

Synopsys FPGA Synthesis Synplify Pro for Microsemi Edition User Guide

May 2013

SYNOPSYS®

Copyright Notice and Proprietary Information

Copyright © 2013 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only.

Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

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

Destination Control Statement

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

Disclaimer

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

Registered Trademarks (®)

Synopsys, AEON, AMPS, Astro, Behavior Extracting Synthesis Technology, Cadabra, CATS, Certify, CHIPit, CoMET, CODE V, Design Compiler, DesignWare, EMBED-IT!, Formality, Galaxy Custom Designer, Global Synthesis, HAPS, HapsTrak, HDL Analyst, HSIM, HSPICE, Identify, Leda, LightTools, MAST, METeor, ModelTools, NanoSim, NOVeA, OpenVera, ORA, PathMill, Physical Compiler, PrimeTime, SCOPE, Simply Better Results, SiVL, SNUG, SolvNet, Sonic Focus, STAR Memory System, Syndicated, Synplicity, the Synplicity logo, Synplify, Synplify Pro, Synthesis Constraints Optimization Environment, TetraMAX, UMRBus, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

Trademarks (™)

AFGen, Apollo, ARC, ASAP, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, BEST, Columbia, Columbia-CE, Cosmos, CosmosLE, CosmosScope, CRITIC, CustomExplorer, CustomSim, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, DesignPower, DFTMAX, Direct Silicon Access, Discovery, Eclipse, Encore, EPIC, Galaxy, HANEX, HDL Compiler, Hercules, Hierarchical Optimization Technology, High-performance ASIC Prototyping System, HSIMplus, i-Virtual Stepper, IICE, in-Sync, iN-Tandem, Intelli, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Macro-PLUS, Magellan, Mars, Mars-Rail, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, MultiPoint, ORAengineering, Physical Analyst, Planet, Planet-PL, Polaris, Power Compiler, Raphael, RippledMixer, Saturn, Scirocco, Scirocco-i, SiWare, Star-RCXT, Star-SimXT, StarRC, System Compiler, System Designer, Taurus, Total-Recall, TSUPREM-4, VCSi, VHDL Compiler, VMC, and Worksheet Buffer are trademarks of Synopsys, Inc.

Service Marks (sm)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.

ARM and AMBA are registered trademarks of ARM Limited.

Saber is a registered trademark of SabreMark Limited Partnership and is used under license.

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

Printed in the U.S.A

May 2013

Contents

Chapter 1: Introduction

Synopsys FPGA and Prototyping Products	16
FPGA Implementation Tools	16
Synopsys FPGA Tool Features	18
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	26
-----------------------------------	----

Chapter 3: Preparing the Input

Setting Up HDL Source Files	30
Creating HDL Source Files	30
Using the Context Help Editor	32
Checking HDL Source Files	33
Editing HDL Source Files with the Built-in Text Editor	34
Setting Editing Window Preferences	38
Using an External Text Editor	40
Using Mixed Language Source Files	41
Working with Constraint Files	46
When to Use Constraint Files over Source Code	46
Using a Text Editor for Constraint Files (Legacy)	47
Tcl Syntax Guidelines for Constraint Files	48
Checking Constraint Files	49

Generating Constraint Files for Forward Annotation	50
--	----

Chapter 4: Specifying Constraints

Using the SCOPE Editor	52
Creating Constraints in the SCOPE Editor	52
Specifying SCOPE Constraints	57
Entering and Editing Scope Constraints	57
Setting Clock and Path Constraints	59
Defining Input and Output Constraints	60
Specifying Standard I/O Pad Types	62
Using the TCL View of SCOPE GUI	62
Guidelines for Entering and Editing Constraints	65
Specifying Timing Exceptions	68
Defining From/To/Through Points for Timing Exceptions	68
Defining Multicycle Paths	72
Defining False Paths	73
Finding Objects with Tcl find and expand	74
Specifying Search Patterns for Tcl find	74
Refining Tcl Find Results with -filter	74
Using the Tcl Find Command to Define Collections	76
Using the Tcl expand Command to Define Collections	78
Checking Tcl find and expand Results	79
Using Tcl find and expand in Batch Mode	80
Combining Tcl find with Other Operations	81
Using Collections	82
Comparison of Methods for Defining Collections	82
Creating and Using Scope Collections	83
Creating Collections using Tcl Commands	85
Viewing and Manipulating Collections with Tcl Commands	88
Converting SDC to FDC	92
Using the SCOPE Editor (Legacy)	93
Entering and Editing SCOPE Constraints (Legacy)	95
Specifying SCOPE Timing Constraints (Legacy)	97
Entering Default Constraints	97
Setting Clock and Path Constraints	97
Defining Clocks	100
Defining Input and Output Constraints (Legacy)	107
Defining False Paths (Legacy)	108

Chapter 5: Setting up a Logic Synthesis Project

Setting Up Project Files	112
Creating a Project File	112
Opening an Existing Project File	115
Making Changes to a Project	116
Setting Project View Display Preferences	117
Updating Verilog Include Paths in Older Project Files	119
Managing Project File Hierarchy	120
Creating Custom Folders	120
Manipulating Custom Project Folders	123
Manipulating Custom Files	124
Setting Up Implementations	126
Working with Multiple Implementations	126
Setting Logic Synthesis Implementation Options	129
Setting Device Options	129
Setting Optimization Options	132
Specifying Global Frequency and Constraint Files	133
Specifying Result Options	135
Specifying Timing Report Output	137
Setting Verilog and VHDL Options	137
Specifying Attributes and Directives	142
Specifying Attributes and Directives in VHDL	143
Specifying Attributes and Directives in Verilog	145
Specifying Attributes Using the SCOPE Editor	146
Specifying Attributes in the Constraints File	149
Searching Files	150
Identifying the Files to Search	151
Filtering the Files to Search	151
Initiating the Search	152
Search Results	152
Archiving Files and Projects	153
Archive a Project	153
Un-Archive a Project	157
Copy a Project	160

Chapter 6: Inferring High-Level Objects

Defining Black Boxes for Synthesis	166
Instantiating Black Boxes and I/Os in Verilog	166

Instantiating Black Boxes and I/Os in VHDL	168
Adding Black Box Timing Constraints	170
Adding Other Black Box Attributes	174
Defining State Machines for Synthesis	175
Defining State Machines in Verilog	175
Defining State Machines in VHDL	176
Specifying FSMs with Attributes and Directives	177
Inferring RAMs	180
Inference Versus Instantiation	180
Basic Guidelines for Coding RAMs	181
Specifying RAM Implementation Styles	185
Initializing RAMs	186
Initializing RAMs in Verilog	186
Initializing RAMs in VHDL	187

Chapter 7: Specifying Design-Level Optimizations

Tips for Optimization	192
General Optimization Tips	192
Optimizing for Area	193
Optimizing for Timing	194
Retiming	196
Controlling Retiming	196
Retiming Example	198
Retiming Report	199
How Retiming Works	200
Preserving Objects from Optimization	203
Using syn_keep for Preservation or Replication	204
Controlling Hierarchy Flattening	207
Preserving Hierarchy	207
Optimizing Fanout	209
Setting Fanout Limits	209
Controlling Buffering and Replication	211
Sharing Resources	213
Inserting I/Os	218
Optimizing State Machines	218
Deciding when to Optimize State Machines	219
Running the FSM Compiler	220
Running the FSM Explorer	224

Inserting Probes	227
Specifying Probes in the Source Code	227
Adding Probe Attributes Interactively	228

Chapter 8: Synthesizing and Analyzing Results

Synthesizing Your Design	232
Running Logic Synthesis	232
Using Up-to-date Checking for Job Management	232
Checking Log Results	237
Viewing the Log File	237
Analyzing Results Using the Log File Reports	241
Using the Watch Window	242
Handling Messages	244
Checking Results in the Message Viewer	244
Filtering Messages in the Message Viewer	246
Filtering Messages from the Command Line	248
Automating Message Filtering with a Tcl Script	249
Log File Message Controls	251
Handling Warnings	254

Chapter 9: Analyzing with HDL Analyst and FSM Viewer

Working in the Schematic Views	256
Differentiating Between the Views	257
Opening the Views	257
Viewing Object Properties	258
Selecting Objects in the RTL/Technology Views	263
Working with Multisheet Schematics	265
Moving Between Views in a Schematic Window	266
Setting Schematic View Preferences	267
Managing Windows	269
Exploring Design Hierarchy	270
Traversing Design Hierarchy with the Hierarchy Browser	270
Exploring Object Hierarchy by Pushing/Popping	271
Exploring Object Hierarchy of Transparent Instances	276
Finding Objects	278
Browsing to Find Objects in HDL Analyst Views	278
Using Find for Hierarchical and Restricted Searches	280
Using Wildcards with the Find Command	283
Combining Find with Filtering to Refine Searches	288
Using Find to Search the Output Netlist	288

Crossprobing	291
Crossprobing within an RTL/Technology View	291
Crossprobing from the RTL/Technology View	292
Crossprobing from the Text Editor Window	294
Crossprobing from the Tcl Script Window	297
Crossprobing from the FSM Viewer	298
Analyzing With the HDL Analyst Tool	299
Viewing Design Hierarchy and Context	300
Filtering Schematics	303
Expanding Pin and Net Logic	305
Expanding and Viewing Connections	309
Flattening Schematic Hierarchy	311
Minimizing Memory Usage While Analyzing Designs	315
Using the FSM Viewer	315

Chapter 10: Analyzing Timing

Analyzing Timing in Schematic Views	322
Viewing Timing Information	322
Annotating Timing Information in the Schematic Views	323
Analyzing Clock Trees in the RTL View	325
Viewing Critical Paths	325
Handling Negative Slack	328
Generating Custom Timing Reports with STA	329
Using Analysis Design Constraints	332
Scenarios for Using Analysis Design Constraints	333
Creating an ADC File	334
Using Object Names Correctly in the adc File	338
Using Auto Constraints	339
Results of Auto Constraints	341

Chapter 11: Optimizing for Microsemi Designs

Optimizing Microsemi Designs	346
Using Predefined Microsemi Black Boxes	346
Using Smartgen Macros	347
Working with Radhard Designs	347
Specifying syn_radhardlevel in the Source Code	348

Chapter 12: Working with Synthesis Output

Passing Information to the P&R Tools	352
--	-----

Specifying Pin Locations	352
Specifying Locations for Microsemi Bus Ports	353
Specifying Macro and Register Placement	353
Generating Vendor-Specific Output	354
Targeting Output to Your Vendor	354
Customizing Netlist Formats	355

Chapter 13: Running Post-Synthesis Operations

Running P&R Automatically after Synthesis	358
Working with the Identify Tools	359
Launching from the Synplify Pro Tool	359
Handling Problems with Launching Identify	361
Using the Identify Tool	362
Using Compile Points with the Identify Tool	364
Simulating with the VCS Tool	366

Chapter 14: Working with IP Input

Generating IP with SYNCore	372
Specifying FIFOs with SYNCore	372
Specifying RAMs with SYNCore	378
Specifying Byte-Enable RAMs with SYNCore	386
Specifying ROMs with SYNCore	392
Specifying Adder/Subtractors with SYNCore	397
Specifying Counters with SYNCore	404
The Synopsys FPGA IP Encryption Flow	410
Overview of the Synopsys FPGA IP Flow	410
Encryption and Decryption	411
Working with Encrypted IP	416
Encrypting Your IP	416
Encrypting IP with the encryptP1735.pl Script	418
Encrypting IP with the encryptIP Script	419
Specifying the Script Output Method	421
Preparing the IP Package	422
Using Hyper Source	426
Using Hyper Source for Prototyping	426
Using Hyper Source for IP Designs	426
Threading Signals Through the Design Hierarchy of an IP	427

Chapter 15: Working with Compile Points

Compile Point Basics	432
Advantages of Compile Point Design	432
Manual Compile Points	434
Nested Compile Points	435
Compile Point Types	436
Compile Point Synthesis Basics	441
Compile Point Constraint Files	441
Interface Logic Models	444
Interface Timing for Compile Points	444
Compile Point Synthesis	447
Incremental Compile Point Synthesis	450
Forward-annotation of Compile Point Timing Constraints	451
Synthesizing Compile Points	451
The Manual Compile Point Flow	452
Creating a Top-Level Constraints File for Compile Points	454
Defining Manual Compile Points	455
Setting Constraints at the Compile Point Level	458
Analyzing Compile Point Results	460
Using Compile Points with Other Features	462
Combining Compile Points with Multiprocessing	462
Resynthesizing Incrementally	463
Resynthesizing Compile Points Incrementally	463

Chapter 16: Optimizing Processes for Productivity

Using Batch Mode	468
Running Batch Mode on a Project File	468
Running Batch Mode with a Tcl Script	469
License Queuing	470
Working with Tcl Scripts and Commands	474
Using Tcl Commands and Scripts	474
Generating a Job Script	475
Setting Number of Parallel Jobs	475
Creating a Tcl Synthesis Script	476
Using Tcl Variables to Try Different Clock Frequencies	478
Using Tcl Variables to Try Several Target Technologies	479
Running Bottom-up Synthesis with a Script	480
Automating Flows with synhooks.tcl	481

Chapter 17: Using Multiprocessing

Multiprocessing With Compile Points	486
Setting Maximum Parallel Jobs	486
License Utilization	487

CHAPTER 1

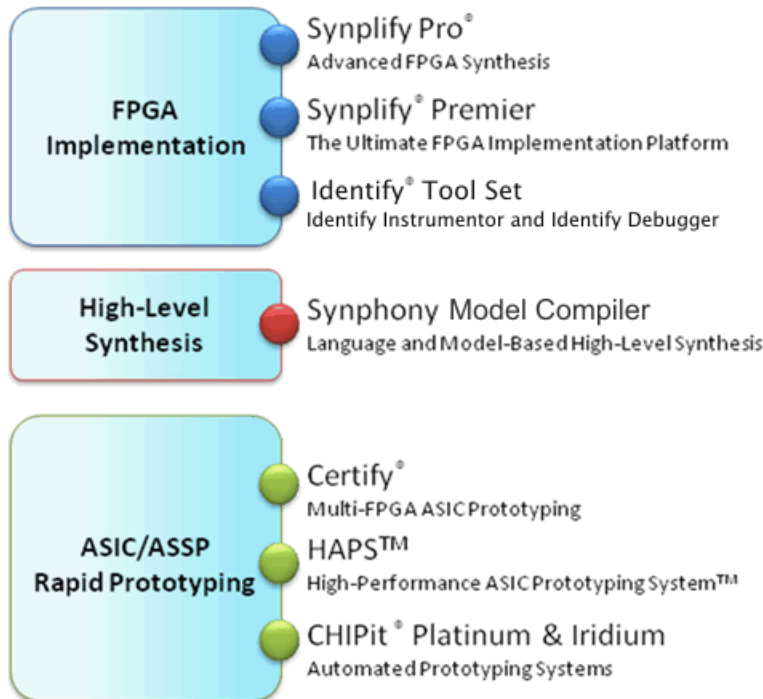
Introduction

This introduction to the Synplify Pro[®] software 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

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 Product

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 Product

The Synplify Premier solution is a superset of the Synplify Pro product functionality and is the ultimate FPGA implementation and debug environment. It provides a comprehensive suite of tools and technologies for advanced FPGA designers, as well as ASIC prototypers targeting single FPGA-based prototypes. The Synplify Premier software is a technology independent solution that addresses the most challenging aspects of FPGA design including timing closure, logic verification, IP usage, ASIC compatibility, DSP implementation, debug, and tight integration with FPGA vendor back-end tools.

The Synplify Premier product offers FPGA designers and ASIC prototypers, targeting single FPGA-based prototypes, with the most efficient method of design implementation and debug. The Synplify Premier software provides in-system verification of FPGAs, dramatically accelerates the debug process, and provides a rapid and incremental method for finding elusive design problems.

Features exclusively supported in the Synplify Premier tool are the following:

- Fast and Enhanced Synthesis Modes
- Physical Synthesis
- Design Planning (Optional)
- DesignWare Support
- Integrated RTL Debug (Identify Tool Set)
- Power Switching Activity (SAIF Generation)

Synopsys FPGA Tool Features

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

	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
Code Analysis				
SCOPE® Spreadsheet	x	x	x	x
HDL Analyst®	Option	x	x	x
Timing Analyzer – Point-to-point		x	x	x
FSM Viewer		x	x	x
Crossprobing		x	x	x
Probe Point Creation		x	x	x
Physical Design				
Design Plan File				x
Logic Assignment to Regions				x
Area Estimation and Region Capacity				x
Pin Assignment				x

	Synplify	Synplify Pro	Synplify Premier	Synplify Premier DP
Physical Synthesis Optimizations				x
Graph-based Physical Synthesis			x	x
Physical Analyst			x	x
Prototyping			x	x
Synopsys DesignWare Foundation Library			x	x
Runtime Advantages				
Enhanced Optimization			x	x
Fast Synthesis			x	x
Team Design				
Mixed Language Design		x	x	x
Compile Points		x	x	x
True Batch Mode (Floating licenses only)		x	x	x
GUI Batch Mode (Floating licenses)	x	x	x	x
Batch Mode Post-synthesis P&R Run	-	x	x	x
Back-annotation of P&R Data	-	-	-	x
Formal Verification Flow		x	x (Physical synthesis disabled)	x (Physical synthesis disabled)
Identify Integration	Limited	x	x	x
Back-annotation of P&R Data				x
Design Environment				
Technical Resource Center	x	x	x	x
Text Editor View	x	x	x	x

	Synplify	Synplify Pro	Synplify Premier	Synplify Premier DP
Watch Window		x	x	x
Message Window		x	x	x
Tcl Window		x	x	x
Workspaces		x	x	x
Multiple Implementations		x	x	x
Vendor Technology/Family Support	x	x	Limited	Limited

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 a reference manual and a tutorial. 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 [User Interface Overview](#) chapter in the *Synopsys FPGA Synthesis 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

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

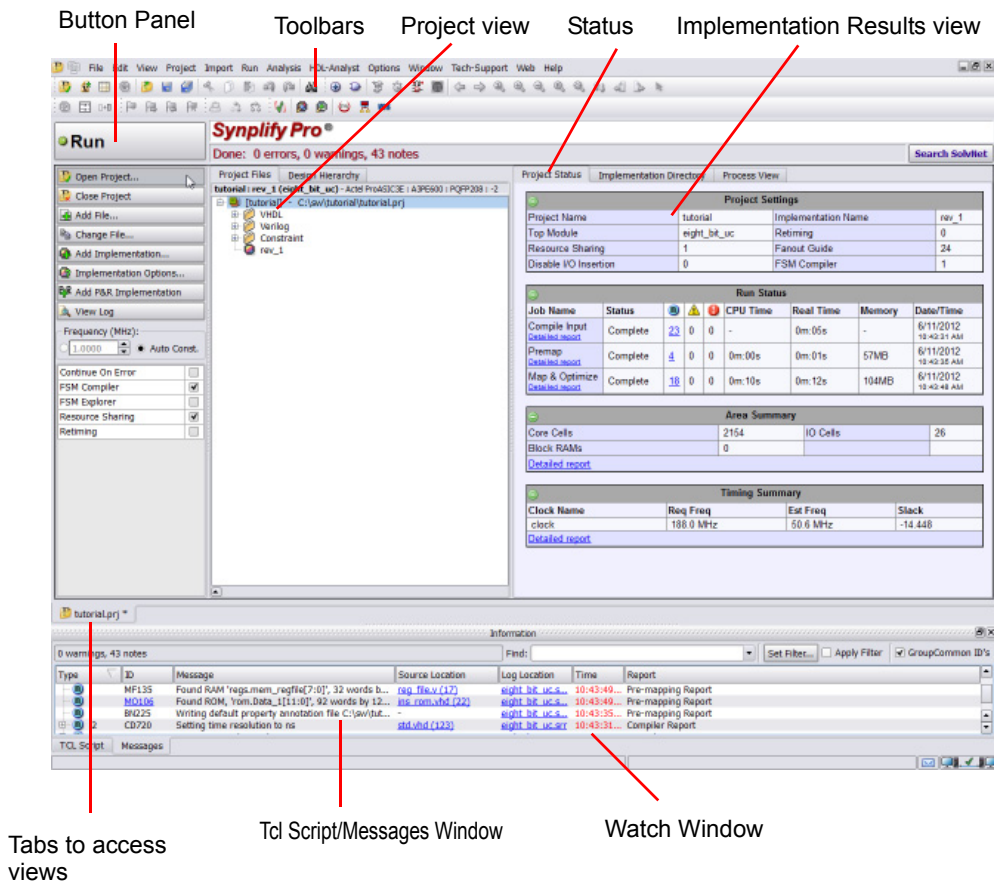
Before you call Synopsys Support, look through the documented information. You can access the information online from the Help menu, or refer to the PDF version. The following table shows you how the information is organized.

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

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



CHAPTER 2

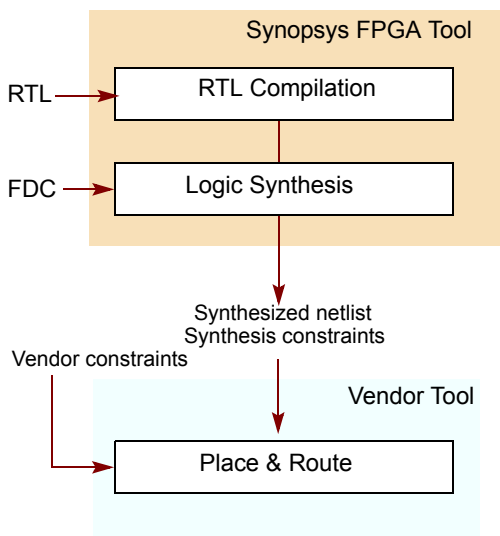
FPGA Synthesis Design Flows

This chapter describes the [Logic Synthesis Design Flow](#), on page 26.

Logic Synthesis Design Flow

The Synopsys FPGA tools synthesize logic by first compiling the RTL source, and then doing logical mapping and optimizations. After logic synthesis, you get a vendor-specific netlist and constraint file that you 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. You can use the Synplify Pro synthesis software for this flow. 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

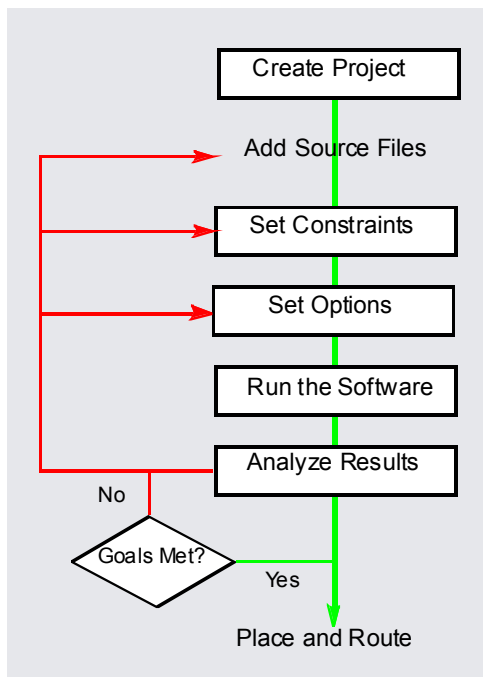
For a design flow with step-by-step instructions based on specific design data, download the tutorial from the website. The following steps summarize the process, which is also illustrated in the figure that follows.

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 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:



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 30
- [Using Mixed Language Source Files](#), on page 41
- [Working with Constraint Files](#), on page 46

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 112](#). 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 30](#)
- [Using the Context Help Editor, on page 32](#)
- [Checking HDL Source Files, on page 33](#)
- [Editing HDL Source Files with the Built-in Text Editor, on page 34](#)
- [Using an External Text Editor, on page 40](#)
- [Setting Editing Window Preferences, on page 38](#)

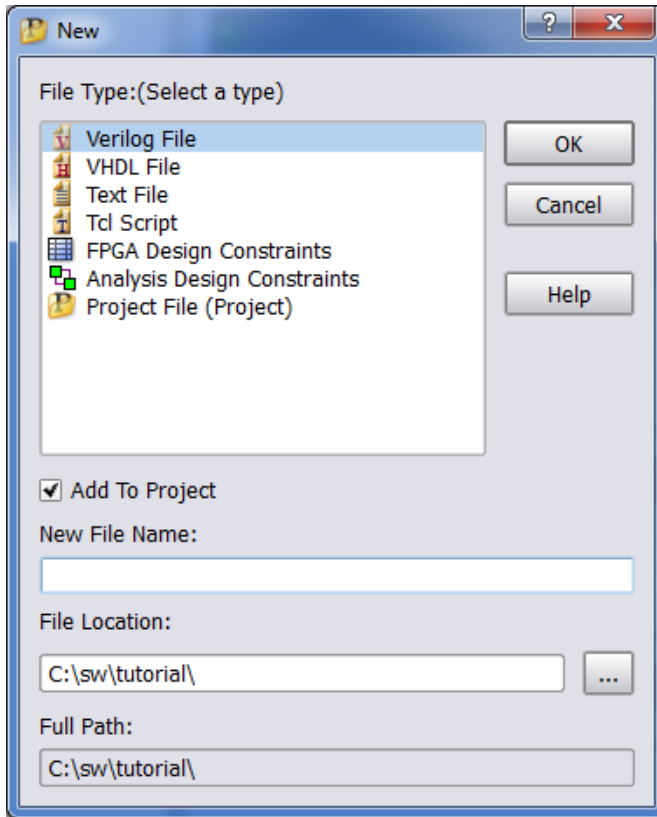
Creating HDL Source Files

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 33](#) and [Editing HDL Source Files with the Built-in Text Editor, on page 34](#)).

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 41](#).

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 Verilog designs that contain SystemVerilog constructs in the source file. For more information, see [Using the Context Help Editor, on page 32](#).

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 34](#) 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 ().

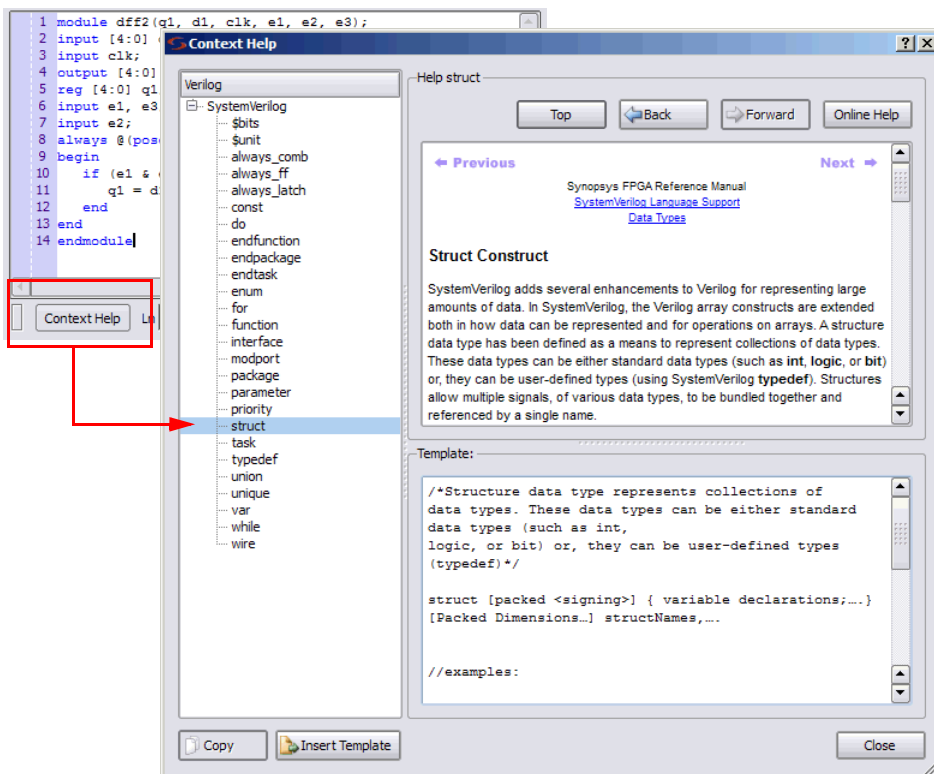
Once you have created a source file, you can check that you have the right syntax, as described in [Checking HDL Source Files, on page 33](#).

Using the Context Help Editor

When you create or open a Verilog design file, use the Context Help button displayed at the bottom of the window to help you code with SystemVerilog constructs in the source file. This feature is currently supported for some of the SystemVerilog constructs.

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 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 shown in the template window is inserted into your SystemVerilog file at the location of the cursor. This allows you to easily insert code 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 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.
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 40](#).

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 38](#).

```

1 module clock_generator (CLK, RESET);
2 /* synthesis syn_noprune = 1 */
3 /* synthesis syn_black_box */
4 output CLK, RESET;
5 endmodule
6
7 module WR_STATE (CLK, RESET, WR_CMD, SNDR_RDY, RCVR_RDY, WREN, CNTEN);
8 output WREN, CNTEN;
9 input CLK, RESET, SNDR_RDY, RCVR_RDY, WR_CMD;
10
11 parameter IDLE = 5'h1, WRO = 5'h2, WR1 = 5'h4;
12 parameter WR2 = 5'h8, WR3 = 5'h10;
13 reg [4:0] present_state, next_state;
14 reg WREN, CNTEN;
15
16 always @ (posedge CLK or posedge RESET)

```

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.

To...	Do...
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 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.

To...	Do...
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 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 crossprobe 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 294](#) 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.

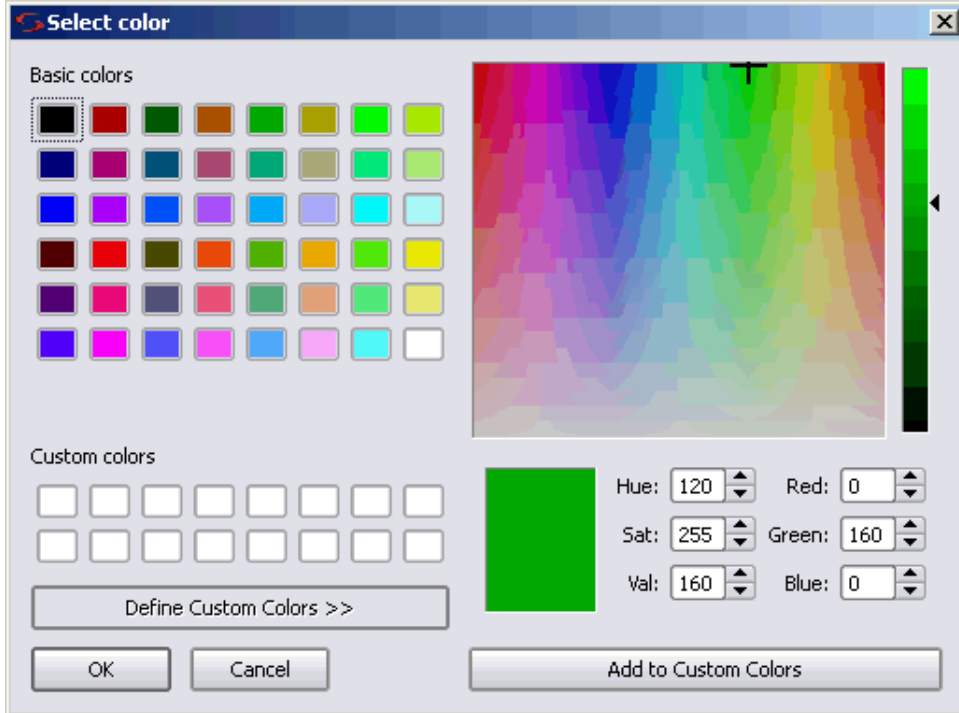
1. Select Options->Editor Options and either Synopsys Editor or External Editor. For more information about the external editor, see [Using an External Text Editor, on page 40](#).
2. Then depending on the type of file you open, you can to set the background, syntax coloring, and font preferences to 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.

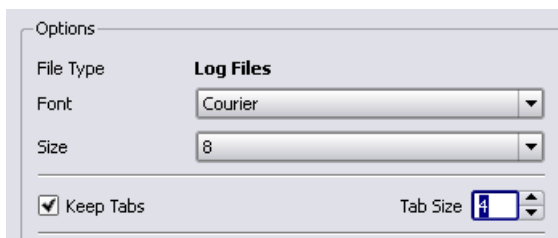
3. You can set syntax colors for some common syntax options, such as keywords, strings, and comments. For example in the log file, warnings and errors can be color-coded for easy recognition.

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



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**.

