and output constraints
defining 155

input constraints, setting 155

input constraints, setting (Legacy) 203

Inputs/Outputs panel
using SCOPE 152

instances
preserving with syn_noprune 429
properties 246, 294
properties of pins 246, 295

ILM *See* interface logic models

© 2016 Synopsys, Inc.

interface logic models 460
interface timing 461
IP
  encryption-decryption flow 525
  re-encryption 528
IP design hierarchy
  hyper source 543
IP encryption flow overview 524
IP encryption scheme 529
IP vendors
  directory structure for package 541
  encrypting IP 529
  package file list for encrypted IP flow 541
  packaging for evaluation 540
  supplying vendor information 541
IPs
  encrypting 529
  encryption flow 524
  SYNCore 484
  SYNCore byte-enable RAMs 497
  SYNCore counters 516
  SYNCore FIFOs 484
  SYNCore RAMs 490
  SYNCore ROMs 504
  SYNCore subtractors 509
  using hyper source for debug 543
Isolate Paths command
  different from Expand Paths 345, 346
ispLEVER
  forward-annotating constraints for 621

J

job management
  up-to-date checking 208

K

key assignments
  customizing 564
key block 527
keywords
  completing words in Text Editor 47

L

Lattice
  constraint file 64
  forward annotation 64
  I/O insertion 439
  macro libraries 610
  PICs 612
Lattice netlist 629
library extensions 53
license queuing 553
  blocking-style 553
license release (synthesis)
  after P&R 636
license_release 636
log file
  remote access 219
log files
  checking information 213
  pipelining description 421
  retiming report 425
  setting default display 213
  state machine descriptions 442
  viewing 213
logic
  expanding between objects 345
  expanding from net 278, 342
  expanding from pin 276, 341
logic preservation
  syn_hier 433
  syn_keep for nets 429
  syn_keep for registers 429
  syn_noprune 429
  syn_preserve 429
logical folders
  creating 77

M

macro libraries
  Lattice 610
manual compile points
  flow 468
max_parallel_jobs variable 558
maximum parallel jobs 557, 568
MaxParallelJobs variable 557

memory usage
maximizing with HDL Analyst 351
Message viewer
filtering messages 229
keyboard shortcuts 228
saving filter expressions 230
searching 228
using 227
using the F3 key to search forward 228
using the Shift-F3 key to search backward 228
messagefilter.txt file 236
messages
demoting 234
filtering 229
promoting 234
saving filter information from command line 231
saving filter information from GUI 230
severity levels 235
suppressing 234
writing messages to file 232
mixed block hierarchical synthesis flow 38
mixed designs
troubleshooting 59
mixed language files 56
mouse strokes
pushing/popping objects 257, 307
multicycle constraints
forward-annotating 621
multicycle paths
setting constraints 153
setting constraints (Legacy) 194
multiple implementations 83
hierarchical projects 117
running from project 83
multiprocessing
maximum parallel jobs 557, 568
multisheet schematics 300
for nested internal logic 338
searching just one sheet 316
transparent instances 301

N

name spaces
output netlist 324
technology view 318
navigating among design views 254, 302
netlists for different vendors 629
nets
expanding logic from 278, 342
preserving for probing with syn_probe 429
preserving with syn_keep 429
properties 246, 294
selecting drivers 345
New property 297
notes
filtering 228
sorting 228
notes, definition 46

O

objects
finding on current sheet 316
flagging by property 295
selecting/deselecting 299
open_design
with find and expand 176
optimization
for area 415
for timing 416
logic preservation. *See* logic preservation.
preserving hierarchy 433
preserving objects 429
tips for 414
orig_inst_of property 298
output constraints, setting 155
output constraints, setting (Legacy) 203
output files 629
specifying 91
output netlists
finding objects 324
overutilization 226

P

package library, adding 70
pad types

© 2016 Synopsys, Inc.