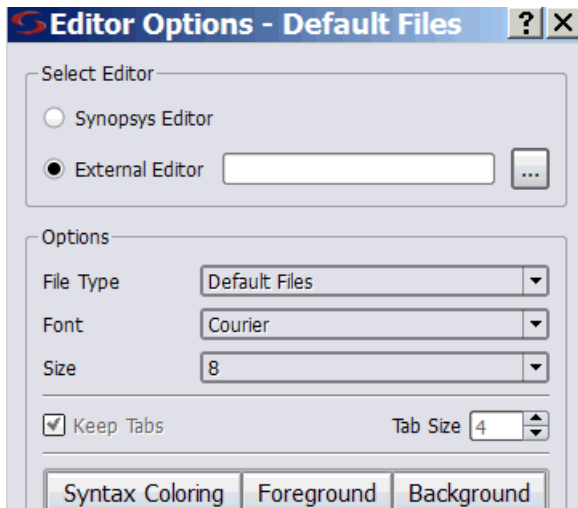


6. Click OK on the Editor Options form.

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 34](#).

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.



- 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.

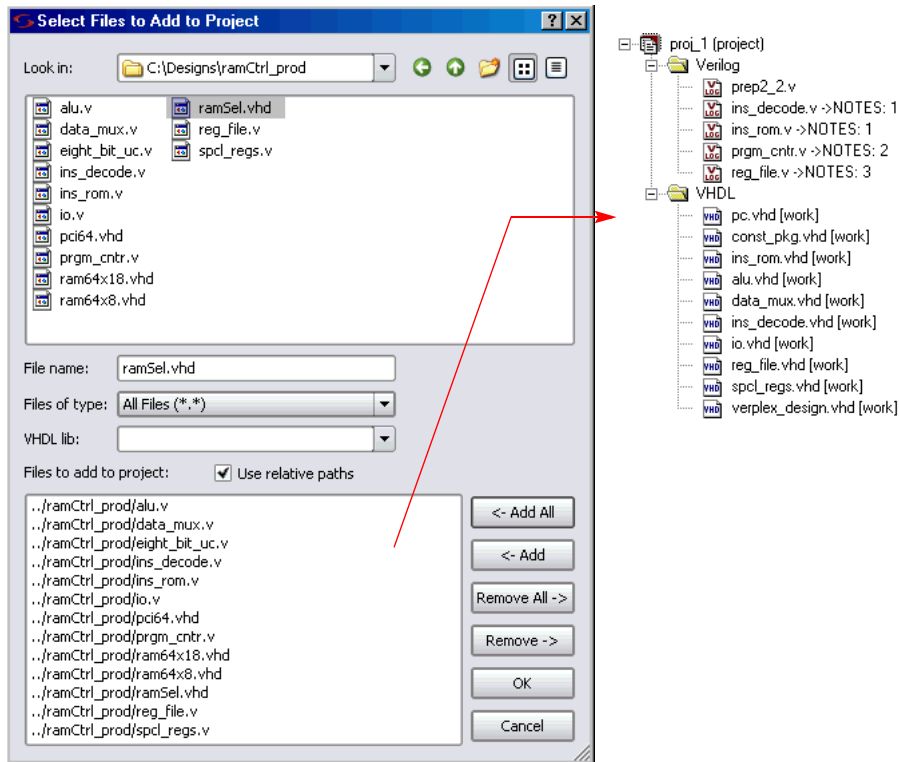
Using Mixed Language Source Files

With the Synplify Pro 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*.

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, *.vhd, *.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 116](#).



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 129](#).
 - 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 159](#).

The screenshot shows the Synplify Pro configuration window with two tabs: Verilog and VHDL.

Verilog Tab:

- Top Level Module:** eight_bit_uc
- Verilog Language:**
 - ☒ Verilog 2001
 - ☐ System Verilog
- ☒ Push Tristates
- ☐ Allow Duplicate Modules
- ☐ Multiple File Compilation Unit
- Compiler Directives and Parameters:**

Parameter Name	Value
- Compiler Directives:** e.g. SIZE=8
- Extract Parameters** button

VHDL Tab:

- Top Level Entity:** eight_bit_uc
- Default Enum Encoding:** default
- ☒ Push Tristates
- ☐ Synthesis On/Off Implemented as Translate On/Off
- ☐ VHDL 2008

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 129](#).
 - 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 44](#) 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

```
Entity abc is
Generic
(
  Number_Bits      : integer := 0;
  Divide_Bit       : boolean  := False;
);
```

Verilog Instantiation

```
abc #(
  .Number_Bits (16),
  .Divide_Bit  (1'b0)
```

```
)
```

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. Note that 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 57](#)), 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 46
- `i:statemod.statereg[*]`, on page 49
- [Tcl Syntax Guidelines for Constraint Files](#), on page 48
- [Generating Constraint Files for Forward Annotation](#), on page 50

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 52](#) 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 142](#) for more information on adding directives to source code.

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 46](#).

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 Constraint File (SCOPE) option. Type a name for the file and click OK.
 - 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 48](#).
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 149](#) for more information.
5. Save the file.
6. Add the file to the project as described in [Making Changes to a Project, on page 116](#), and run synthesis.

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 {}.
 - 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 440](#) of the *Reference Manual*.

Generating Constraint Files for Forward Annotation

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. Set attributes to control forward annotation.

To forward-annotate timing constraints, set the clock period, max delay, input delay, output delay, multiple-cycle paths, and false paths in the SCOPE interface.

For details about these attributes, see the *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 `set_input_delay`, `set_output_delay`, `set_clock` (including the `set_clock` constraints generated by auto constraining), `set_multicycle_path`, `set_false_path`, `set_max_delay`, and global-frequency constraints into corresponding commands in the `filename_sdc.sdc` file for Microsemi.

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

CHAPTER 4

Specifying Constraints

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

- [Using the SCOPE Editor](#), on page 52
- [Specifying SCOPE Constraints](#), on page 57
- [Specifying Timing Exceptions](#), on page 68
- [Finding Objects with Tcl find and expand](#), on page 74
- [Using Collections](#), on page 82
- [Converting SDC to FDC](#), on page 92
- [Using the SCOPE Editor \(Legacy\)](#), on page 93

The following chapters discuss related information:

- [Chapter 5, *Constraints*](#), for an overview of constraints
- [Chapter 6, *SCOPE Constraints Editor*](#), for a description of the SCOPE editor

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 92](#).
- For new designs, use the SCOPE editor. See [Creating Constraints in the SCOPE Editor, on page 52](#) for more information.

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.

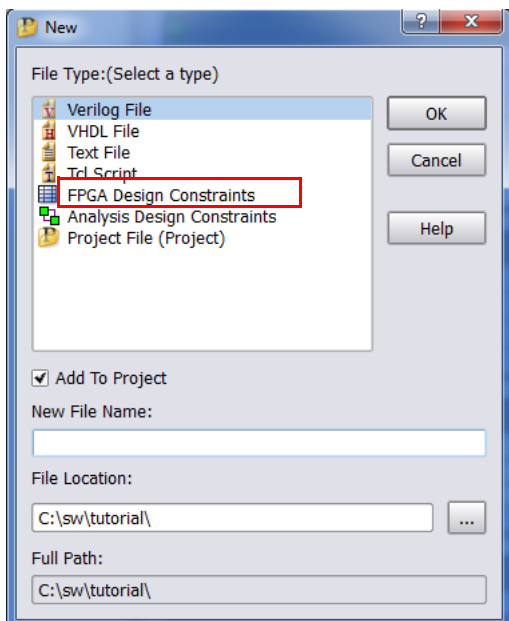
1. To create a new constraint file, follow these steps:
 - Compile the design (F7).
 - Open the SCOPE window by:

Clicking the SCOPE icon in the toolbar ().

This brings up the New Constraint File dialog box.

OR

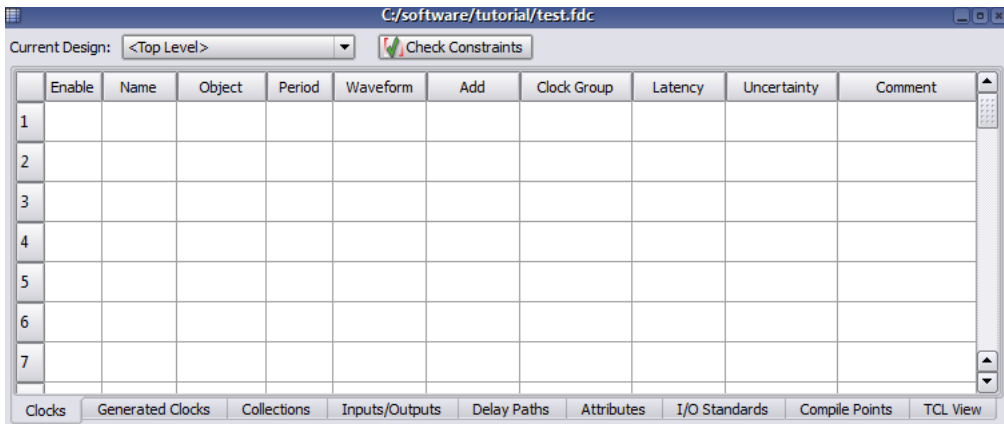
Pressing Ctrl-n or selecting File -> New. This brings up the New dialog box.



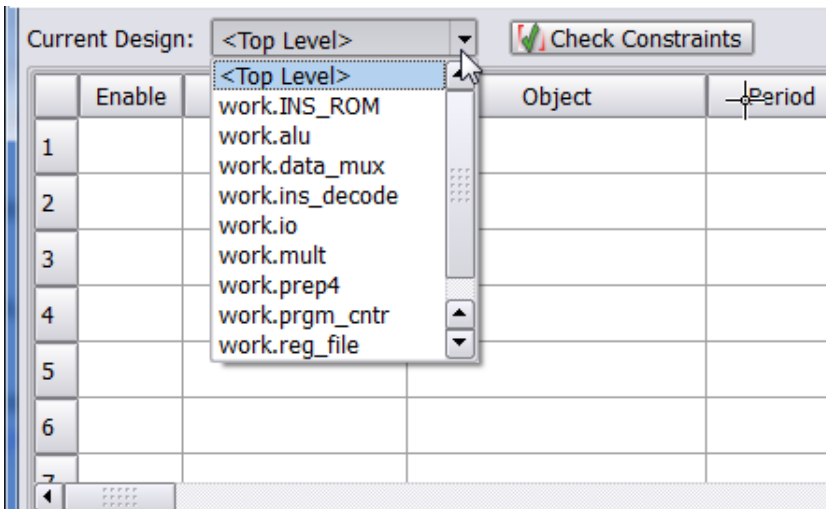
2. To open an existing file, do one of the following:

- 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.



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 57](#).

For information about ways to enter constraints within the SCOPE editor, see [Guidelines for Entering and Editing Constraints, on page 65](#).

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 62](#).
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.

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_89\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:Norm
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{idh}} -from {b:ena}
set_false_path -from $fdc_cmd_2 -to {i:abc.def.g_reg} -through {n:bar}
set_false_path -from {b:boing} -to {c:dc:clk0fx_derived_clock[2]} -through {n:fudge}
set_false_path -from {c:dc:clk0fx_derived_clock} -to {i:abc.def.g_reg[0] i:abc} -through {n:fudge}
##### END "Delay Paths"
```


Specifying SCOPE Constraints

Timing constraints define the performance goals for a design. The FPGA synthesis tool supports a subset of the Synopsys FDC Standard timing constraints (for example, `create_clock`, `set_input_delay`, and `set_false_path`). For additional support, see [Synopsys Standard Timing Constraints, on page 58](#).

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 68](#).

For information about collections, see [Using Collections, on page 82](#).

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

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
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). For complete information about using the FPGA timing constraints with your project, see:

- For specific information on individual constraint options and arguments, see the *Synthesis Commands* PDF document at https://solvnet.synopsys.com/dow_retrieve/G-2012.06/manpages/ni/syn2.pdf.
- For information on which options and arguments are supported, see the *FDC Standard for FPGA Synthesis* document on SolvNet.
- For general information on the Design Constraints Format, see the *Using the Synopsys Design Constraints Format Application Note* on SolvNet.

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	<p>Select the clock object (Clock).</p> <p>Specify a clock name (Clock Alias), if required.</p> <p>Type a period (Period).</p> <p>Change the rise and fall edge times for the clock waveforms of the clock in nanoseconds, if needed.</p> <p>Change the default clock group, if needed</p> <p>Check the Enabled box.</p> <p>See Defining Clocks, on page 100 for information about clock attributes.</p>
Generated Clocks	Generated Clocks	<p>Select the generated clock object.</p> <p>Specify the master clock source (a clock source pin in the design).</p> <p>Specify whether to use invert for the generated clock signal.</p> <p>Specify whether to use: edges, divide_by, or multiply_by.</p> <p>Check the Enabled box.</p>
Input/output delays	Inputs/Outputs	<p>See Defining Input and Output Constraints, on page 60 for information about setting I/O constraints.</p>
Maximum path delay	Delay Paths	<p>Select the Delay Type path of Max Delay.</p> <p>Select the start/from point for either a port or register (From/Through). See Defining From/To/Through Points for Timing Exceptions, on page 68 for more information.</p> <p>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).</p> <p>Set the delay value (Max Delay).</p> <p>Check the Enabled box.</p>
Multicycle paths	Delay Paths	<p>See Defining Multicycle Paths, on page 72.</p>

To define...	Pane	Do this to set the constraint...
False paths	Delay Paths	See Defining False Paths, on page 73 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: <ul style="list-style-type: none"> • 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 97](#), 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.
 - Use the `set_clock_route_delay` command to translates 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.
 - 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 377](#).

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

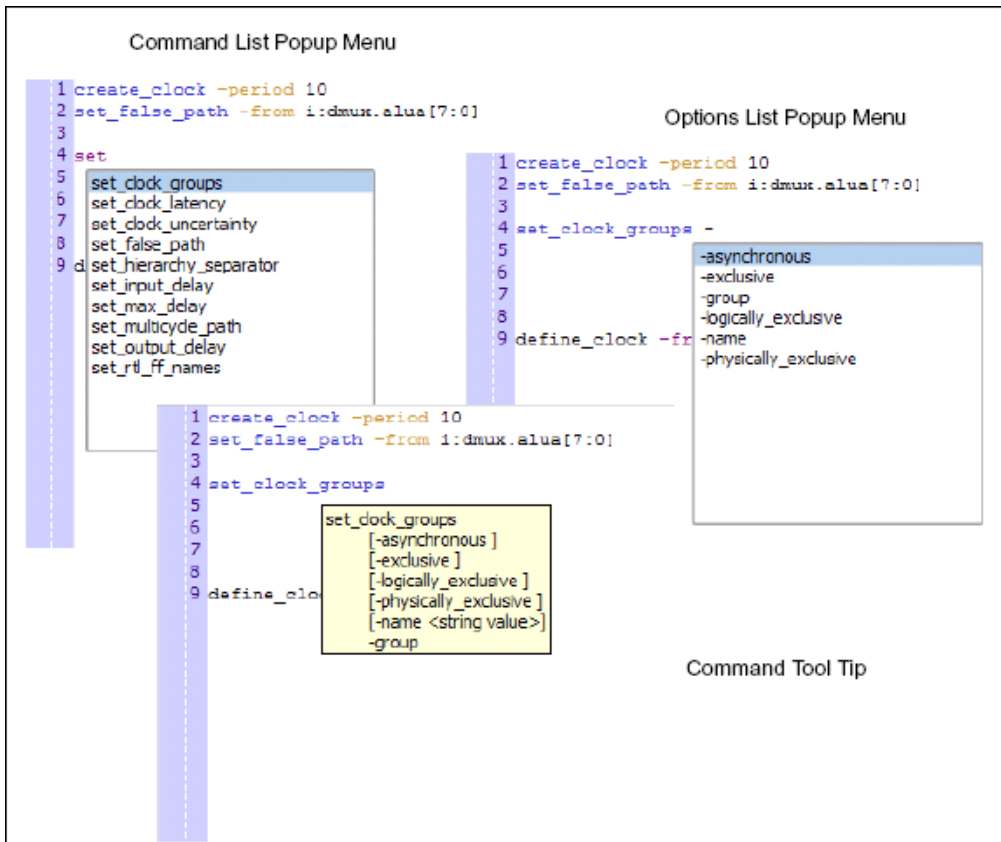
The TCL View of the SCOPE GUI is an advanced text file editor used for FPGA timing and design constraints. This text editor provides the following capabilities:

- 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

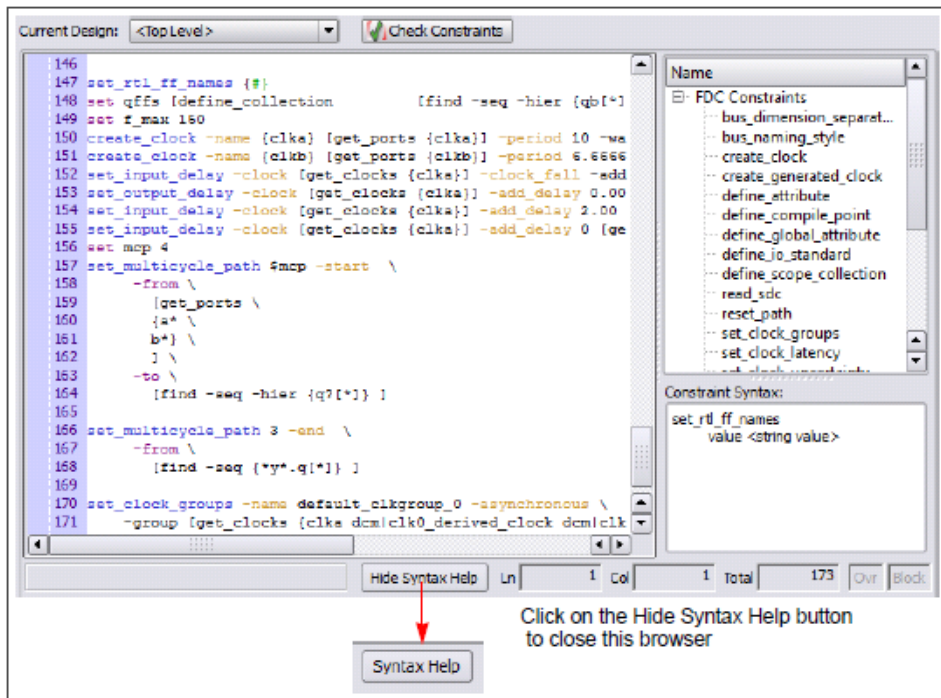
- Checks command syntax and uses color indicators that
 - Validates commands and command syntax
 - Distinguishes between FPGA design constraints and SCOPE legacy constraints
- Allows for standard editor commands, such as copy, paste, comment/un-comment a group of lines, and highlighting of keywords

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.



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



- 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 the FDC constraint file to open it in the text editor.

Guidelines for Entering and Editing Constraints

- 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.

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...
<i>v:design_name</i>	hierarchies or “views” (modules)
<i>c:clock_name</i>	clocks
<i>i:instance_name</i>	instances (blocks)
<i>p:port_name</i>	ports (off-chip)
<i>t:pin_name</i>	hierarchical ports, and pins of instantiated cells
<i>b:name</i>	bits of a bus (port)
<i>n:net_name</i>	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 57](#).

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.

- Make sure the object identifiers map as expected:

Synopsys Design Constraint Identifiers	FPGA Synthesis Constraint Identifiers
get_clocks (Wildcards are not supported)	c:
get_registers	r:
get_nets	n:
get_ports	p:
get_cells	i:
get_pins	t:

4. 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 { }

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 16, *Batch Commands and Scripts*](#) in the *Reference Manual*.

- [Defining From/To/Through Points for Timing Exceptions](#), on page 68
- [Defining Multicycle Paths](#), on page 72
- [Defining False Paths](#), on page 73

For information about resolving timing exception conflicts, see [Conflict Resolution for Timing Exceptions, on page 393](#) 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 393](#) 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 bi-directional 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:

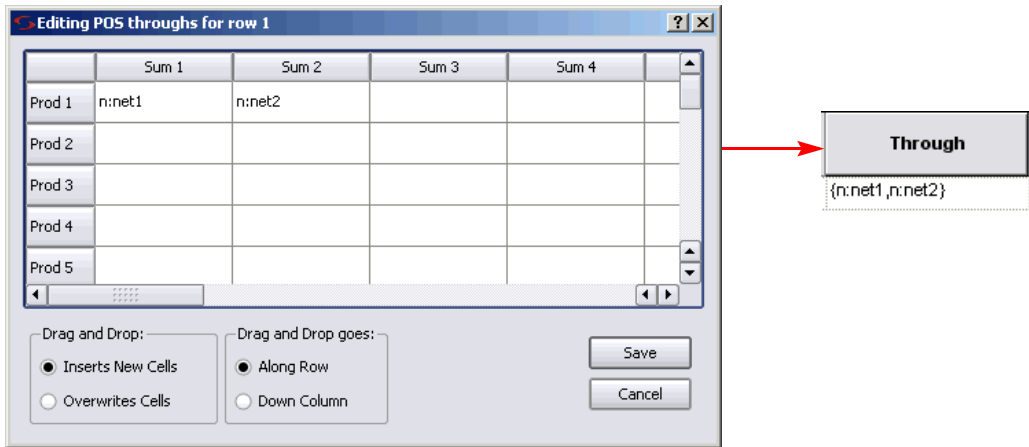
```
set_multicycle_path -from {i:aq i:bq} 2
```

```
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 bi-directional 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.

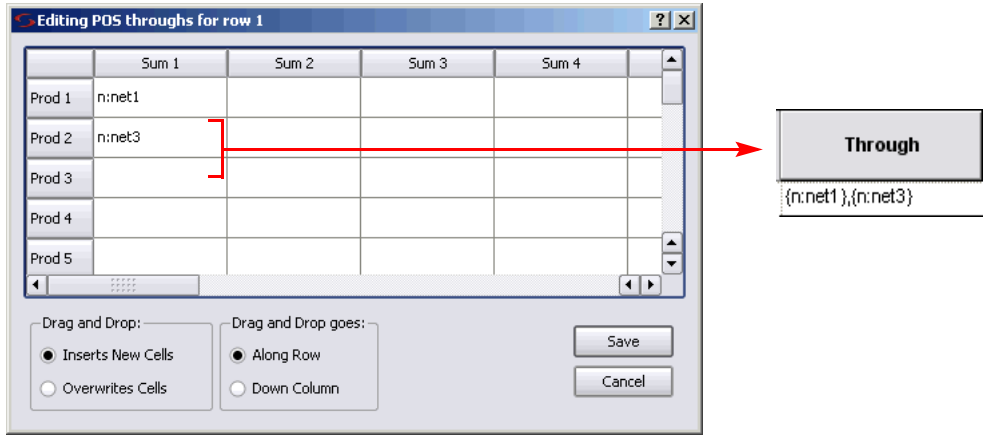
- To specify an OR when constraining a list of through points, you can type the net names in the Through field or you can use the POS UI. To do this:
 - Click in the Through field and click the arrow. This opens the Product of Sums interface.
 - Either type the first net name in a cell in a Prod row 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.



- 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*.

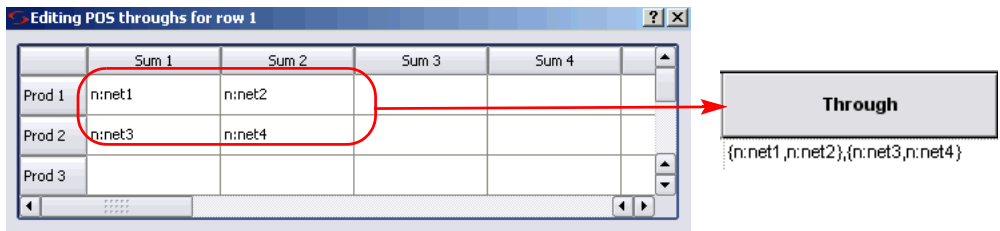
- To specify an AND when constraining a list of through points, type the names in the **Through** field or do the following:
 - Open the Product of Sums interface as described previously.
 - Either type the first net name in the first cell in a Sum column or drag the net from an HDL Analyst view into 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.



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 68](#) 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:

$$\text{multicycle_distance} = \text{clock_distance} + (\text{cycles} - 1) * \text{reference_clock_period}$$

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 68](#) 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.

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 74
- [Refining Tcl Find Results with -filter](#), on page 74
- [Using the Tcl Find Command to Define Collections](#), on page 76
- [Using the Tcl expand Command to Define Collections](#), on page 78
- [Checking Tcl find and expand Results](#), on page 79
- [Using Tcl find and expand in Batch Mode](#), on page 80

Once you have located objects with the find or expand commands, you can group them into collections, as described in [Using Collections, on page 82](#), 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 1129](#) of the *Reference Manual*.

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.`

Option	Value
Fanout Guide	10000
Disable I/O Insertion	<input type="checkbox"/>
Update Compile Point Timing Data	<input type="checkbox"/>
Read Write Check on RAM	<input type="checkbox"/>
Annotated Properties for Analyst	<input checked="" type="checkbox"/>
Resolve Mixed Drivers	<input type="checkbox"/>

Click on an option for description

- 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 *searchPattern*, and then further filters the found objects 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 74
find syntax details	Tcl find Command, on page 1128 in the Reference Manual
find -filter syntax details	Tcl Find -filter Command, on page 1138 in the Reference Manual

Examples of Useful Find -filter Commands

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

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 82](#).

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 74](#)). 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 1129](#) of the *Reference Manual*.

2. Check your find constraints. See [Checking Tcl find and expand Results, on page 79](#).
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 88](#):
 - 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 88](#).
5. Combine the Tcl `find` command with other commands:

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

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 [Tcl expand Command, on page 1144](#) in the *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}
```

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.

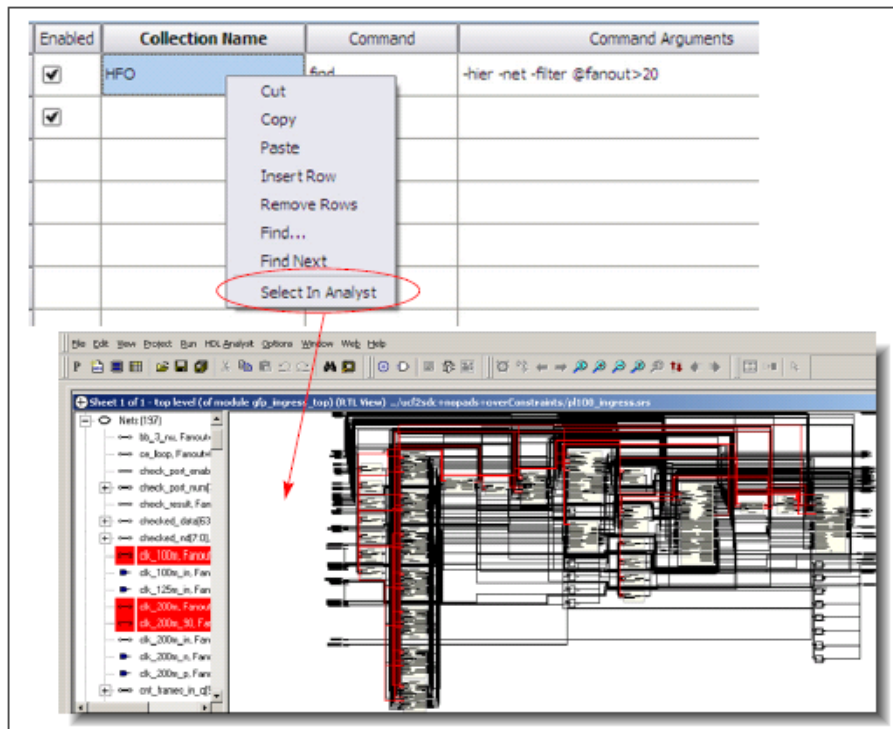
```
expand -pin -from {t:i_and3.z}
```

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 85](#)).

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.



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.

This batch script uses the find command to find DSP48Es and negative slack, and then writes out the results to separate text files:

```
open_design implementation_a/top.srm
set find_DSP48Es [find -hier -inst{*} -filter @view == {DSP48E*}]
set find_negslack [find -hier -seq -inst {*} -filter @slack
< {-0.0}]
```



```
c_print $find_DSP48Es -file DSP48Es.txt  
c_print -prop slack -prop view $find_negslack -file negslack.txt
```

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. The `DSP48s.txt` file lists the DSP48Es, and the `negslack.txt` file lists the instances with negative slack.

Combining Tcl find with Other Operations

When combined with or embedded in other commands, the find command allows you to isolate or group objects, and then manipulate them easily.

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.

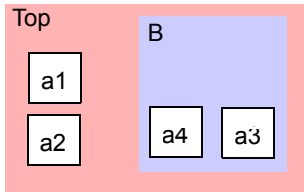
- [Comparison of Methods for Defining Collections](#), on page 82
- [Creating and Using Scope Collections](#), on page 83
- [Creating Collections using Tcl Commands](#), on page 85
- [Viewing and Manipulating Collections with Tcl Commands](#), on page 88

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.



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 85](#)) for the reasons described in [Comparison of Methods for Defining Collections, on page 82](#).

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.

	Enabled	Collection Name	Command	Command Arguments	Comment	
1	<input checked="" type="checkbox"/>	find_all	find	-hier -inst {*usbSlaveControl.u_endpMux.*}		
2	<input checked="" type="checkbox"/>	find_reg	find	-hier -seq {*usbSlaveControl.u_endpMux.*}		
3	<input checked="" type="checkbox"/>	find_comb	find	-hier -inst {*usbSlaveControl.u_endpMux.*} -filter @is_combination		
4						
5						
6						

Collections

- In the Commands column, enter the command. See the *Reference Manual* for complete syntax details. Additional information about specifying search patterns is described in [Specifying Search Patterns for Tcl find, on page 74](#) and [Specifying Search Patterns for Tcl find, on page 74](#).

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 combinatorial 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 88

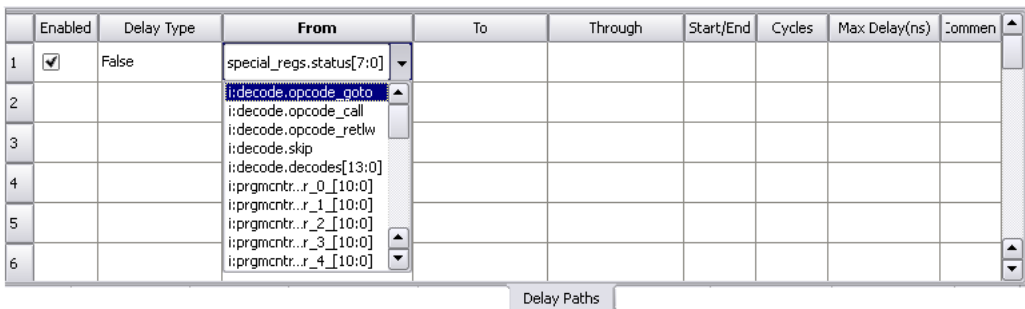
2. To create a collection that is made up of other collections, do this:
 - Define the collections as described in the previous step. These collections must be defined before you can concatenate them or add them together in a new collection.
 - To concatenate collections or add to collections, type a name for the new collection in the Name column. Type the appropriate operator commands like c_union or c_diff in the Command column. See [Creating Collections using Tcl Commands](#), on page 85 for a list of available commands and the *Reference Manual* 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.

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.



	Enabled	Delay Type	From	To	Through	Start/End	Cycles	Max Delay(ns)	Comment
1	<input checked="" type="checkbox"/>	False	special_regs.status[7:0]						
2	<input type="checkbox"/>		i:decode.opcode_goto						
3	<input type="checkbox"/>		i:decode.opcode_call						
4	<input type="checkbox"/>		i:decode.opcode_retlw						
5	<input type="checkbox"/>		i:decode.skip						
6	<input type="checkbox"/>		i:decode.decodes[13:0]						
			i:prgmcntr..r_0[10:0]						
			i:prgmcntr..r_1[10:0]						
			i:prgmcntr..r_2[10:0]						
			i:prgmcntr..r_3[10:0]						
			i:prgmcntr..r_4[10:0]						

Delay Paths

- 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 83](#)). 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 82](#).

For details of the syntax for the commands described here, refer to [synhooks File Syntax, on page 1118](#) 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 76](#) and [Specifying Search Patterns for Tcl find, on page 74](#)). 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	<code>c_union</code> . See Examples: c_union Command, on page 87
Concatenate collections	<code>c_union</code> . See Examples: c_union Command, on page 87 .
Isolate differences between collections	<code>c_diff</code> . See Examples: c_diff Command, on page 87 .
Find common objects between collections	<code>c_intersect</code> . See Examples: c_intersect Command, on page 87 .
Find objects that belong to just one collection	<code>c_symdiff</code> . See Examples: c_symdiff Command, on page 88 .

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

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 88](#)), but you cannot apply constraints to the collection.

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 `collection1` and `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 360](#) 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).
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 91](#).

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 360](#) in the *Reference Manual*.

To...	Do this...
Copy a collection	<p>Create a new variable for the copy and copy the original collection to it with the <code>set</code> command. When you make changes to the original, it does not affect the copy, and vice versa.</p> <pre>set my_collection_copy \$my_collection</pre>
List the objects in a collection	<p>Use the <code>c_print</code> command to view the objects in a collection, and optionally their properties, in column format:</p> <pre>"v:top" "v:block_a" "v:block_b"</pre> <p>Alternatively, you can use the <code>-print</code> option to an operation command to list the objects.</p>

To...	Do this...
Generate a Tcl list of the objects in a collection	<p>Use the <code>c_list</code> 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.</p> <p>This is an example of <code>c_list</code> results:</p> <pre>{ "v:top" "v:block_a" "v:block_b" }</pre> <p>Alternatively, you can use the <code>-print</code> option to an operation command to list the objects.</p>

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 [FPGA Design Constraint \(FDC\) File, on page 761](#).

For details about the translated files and troubleshooting guidelines, see [sdc2fdc Tcl Shell Command, on page 760](#).

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_multicycle_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 <code>define_clock</code> . See Defining Clocks, on page 100 for additional information.
	Clock frequency other than the one implied by the signal on the clock pin <code>syn_reference_clock</code> (attribute). See Defining Clocks, on page 100 for additional information
	Clock domains with asymmetric duty cycles <code>define_clock</code> . See Defining Clocks, on page 100 for additional information

Use SCOPE...	To Define...
Clock to Clock	Edge-to-edge clock delays define_clock_delay. See Defining Clocks, on page 100 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 107 for additional information Output delays from your FPGA define_output_delay. See Defining Input and Output Constraints (Legacy), on page 107 for additional information
Delay Paths	Paths with multiple clock cycles define_multicycle_path. See Defining Multicycle Paths, on page 72 for additional information False paths (certain technologies) define_false_path. See Defining False Paths (Legacy), on page 108 for additional information. Path delays define_path_delay. See Defining From/To/Through Points for Timing Exceptions, on page 68 for additional information
Attributes	Assign attributes for objects specifying their values

Use SCOPE...	To Define...
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.

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 339](#).

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 83 .	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 <code>syn_reference_clock</code> , that were not entered in the source files	Attributes

To define...	Click...
I/O standards for any port in the I/O Standard panel of the SCOPE window.	/O Standard
Compile points in a top-level constraint file. See Synthesizing Compile Points, on page 451 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 46](#) 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 116](#).

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 47.

The SCOPE GUI is much easier to use, and you can define various timing constraints in it. For the equivalent Tcl syntax, see [Chapter 16, *Batch Commands and Scripts*](#) in the *Reference Manual*. See the following for different timing constraints:

- [Entering Default Constraints](#), on page 97
- [Setting Clock and Path Constraints](#), on page 97
- [Defining Clocks](#), on page 100
- [Defining Input and Output Constraints \(Legacy\)](#), on page 107
- [Specifying Standard I/O Pad Types](#), on page 62

To set constraints for timing exceptions like false paths and multicycle paths, see [Specifying Timing Exceptions](#), on page 68.

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 97). For auto constraints, see [Using Auto Constraints](#), on page 339. To apply the constraints, add the file to the project according to the procedure described in [Making Changes to a Project](#), on page 116. The constraints file has an `fdc` extension. See [Working with Constraint Files](#), on page 46 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 451 for more information about compile points and [Specifying Attributes Using the SCOPE Editor](#), on page 146. For information about setting default constraints, see [Entering Default Constraints](#), on page 97.

To define...	Pane	Do this to set the constraint...
Clocks	Clock	<p>Select the clock object (Clock).</p> <p>Specify a clock name (Clock Alias), if required.</p> <p>Type a frequency value (Frequency) or a period (Period).</p> <p>Change the default Duty Cycle or set Rise/Fall At, if needed.</p> <p>Change the default clock group, if needed</p> <p>Check the Enabled box.</p> <p>See Defining Clocks, on page 100 for information about clock attributes.</p>
Virtual clocks	Clock	<p>Set the clock constraints as described for clocks, above.</p> <p>Check the Virtual Clock box.</p>
Route delay	Clock Inputs/ Outputs Registers	<p>Specify the route delay in nanoseconds. Refer to Defining Clocks, on page 100, Defining Input and Output Constraints (Legacy), on page 107 and the Register Delays section of this table details.</p>
Edge-to-edge clock delay	Clock to Clock	<p>Select the starting edge for the delay constraint (From Clock Edge).</p> <p>Select the ending edge for the constraint (To Clock Edge).</p> <p>Enter a delay value.</p> <p>Mark the Enabled check box.</p>
Input/output delays	Inputs/ Outputs	<p>See Defining Input and Output Constraints (Legacy), on page 107 for information about setting I/O constraints.</p>
Register delays	Registers	<p>Select the register (Register).</p> <p>Select the type of delay, input or output (Type).</p> <p>Type a delay value (Value).</p> <p>Check the Enabled box.</p> <p>If you do not meet timing goals after place-and-route, adjust the clock constraint as follows:</p> <ul style="list-style-type: none"> • 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.

To define...	Pane	Do this to set the constraint...
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 68 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 72 .
False paths	Delay Paths Clock to Clock	See Defining False Paths (Legacy), on page 108 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: <ul style="list-style-type: none"> • 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 339](#)), 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 100
- [Constraining Clock Enable Paths](#), on page 104
- [Defining Other Clock Requirements](#), on page 106

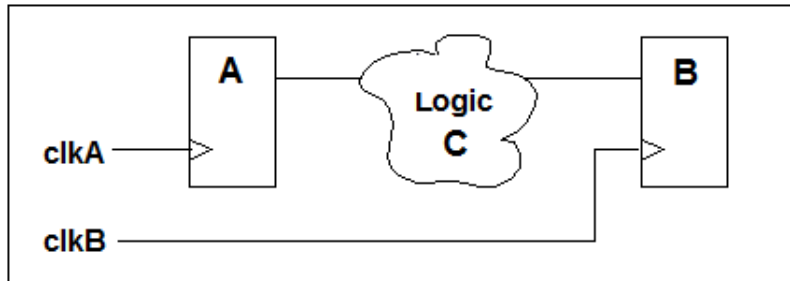
Defining Clock Frequency

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 106](#) 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 106](#) for more information about clock group settings.

The global frequency also applies to any purely combinatorial 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.

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.
Combinatorial 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 104](#) 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

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 72](#)). This is the recommended method.
- Using the `syn_reference_clock` attribute, as described in step 4 of [Defining Clock Frequency, on page 100](#). 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.

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 100](#)), 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 107](#) 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 combinatorial 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.

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 97](#), 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 68](#) 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).
 - 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 5

Setting up a Logic Synthesis Project

When you synthesize a design with the Synopsys FPGA synthesis tools, 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 112
- [Managing Project File Hierarchy](#), on page 120
- [Setting Up Implementations](#), on page 126
- [Setting Logic Synthesis Implementation Options](#), on page 129
- [Specifying Attributes and Directives](#), on page 142
- [Searching Files](#), on page 150
- [Archiving Files and Projects](#), on page 153

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 112
- [Opening an Existing Project File](#), on page 115
- [Making Changes to a Project](#), on page 116
- [Setting Project View Display Preferences](#), on page 117
- [Updating Verilog Include Paths in Older Project Files](#), on page 119

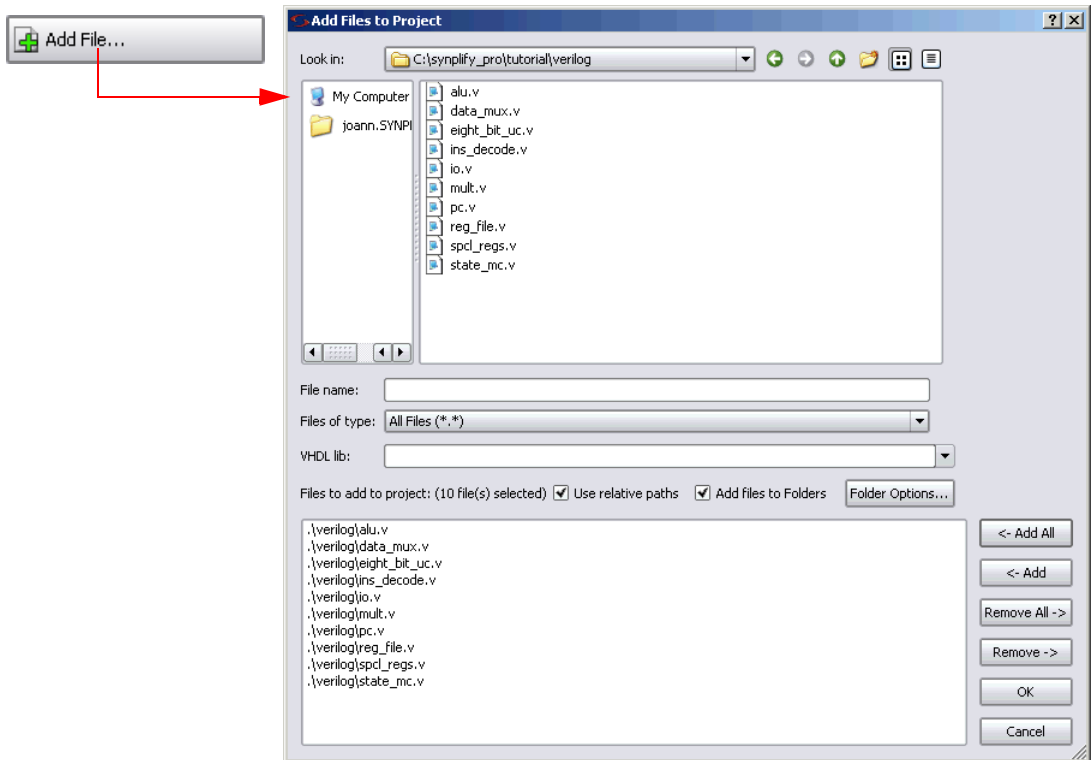
Creating a Project File

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 41.



- 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.

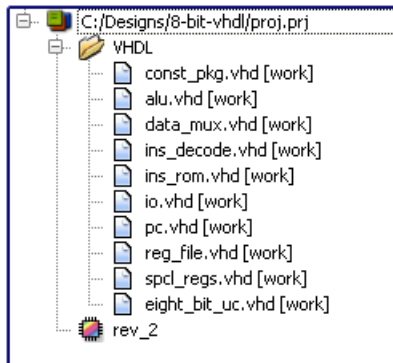
You can add all the files in the directory and then remove the ones you do not need with the Remove button.

If you are adding VHDL files, select the appropriate library from the 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.

- 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. You can have multiple implementations in Synplify Pro. 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 137](#). You can also set these file options later, before running synthesis.

For additional vendor-specific information about using vendor macro libraries and black boxes, see [Optimizing for Microsemi Designs, on page 345](#).

- 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.

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

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 112](#) 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.

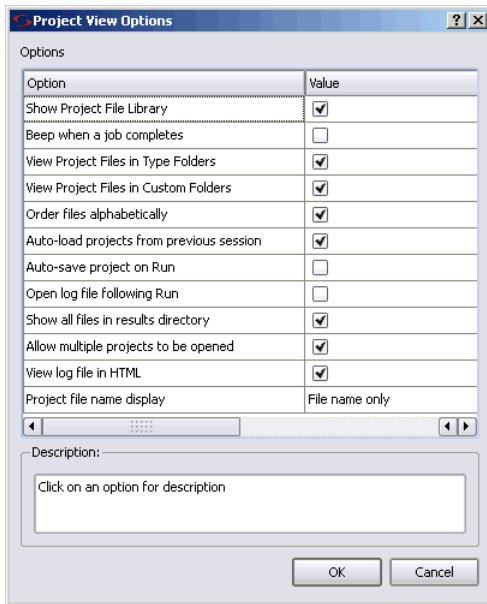
5. To check the time stamp on a file, right click on a file in the Project view and select File Options. Check the time that the file was last modified. Click OK.

Setting Project View Display Preferences

You can customize the organization and display of project files.

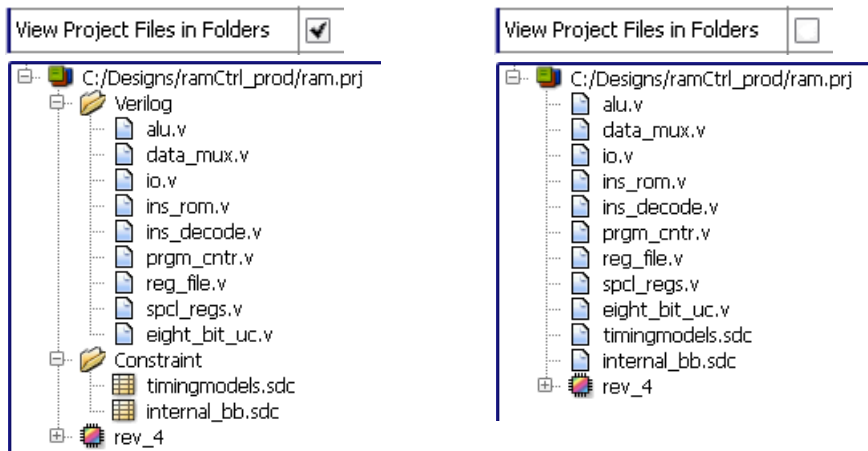
1. Select Options->Project View Options.

The Project View Options form opens.



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

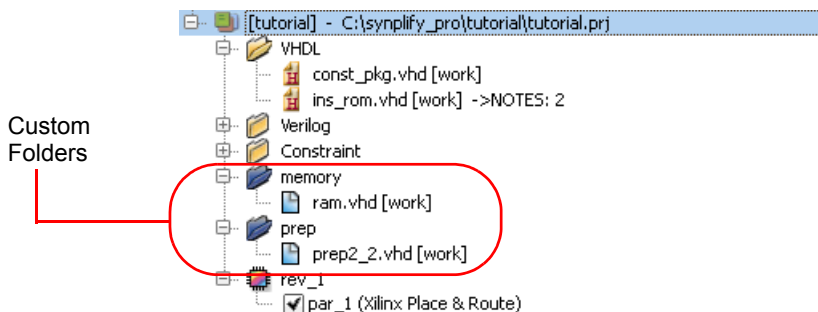
Checking this option creates separate folders in the Project view for constraint files and source files.



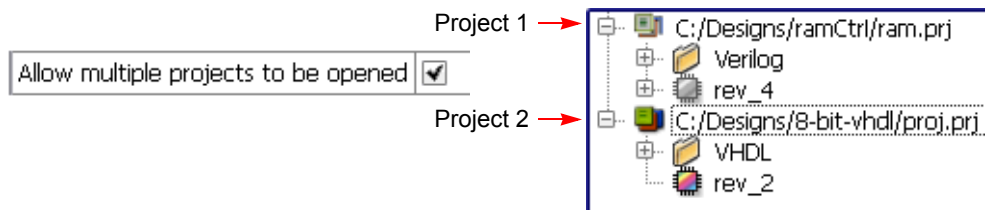
3. Control file display with the following:

- 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 120](#). 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 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.

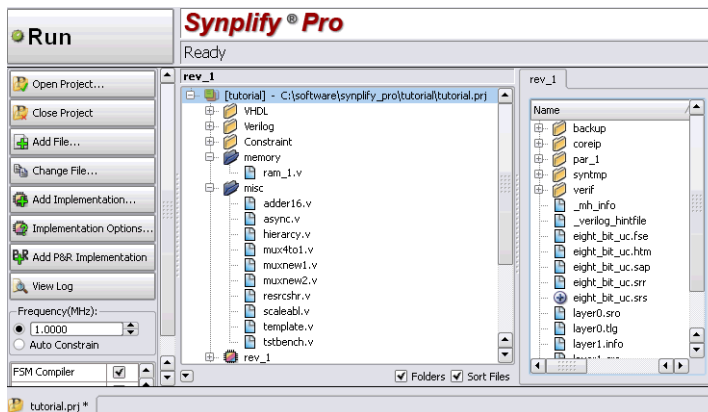
Managing Project File Hierarchy

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](#)

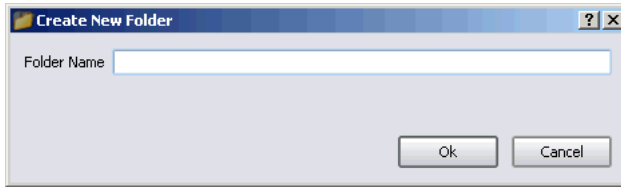
Creating Custom Folders

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.



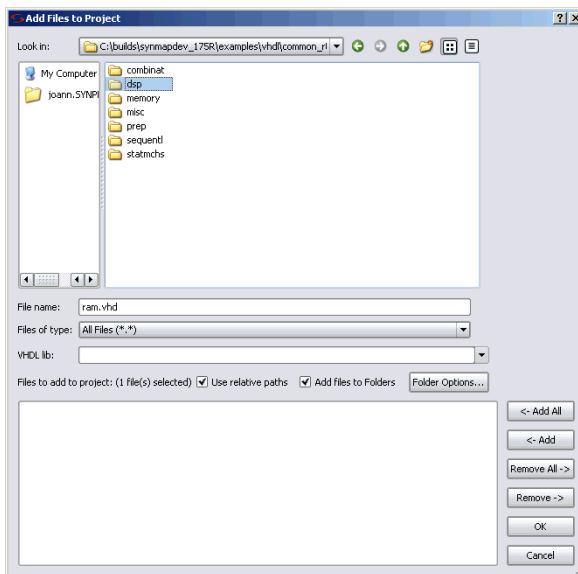
There are several ways to create custom folders and then add files to them in a project. Use one of the following methods:

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.



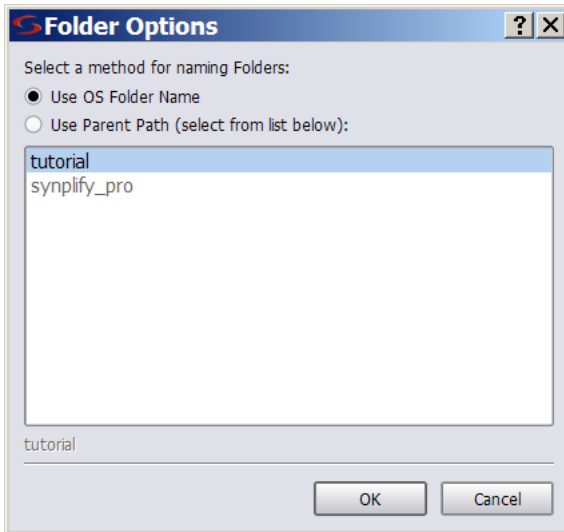
Note that you can arbitrarily name the folder, however do not use the character (/) because this is a hierarchy separator symbol.

- 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.
 - 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:

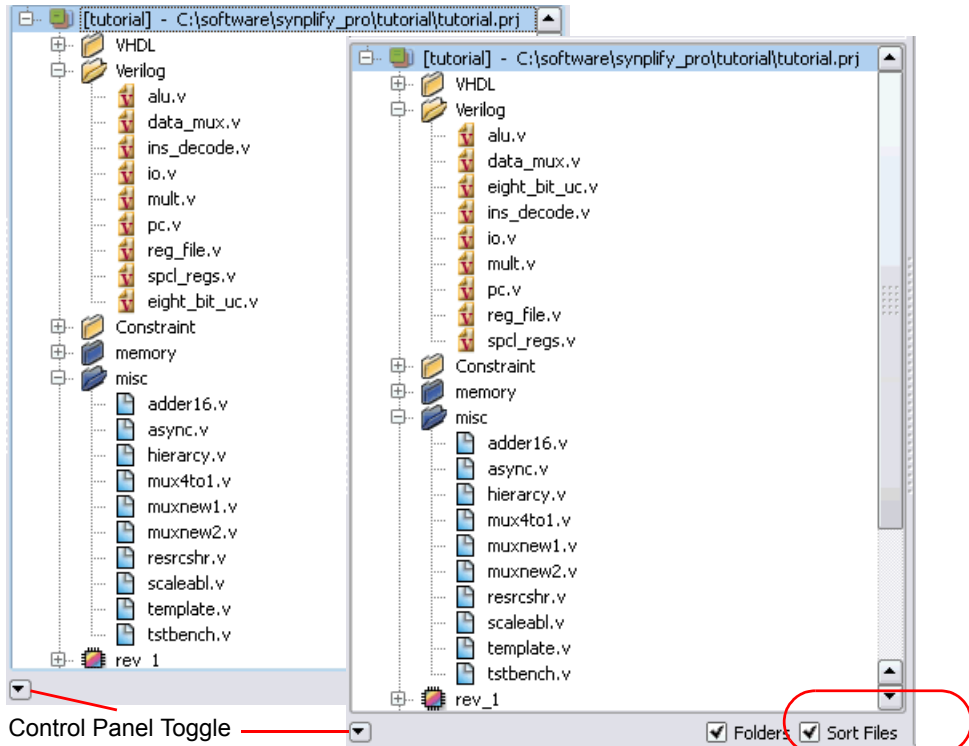
```
/Examples/Verilog/RTL
```

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.



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

Implementations are extensions of the project metaphor used in the Synplify Pro synthesis software. An implementation is one version of the project, implemented with a specific set of constraints and other settings. A project can contain multiple implementations, each with its own settings.

Working with Multiple Implementations

The Synplify Pro tool lets 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, and the `design_fsm.sdc` file generated by FSM Explorer. 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 129](#) for information about setting options.

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 242](#) 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 57](#). For information about overriding global settings with individual attributes or directives, see [Specifying Attributes and Directives, on page 142](#).

This section discusses the following topics:

- [Setting Device Options, on page 129](#)
- [Setting Optimization Options, on page 132](#)
- [Specifying Global Frequency and Constraint Files, on page 133](#)
- [Specifying Result Options, on page 135](#)
- [Specifying Timing Report Output, on page 137](#)
- [Setting Verilog and VHDL Options, on page 137](#)

Setting Device Options

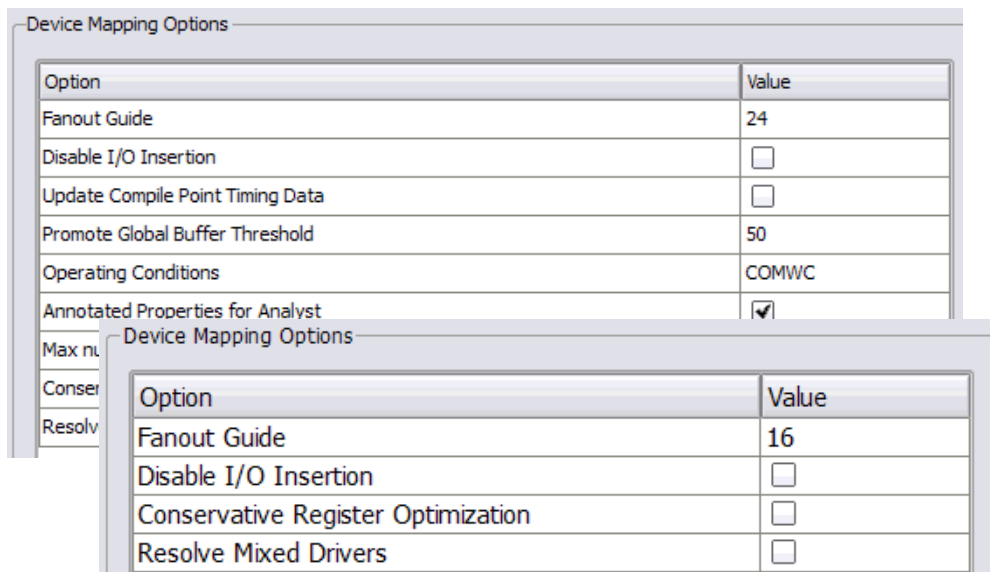
Device options are part of the global options you can set for the synthesis run. They include the part selection (technology, part and speed grade) and implementation options (I/O insertion and fanouts). The options and the implementation of these options can vary from technology to technology, so check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

The image displays two screenshots of the Synplify Pro software interface, specifically the 'Device' selection window. Both screenshots show a 'Device' tab with four dropdown menus: 'Technology:', 'Part', 'Package:', and 'Speed:'.
The top screenshot shows the following selections:
- Technology: Microsemi SmartFusion
- Part: A2F200M3F
- Package: PQFP208
- Speed: Std
The bottom screenshot shows the following selections:
- Technology: Microsemi IGLOO+
- Part: AGLP030V2
- Package: CS201
- Speed: Std

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 and retiming, see [Setting Fanout Limits, on page 209](#), and [Retiming, on page 196](#), respectively. For details about other vendor-specific options, refer to the appropriate vendor chapter and technology family in the *Reference Manual*.



4. Set other implementation options as needed (see [Setting Logic Synthesis Implementation Options, on page 129](#) 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 (<code>set_option...</code>)
Annotated Properties for Analyst	<code>-run_prop_extract {1 0}</code>
Disable I/O Insertion	<code>-disable_io_insertion {1 0}</code>
Fanout Guide	<code>-fanout_limit fanout_value</code>

Option	Tcl Command (set_option...)
Package	-package <i>pkg_name</i>
Part	-part <i>part_name</i>
Resolve Mixed Drivers	-resolve_multiple_driver {1 0}
Speed	-speed_grade <i>speed_grade</i>
Technology	-technology <i>keyword</i>
Update Compile Point Timing Data	-update_models_cp {0 1}

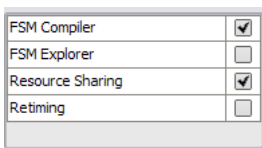
Setting Optimization Options

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.

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 218
FSM Explorer	Running the FSM Explorer, on page 224 <i>Note:</i> Only a subset of the Microsemi technologies support the FSM Explorer option. Use the Project->Implementation Options->Options panel to determine if this option is supported for the device you specify in your tool.
Resource Sharing	Sharing Resources, on page 213
Retiming	Retiming, on page 196

The equivalent Tcl `set_option` command options are as follows:

Option	<code>set_option</code> Tcl Command Option
FSM Compiler	<code>-symbolic_fsm_compiler {1 0}</code>
FSM Explorer	<code>-use_fsm_explorer {1 0}</code>
Resource Sharing	<code>-resource_sharing {1 0}</code>
Retiming	<code>-retiming {1 0}</code>

- Set other implementation options as needed (see [Setting Logic Synthesis Implementation Options, on page 129](#) for a list of choices). Click OK.
- 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.

- To set a global frequency, do one of the following:
 - Type a global frequency in the Project view.

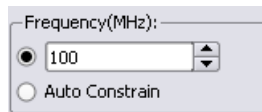
- Open the Implementation Options form by clicking the Implementation Options button or selecting Project->Implementation Options, and click the Constraints tab.

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

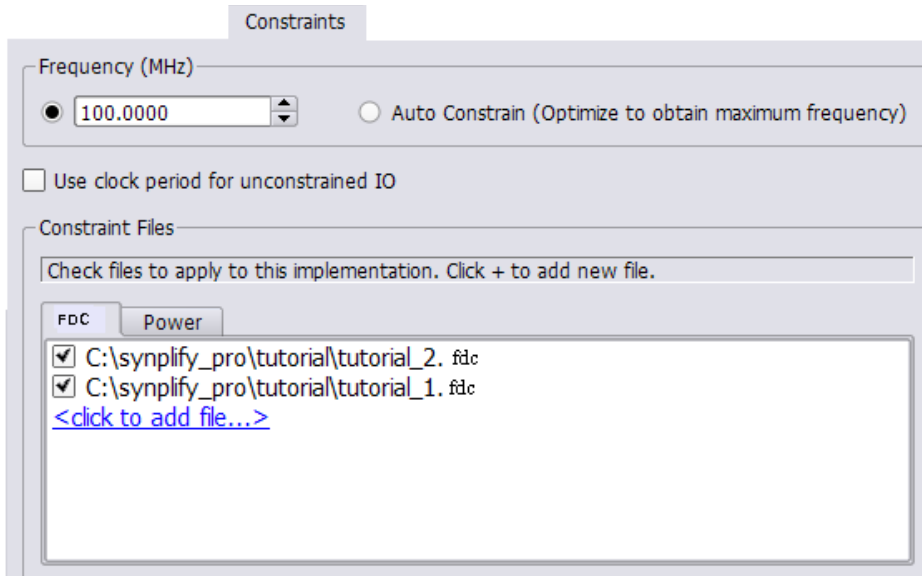
You can override the global frequency with local constraints, as described in [Specifying SCOPE Constraints, on page 57](#). In the Synplify Pro tool, you can automatically generate clock constraints for your design instead of setting a global frequency. See [Using Auto Constraints, on page 339](#) for details.

Global Frequency and Constraints

Project View

A small dialog box titled "Frequency(MHz):". It contains a radio button selected next to a text field with the value "100". Below it is another radio button labeled "Auto Constrain".

Implementation Options->Constraints

A screenshot of the "Constraints" tab in the Implementation Options dialog. The "Frequency (MHz)" section has a radio button selected next to a text field containing "100.0000", and another radio button labeled "Auto Constrain (Optimize to obtain maximum frequency)". Below this is a checkbox labeled "Use clock period for unconstrained IO" which is unchecked. The "Constraint Files" section has a text area with the instruction "Check files to apply to this implementation. Click + to add new file." Below this is a list box with two entries, both checked: "C:\synplify_pro\tutorial\tutorial_2. fdc" and "C:\synplify_pro\tutorial\tutorial_1. fdc". At the bottom of the list box is a blue hyperlink "<click to add file...>". There are tabs labeled "FDC" and "Power" above the list box.

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 57](#).

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.

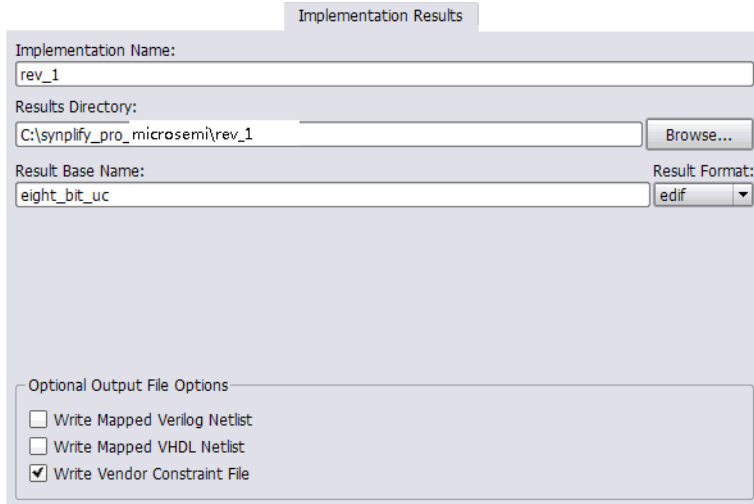
4. Set other implementation options as needed (see [Setting Logic Synthesis Implementation Options, on page 129](#) for a list of choices). Click OK.

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.



The image shows a dialog box titled "Implementation Results". It contains the following fields and options:

- Implementation Name:** A text field containing "rev_1".
- Results Directory:** A text field containing "C:\synplify_pro_microsemi\rev_1" and a "Browse..." button.
- Result Base Name:** A text field containing "eight_bit_uc".
- Result Format:** A dropdown menu currently set to "edif".
- Optional Output File Options:** A group box containing three checkboxes:
 - ☐ Write Mapped Verilog Netlist
 - ☐ Write Mapped VHDL Netlist
 - ☒ Write Vendor Constraint File

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, on page 50](#) 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 129](#) 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.



The screenshot shows a dialog box with a tab labeled "Timing Report". Inside the dialog, there are two input fields. The first field is labeled "Number of Critical Paths:" and contains the value "32". The second field is labeled "Number of Start/End Points:" and contains the value "8".

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 129](#) for a list of choices). Click OK.

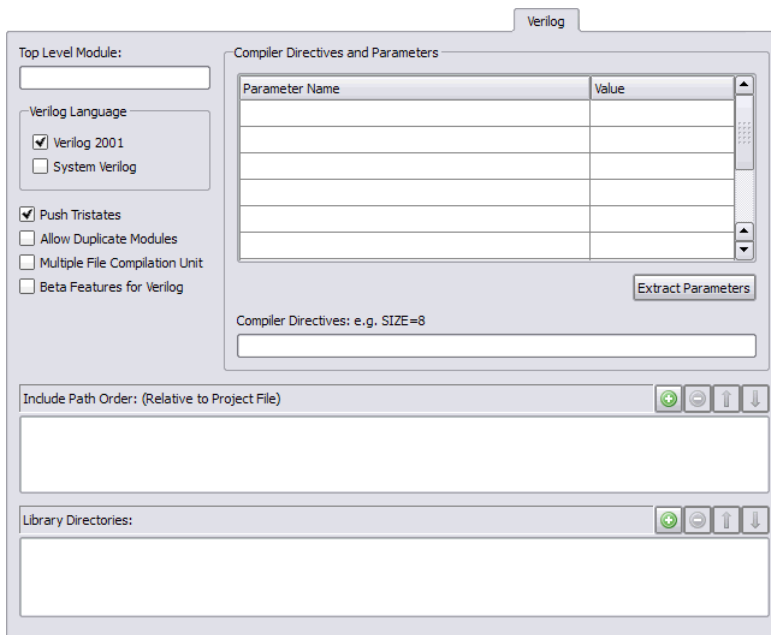
When you synthesize the design, the software compiles and maps the design using the options you set.

Setting Verilog and VHDL Options

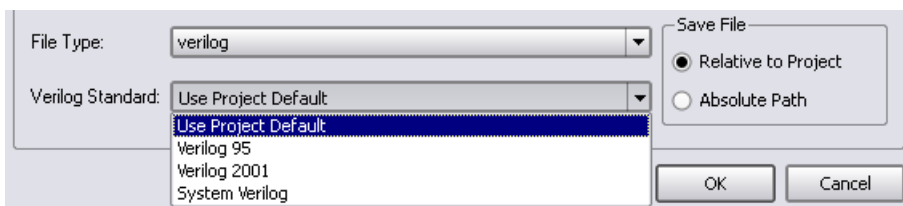
When you set up the Verilog and VHDL source files in your project, you can also specify certain compiler options.

Setting Verilog File Options

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



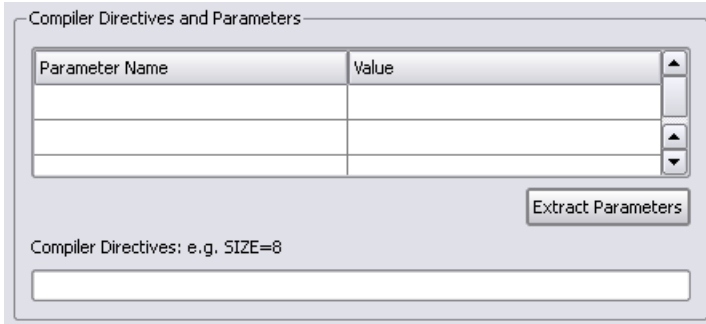
1. Specify the Verilog format to use.
 - To set the compiler globally for all the files in the project, select Project->Implementation Options->Verilog. If you are using Verilog 2001 or SystemVerilog, check the *Reference Manual* for supported constructs.
 - To specify the Verilog compiler on a per file 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.



Parameter Name	Value

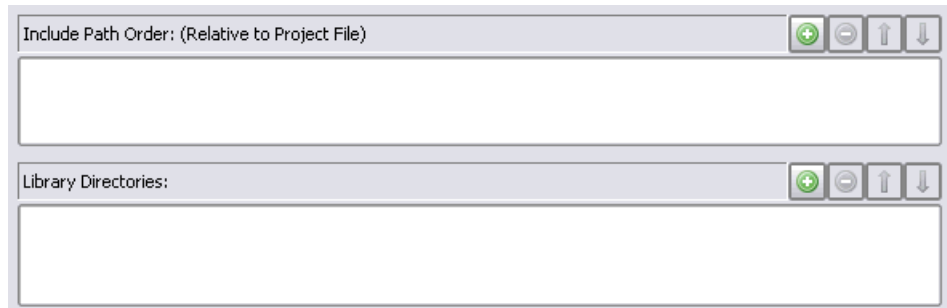
Extract Parameters

Compiler Directives: e.g. SIZE=8

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"
```



Include Path Order: (Relative to Project File)

Library Directories:

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 129](#) 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.