industry standards 156
parallel jobs 557
parameter passing 60
boolean generics 59
parameters
extracting from Verilog source code 95
part selection options 85
path constraints
false paths 167
false paths (Legacy) 204
pathnames
using wildcards for long names (Find) 319
paths
crossprobing 331
tracing between objects 345
tracing from net 278, 342
tracing from pin 276, 341
pattern matching
Find command (Tcl) 169
pattern searching 127
PDF
cutting from 48
PICs 612
pin names, displaying 340
pins
expanding logic from 276, 341
properties 246, 294
pipelining
adding attribute 420
definition 418
prerequisites 418
whole design 418
ports
false path constraint 167
false path constraint (Legacy) 204
properties 246, 294
POS interface
using 163
post-synthesis constraints with adc 369
preferences
crossprobing to place-and-route file 303
displaying Hierarchy Browser 303
displaying labels 304
RTL and Technology views 303
sheet size (UI) 303
primitives
pin name display 340
pushing into with mouse stroke 257, 308
viewing internal hierarchy 336
probes
adding in source code 446
definition 446
retiming 427
Product of Sums interface. *See* POS interface
project command
archiving projects 130
copying projects 137
unarchiving projects 134
project file hierarchy 77
project files
adding files 72
adding source files 68
batch mode 550
creating 68
definition 68
deleting files from 72
opening 72
replacing files in 72
updating include paths 75
VHDL file order 71
VHDL library 70
project status report
remote access 219
projects
archiving 130
copying 137
restoring archives 134
properties
displaying with tooltip 246, 294
finding objects with Tcl find -filter 170
orig_inst_of 298
reporting for collections 184
viewing for individual objects 246, 294
prototyping
using hyper source threading 543
Push/Pop mode
HDL Analyst 307
keyboard shortcut 309
using 257, 258, 307, 309

Q

question mark wildcard, Find command 264, 319

R

RAM inference 398
  using attributes 399
RAMs 407
  compiling with SYNCore 490
  inferring block RAM 401
  initializing 407
  SYNCore 490
  SYNCore, byte-enable 497
register balancing. *See* retiming
register constraints, setting (Legacy) 193
registers
  false path constraint 167
  false path constraint (Legacy) 204
Registers panel
  using SCOPE 152
remote access
  status reports 219
replication
  controlling 436
reports
  gated clock conversion 586
resource sharing 612
  optimization technique 415
  overriding option with syn_sharing 439
  results example 439
  using 438
resource usage 225
resource utilization. *See* resource usage
resynthesis
  compile points 465
  forcing with Resynthesize All 466
  forcing with Update Compile Point Timing Data 466
retiming
  effect on attributes and constraints 426
  example 424
  overview 422
  probes 427
  report 425
  simulation behavior 427
return codes 551
rom.info file 309
ROMs
  SYNCore 504
  viewing data table 309
RTL view
  See also HDL Analyst
  analyzing clock trees 361
  crossprobing collection objects 179
  crossprobing description 327
  crossprobing from 328
  crossprobing from Text Editor 330
  defined 292
  description 291
  filtering 274, 339
  finding objects with Find 316
  finding objects with Hierarchy Browser 262, 314
  flattening hierarchy 283, 347
  highlighting collections 184
  opening 243, 293
  selecting/deselecting objects 299
  setting preferences 303
  state machine implementation 443
  traversing hierarchy 306
RTL views
  hierarchical projects 125
  hierarchical subprojects 116
running P&R
  license release (synthesis) 636

S

safe case 396
safe FSM 396
  using safe case 396
schematics
  multisheet. *See* multisheet schematics
  page size 303
  selecting/deselecting objects 248, 299
SCOPE
  adding attributes 102
  adding probe insertion attribute 447
  Attributes panel 152
  case sensitivity for Verilog designs 169
  Clocks panel 151
  collections compared to Tcl script window 177
  Collections panel 151

© 2016 Synopsys, Inc.

Compile Points panel 152
creating compile-point constraint file 475
defining compile points 471
Delay Paths panel 152
drag and drop 160
editing operations 161
Generated Clocks panel 151
I/O pad type 156
I/O Standards panel 152
Inputs/Outputs panel 152
multicycle paths 166
pipelining attribute 419
Registers panel 152
setting compile point constraints 475
setting constraints (FDC) 144
specifying constraints 151
state machine attributes 393
TCL View 152