VHDL

Top Level Entity:

Default Enum Encoding: default

☒ Push Tristates

☐ Synthesis On/Off Implemented as Translate On/Off

☐ VHDL 2008

☐ Beta Features for VHDL

Generics

Generic Name	Value

Extract Generic Constants

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 159](#).

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:

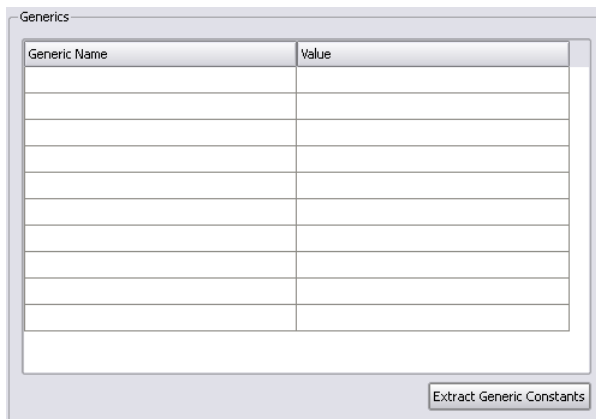
- Specify the kind of encoding you want to use.
- Disable the FSM compiler.

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 176](#).

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.



4. To push tristates across process/block boundaries, check that Push Tristates is enabled. For details, see [Push Tristates Option, on page 169](#) in 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 1061 in the *Reference Manual* for more information.
6. Set other implementation options as needed (see [Setting Logic Synthesis Implementation Options](#), on page 129 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. This table describes the methods that are available to create attribute and directive 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 priority when there are conflicts.

For further details, refer to the following:

- [Specifying Attributes and Directives in VHDL](#), on page 143
- [Specifying Attributes and Directives in Verilog](#), on page 145
- [Specifying Attributes Using the SCOPE Editor](#), on page 146
- [Specifying Attributes in the Constraints File](#), on page 149

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 142](#). 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 736](#) in the *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 simplifiedff 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 736](#) 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 simplifiedff 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 142](#). 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 539](#) in 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

```
/* synthesis attributeName = value */
/* synthesis directoryName = value */
```

Verilog Line Comment Syntax

```
// synthesis attributeName = value
// synthesis directoryName = value
```

For details of the syntax rules, see [Verilog Attribute and Directive Syntax, on page 539](#) in the *Reference Manual*. The following are examples:

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

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 */;
```

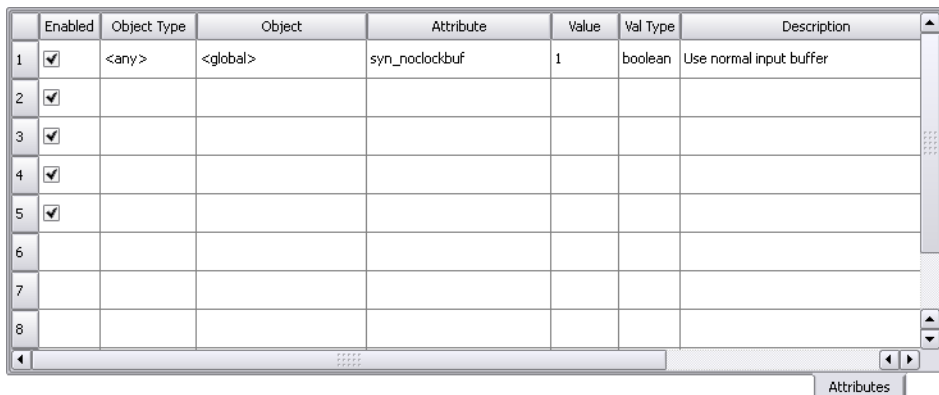
The `syn_preserve` attribute is only applied to `q_d`. This is the expected behavior for the synthesis tools. To apply this attribute to all registers, you must use a separate Verilog `reg` statement for each register and apply the attribute.

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 143](#) or [Specifying Attributes and Directives in Verilog, on page 145](#)). 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).



	Enabled	Object Type	Object	Attribute	Value	Val Type	Description
1	<input checked="" type="checkbox"/>	<any>	<global>	syn_noclockbuf	1	boolean	Use normal input buffer
2	<input checked="" type="checkbox"/>						
3	<input checked="" type="checkbox"/>						
4	<input checked="" type="checkbox"/>						
5	<input checked="" type="checkbox"/>						
6							
7							
8							

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.

	Enabled	Object Type	Object	Attribute	Value	Val Type	Description	Comment
1	<input checked="" type="checkbox"/>		<global>	▼				
2				syn_global_buffers				
3				syn_loc				
4				syn_netlist_hierarchy				
5				syn_noarrayports				
				syn_noclockbuf				
				syn_ramstyle				
				syn_replicate				

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.

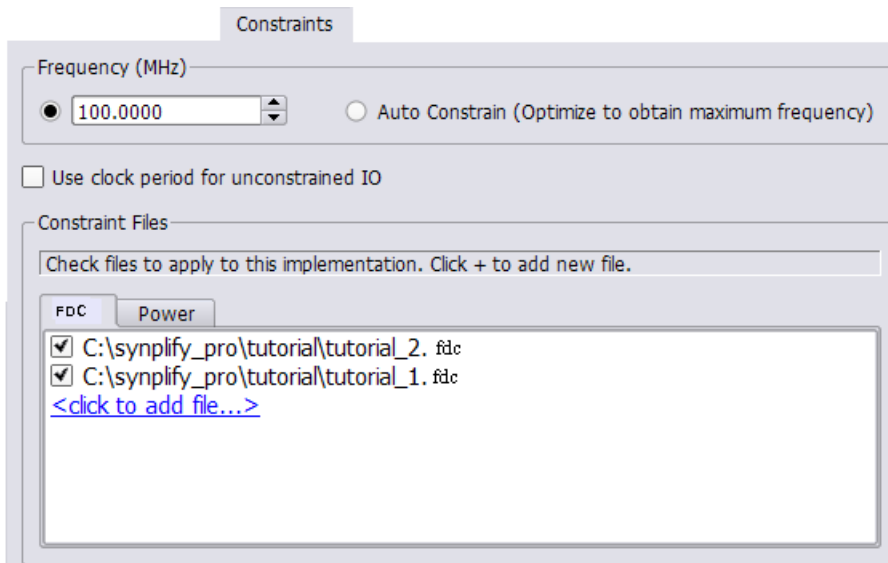
5. Fill out the value. Hold down the mouse button in the Value column, and select from the list. You can also type in a value.

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 904](#) 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.

Specifying Attributes in the Constraints File

When you use the SCOPE window ([Specifying Attributes Using the SCOPE Editor, on page 146](#)), 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 [Attribute and Directive Summary \(Alphabetical\), on page 907](#) in the *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 [i:statemod.statereg\[*\], on page 49](#).

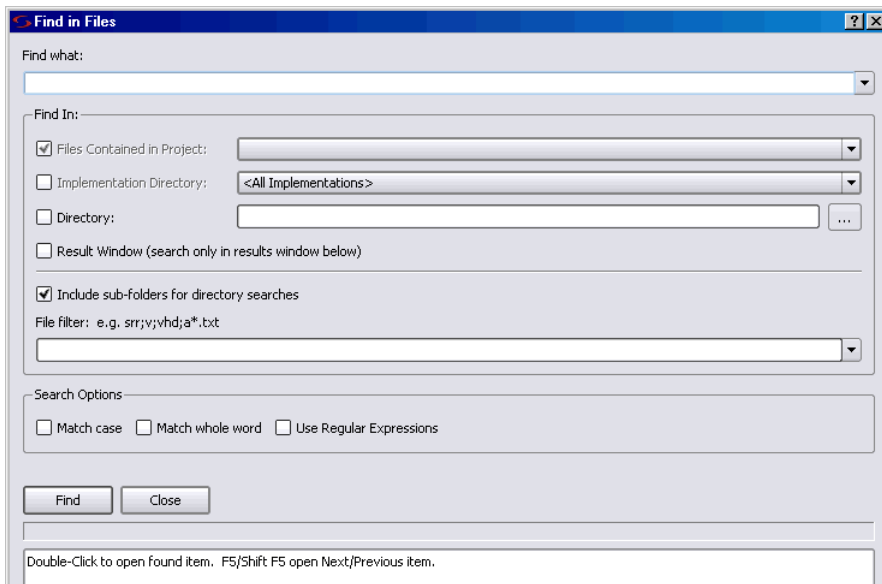
Searching Files

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 crossprobe 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.



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.
- **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 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.

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.

Whenever you have a sar file that contains relative or absolute include paths for the files in the project, 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_ Directive, on page 167](#).

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

- [Archive a Project](#)
- [Un-Archive a Project](#)
- [Copy a Project](#)

Archive a Project

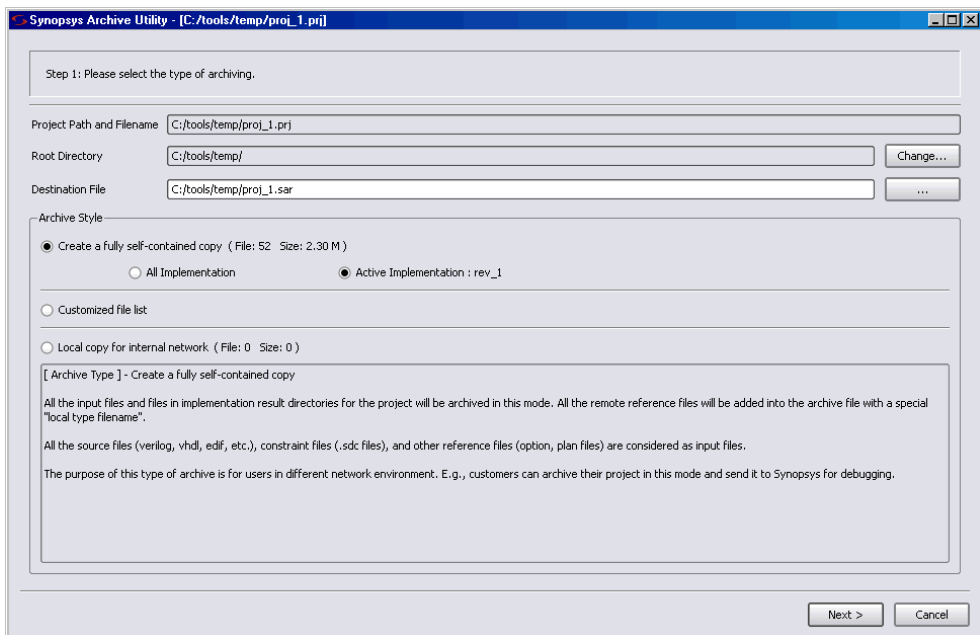
Use the archive utility to store the files for a design project into a single archive file in a proprietary format (sar). You can archive an entire project or selected files from a project. If you want to create a copy of a project without archiving the files, see [Copy a Project, on page 160](#).

Here are the steps to create an archive:

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

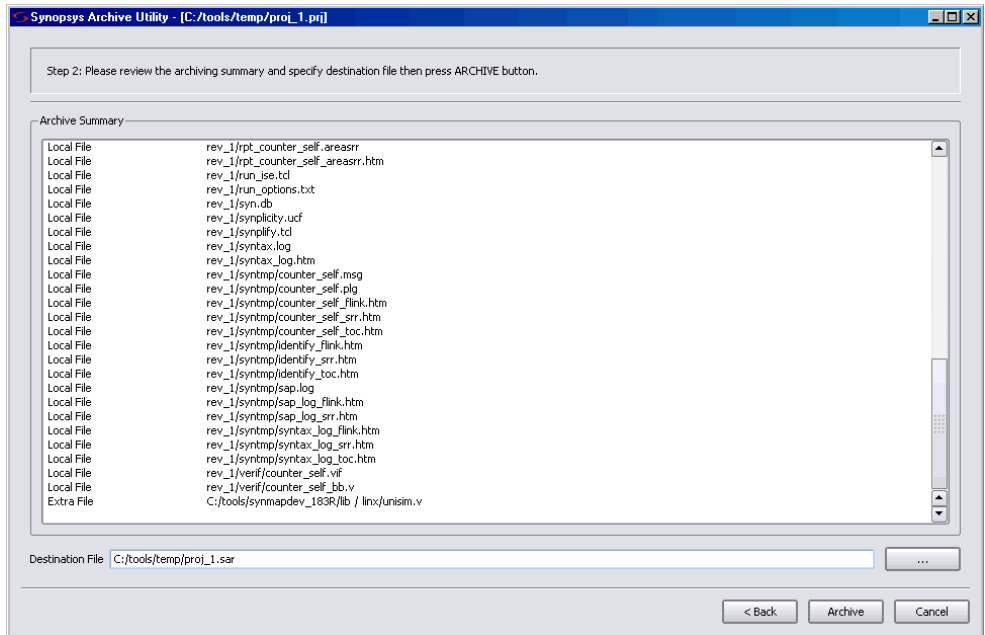
The Tcl command equivalent is `project -archive`. For a complete description of the project Tcl command options for archiving, see [project, on page 1084](#) of the *Reference Manual*.

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.



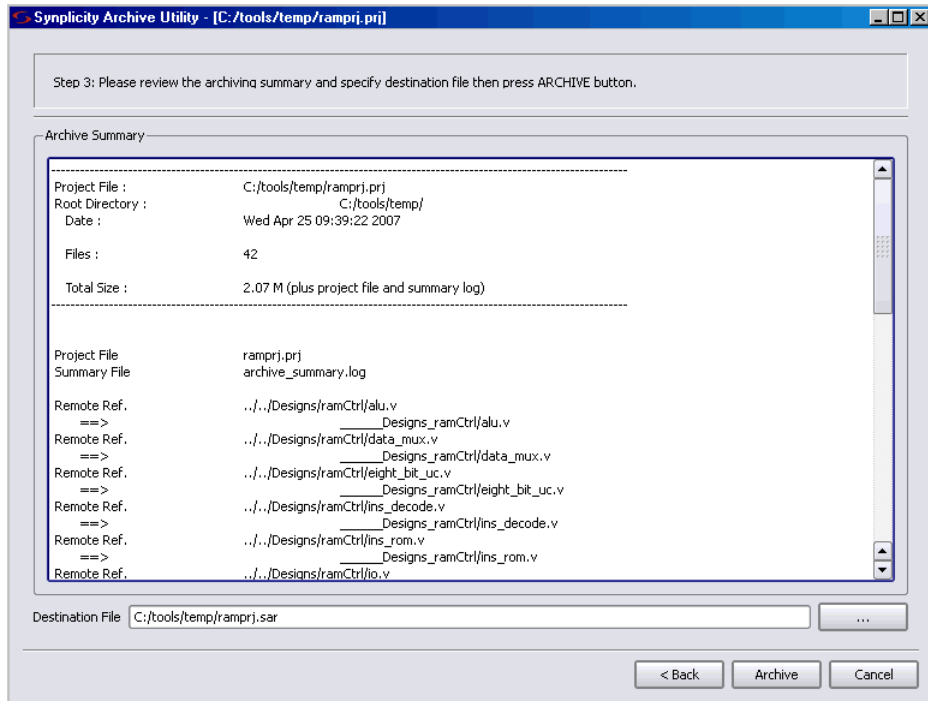
2. Do the following on the first page of the wizard:

- 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, and use the check boxes to include files in or exclude files from the archive. Use the Add Extra Files button on the second page to include additional files in the project.

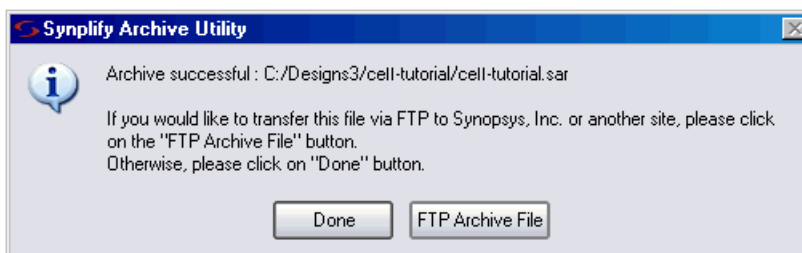


- Click Next.

The tool summary displays all the files in the archive and shows the full uncompressed file size. The actual size is smaller after the archiving operation as there is no duplication of files.



3. Use the Back button to correct directory or file information and/or follow-up on any missing files, as appropriate.
4. Verify that the current archive contains the files that you want, then click Archive which creates the project archive sar file and displays the following prompt:



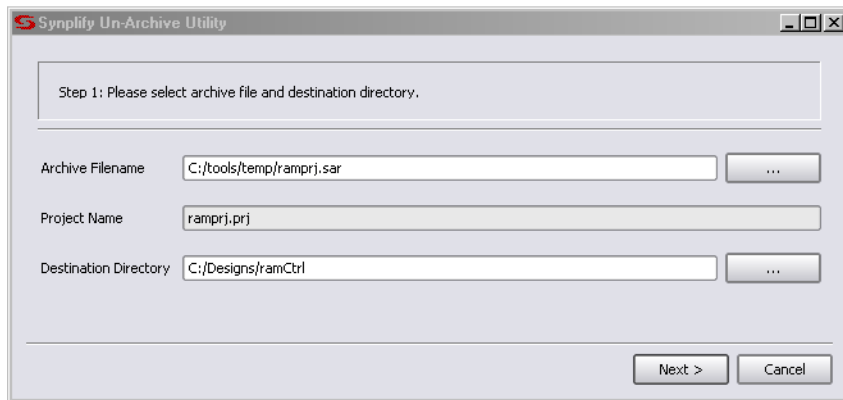
Click Done if you are finished.

Un-Archive a Project

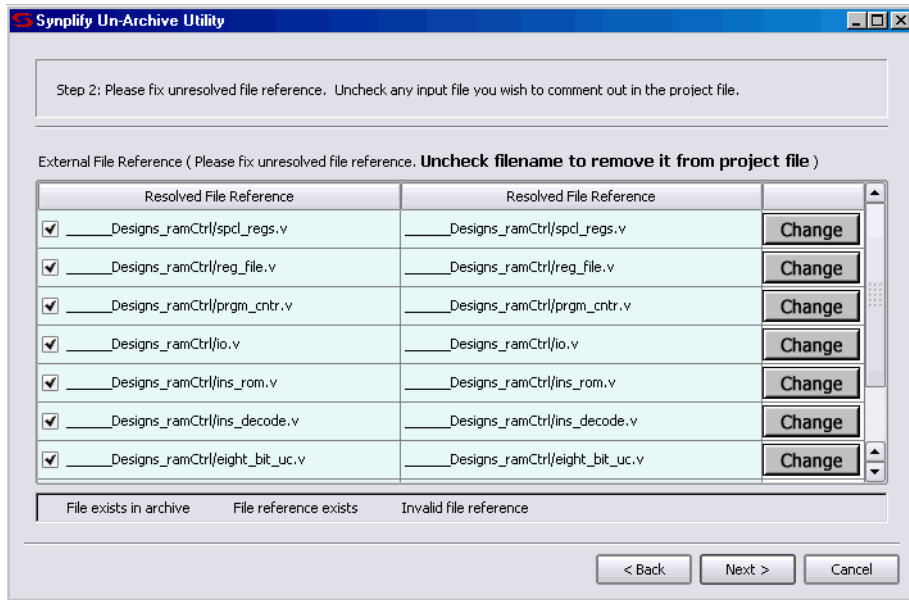
Use this procedure to extract design project files from an archive file (sar).

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

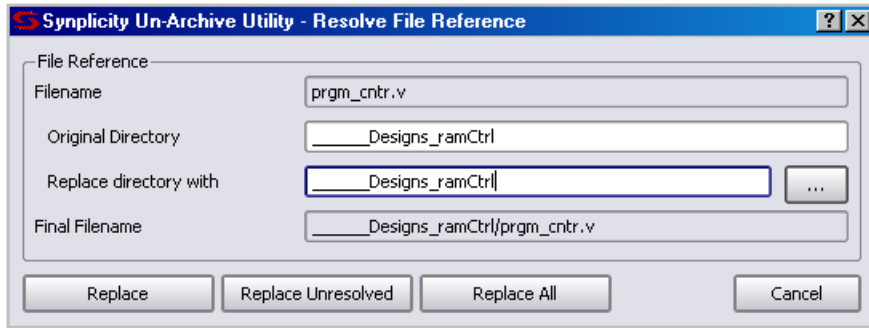
The Tcl command equivalent is project [-unarchive](#). For a complete description of the project Tcl command options for archiving, see [project](#), on [page 1084](#) of the *Reference Manual*.



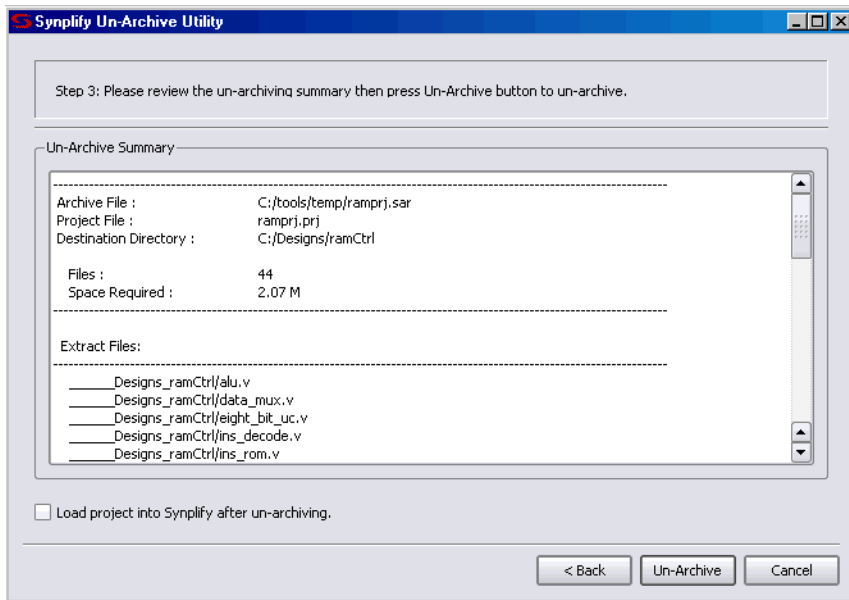
2. In the wizard, enter the following:
 - Name of the sar file containing the project files.
 - Name of project to extract (un-archive). This field is automatically extracted from the sar file and cannot be changed.
 - Pathname of directory in which to write the project files (destination).
 - Click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



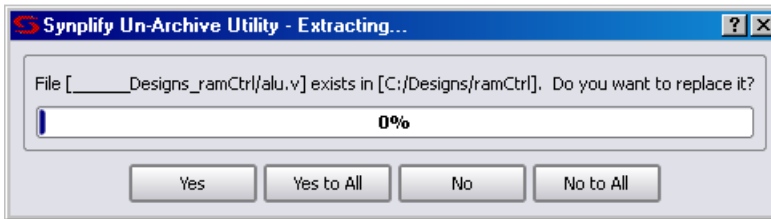
4. Click Next and verify that the project files you want are displayed in the Un-Archive Summary.



5. If you want to load this project in the UI after files have been extracted, enable the Load project into Synplicity after un-archiving option.
6. Click Un-Archive.

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

7. 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

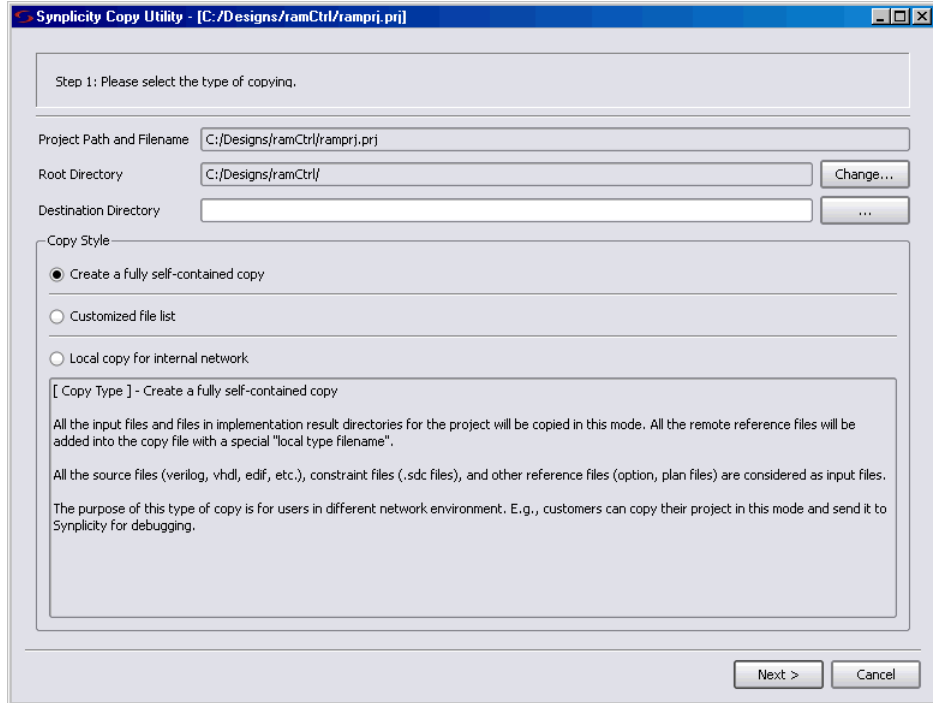
Use this utility to create an unarchived copy of a design project. You can copy an entire project or just selected files from the project. However, if you want to create an archive of the project, where the entire project is stored as a single file, see [Archive a Project, on page 153](#).

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

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

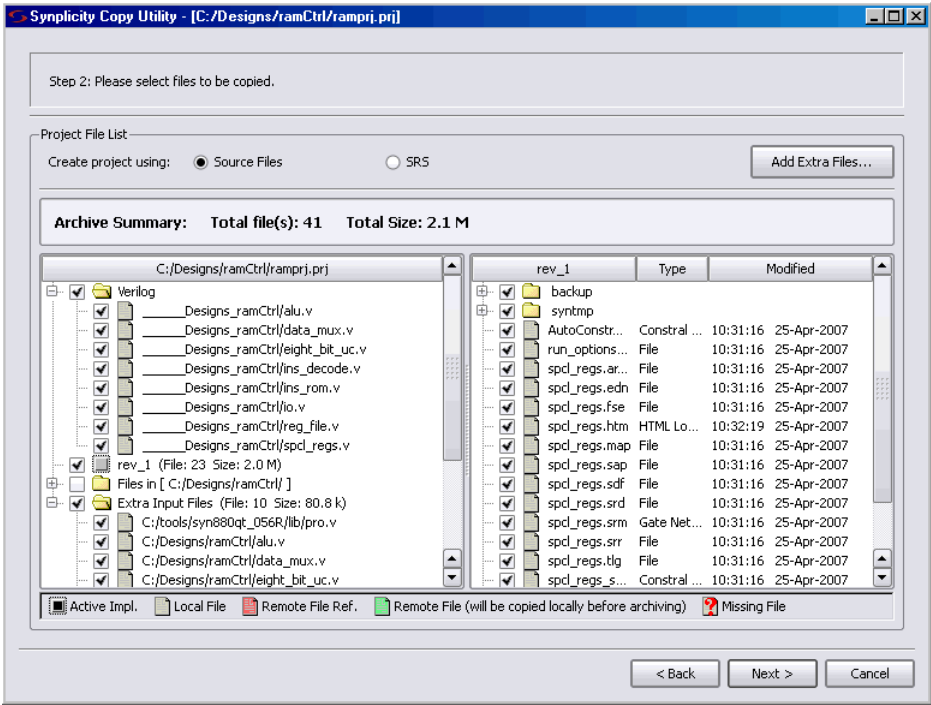
The Tcl command equivalent is `project -copy`. For a complete description of the project Tcl command options for archiving, see [project, on page 1084](#) of the *Reference Manual*.

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 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.

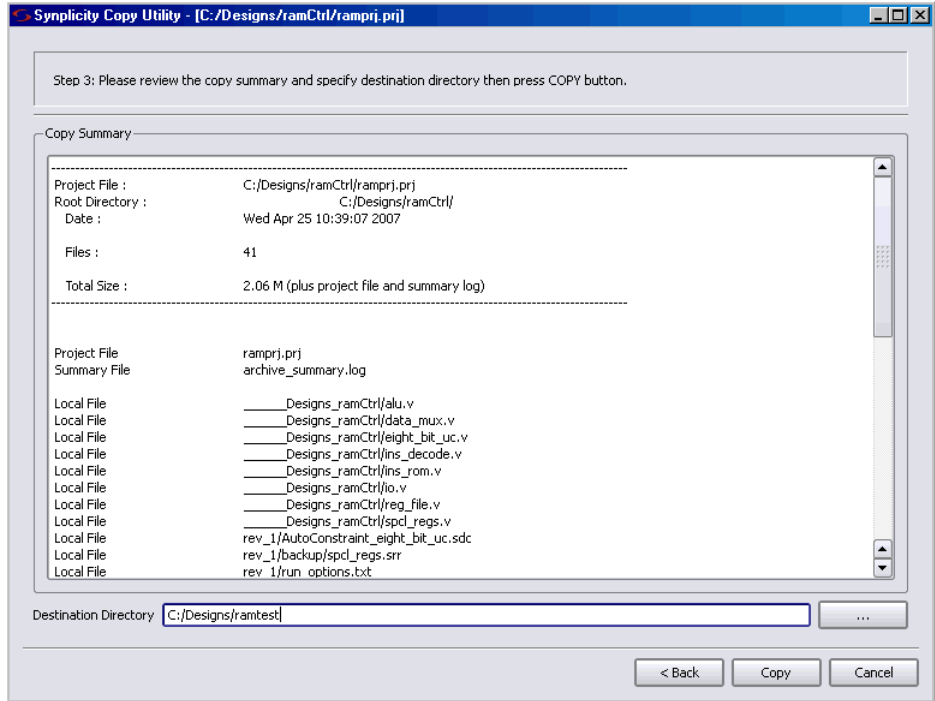


2. Do the following in the wizard:

- 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.



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.

CHAPTER 6

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 166
- [Defining State Machines for Synthesis](#), on page 175
- [Inferring RAMs](#), on page 180
- [Initializing RAMs](#), on page 186

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 166
- [Instantiating Black Boxes and I/Os in VHDL](#), on page 168
- [Adding Black Box Timing Constraints](#), on page 170
- [Adding Other Black Box Attributes](#), on page 174

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 168.

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" */
endmodule
```

- Make an instance of the stub in your design.
- 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 170](#) and [Adding Other Black Box Attributes, on page 174](#).
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 166](#).