SCOPE editor
using 144

scope of the document 21

SCOPE panels
entering and editing constraints 151

SCOPE TCL View
using 157

search
browsing objects with the Find command 315
browsing with the Hierarchy Browser 262, 314
finding objects on current sheet 316
setting limit for results 318
setting scope 317
using the Find command in HDL Analyst views 316

search in Analyst
browsing objects with the Find command 263

See also search

set command
collections 185

set_option command 86

sheet connectors
navigating with 302

sheet size
setting number of objects 303

shematic view
setting preferences (New Anallyst) 254

Shift-F3 key
Message Viewer 228

Show Cell Interior option 336

Show Context command
different from Expand 338
using 338

signals
threading with hyper source. *See* hyper source

simulation, effect of retiming 427

single-port RAMs
SYNCore parameters 493

slack 364
setting margins 361

slack time display 358

Slow property 296

source code
adding pipelining attribute 420
commenting with synthesis on/off 98
crossprobing from Tcl window 333
defining FSMs 391
fixing errors 49
opening automatically to crossprobe 329
optimizing 414

source files
See also Verilog, VHDL.
adding comments 48
adding files 68
checking 45
column editing 48
copying examples from PDF 48
creating 42
crossprobing 330
editing 47
editing operations 47
mixed language 56
specifying default encoding style 97
specifying top level file for mixed language projects 57
specifying top level in Project view 71
specifying top-level file 97
state machine attributes 393
using bookmarks 48

special characters
Tcl collections 183

STA 365
STA, generating custom timing reports 365
STA, using analysis design constraints (adc) 368
stand-alone timing analyst. *See* STA
state machines
  See also FSM Compiler, FSM Explorer, FSM viewer, FSMs.
  attributes 393
  descriptions in log file 442
  implementation 443
  parameter and 'define comparison 392
statemachine.info file 355
sub-projects
  creating 108
  linking updates to top level 110
  multiple implementations 117
  synchronizing device options with top level 124
subprojects
  block-based 108
  compiling 114
  nested 111
subtractors
  SYNCore 509
syn_allow_retiming
  using for retiming 423
syn_allowed_resources
  compile points 477
syn_encoding attribute 394
syn_enum_encoding directive
  FSM encoding 395
syn_force_pad attribute
  using 439, 620
syn_hier attribute
  controlling flattening 432
  preserving hierarchy 433
  using with compile points 477
syn_isclock
  black box clock pins 390
syn_keep
  inferring Lattice PICs 613
  replicating redundant logic 430
syn_keep attribute
  preserving nets 429
  preserving shared registers 429
syn_keep directive
  effect on buffering 436
syn_macro
  specifying encrypted IP as white box 538
syn_maxfan attribute
  setting fanout limits 434
syn_noprune directive
  preserving instances 429
syn_pipeline attribute 420
syn_preserve
  effect on buffering 436
  preserving power-on for retiming 424
syn_preserve directive
  preserving FSMs from optimization 393
  preserving logic 429
syn_probe attribute 446
  inserting probes 446
  preserving nets 429
syn_reference_clock constraint (Legacy) 188
syn_replicate attribute
  using buffering 436
syn_sharing directive
  overriding default 439
syn_state_machine directive
  using with value=0 444
SYN_TCL_HOOKS environment variable 563
syn_tco attribute
  adding in SCOPE 389
syn_tco directive 388
  adding black box constraints 387
syn_tpd attribute
  adding in SCOPE 389
syn_tpd directive 388
  adding black box constraints 387
syn_tsu attribute
  adding in SCOPE 389
syn_tsu directive 388
  adding black box constraints 387
syn_useioff
  preventing flops from moving during retiming 424

© 2016 Synopsys, Inc.

SYNCore
adders 509
counters 516
FIFO compiler 484
RAMs 490
RAMs, byte-enable 497
RAMs, dual-port parameters 494
RAMs, single-port parameters 493
ROMs 504
ROMs, parameters 508
subtractors 509

synhooks
automating message filtering 232

synhooks.tcl file 563

Synopsys
FPGA product family 16

Synplify Pro synthesis tool
overview 16

synplify_pro UNIX command 22

SYNPLIFY_REMOTE_REPORT_LOCATION 221

syntax
checking source files 45

syntax check 45

synthesis
hierarchical projects 121

synthesis check 45

synthesis_on/off
using 98

SystemVerilog keywords
context help 44

T

ta file 365

Tcl
max_parallel_jobs variable 558

tcl callbacks
customizing key assignments 564

Tcl commands
batch script 552
running 556

Tcl expand
using 168

Tcl expand command
crossprobing objects 179
usage tips 173
using in SCOPE 178

Tcl files 556
creating 558
for bottom-up synthesis 562
guidelines 61
naming conventions 62
recording from commands 557
synhooks.tcl 563
using variables 560
wildcards 62