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.

```

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);

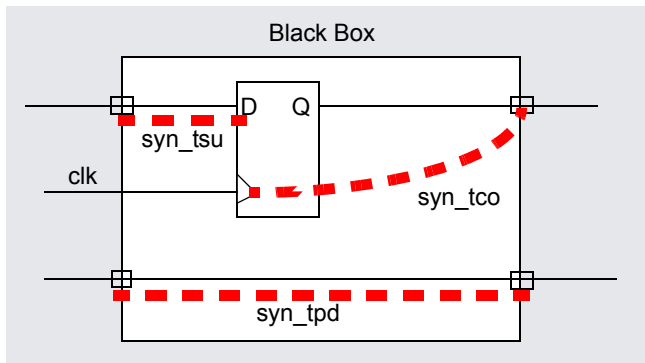
```

4. Add timing constraints and attributes. See [Adding Black Box Timing Constraints](#), on page 170 and [Adding Other Black Box Attributes](#), on page 174.

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`.



1. Define the instance as a black box, as described in [Instantiating Black Boxes and I/Os in Verilog](#), on page 166 or [Instantiating Black Boxes and I/Os in VHDL](#), on page 168.

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_tcoll : string;
attribute syn_tcol2 : string;
```

- Define the constraints in either of these ways:

VHDL syntax	attribute <i>attributeName</i> < <i>n</i> > : " <i>att_value</i> "
----------------	--

Verilog-style notation	attribute <i>attributeName</i> < <i>n</i> > of <i>bbox_name</i> : component is " <i>att_value</i> "
---------------------------	--

The following table shows the appropriate syntax for *att_value*. See the *Reference Manual* for complete syntax information.

Attribute	Value Syntax
<code>syn_tsu<n></code>	<code>bundle -> [!]clock = value</code>
<code>syn_tco<n></code>	<code>[!]clock -> bundle = value</code>
<code>syn_tpd<n></code>	<code>bundle -> bundle = value</code>

- `<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 *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.

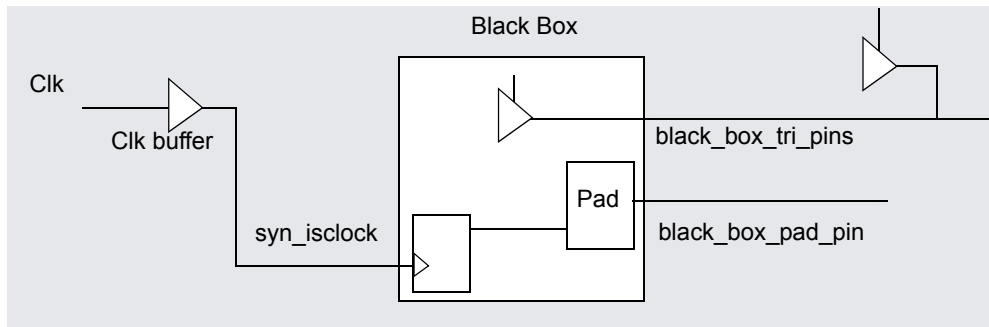
The resulting constraint file contains syntax like this:

```
define_attribute v:{blackboxModule} attribute<n> {attributeValue}
```

6. Synthesize the design, and check black box timing.

Adding Other Black Box Attributes

Besides black box timing constraints, you can also add other attributes to define pin types on the black box. You cannot use the attributes for all technologies. Check the *Reference Manual* for details about which technologies are supported.



1. To specify that a clock pin on the black box has access to global clock routing resources, use `syn_isclock`.

Depending on the technology, different clock resources are inserted. For Microsemi it inserts CLKBUF.

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, like Microsemi technologies.

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 175
- [Defining State Machines in VHDL](#), on page 176
- [Specifying FSMs with Attributes and Directives](#), on page 177

For information about the attributes used to define state machines, see [Running the FSM Compiler](#), on page 220.

Defining State Machines in Verilog

The synthesis software recognizes and automatically extracts state machines from the Verilog source code if you follow these coding guidelines. The software attaches the `syn_state_machine` attribute to each extracted FSM.

For alternative ways to define state machines, see [Defining State Machines in VHDL](#), on page 176 and [Specifying FSMs with Attributes and Directives](#), on page 177.

- 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.

- Use explicit state values for states using `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 whereas `parameter` definitions are local. Local definitions make it easier to reuse certain state names in multiple FSM designs. For example, you might want to reuse common state names like RESET, IDLE, READY, READ, WRITE, ERROR, and DONE. If you use `'define` to assign state names, you cannot reuse a state name because the name has already been taken in the global name space. To use the names multiple times, you have to `'undef` state names between modules and redefine them with `'define` state names in the new FSM modules. 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 coding guidelines. For alternative ways to define state machines, see [Defining State Machines in Verilog, on page 175](#) and [Specifying FSMs with Attributes and Directives, on page 177](#).

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 (see [Optimizing State Machines, on page 218](#)), or you can manually specify attributes to define the state machines. You attach the attributes to the state registers. For detailed information about the attributes and their syntax, see the *Reference Manual*.

The following steps show you how to use attributes to define FSMs for extraction. For alternative ways to define state machines, see [Defining State Machines in Verilog, on page 175](#) and [Defining State Machines in VHDL, on page 176](#).

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	<code>syn_state_machine=1</code>
Prevent state machines from being extracted and optimized	<code>syn_state_machine=0</code>
Prevent the state machine from being optimized away	<code>syn_preserve=1</code>

For information about how to add attributes, see [Specifying Attributes and Directives, on page 142](#).

2. To determine the encoding style used 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 and the FSM Explorer honor this setting. The different values for this attribute are briefly described here:

Situation: If...	syn_encoding Value	Explanation
Area is important	<code>sequential</code>	One of the smallest encoding styles.
Speed is important	<code>onehot</code>	Usually the fastest style and suited to most FPGA styles.
Recovery from an invalid state is important	<code>safe</code> , with another style. For example: <pre>/* synthesis syn_encoding = "safe, onehot" */</pre>	Forces the state machine to reset. For example, where an alpha particle hit in a hostile operating environment causes a spontaneous register change, you can use <code>safe</code> to reset the state machine.
There are <5 states	<code>sequential</code>	Default encoding.
Large output decoder follows the FSM	<code>sequential</code> or <code>gray</code>	Could be faster than <code>onehot</code> , even though the value must be decoded to determine the state. For <code>sequential</code> , more than one bit can change at a time; for <code>gray</code> , only one bit changes at a time, but more than one bit can be hot.
There are a large number of flip-flops	<code>onehot</code>	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 this method for user-defined FSM encoding. For example:

```
attribute syn_enum_encoding of state_type : type is "001 010 101";
```

Inferring RAMs

There are two methods of handling RAMs: instantiation and inference. The software can automatically infer RAMs if they are structured correctly in your source code. For details, see the following sections:

- [Inference Versus Instantiation](#), on page 180
- [Basic Guidelines for Coding RAMs](#), on page 181
- [Specifying RAM Implementation Styles](#), on page 185

For information about generating RAMs with SYNCore, see [Specifying RAMs with SYNCore](#), on page 378.

Inference Versus Instantiation

There are two methods to handle RAMs: instantiation and inference. Many FPGA families provide technology-specific RAMs that you can instantiate in your HDL source code. The software supports instantiation, but you can also set up your source code so that it infers the RAMs. The following table sums up the pros and cons of the two approaches.

Inference in Synthesis

Advantages

Portable coding style
Automatic timing-driven synthesis
No additional tool dependencies

Limitations

Glue logic to implement the RAM might result in a sub-optimal implementation.
Can only infer synchronous RAMs
No support for address wrapping
Pin name limitations means some pins are always active or inactive

Instantiation

Advantages

Most efficient use of the RAM primitives of a specific technology
Supports all kinds of RAMs

Limitations

Source code is not portable because it is technology-dependent.
Limited or no access to timing and area data if the RAM is a black box.
Inter-tool access issues, if the RAM is a black box created with another tool.

Basic Guidelines for Coding RAMs

Read through the limitations before you start. See [Inference Versus Instantiation, on page 180](#) for information. The following steps describe general rules for coding RAMs so that the compiler infers them; to ensure that they are mapped to the vendor-specific implementation you want, see [Specifying RAM Implementation Styles, on page 185](#).

1. Make sure that the RAM meets minimum size and address width requirements for your technology. The software implements RAMs that are smaller than the minimum as registers.
2. Structure the assignment to a VHDL signal/Verilog register as follows:
 - To infer a RAM, structure the code as an indexed array or a case structure. Code it as a two-dimensional array (VHDL) or memory (Verilog) with writes to one process.
 - Control the structure with a clock edge and a write enable.

The software extracts RAMs even if write enables are tied to true (VCC), if you have complex write enables coded in nested if statements, or if you have RAMs with synchronous resets.

3. For a single-port RAM, make the address for indexing the write-to the same as the address for the read-from. The following code and figure illustrate how the software infers a single-port RAM.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity ramtest is
port (q : out std_logic_vector(3 downto 0);

      d : in std_logic_vector(3 downto 0);
      addr : in std_logic_vector(2 downto 0);
      we : in std_logic;
      clk : in std_logic);
end ramtest;

architecture rtl of ramtest is

type mem_type is array (7 downto 0) of std_logic_vector
  (3 downto 0);
signal mem : mem_type;
```

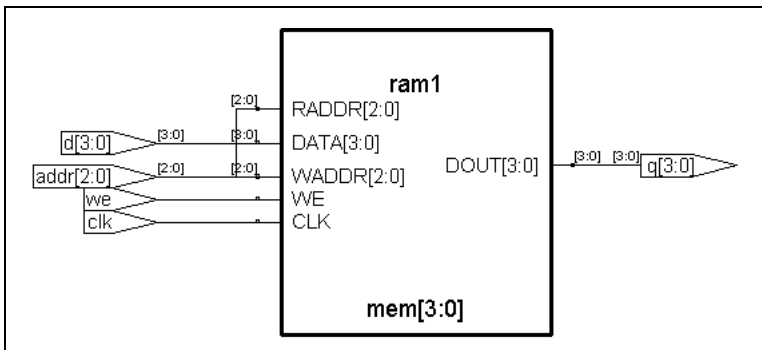
```

begin
  q <= mem(conv_integer(addr));

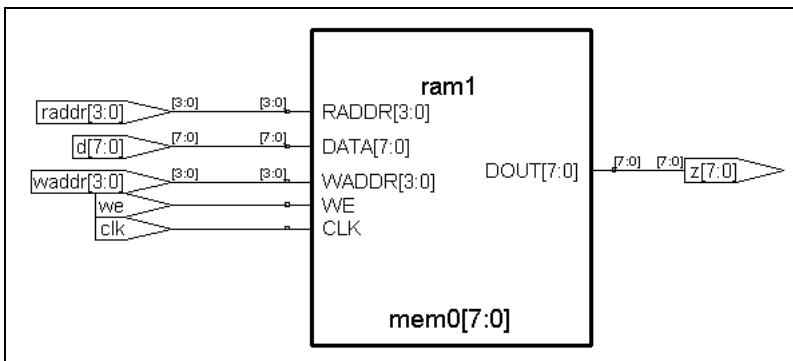
  process (clk) begin
    if rising_edge(clk) then
      if (we = '1') then
        mem(conv_integer(addr)) <= d;
      end if;
    end if;
  end process;

end rtl;

```



4. For a dual-port RAM, make the write-to and read-from addresses different. The following figure and code example illustrate how the software infers a dual-port RAM.



```
module ram16x8(z, raddr, d, waddr, we, clk);
output [7:0] z;
input [7:0] d;
input [3:0] raddr, waddr;
input we;
input clk;
reg [7:0] z;
reg [7:0] mem0, mem1, mem2, mem3, mem4, mem5, mem6, mem7;
reg [7:0] mem8, mem9, mem10, mem11, mem12, mem13, mem14, mem15;
always @(mem0 or mem1 or mem2 or mem3 or mem4 or mem5 or mem6 or
        mem7 or mem8 or mem9 or mem10 or mem11 or mem12 or mem13 or
        mem14 or mem15 or raddr)
begin
    case (raddr[3:0])
        4'b0000: z = mem0;
        4'b0001: z = mem1;
        4'b0010: z = mem2;
        4'b0011: z = mem3;
        4'b0100: z = mem4;
        4'b0101: z = mem5;
        4'b0110: z = mem6;
        4'b0111: z = mem7;
        4'b1000: z = mem8;
        4'b1001: z = mem9;
        4'b1010: z = mem10;
        4'b1011: z = mem11;
        4'b1100: z = mem12;
        4'b1101: z = mem13;
        4'b1110: z = mem14;
        4'b1111: z = mem15;
    endcase
end

always @(posedge clk) begin
    if(we) begin
        case (waddr[3:0])
            4'b0000: mem0 = d;
            4'b0001: mem1 = d;
            4'b0010: mem2 = d;
            4'b0011: mem3 = d;
            4'b0100: mem4 = d;
            4'b0101: mem5 = d;
            4'b0110: mem6 = d;
            4'b0111: mem7 = d;
            4'b1000: mem8 = d;
            4'b1001: mem9 = d;
            4'b1010: mem10 = d;
```

```

        4'b1011: mem11 = d;
        4'b1100: mem12 = d;
        4'b1101: mem13 = d;
        4'b1110: mem14 = d;
        4'b1111: mem15 = d;
    endcase
end
end
endmodule

```

5. To infer multi-port RAMs or nrams (certain technologies only), do the following:
 - Target a technology that supports multi-port RAMs.
 - Register the read address.
 - Add the `syn_ramstyle` attribute with a value of `no_rw_check`. If you do not do this, the compiler errors out.
 - Make sure that the writes are to one process. If the writes are to multiple processes, use the `syn_ramstyle` attribute to specify a RAM.
6. For RAMs where inference is not the best solution, use either one of these approaches:
 - Implement them as regular logic using the `syn_ramstyle` attribute with a value of `registers`. You might want to do this if you have to conserve RAM resources.
 - Instantiate RAMs using the black box methodology. Use this method in cases where RAM is implemented in two cells instead of one because the RAM address range spans the word limit of the primitive and the software does not currently support address wrapping. If the address range is 8 to 23 and the RAM primitive is 16 words deep, the software implements the RAM as two cells, even though the address range is only 16 words deep. Refer to the list of limitations in [Inference Versus Instantiation, on page 180](#) and the vendor-specific information referred to in the previous step to determine whether you should instantiate RAMs.
7. Synthesize your design.

The compiler infers one of the following RAMs from the source code. You can view them in the RTL view:

RAM1	RAM
RAM2	Resettable RAM
NRAM	Multi-port RAM

If the number of words in the RAM primitive is less than the required address range, the compiler generates two RAMs instead of one, leaving any extra addresses unused.

Once the compiler has inferred the RAMs, the mapper implements the inferred RAMs in the technology you specified. For details of how to map the RAM inferred by the compiler to the implementation you want, see [Specifying RAM Implementation Styles, on page 185](#).

Specifying RAM Implementation Styles

You can manually influence how RAMs are implemented with the `syn_ramstyle` attribute, as described in the following procedure. The valid values vary slightly, depending on the technology you use. Check the *Reference Manual* for the values that apply to the technology you choose.

1. If you do not want to use RAM resources, attach the `syn_ramstyle` attribute with a value of `registers` to the RAM instance name or to the signal driven by the RAM.

Initializing RAMs

You can specify startup values for RAMs and pass them on to the place-and-route tools. See the following for ways to set the initial values:

- [Initializing RAMs in Verilog](#), on page 186
- [Initializing RAMs in VHDL](#), on page 187

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 the 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 490](#) in 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 in Verilog, on page 486](#) in 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 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 190](#).

See [RAM Initialization Example, on page 489](#) in the *Reference Manual* for an example of a Verilog single-port RAM.

Initializing RAMs in VHDL

There are two ways to initialize RAMs 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 189](#).

```
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",
    "111110011011101010011110001",
    "111001111000111100101100111",
    "110010110011101110011110001",
    "101001111000111111100110111",
    "10000000000000111111111111",
    "010110000111001111100110111",
    "001101001100011110011110001",
    "000110000111001100101100111",
    "000001100100011010011110001",
    "00000000000000100000000000",
    "000001100100010101100001110",
    "000110000111000011010011000",
    "001101001100010001100001110",
    "010110000111000000011001000",
    "011111111111111000000000000",
    "101001111000110000011001000",
    "110010110011100001100001110",
    "111001111000110011010011000",
    "111110011011100101100001110",
    "11111111111111011111111111",
    "111110011011101010011110001",
    "111001111000111100101100111",
    "110010110011101110011110001",
    "101001111000111111100110111",
    "10000000000000111111111111",
    ,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;

```

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 187](#) and [Initializing RAMs with \\$readmemb and \\$readmemh, on page 190](#).

```

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 490](#) 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 in Verilog, on page 486](#) in the *Reference Manual*.

CHAPTER 7

Specifying Design-Level Optimizations

This chapter covers techniques for optimizing your design using built-in tools or attributes. For vendor-specific optimizations, see [Chapter 11, *Optimizing for Microsemi Designs*](#). It describes the following:

- [Tips for Optimization](#), on page 192
- [Retiming](#), on page 196
- [Preserving Objects from Optimization](#), on page 203
- [Optimizing Fanout](#), on page 209
- [Sharing Resources](#), on page 213
- [Inserting I/Os](#), on page 218
- [Optimizing State Machines](#), on page 218
- [Inserting Probes](#), on page 227

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 192
- [Optimizing for Area](#), on page 193
- [Optimizing for Timing](#), on page 194

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 193](#). For timing optimization, see [Optimizing for Timing, on page 194](#).

- 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 194](#). General optimization tips are in [General Optimization Tips, on page 192](#).

- 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 209](#) 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 213](#) 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 177](#).
- 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 177](#).
- 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.

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 193](#). General optimization tips are in [General Optimization Tips, on page 192](#).

- 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. See [Retiming, on page 196](#) for details.

- 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 209](#) 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 211](#) 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.

Retiming

Some Microsemi technologies. Retiming improves the timing performance of sequential circuits without modifying the source code. It automatically moves registers (register balancing) across combinatorial 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 and DSPs 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.

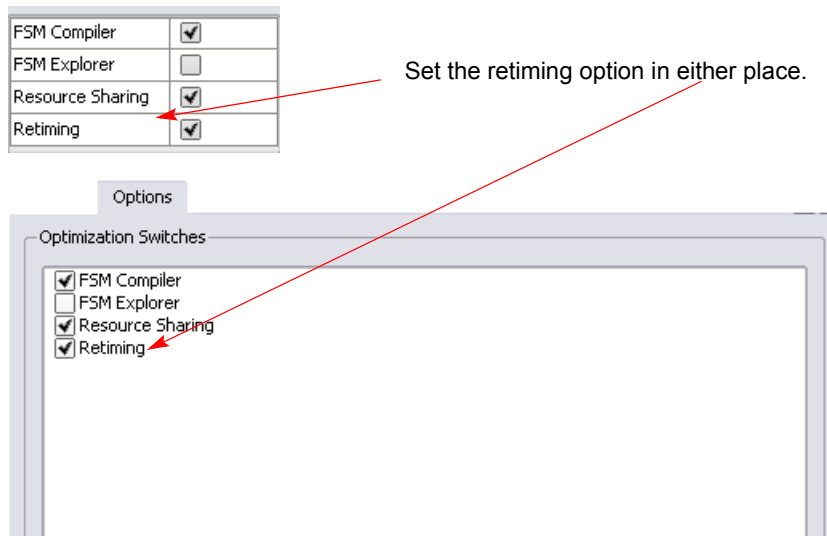
- [Controlling Retiming](#), on page 196
- [Retiming Example](#), on page 198
- [Retiming Report](#), on page 199
- [How Retiming Works](#), on page 200

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.



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 200](#) 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
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 200](#) for details.
- 5. Click Run to start synthesis.

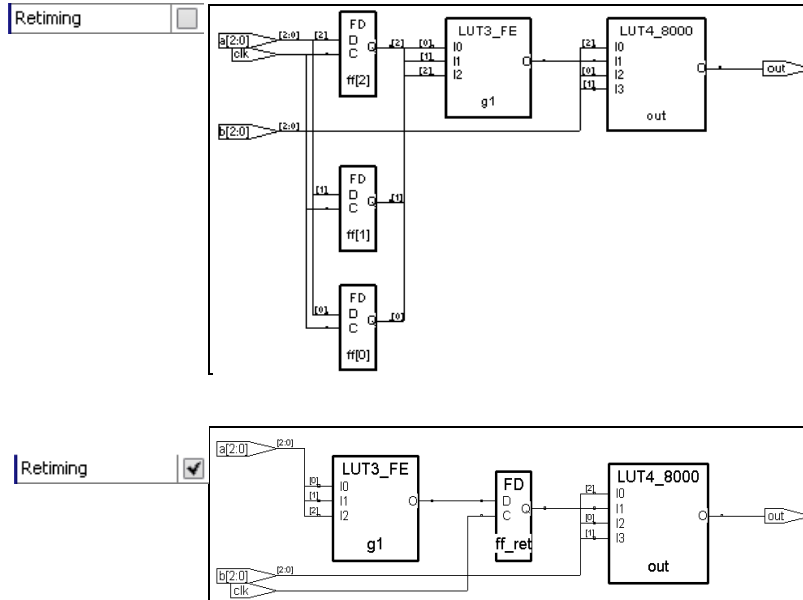
After the LUTs are mapped, the software moves registers to optimize timing. See [Retiming Example, on page 198](#) for an example. The software honors other attributes you set, like `syn_preserve`, `syn_useioff`, and `syn_ramstyle`. See [How Retiming Works, on page 200](#) for details.

Note that the tool might retime registers associated with RAMs and DSPs, 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 199](#) 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.

How Retiming Works

This section describes how retiming works when it moves sequential components (flip-flops). Registers associated with RAMs and DSPs 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 combinatorial 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 combinatorial logic has `syn_ramstyle` set to `registers`, the registers can be retimed into the combinatorial logic.
- Retimed flip-flops are only moved through combinatorial 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 <code>define_reg_input_delay</code> and <code>define_reg_output_delay</code> 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 <code>max_delay</code> constraint. In this case, retiming is performed but the constraint is not forward annotated. (The <code>max_delay</code> value would no longer be valid.)
<code>syn_hier=macro</code>	Does not retime registers in a macro with this attribute.
<code>syn_keep</code>	Does not retime across <code>keepbufs</code> generated because of this attribute.
<code>syn_hier=macro</code>	Does not retime registers in a macro with this attribute.
<code>syn_preserve</code>	Does not retime flip-flops with this attribute set.
<code>syn_probe</code>	Does not retime net drivers with this attribute. If the net driver is a LUT or gate, no flip-flops are retimed across it.
<code>syn_reference_clock</code>	On a critical path, does not retime registers with different <code>syn_reference_clock</code> values together, because the path effectively has two different clock domains.
<code>syn_useioff</code>	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.
<code>syn_allow_retiming</code>	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.

- 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 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 Optimization

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 *Reference Manual*.

To Preserve...	Attach...	Result
Nets	<code>syn_keep</code> on wire or reg (Verilog), or signal (VHDL). For Microsemi designs, use <code>alspreserve</code> as well as <code>syn_keep</code> .	Keeps net for simulation, a different synthesis implementation, or for passing to the place-and-route tool.
Nets for probing	<code>syn_probe</code> on wire or reg (Verilog), or signal (VHDL)	Preserves internal net for probing.
Shared registers	<code>syn_keep</code> on input wire or signal of shared registers	Preserves duplicate driver cells, prevents sharing. See Using <code>syn_keep</code> for Preservation or Replication , on page 204 for details on the effects of applying <code>syn_keep</code> to different objects.
Sequential components	<code>syn_preserve</code> on reg or module (Verilog), signal or architecture (VHDL)	Preserves logic of constant-driven registers, keeps registers for simulation, prevents sharing
FSMs	<code>syn_preserve</code> 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	<code>syn_noprune</code> 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 204
- [Controlling Hierarchy Flattening](#), on page 207
- [Preserving Hierarchy](#), on page 207

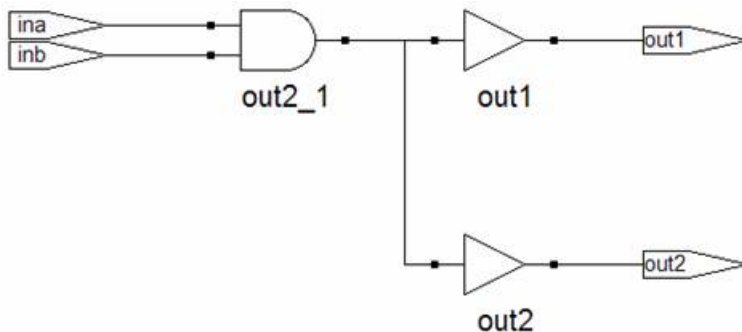
Using syn_keep for Preservation or Replication

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.



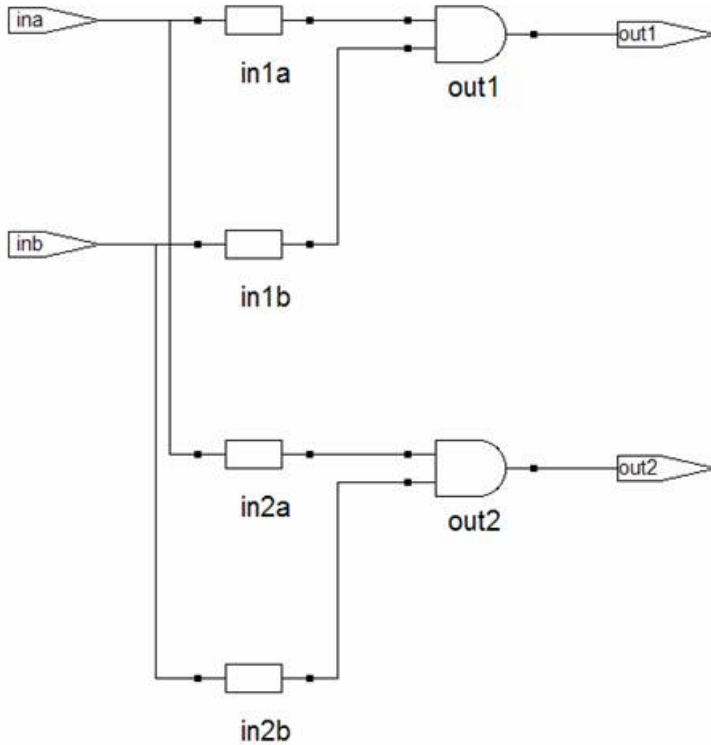
To replicate the AND gate in the previous example, apply `syn_keep` to the input wires, as shown below:

```
module redundant1d(ina,inb,out1,out2);
  input ina,inb;
  output out1,out2;
  wire out1;
  wire out2;

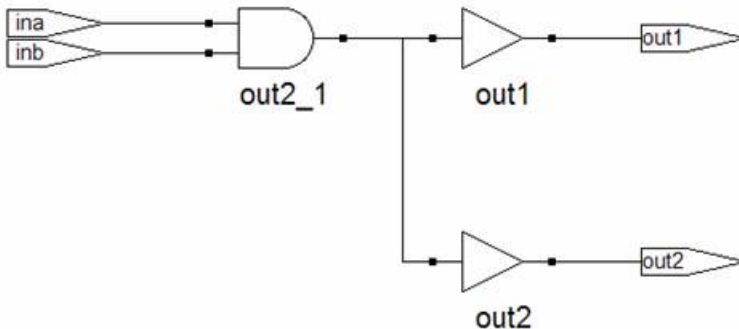
  wire in1a /*synthesis syn_keep = 1*/;
  wire in1b /*synthesis syn_keep = 1*/;
  wire in2a /*synthesis syn_keep = 1*/;
  wire in2b /*synthesis syn_keep = 1 */;

  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.



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 207](#).

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	<code>flatten</code>
Remove the current level of hierarchy without affecting the lower levels	<code>remove</code>
Remove the current level of hierarchy and the lower levels	<code>flatten, remove</code>
Flatten the current level (if needed for optimization)	<code>soft</code>

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 954](#) in the *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 989](#) in the *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 207](#).

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 with no exceptions	hard
Preserve the interface and contents with no exceptions (except ProASIC3 families)	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.

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 *Reference Manual*. This section describes the following:

- [Setting Fanout Limits](#), on page 209
- [Controlling Buffering and Replication](#), on page 211

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 211](#) 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. For certain Microsemi technologies, you can set a global hard fanout limit by doing the following:
 - Select Project-> Implementation Options->Device and type a value for the Fanout Guide option, as described in the previous step.
 - On the same tab, check the Hard Fanout Limit option.

This makes the specified value a global hard fanout limit for the design.

3. 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 (including a global hard limit in Microsemi technologies). However, these limits still function as soft limits, and are replicated or buffered, as described in [Controlling Buffering and Replication, on page 211](#).

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.

4. 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 211](#).

5. 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`.

6. 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.

Controlling Buffering and Replication

To honor fanout limits (see [Setting Fanout Limits, on page 209](#)) 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 combinatorial 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...
<code>syn_maxfan</code> is set on a register output	<code>syn_maxfan</code> is set on input ports in Microsemi ProASIC3 families
<code>syn_replicate</code> is 1	<p><code>syn_replicate</code> is 0.</p> <p>Note that the <code>syn_replicate</code> attribute must be used in conjunction with the <code>syn_maxfan</code> attribute for Microsemi families. The <code>syn_replicate</code> attribute is used only to turn off the replication.</p>
	<code>syn_maxfan</code> is set on a port/net that is driven by a port or I/O pad
	The net driver has a <code>syn_keep</code> or <code>syn_preserve</code> 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.

Sharing Resources

One of the ways you can optimize area is to use resource sharing. 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.

1. Specify resource sharing globally for the whole design, using one of the methods below. Enable the option to improve area; disable it to improve timing.
 - Select Project->Implementation Options->Options, and enable or disable Resource Sharing. Alternatively, enable 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 1041 of the *Reference Manual* for details of the syntax.

```
Verilog  module top(out, in, clk_in) /* synthesis syn_sharing = "on" */;
```

```
VHDL    architecture rtl of top is
        attribute syn_sharing : string;
        attribute syn_sharing of rtl : architecture is "off";
```

- Edit your project file and include the following command. 0 disables and 1 enables resource sharing:

```
set_option -resource_sharing 1|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 mapper does not consider the resource sharing setting, so even if resource sharing is disabled, it can perform resource sharing optimizations to improve results.

2. To specify resource sharing on an individual basis, or to override the global setting, specify the `syn_sharing` attribute for the lower-level module/architecture, using the syntax described in the previous step.

The following examples illustrate resource sharing.

Verilog Example

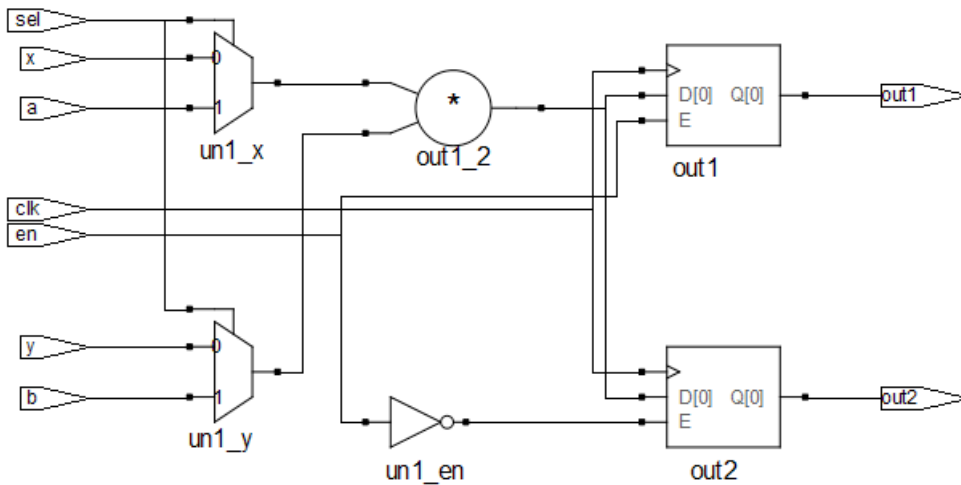
The following example illustrates resource sharing in Verilog. The first diagram shows the equivalent logic with resource sharing enabled, and the second diagram shows the same logic with resource sharing disabled.

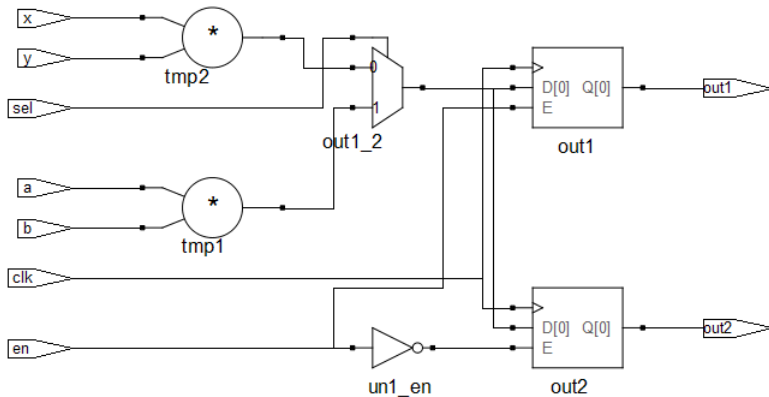
```

module add (a, b, x, y, out1, out2, sel, en, clk);
  input a, b, x, y, sel, en, clk;
  output out1, out2;
  wire tmp1, tmp2;
  assign tmp1 = a * b;
  assign tmp2 = x * y;
  reg out1, out2;

  always@(posedge clk)
    if (en)
      begin
        out1 <= sel ? tmp1: tmp2;
      end
    else
      begin
        out2 <= sel ? tmp1: tmp2;
      end
    end
endmodule

```





VHDL Example

The following example illustrates resource sharing in VHDL. The first diagram shows the equivalent logic with resource sharing enabled, and the second diagram shows the same logic with resource sharing disabled.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity add is
  port (a, b : in std_logic_vector(1 downto 0);
        x, y : in std_logic_vector(1 downto 0);
        clk, sel, en: in std_logic;
        out1 : out std_logic_vector(3 downto 0);
        out2 : out std_logic_vector(3 downto 0)
  );
end add;

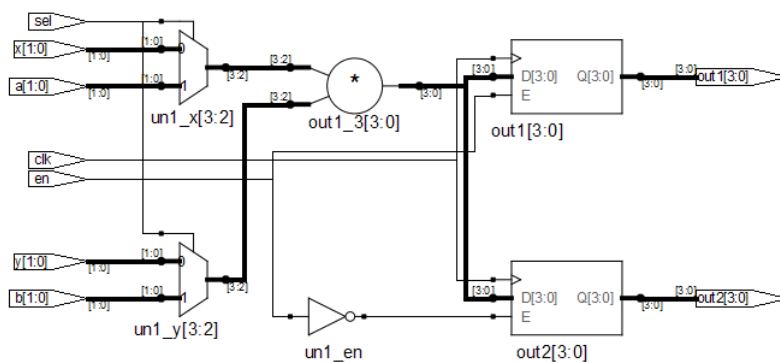
architecture rtl of add is
  signal tmp1, tmp2: std_logic_vector(3 downto 0);
begin
  tmp1 <= a * b;
  tmp2 <= x * y;

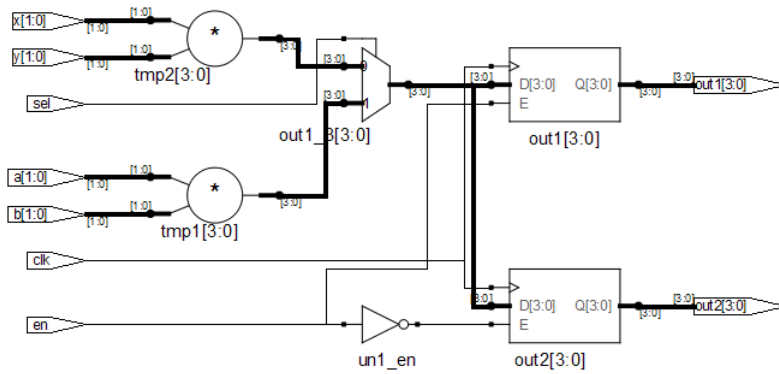
```

```

process(clk) begin
    if clk'event and clk='1' then
        if (en='1') then
            if (sel='1') then
                out1 <= tmp1;
            else
                out1 <= tmp2;
            end if;
        else
            if (sel='1') then
                out2 <= tmp1;
            else
                out2 <= tmp2;
            end if;
        end if;
    end if;
end process;
end rtl;

```





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.

Optimizing State Machines

You can optimize state machines with the symbolic FSM Compiler and the FSM Explorer tools.

- The Symbolic FSM Compiler
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.

- **The FSM Explorer**
A specialized state machine optimizer that explores different encoding styles before selecting the best style. It uses the FSM Compiler to extract state machines, and runs the FSM Compiler automatically if it has not been run.

For more information, see the following:

- [Deciding when to Optimize State Machines](#), on page 219
- [Running the FSM Compiler](#), on page 220
- [Running the FSM Explorer](#), on page 224

Deciding when to Optimize State Machines

The FSM Explorer and the FSM Compiler are automatic tools 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 177](#).

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 315](#).

- To run the FSM Explorer

The FSM Explorer is a tool that examines all the encoding styles before selecting the best option, based on the state machine extraction done by the FSM Compiler. If the FSM Compiler has not been run previously, the Explorer automatically runs it. For more information about using the FSM Explorer, see [Running the FSM Explorer, on page 224](#).

If you are trying to decide whether to use the FSM Compiler or the FSM Explorer to optimize your state machines, remember these points:

- The FSM Explorer runs the FSM Compiler if it has not already been run, because it picks encoding styles based on the state machines that the FSM Compiler extracts.
- Like the FSM Compiler, you use the FSM Explorer to generate better results for your state machines. Unlike the FSM Compiler, which picks an encoding style based on the number of states, the FSM Explorer tries out different encoding styles and picks the best style for the state machine based on overall design constraints.

The trade-off is that the FSM Explorer takes longer to run than the FSM Compiler.

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 220
- [Running the FSM Compiler on Individual FSMs](#), on page 222

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 177.

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 315](#).
 - 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.

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;
```

- 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 177](#). 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 177](#). 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 315](#).

Running the FSM Explorer

1. If you need to customize the extraction process, set attributes.
 - Use `syn_state_machine=0` to specify state machines you do not want to extract and optimize.

```
Verilog  reg [3:0] curstate /* synthesis state_machine */ ;
```

```
VHDL    signal curstate : state_type;
        attribute syn_state_machine : boolean;
        attribute syn_state_machine of curstate : signal is true;
```

- Use `syn_encoding` if you want to set a specific encoding style.

```
Verilog  reg [3:0] curstate /* synthesis syn_encoding="gray" */ ;
```

```
VHDL    signal curstate : state_type;
        attribute syn_encoding : string;
        attribute syn_encoding of curstate : signal is "gray";
```

The FSM Compiler honors the `syn_state_machine` attribute when it extracts state machines, and the FSM Explorer honors the `syn_encoding` attribute when it sets encoding styles. See [Specifying FSMs with Attributes and Directives, on page 177](#) for details.

2. Enable the FSM Explorer by checking the FSM Explorer 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 or Implementation Options buttons.

If you have not checked the FSM Compiler option, checking the FSM Explorer option automatically selects the FSM Compiler option.

3. Click Run to run synthesis.

The FSM Explorer uses the state machines extracted by the FSM Compiler. If you have not run the FSM Compiler, the FSM Explorer invokes the compiler automatically to extract the state machines, instantiate state machine primitives, and optimize them. Then, the FSM Explorer runs through each encoding style for each state machine that does not have a `syn_encoding` attribute and picks the best style. If you have defined an encoding style with `syn_encoding`, it uses that style.

The FSM Compiler writes a description of each state machine extracted and the set of reachable states for each state machine in the log file. The FSM Explorer adds the selected encoding styles. The FSM Explorer also generates a `<design>_fsm.sdc` file that contains the encodings and which is used for mapping.

4. Select View->View Log File and check the log file for the descriptions. The following extract shows the state machine and the reachable states as well as the encoding style, `gray`, set by FSM Explorer.

```

Extracted state machine for register cur_state
State machine has 7 reachable states with original encodings of:
0000001
0000010
0000100
0001000
0010000
0100000
1000000
....
Adding property syn_encoding, value "gray", to instance
cur_state[6:0]
List of partitions to map:
    view:work.Control(verilog)

Encoding state machine work.Control(verilog) -
cur_state_h.cur_state[6:0]
original code -> new code
0000001 -> 000
0000010 -> 001

```

```
0000100 -> 011
0001000 -> 010
0010000 -> 110
0100000 -> 111
1000000 -> 101
```

5. Check the state machine implementation in the RTL and Technology views and in the FSM viewer.

For information about the FSM viewer, see [Using the FSM Viewer, on page 315](#).

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 228](#).

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 *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;
```

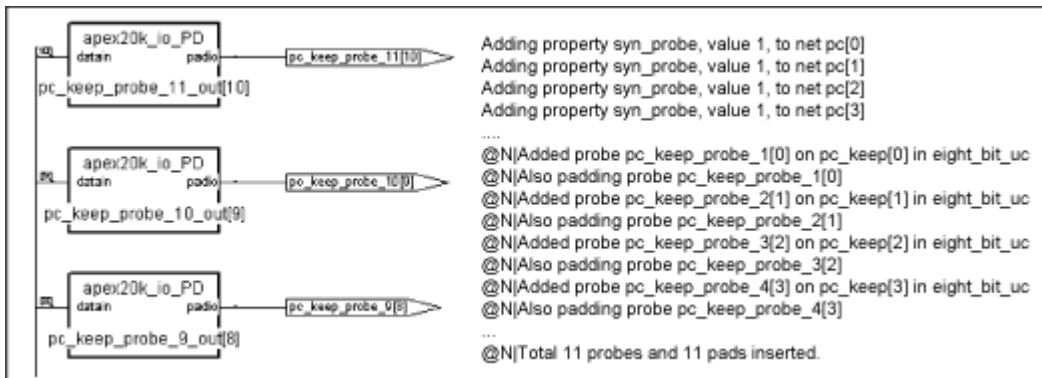
For detailed information about VHDL attributes and sample files, see the *Reference Manual*.

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 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 227](#).

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.

- 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.

CHAPTER 8

Synthesizing and Analyzing 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 232
- [Checking Log Results](#), on page 237
- [Handling Messages](#), on page 244

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](#), on page 232
- [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 234](#).
- 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                                10dev, Build
                                                           reserved
@N: : | premapping output is up to date. No run necessary.
To force a re-synthesis, select [Resynthesize All] in menu [Run]. cks\test.sdc

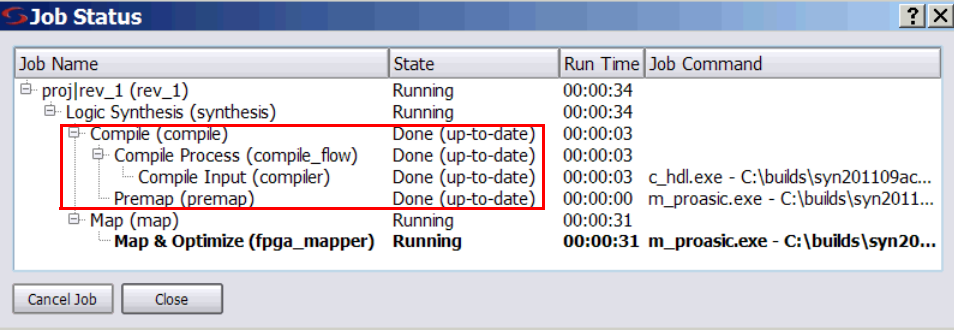
Log file from previous run:                                ds\syn201103_
@L:top_premapping.srr                                     file <C:\buil
#####]                                                    :22|Net U_reg
                                                           4|Net U_gt_re

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
#####]
```



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 Re-synthesis Run



Job Name	State	Run Time	Job Command
proj rev_1 (rev_1)	Running	00:00:34	
Logic Synthesis (synthesis)	Running	00:00:34	
Compile (compile)	Done (up-to-date)	00:00:03	
Compile Process (compile_flow)	Done (up-to-date)	00:00:03	
Compile Input (compiler)	Done (up-to-date)	00:00:03	c_hdl.exe - C:\builds\syn201109ac...
Prelude (premap)	Done (up-to-date)	00:00:00	m_proasic.exe - C:\builds\syn2011...
Map (map)	Running	00:00:31	
Map & Optimize (fpga_mapper)	Running	00:00:31	m_proasic.exe - C:\builds\syn20...

Buttons: Cancel Job, Close

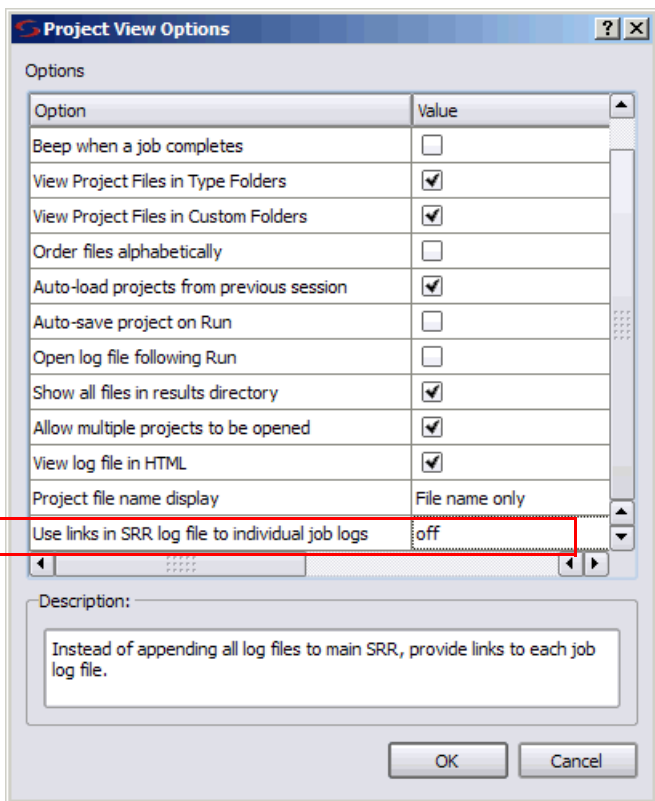
See also:

- [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.



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.

Checking Log Results

You can check the log file for information about the synthesis run. In addition, the Synplify Pro 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 the Log File](#), on page 237
- [Analyzing Results Using the Log File Reports](#), on page 241
- [Using the Watch Window](#), on page 242

Viewing the Log File

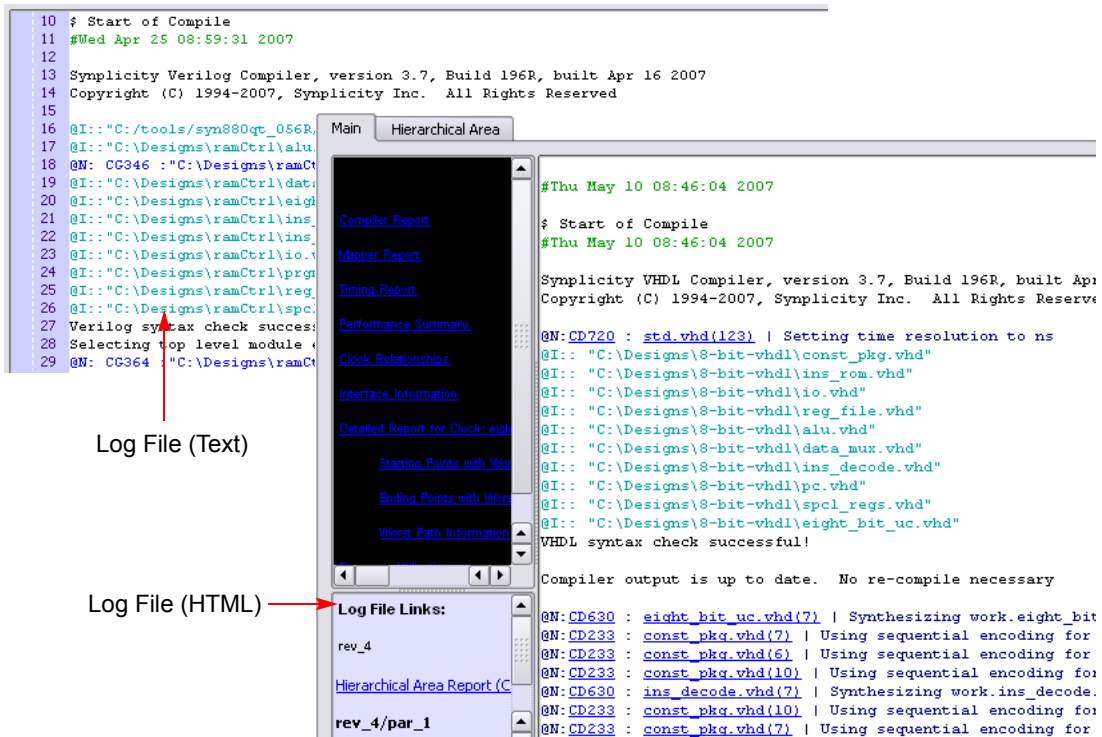
The log file contains the most comprehensive results and information about a synthesis run. The default log file is in HTML format, but there is a text version available too.

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 242](#)) instead of the log file. For details, read through the log file.

1. To view the log file, do one of the following:
 - To view the log file in the default HTML format, select View->Log File or click the View Log button in the Project window. You see the log file in HTML format. Alternatively you can double-click the *designName_srr.htm* file in the Implementation Results view to open the HTML log file.
 - To see a text version of the log file, double-click the *designName.srr* file in the Implementation Results view. A Text Editor window opens with the log file.

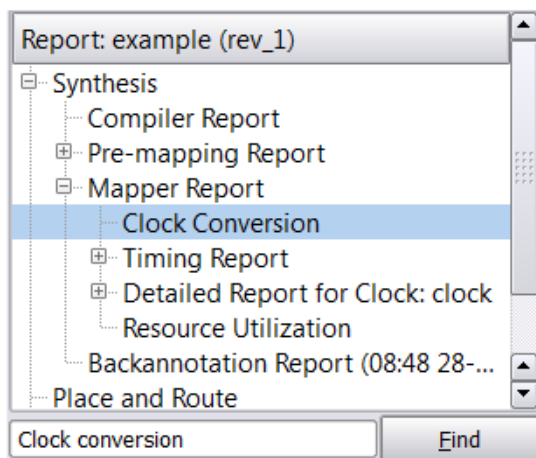
Alternatively, you can set the default to show the text file version instead of the HTML version. Select Options->Project View Options, and toggle off the View log file in HTML option.

The log file lists the compiled files, details of the synthesis run, color-coded errors, warnings and notes, and a number of reports. For information about the reports, see [Analyzing Results Using the Log File Reports, on page 241](#).



2. To navigate in the log file, use the following techniques:

- Use the scroll bars.
- Use the Find command as described in the next step.
- In the HTML file, click the appropriate header to jump to that point in the log file. For example, you can jump to the Starting Points with Worst Slack section.
- In the HTML file, you can use the search field to find a header in the table of contents. Enter all or part of the header name in the search field, then click Find. The log file displays the resulting section.



3. To find information in the log file, you can also select Edit->Find or press Ctrl-f. Fill out the criteria in the form and click OK.

For general information about working in an Editing window, including adding bookmarks, see [Editing HDL Source Files with the Built-in Text Editor](#), on page 34.

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 Synplify Pro product also lets you generate a report for a path between any two designated points, see [Generating Custom Timing Reports with STA, on page 329](#). 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
Detailed information about slack times, constraints, arrival times, etc.	Interface Information
Resource usage	Resource Usage Report

4. 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 244](#) for information. Notes are informational and usually can be ignored. For details about crossprobing and fixing errors, see [Handling Warnings, on page 254](#), [Editing HDL Source Files with the Built-in Text Editor, on page 34](#), and [Crossprobing from the Text Editor Window, on page 294](#).

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.

5. 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 34](#).
6. 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.

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 by going to the Performance Summary section of the log file.
 - Check the slack times. See [Handling Negative Slack, on page 328](#) 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 Synplify Pro timing information about a path between any two designated points, see [Generating Custom Timing Reports with STA, on page 329](#).

2. To check buffers,
 - 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,
 - Go to the Resource Usage Report section at the end of the log file.
 - Check the number and types of components used to determine if you have used too much of your resources.

Using the Watch Window

The Synplify Pro 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 the Log File, on page 237](#).

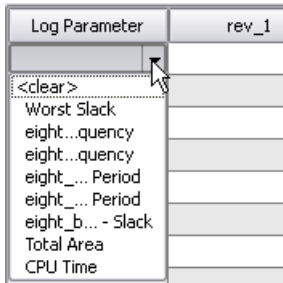
1. Open the Watch window, if needed, by checking View->Watch Window.

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 58](#) 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 192](#).

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

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:

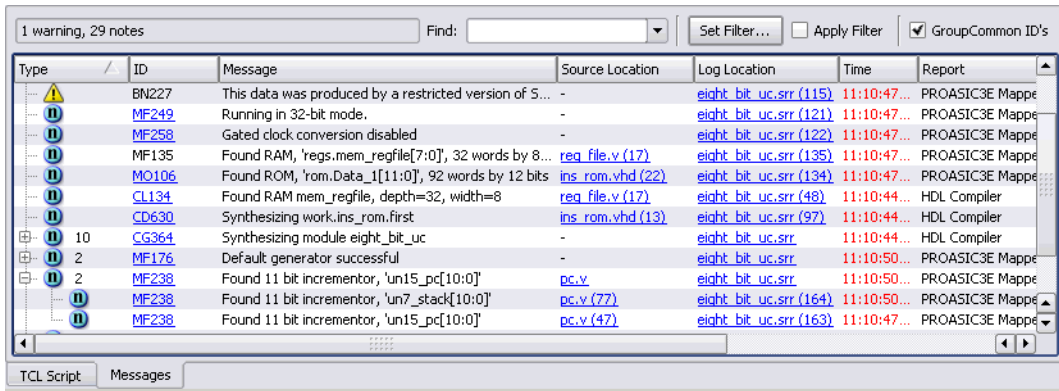
- [Checking Results in the Message Viewer, on page 244](#)
- [Filtering Messages in the Message Viewer, on page 246](#)
- [Filtering Messages from the Command Line, on page 248](#)
- [Automating Message Filtering with a Tcl Script, on page 249](#)
- [Log File Message Controls, on page 251](#)
- [Handling Warnings, on page 254](#)

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 62](#) in the *Reference Manual* for a full description of the window.



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.

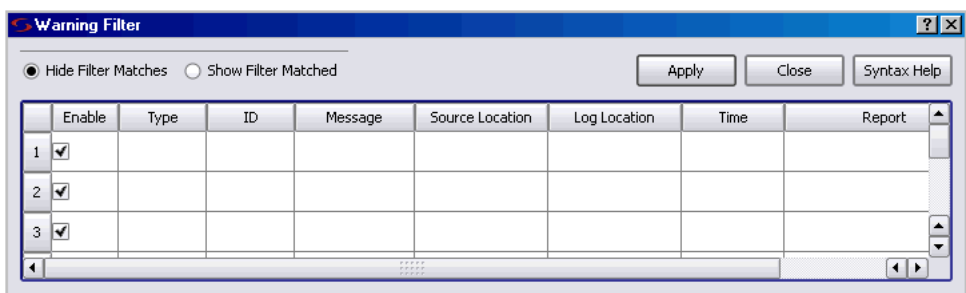
4. To filter the messages, use the procedure described in [Filtering Messages in the Message Viewer, on page 246](#). Crossprobe errors from the message window:
 - If you need more information about how to handle a particular message, click the message ID in the ID column. This opens the documentation 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 254](#).
 - 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 244](#). For the command line equivalent of this procedure, see [Filtering Messages from the Command Line, on page 248](#).

1. Open the message viewer by clicking the Messages tab in the Tcl window as previously described.
2. Click Filter in the message window.

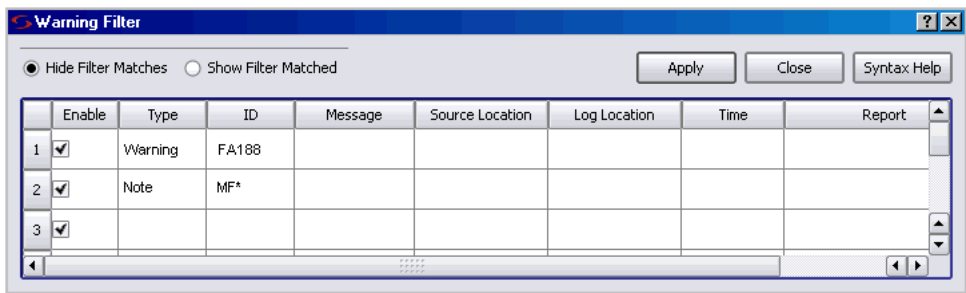
The Warning Filter spreadsheet opens, where you can set up filtering expressions. Each line is one filter expression.



3. Set your display preferences.
 - To hide your filtered choices from the list of messages, click Hide Filter Matches in the Warning Filter window.

- 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 246](#).

1. Type your filter expressions in the Tcl window using the `log_filter` command. For details of the syntax, see [log_filter Tcl Command, on page 1120](#) 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
```

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 246](#) or [Filtering Messages from the Command Line, on page 248](#) for details about creating this file.)

```
log_filter -clear
log_filter -hide_matches
log_filter -field report=="ProASIC3E 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 481](#).

3. Edit the `synhooks.tcl` file so that it reads like the following example. For syntax details, see [synhooks File Syntax, on page 1118](#) 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.

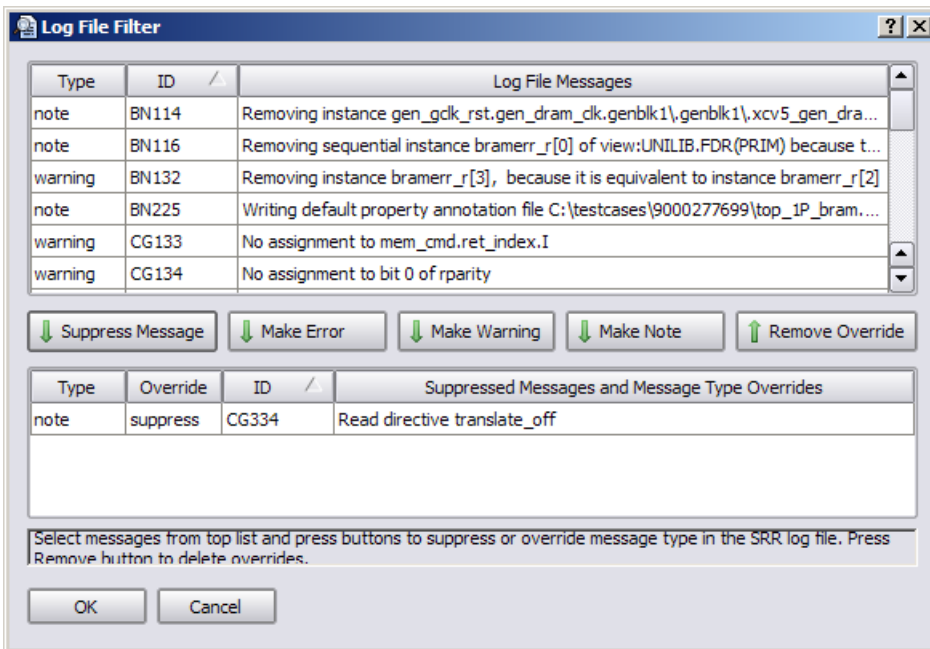
```
source "d:/tcl/smtp_setup.tcl"
proc send_simple_message {recipient email_server subject body}{
    set token [mime::initialize -canonical text/plain -string
        $body]
    mime::setheader $token Subject $subject
    smtp::sendmessage $token -recipients $recipient -servers
        $email_server
    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.

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 *project-Name.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
- 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.

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.

CHAPTER 9

Analyzing with HDL Analyst and FSM Viewer

This chapter describes how to analyze logic in the HDL Analyst and FSM Viewer.

See the following for detailed procedures:

- [Working in the Schematic Views](#), on page 256
- [Exploring Design Hierarchy](#), on page 270
- [Finding Objects](#), on page 278
- [Crossprobing](#), on page 291
- [Analyzing With the HDL Analyst Tool](#), on page 299
- [Using the FSM Viewer](#), on page 315

For information about analyzing timing, see [Chapter 10, *Analyzing Timing*](#).

Working in the Schematic Views

The HDL Analyst includes the RTL and Technology views, which are schematic views used to graphically analyze your design.

For detailed descriptions of these views, see Chapter 2 of the *Reference Manual*. This section describes basic procedures you use in the RTL and Technology views. The information is organized into these topics:

- [Differentiating Between the Views](#), on page 257
- [Opening the Views](#), on page 257
- [Viewing Object Properties](#), on page 258
- [Selecting Objects in the RTL/Technology Views](#), on page 263
- [Working with Multisheet Schematics](#), on page 265
- [Moving Between Views in a Schematic Window](#), on page 266
- [Setting Schematic View Preferences](#), on page 267
- [Managing Windows](#), on page 269

For information on specific tasks like analyzing critical paths, see the following sections:



- [Exploring Object Hierarchy by Pushing/Popping](#), on page 271
- [Exploring Object Hierarchy of Transparent Instances](#), on page 276
- [Browsing to Find Objects in HDL Analyst Views](#), on page 278
- [Crossprobing](#), on page 291
- [Analyzing With the HDL Analyst Tool](#), on page 299

Differentiating Between the Views

- The difference between the RTL and Technology views is that the RTL view is the view generated after compilation, while the Technology view is the view generated after mapping. The RTL view displays your design as a high-level, technology-independent schematic. At this high level of abstraction, the design is represented with technology-independent components like variable-width adders, registers, large muxes, state machines, and so on. This view corresponds to the `srs` netlist file generated by the software in the Synopsys proprietary format. For a detailed description, see Chapter 2 of the *Reference Manual*.
- The Technology view contains technology-specific primitives. It shows low-level, vendor-specific components such as look-up tables, cascade and carry chains, muxes, and flip-flops, which can vary with the vendor and the technology. This view corresponds to the `srm` netlist file, generated by the software in the Synopsys proprietary format. For a detailed description, see Chapter 2 of the *Reference Manual*.

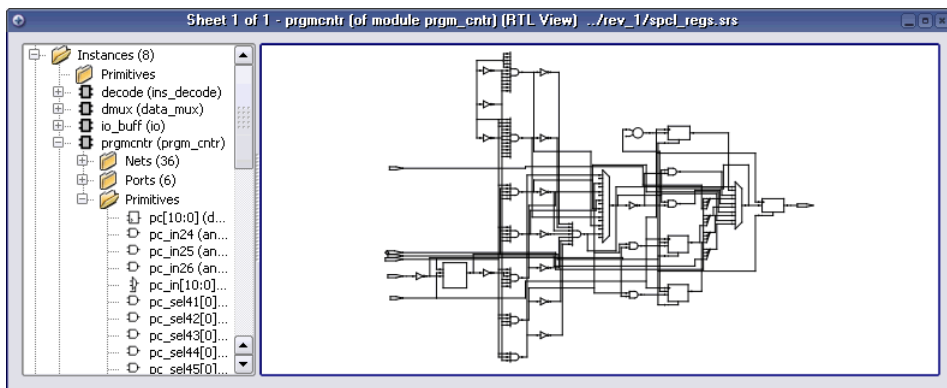
Opening the Views

The procedure for opening an RTL or Technology view is similar; the main difference is the design stage at which these views are available.

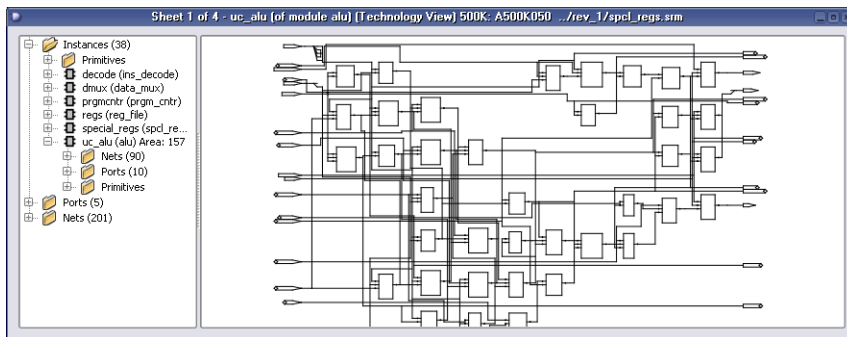
To open an RTL view...	<p>Start with a compiled design.</p> <p>To open a hierarchical RTL view, do one of the following:</p> <ul style="list-style-type: none"> • Select HDL Analyst->RTL->Hierarchical View. • Click the RTL View icon () (a plus sign inside a circle). • Double-click the <code>srs</code> file in the Implementation Results view. <p>To open a flattened RTL view, select HDL Analyst->RTL->Flattened View.</p>
To open a Technology view...	<p>Start with a mapped (synthesized) design.</p> <p>To open a hierarchical Technology view, do one of the following:</p> <ul style="list-style-type: none"> • Select HDL Analyst->Technology->Hierarchical View. • Click the Technology View icon (NAND gate icon ) (a NAND gate icon). • Double-click the <code>srm</code> file in the Implementation Results view. <p>To open a flattened Technology view, select HDL Analyst->Technology->Flattened View.</p>

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 the window contains the name of the view, the kind of view, hierarchical level, and the number of sheets in the schematic. See [Hierarchy Browser, on page 71](#) in the *Reference Manual* for a description of the Hierarchy Browser.