Tcl find
batch mode 176
filtering results by property 170
search patterns 168
using 168

Tcl find command
annotating properties 170
case sensitivity 169
crossprobing objects 179
database differences 178
pattern matching 169
Tcl window vs SCOPE 177
usage tips 172
useful -filter examples 172
using in SCOPE 178

Tcl Script window
crossprobing 333
message viewer 227

Tcl script window
collections compared to SCOPE 177

Tcl scripts
See Tcl files.

TCL View 157
using 157
using SCOPE 152

-tclcmd 550

team design. *See* hierarchical projects, hierarchical project management flows

Technology view
See also HDL Analyst
critical paths 361
crossprobing 327, 328
crossprobing collection objects 179
crossprobing from source file 330
filtering 274, 339
finding objects 318

finding objects with Find 316
finding objects with Hierarchy Browser 262, 314
flattening hierarchy 283, 347
general description 291
highlighting collections 184
opening 293
selecting/deselecting objects 299
setting preferences 303
state machine implementation in 443
traversing hierarchy 306

text editor
built-in 46
external 51
using 46

Text Editor view
crossprobing 328

Text Editor window
colors 49
crossprobing 49
fonts 49

text files
crossprobing 330

The Synopsys FPGA Product Family 16

third-party vendor tools
invoking 630

through constraints 163
AND lists 164
OR lists 163

time stamp, checking on files 73

timing analysis 358

timing analysis using STA 365

timing budgeting
compile points 461

timing constraints (Legacy) 188

timing exceptions, adding constraints after synthesis 369

timing exceptions, modifying with adc 369

timing failures 364

timing information commands 358

timing information in HDL views 359

timing information, critical paths 363

timing optimization 416

timing report, stand-alone 365

timing reports
specifying format options 93

timing reports, custom 365

tips
memory usage 351

to constraints
specifying 163

tool tags
creating 630
definition 630

top level
specifying 97

top-down design flow
compile point advantages 450

top-down hierarchical development flow
subprojects 108

top-down hierarchical synthesis flow 36

top-first hierarchical development flow 32

transparent instances
flattening 347
lower-level logic on multiple sheets 301

U

UNIX commands
synplify_pro 22

up-to-date checking 208
copying job logs to log file 211
limitations 212

V

vendor-specific netlists 629

Verilog
‘define statements 95
adding attributes and directives 101
adding probes 446
black boxes 382
black boxes, instantiating 382
case sensitivity for Tcl Find command 169
checking source files 45
choosing a compiler 94
creating source files 42
crossprobing from HDL Analyst view 328

© 2016 Synopsys, Inc.

defining FSMs 391
defining state machines with parameter and 'define 392
editing operations 47
extracting parameters 95
include paths, updating 75
initializing RAMs 407
mixed language files 56
specifying compiler directives 95
specifying top-level module 97
using library extensions 53

Verilog 2001
setting global option from the Project view 94
setting option per file 94

Verilog library files
using library extensions 53

Verilog macro libraries
Lattice 610

Verilog model (.vmd) 460

VHDL
adding attributes and directives 100
adding probes 446
black boxes 384
black boxes, instantiating 384
case sensitivity for Tcl Find command 169
checking source file 45
constants 97
creating source files 42
crossprobing from HDL Analyst view 328
defining FSMs 392
editing operations 47
extracting generics 97
file order in mixed designs 59
global signals in mixed designs 59
initializing RAMs with variable declarations 410
initializing with signal declarations 408
mixed language files 56
specifying top-level entity 97

VHDL files
adding library 70
adding third-party package library 70
order in project file 71
ordering automatically 71

VHDL macro libraries
Lattice 611

vi text editor 51

virtual clock, setting (Legacy) 193

W

warning messages
definition 46

warnings
feedback muxes 417
filtering 228
handling 237
sorting 228

Watch window 224
moving 224, 227
multiple implementations 84
resizing 224, 227

wildcards
effect of search scope 319
Find command (Tcl) 169
message filter 230

wildcards (Find)
examples 321
how they work 264, 319