RTL View



Technology View



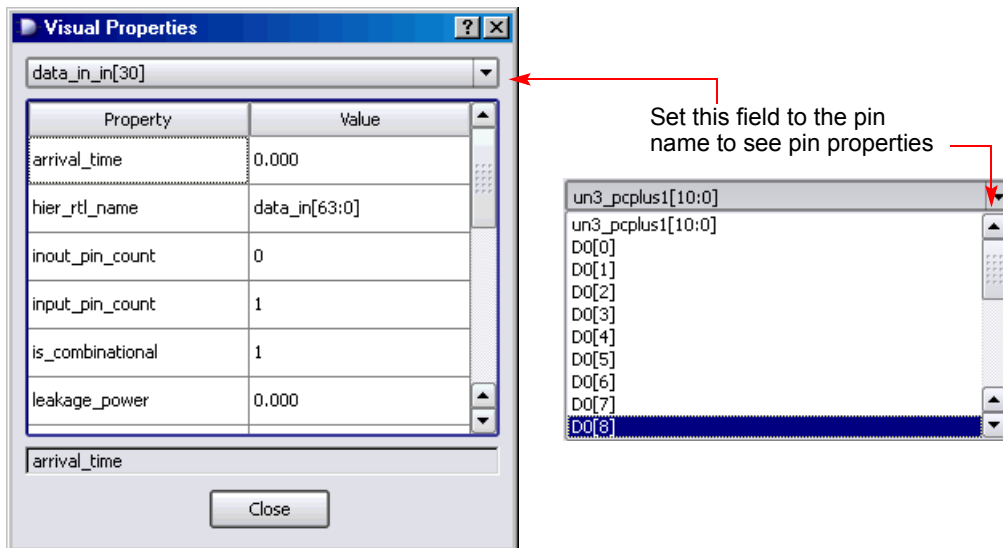
Viewing Object Properties

There are a few ways in which you can view the properties of objects.

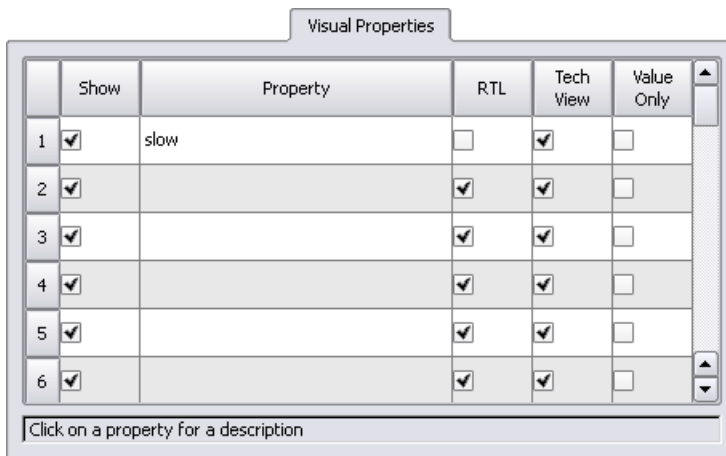
1. To temporarily display the properties of a particular object, hold the cursor over the object. A tooltip temporarily displays the information at the cursor and in the status bar at the bottom of the tool window.

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.



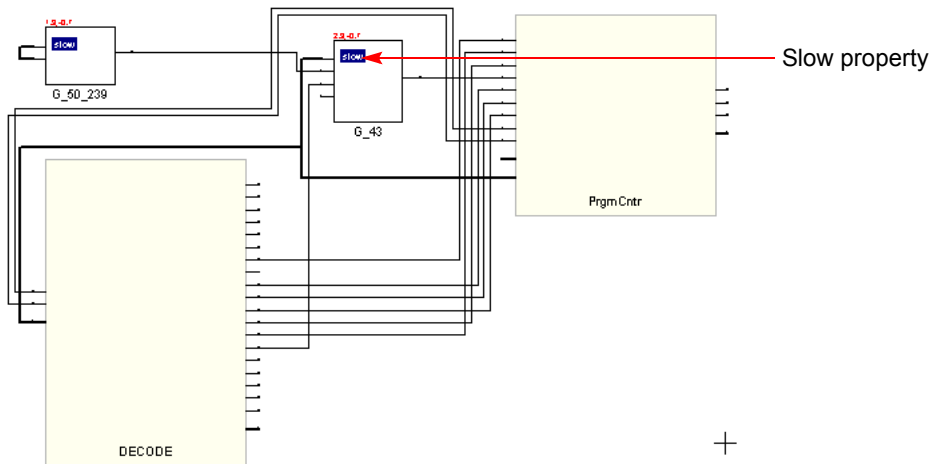
3. To flag objects by property, do the following with an open RTL/Technology view:
 - Set the properties you want to see by selecting Options->HDL Analyst Options->Visual Properties, and selecting the properties from the pull-down list. Some properties are only available in certain views.



- 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 HDL Analyst annotates all objects in the current view that have the specified property with a rectangular flag that contains the property name and value. The software uses different colors for different properties, so you can enable and view many properties at the same time.

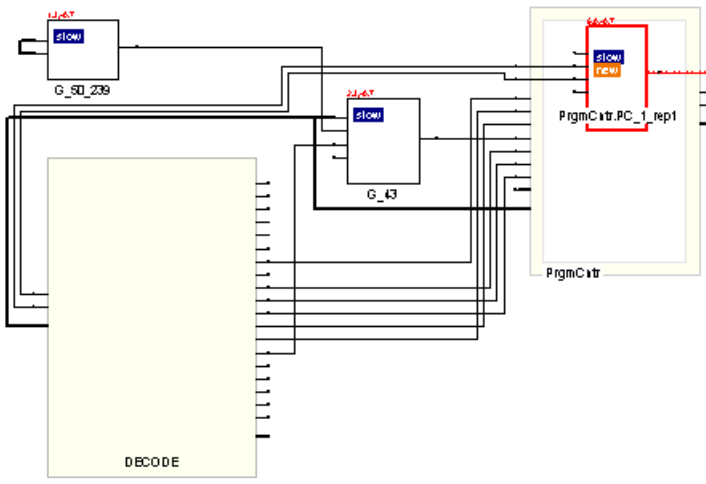
Example: Slow and New Properties

You can view objects with the slow property when you are analyzing your critical path. All objects with this property do not meet the timing criteria. The following figure shows a filtered view of a critical path, with slow instances flagged in blue.



When you are working with filtered views, you can use the New property to quickly identify 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. This can be useful when you are debugging your design.

The following figure expands one of the pins from the previous filtered view. The new instance added to the view has two flags: new and slow.



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, 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.

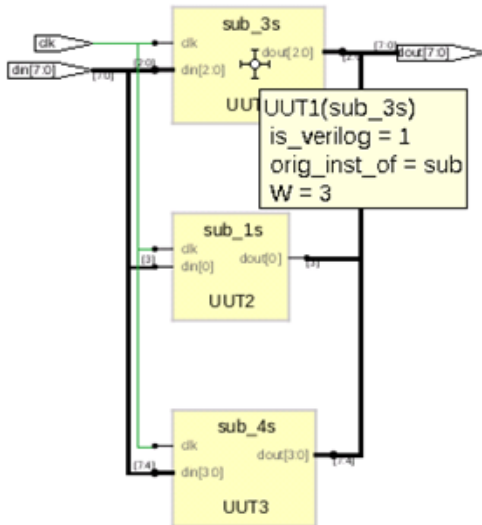
Top.v

```
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.
Multiple objects	Use one of these methods: <ul style="list-style-type: none"> • 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 278. • Use Find to select the objects you want. See Using Find for Hierarchical and Restricted Searches, on page 280.
Objects by type (instances, ports, nets)	Use Edit->Find to select the objects (see Browsing With the Find Command, on page 280), 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: <ul style="list-style-type: none"> • 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 291](#) 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.

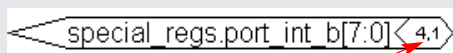
These options set a limit on the number of objects displayed on an unfiltered and filtered schematic sheet, respectively. A low Filtered Instances value can cause lower-level logic inside a transparent instance to be displayed on a separate sheet. The sheet numbers are indicated inside the empty transparent instance.

2. To navigate through a multisheet schematic, refer to this table. It summarizes common operations and ways to navigate.

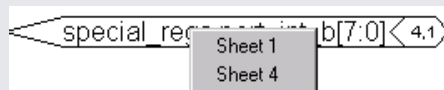
To view...	Use one of these methods...
Next sheet or previous sheet	<p>Select View->Next/Previous Sheet.</p> <p>Press the right mouse button and draw a horizontal mouse stroke (left to right for next sheet, right to left for previous sheet).</p> <p>Click the icons: Next Sheet (📄➡) or Previous Sheet (📄⬅).</p> <p>Press Shift-right arrow (Next Sheet) or Shift-left arrow (Previous sheet).</p> <p>Navigate with View->Back and View ->Forward if the next/previous sheets are part of the display history.</p>
A specific sheet number	<p>Select View->View Sheets and select the sheet.</p> <p>Click the right mouse button, select View Sheets from the popup menu, and then select the sheet you want.</p> <p>Press Ctrl-g and select the sheet you want.</p>
Lower-level logic of a transparent instance on separate sheets	<p>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.</p>

To view...	Use one of these methods...
All objects of a certain type	<p>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.</p> <p>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.</p>
Selected items only	Filter the schematic as described in Filtering Schematics, on page 303 .
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



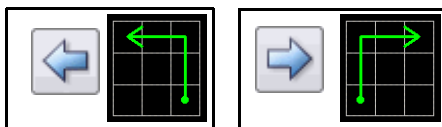
Sheet connector with multisheet popup menu

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 View Preferences

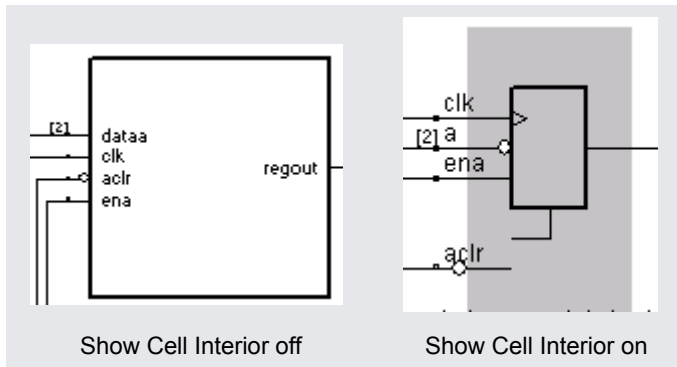
You can set various preferences for the RTL and Technology views from the user interface.

1. Select Options->HDL Analyst Options. For a description of all the options on this form, see [HDL Analyst Options Command, on page 255](#) in the *Reference Manual*.
2. The following table details some common operations:

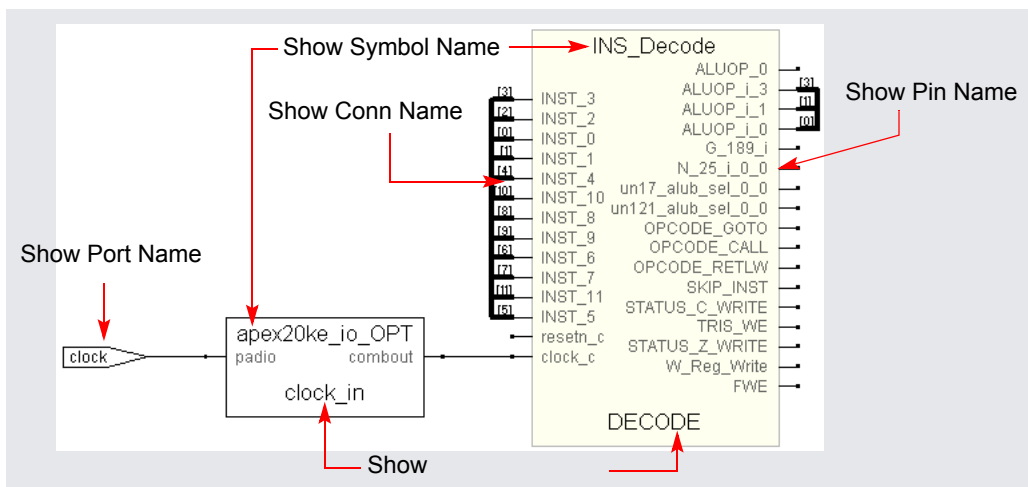
To...	Do this...
Display the Hierarchy Browser	Enable Show Hierarchy Browser (General tab).
Control crossprobing from an object to a P&R text file	Enable Enhanced Text Crossprobing. (General tab)
Determine the number of objects displayed on a sheet.	Set the value with Maximum Instances on the Sheet Size 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 view you open.

3. To view hierarchy within a cell, enable the General->Show Cell Interiors option.



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 307](#) 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 266](#).

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.

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 299](#). The topics include:

- [Traversing Design Hierarchy with the Hierarchy Browser](#), on page 270
- [Exploring Object Hierarchy by Pushing/Popping](#), on page 271
- [Exploring Object Hierarchy of Transparent Instances](#), on page 276

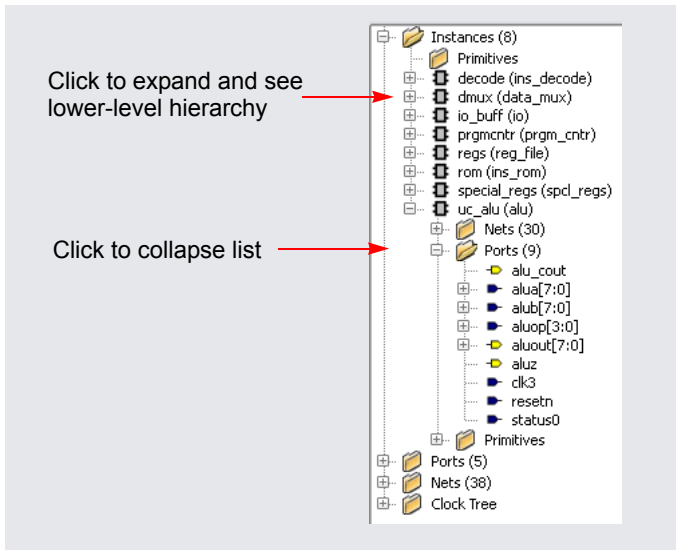
Traversing Design Hierarchy with the Hierarchy Browser

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 271](#) 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 72 in the *Reference Manual* for an explanation of the symbols.

Exploring Object Hierarchy by Pushing/Poppping

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 270. You can access Push/Pop mode with the Push/Pop Hierarchy icon, the Push/Pop Hierarchy command, or mouse strokes.

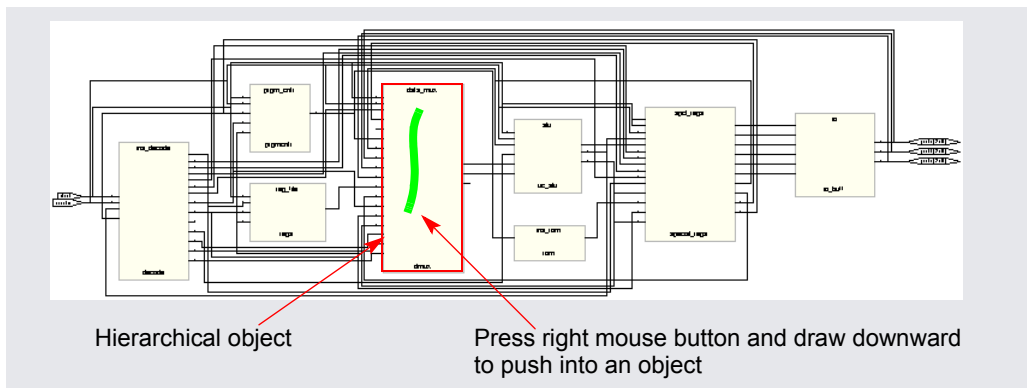
When combined with other commands like filtering and expansion commands, Push/Pop mode can be a very powerful tool for isolating and analyzing logic. See [Filtering Schematics](#), on page 303, [Expanding Pin and Net Logic](#), on page 305, and [Expanding and Viewing Connections](#), on page 309 for 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 272, next
- [Popping up a Hierarchical Level](#), on page 275

Pushing into Objects

In the schematic views, 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).



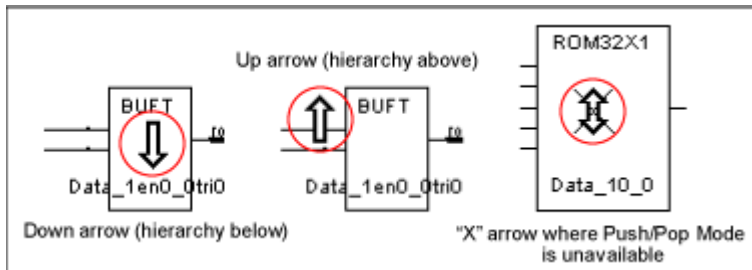
- Technology-specific primitives. The primitives are listed in the Hierarchy Browser in the Technology view, under Instances - Primitives.
- Inferred ROMs and state machines.

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:

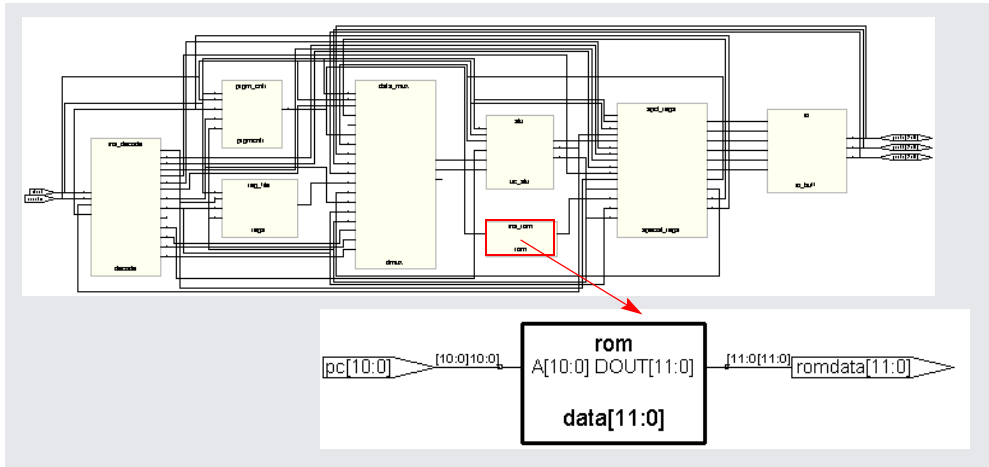
- Select View->Push/Pop Hierarchy.
- Right-click in the Technology view and select Push/Pop Hierarchy from the popup menu.
- Click the Push/Pop Hierarchy icon () in the toolbar (two arrows pointing up and down).
- Press F2.

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.

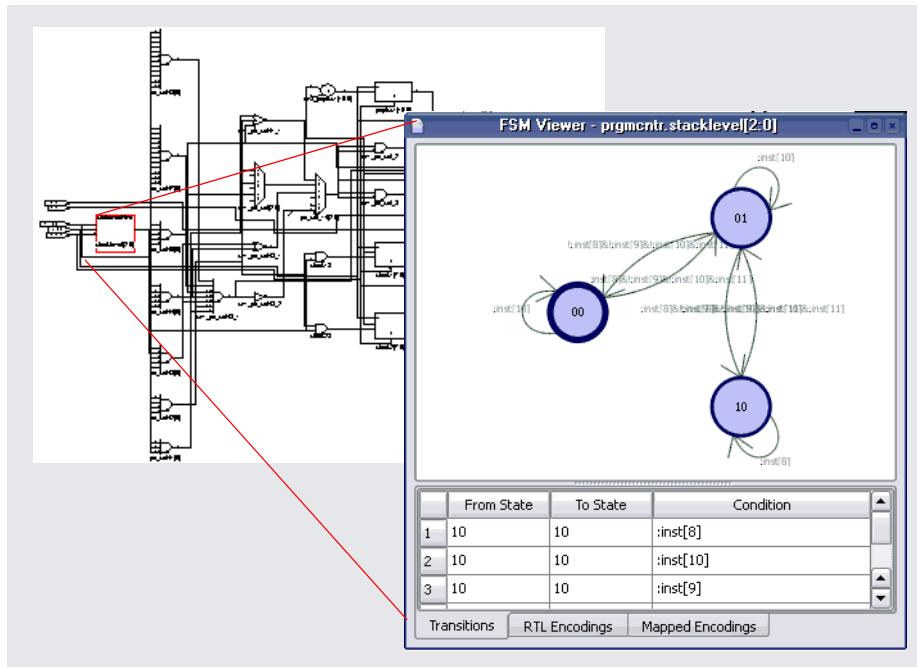


3. To push (descend) into an object, click on the hierarchical object. For a transparent instance, you must click on the pale yellow border. The following figure shows the result of pushing into a ROM.

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.

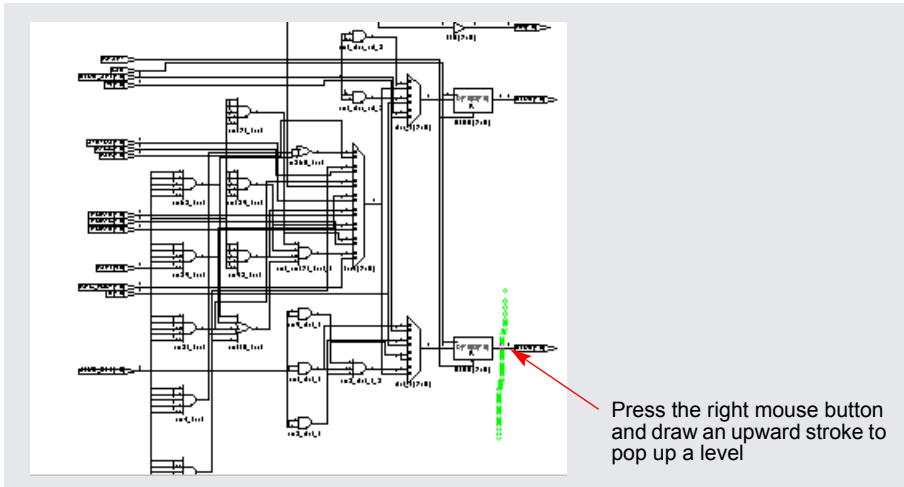


Similarly, you can push into a state machine. When you push into an FSM from the RTL view, you open the FSM viewer where you can graphically view the transitions. For more information, see [Using the FSM Viewer, on page 315](#). If you push into a state machine from the Technology view, you see the underlying 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.



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 272](#) for 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 271](#)).

	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.

Finding Objects

In the schematic views, 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 278
- [Using Find for Hierarchical and Restricted Searches](#), on page 280
- [Using Wildcards with the Find Command](#), on page 283
- [Using Find to Search the Output Netlist](#), on page 288

For information about the Tcl Find command, which you use to locate objects, and create collections, see [Tcl find Command, on page 1128](#) in the *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.

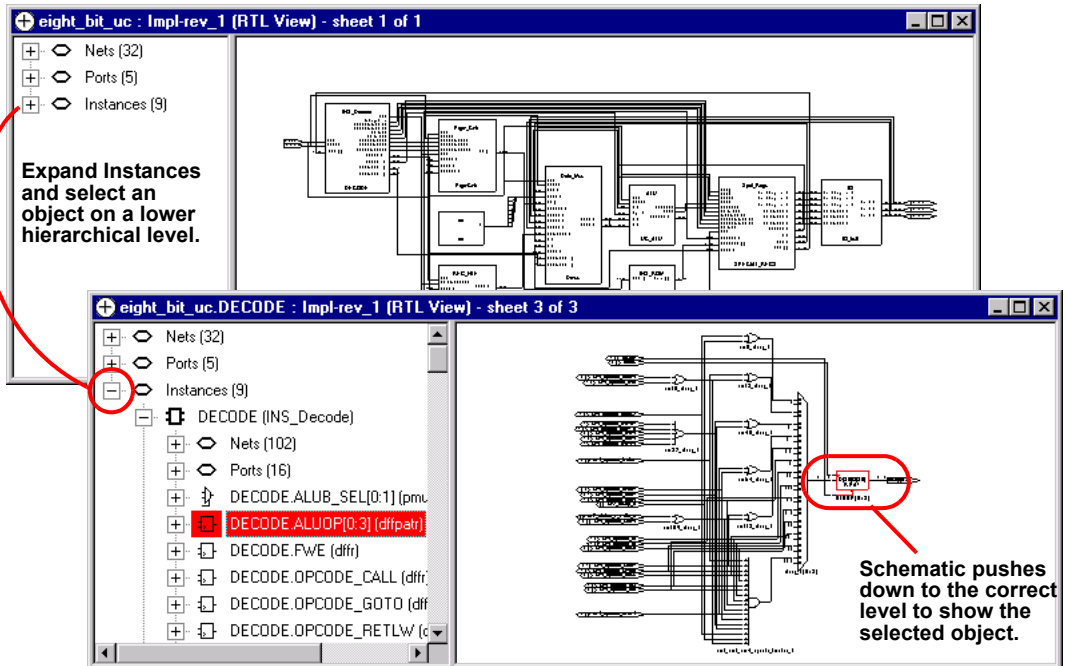
2. To select a range of objects, select the first object in the range. Then, scroll to display the last object in the range. Press and hold the Shift key while clicking the last object in the range.

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 view, 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.
 - 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 278](#)). 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.

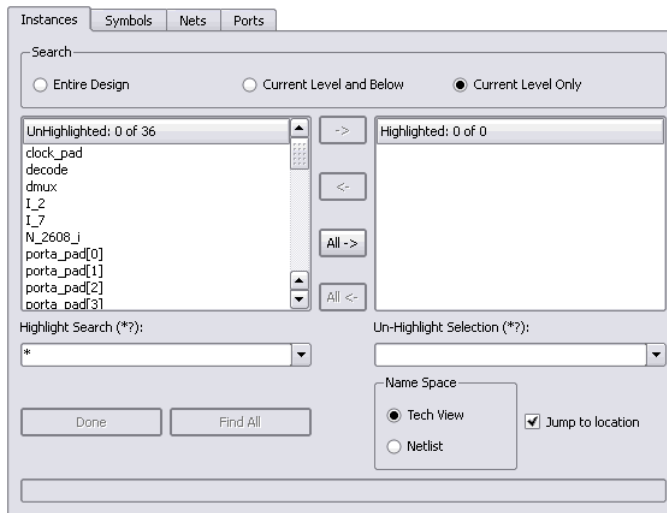
See [Viewing Design Hierarchy and Context, on page 300](#) and [Filtering Schematics, on page 303](#) for details. With a filtered view, the software only searches the filtered instances, unless you set the scope of the search to Entire Design, as described below, in which case Find searches the entire 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 284](#) 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.

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 (`srnm`) 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 288](#).

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 283](#) 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 123](#) 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

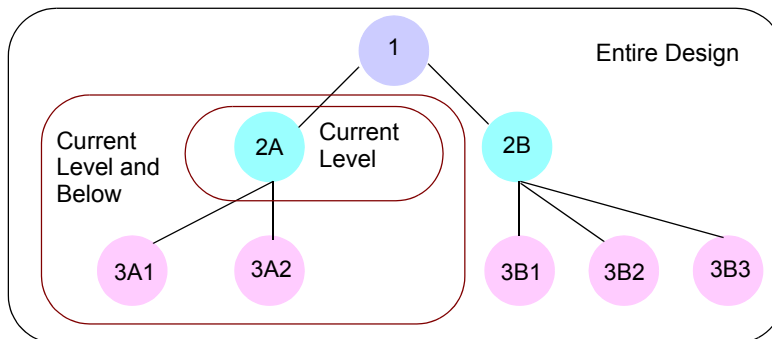
The asterisk and question mark wildcards do not cross hierarchical boundaries, but search each level of hierarchy individually with the search pattern. This default is affected by the following:

- Hierarchical separators

Dots match hierarchy separators, unless you use the backslash escape character in front of the dot (\.). Hierarchical search patterns with a dot (!*.*) 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 285](#) for examples.

How a Wildcard Search Works

1. The starting point of a wildcard search depends on the range set for the search.

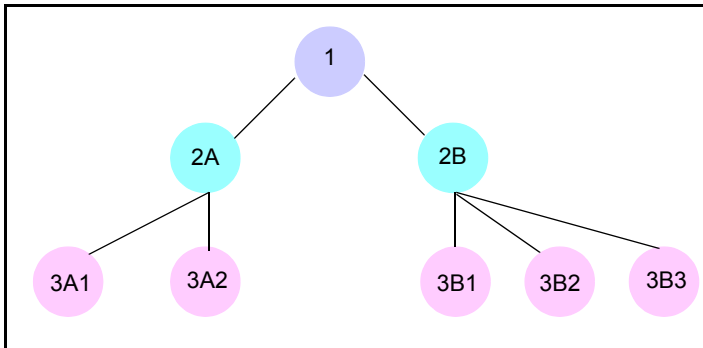
Entire Design	Starts at top level and uses the pattern to search from that level. It then moves to any child levels below the top level and searches them. The software repeats the search pattern at each hierarchical point in the design until it searches the entire design.
Current Level	Starts at the 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 284](#) and [Wildcard Search Examples, on page 285](#) 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.



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

In a simple Tcl search, no character (except the backslash, \) has special meaning. This means that the asterisk matches everything in a string. The FPGA synthesis tools and Synopsys TimeQuest and Design Compiler products confine 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

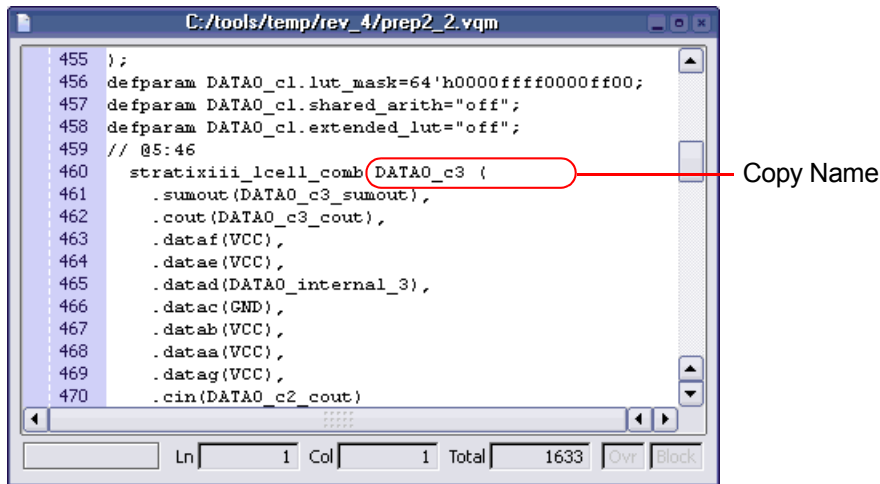
You can combine the Find command with the filtering commands to better effect. Depending on what you want to do, use the Find command first, or a filtering command.

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.

Using Find to Search the Output Netlist

When the synthesis tool creates an output netlist like an edf 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.

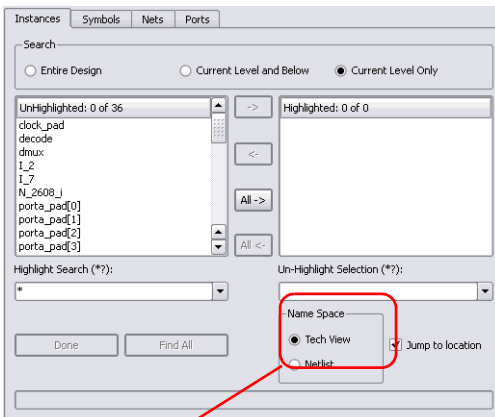
1. Select the output netlist file option in the Implementations Results tab of the Implementation Options dialog box.
2. After you synthesize your design, open your output netlist file and select the name of the object you want to find.



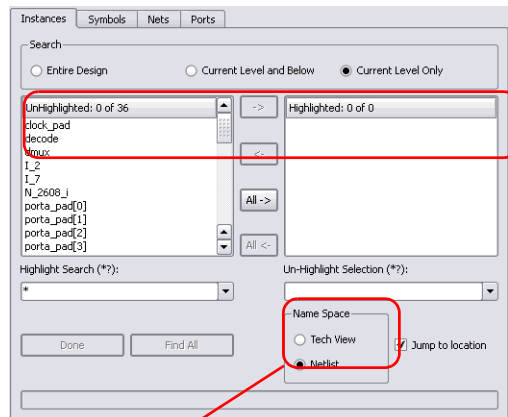
3. Copy the name and open a Technology view.

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



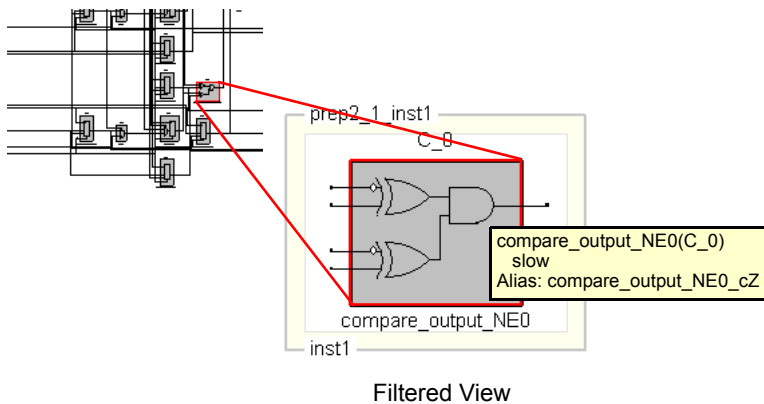
Search by Netlist

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.



The tooltip shows the equivalent name in the Technology view.

6. Double click 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 views. 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:

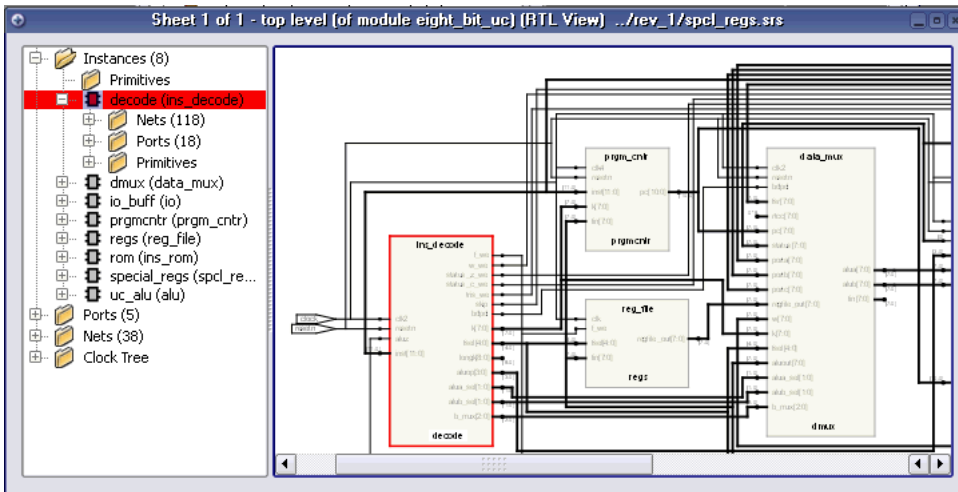
- [Crossprobing within an RTL/Technology View, on page 291](#)
- [Crossprobing from the RTL/Technology View, on page 292](#)
- [Crossprobing from the Text Editor Window, on page 294](#)
- [Crossprobing from the Tcl Script Window, on page 297](#)
- [Crossprobing from the FSM Viewer, on page 298](#)

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

In this example, when you select the DECODE module in the Hierarchy Browser, the DECODE module is automatically selected in the RTL view.

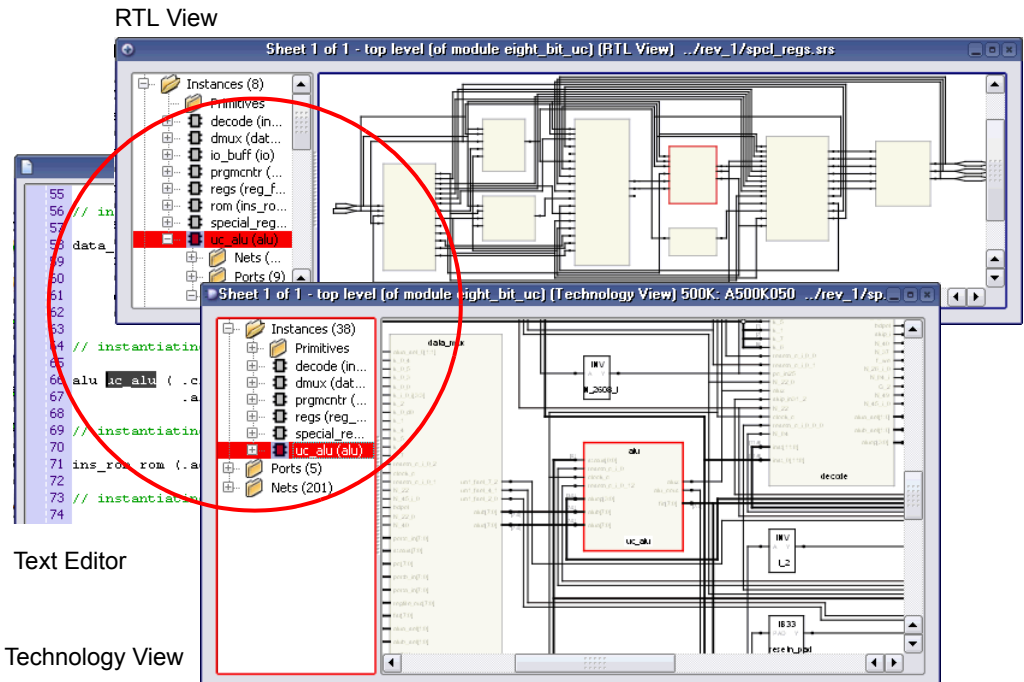


Crossprobing from the RTL/Technology View

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.

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 295](#) 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.

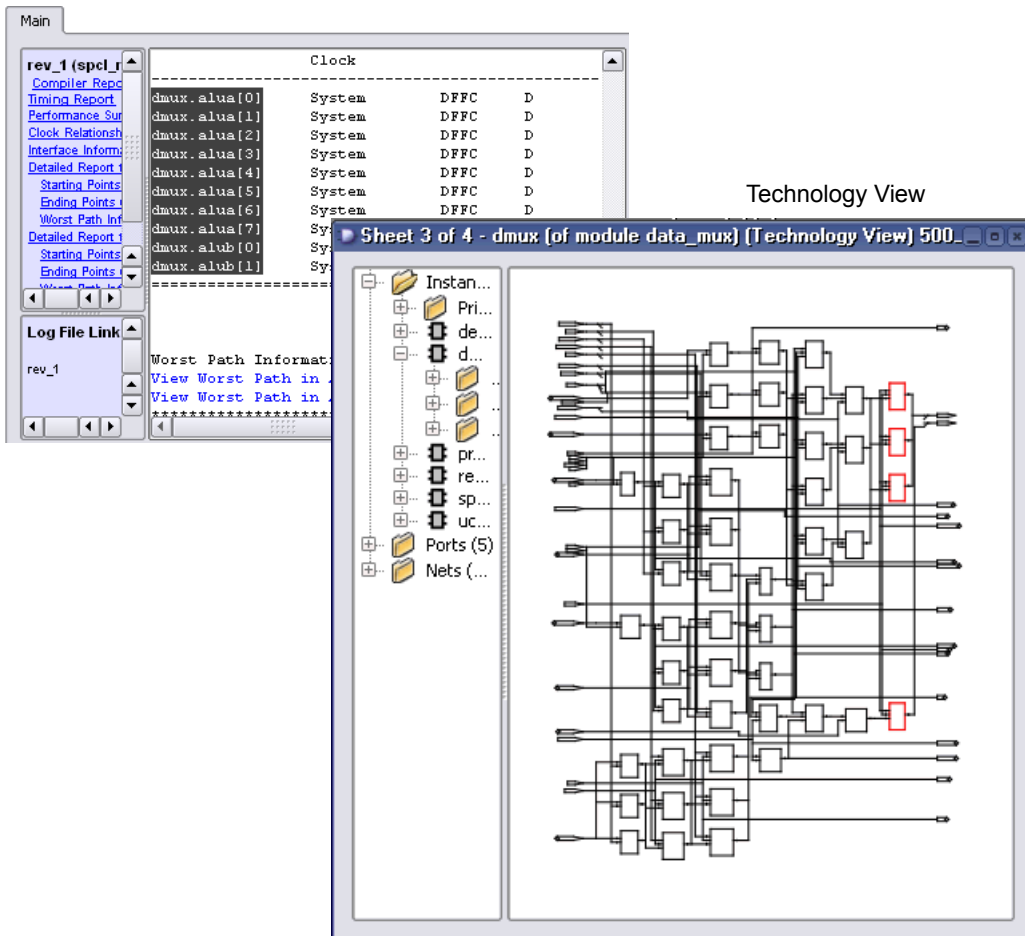
Example of Crossprobing a Path from a Text File

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.

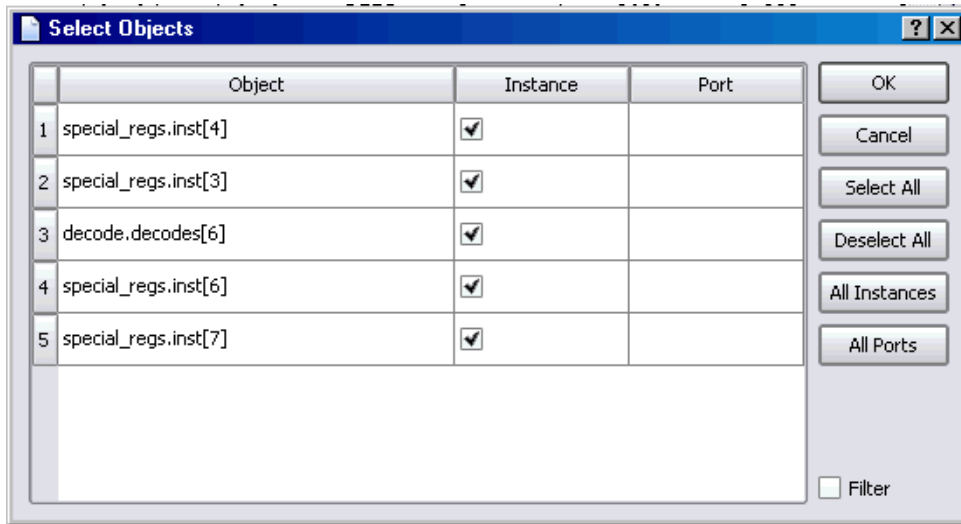
1. Open the log file, the RTL, and Technology views.
2. Select the path objects in the log file.
 - Select the column by pressing **Alt** and dragging the cursor to the end of the column. On the Linux platform, use the key to which the **Alt** function is mapped; this is usually the **Ctrl-Alt** key 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.

Text Editor



- 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.
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 300
- [Filtering Schematics](#), on page 303
- [Expanding Pin and Net Logic](#), on page 305
- [Expanding and Viewing Connections](#), on page 309
- [Flattening Schematic Hierarchy](#), on page 311
- [Minimizing Memory Usage While Analyzing Designs](#), on page 315

For additional information about navigating the HDL Analyst views or using other techniques like crossprobing, see the following:

- [Working in the Schematic Views](#), on page 256
- [Exploring Design Hierarchy](#), on page 270
- [Finding Objects](#), on page 278
- [Crossprobing](#), on page 291

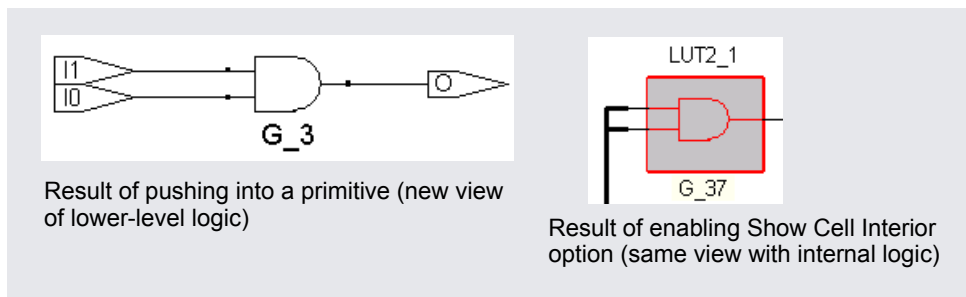
Viewing Design Hierarchy and Context

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 325](#) 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.

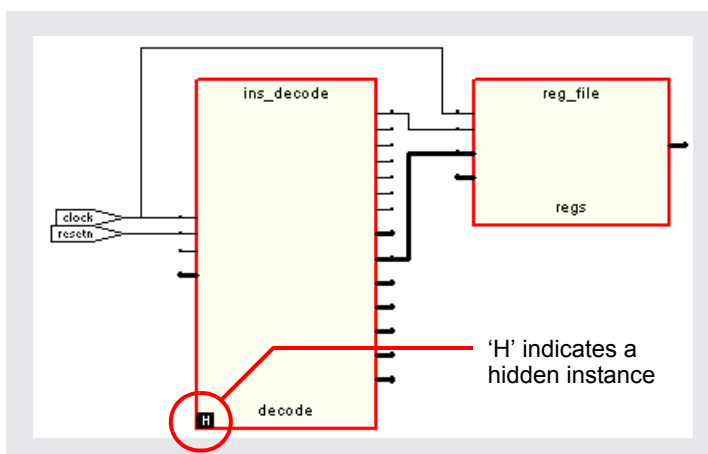
1. To view the internal logic of primitives in your design, do either of the following:
 - To view the logic of an individual primitive, push into it. This generates a new schematic view with the internal details. Click the Back icon to return to the previous view.
 - To view the logic of all primitives 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 view and all subsequent views. If the view is too cluttered with this option on, filter the view (see [Filtering Schematics, on page 303](#)) 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:



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 view.

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.

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 271 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 311 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 311 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

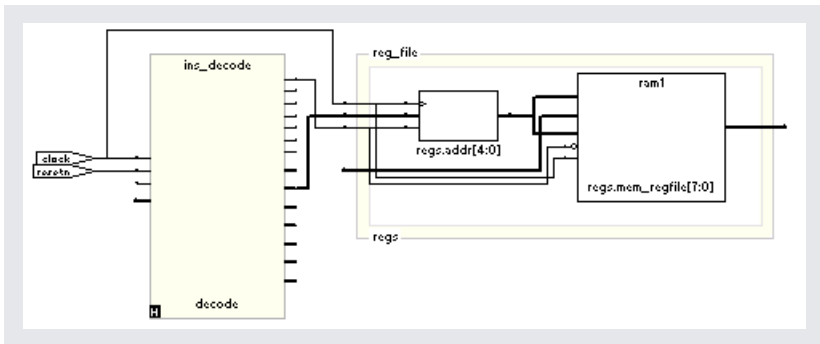
Filtering is a useful first step in analysis, because it focuses analysis on the relevant parts of the design. Some commands, like the Expand commands, automatically generate filtered views; this procedure only discusses manual filtering, where you use the Filter Schematic command to isolate selected 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.

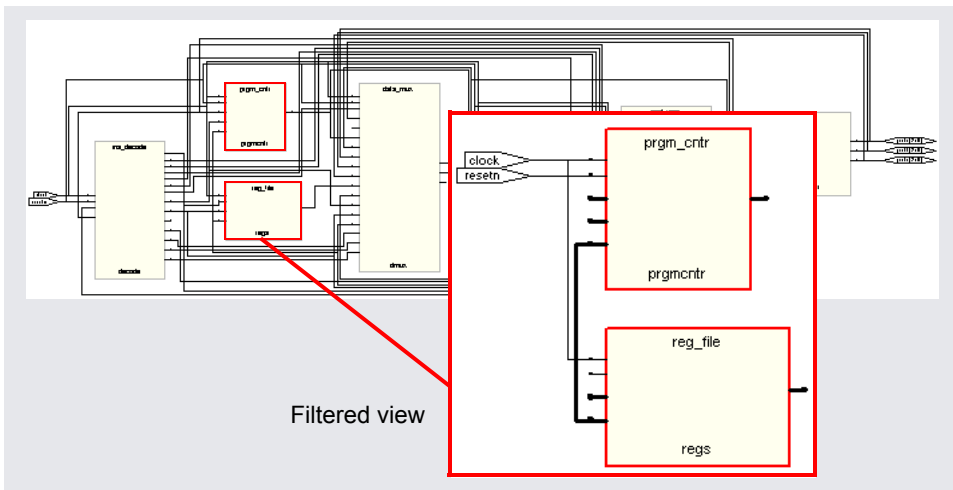
If you filter a hidden instance, the software does not display its internal hierarchy when you filter the design. The following example illustrates this.



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) (.

- 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 300](#) and [Transparent and Opaque Display of Hierarchical Instances, on page 305](#) in 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 300](#), 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 305 and Expanding and Viewing Connections, on page 309
Filter further	Select objects and filter again
Find objects	See Finding Objects, on page 278
Flatten, or hide and flatten	See Flattening Schematic Hierarchy, on page 311 . You can hide transparent or opaque instances.
Crossprobe from filtered view	See Crossprobing from the RTL/Technology View, on page 292

4. To return to the previous schematic view, 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 303](#) and [Flattening Schematic Hierarchy, on page 311](#).

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 309](#).

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 303](#) and [Flattening Schematic Hierarchy, on page 311](#) for details.

1. To expand logic from a pin hierarchically across boundaries, use the following commands.

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 307 .
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 308 .
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 308 .

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.

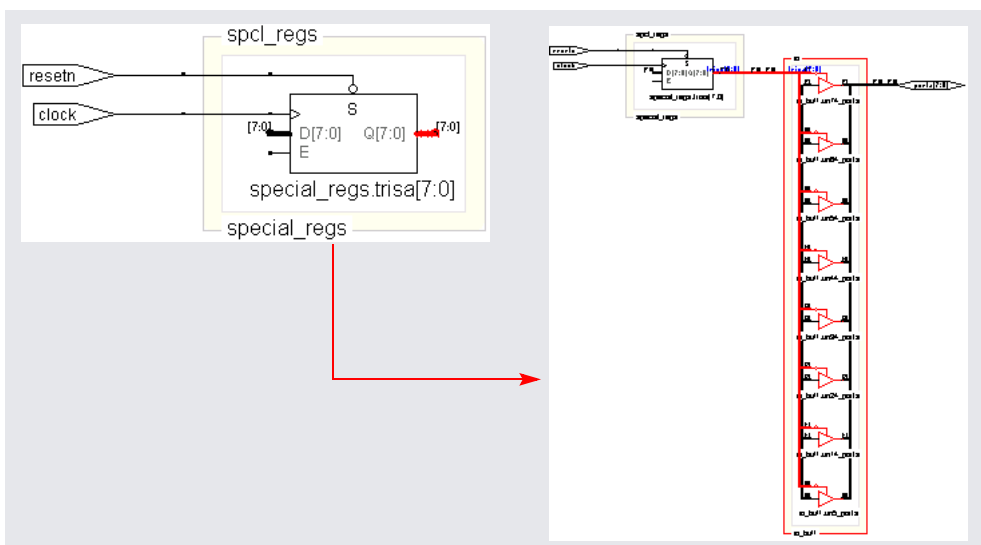
For descriptions of the Expand commands, see [HDL Analyst Menu, on page 230](#) of the *Reference Manual*.

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.

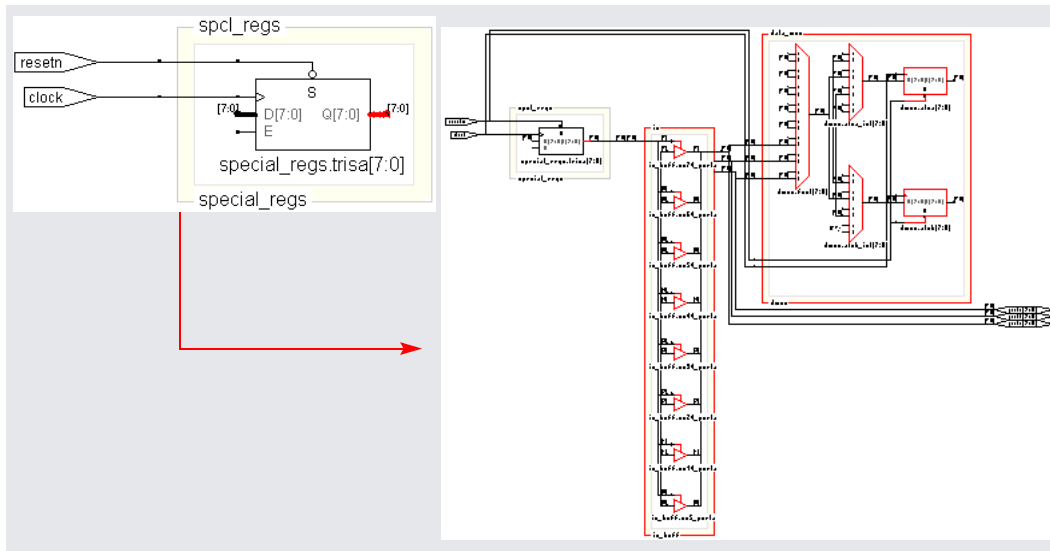
- To expand at the current level only, select the commands from the HDL Analyst->Current Level menu or the right-click popup menu->Current Level.

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 309).
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.

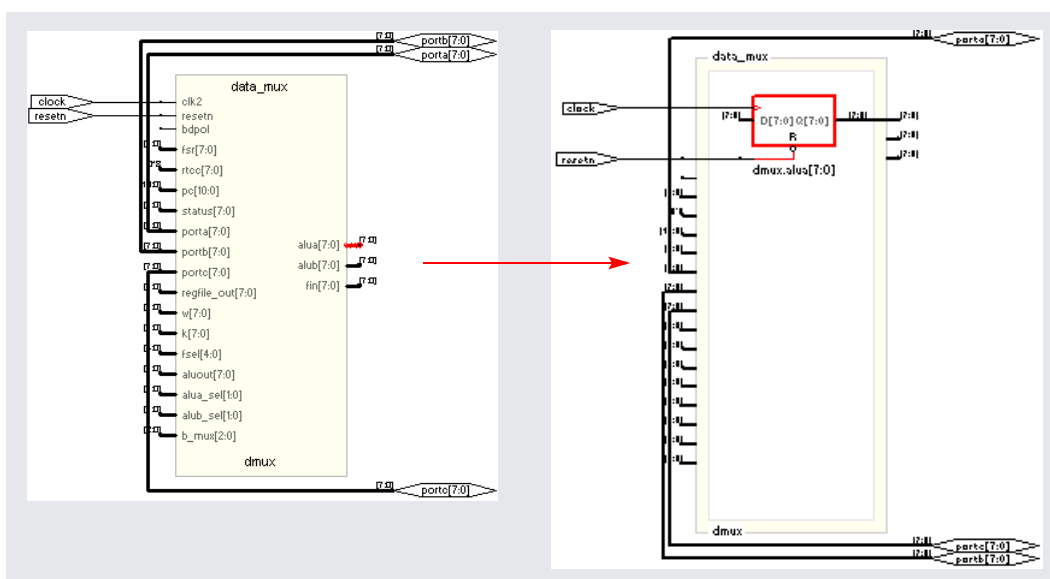
Expanding Filtered Logic Example



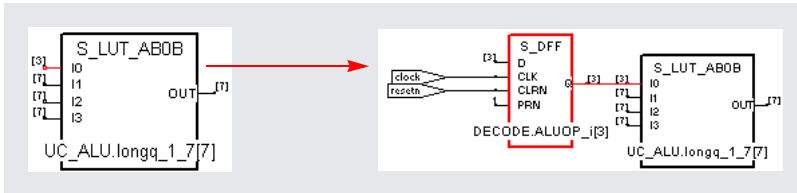
Expanding Filtered Logic to Register/Port Example



Expanding Inwards Example



Selecting the Net Driver Example



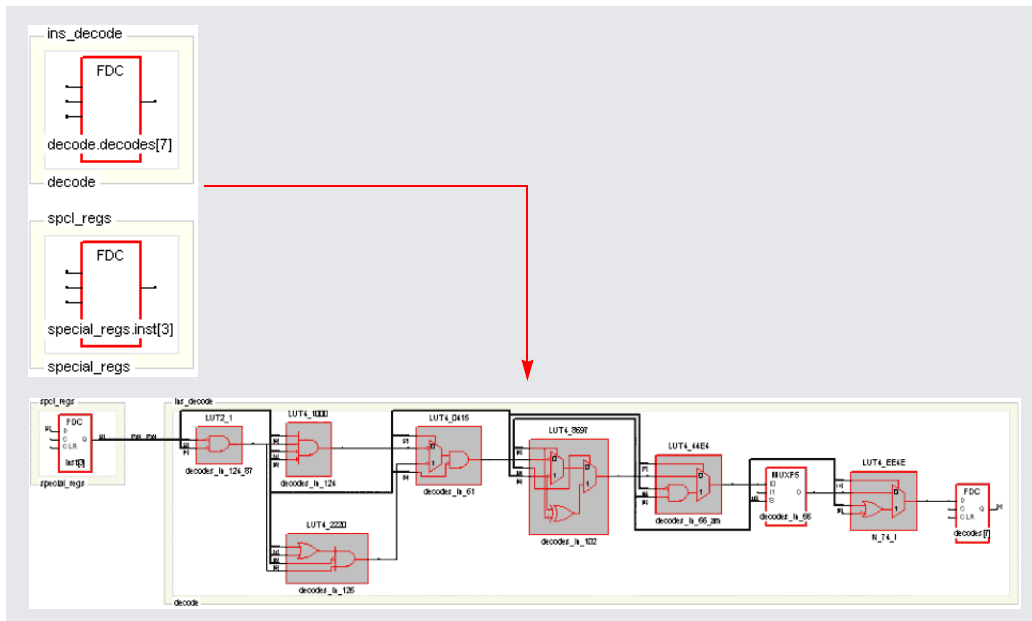
Expanding and Viewing Connections

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 305](#). 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 322](#).

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 327](#) 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.



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.

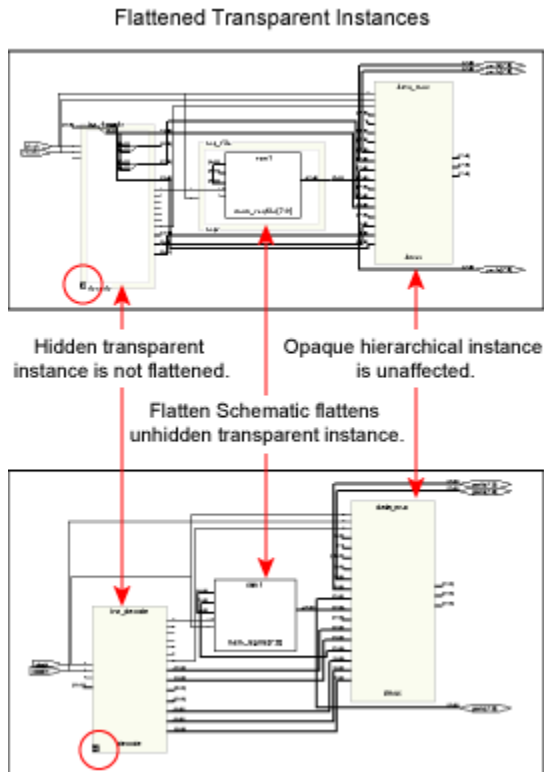
The software flattens the top-level design and displays it in a new window. To return to the top-level design, right-click and select Unflatten Schematic.

Unless you really require the entire design to be flattened, use Push/Pop mode and the filtering commands ([Filtering Schematics, on page 303](#)) 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.



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**.

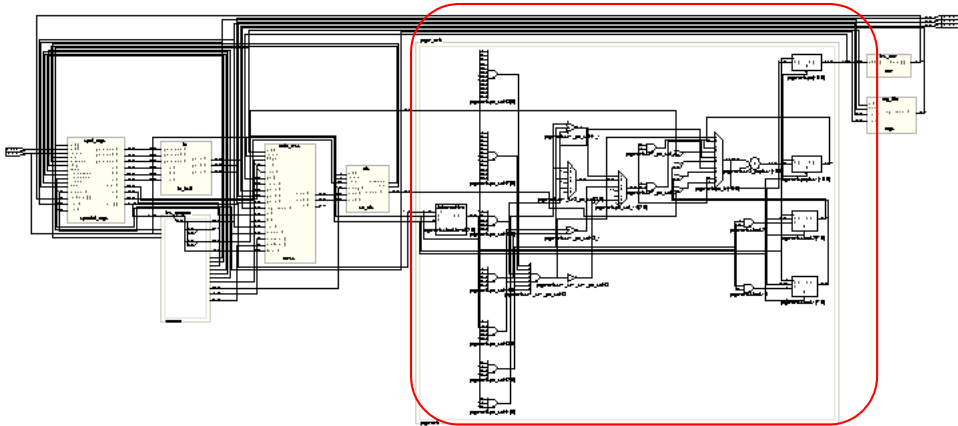
The results differ slightly, depending on the kind of view from which you dissolve instances.

Starting View	Software Generates a...
---------------	-------------------------

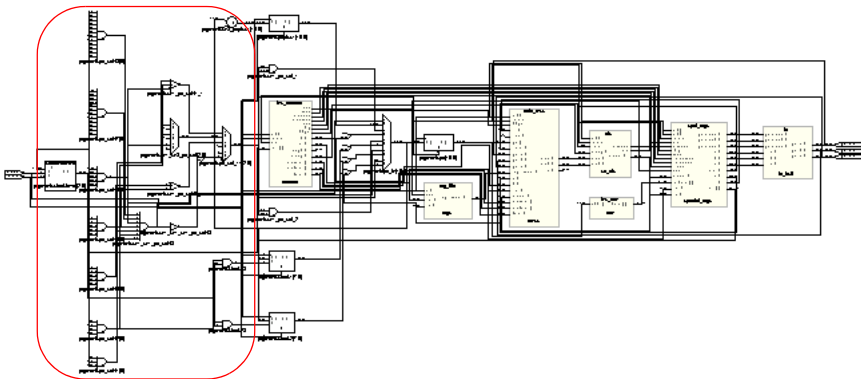
Filtered	Filtered view with the internal logic of dissolved instances displayed within hollow bounding boxes (transparent instances), and the hierarchy of the rest of the design unchanged. If the transparent instance does not display internal logic, use one of the alternatives described in step 4 of Viewing Design Hierarchy and Context, on page 300 . 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.

The following figure illustrates this.

Dissolved logic for prgmctr shown nested when started from filtered view



Dissolved logic for prgmctr 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 Push/Pop mode.

Minimizing Memory Usage While Analyzing Designs

When working with large hierarchical designs, use the following techniques to use memory resources efficiently.

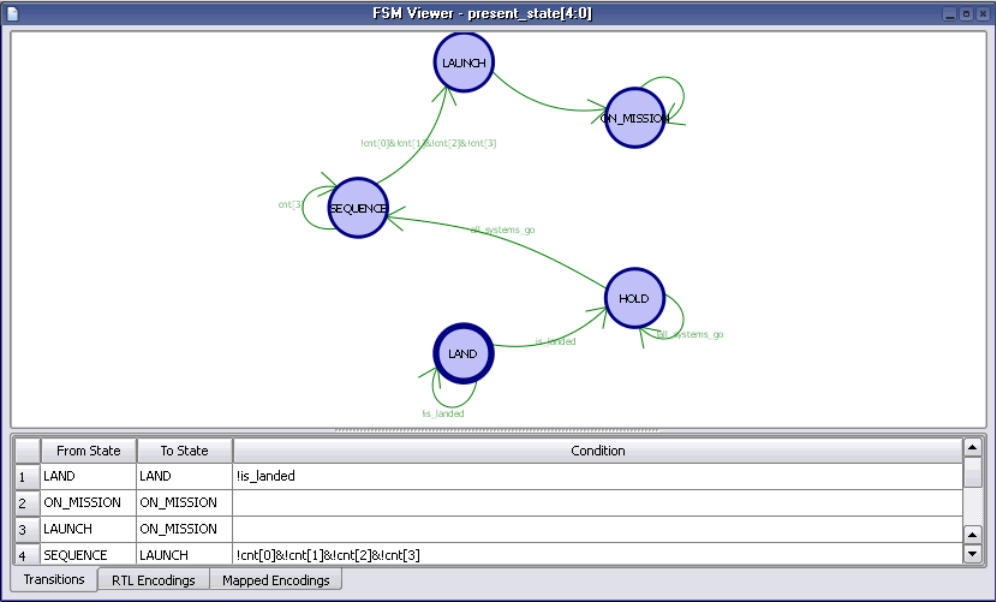
- Before you do any analysis operations such as searching, flattening, expanding, or pushing/popping, hide (HDL Analyst->Hide Instances) the hierarchical instances you do not need. This saves memory resources, because the software does not load the hierarchy of the hidden instances.
- Temporarily divide 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 311](#) for details. For more information on the Expand Paths and Isolate Paths commands, see [RTL View and Technology View Popup Menu Commands, on page 292](#) 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.

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 implemented by either the FSM Compiler or the FSM Explorer. For more information, see [Running the FSM Compiler, on page 220](#) and [Running the FSM Explorer, on page 224](#), respectively.

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).

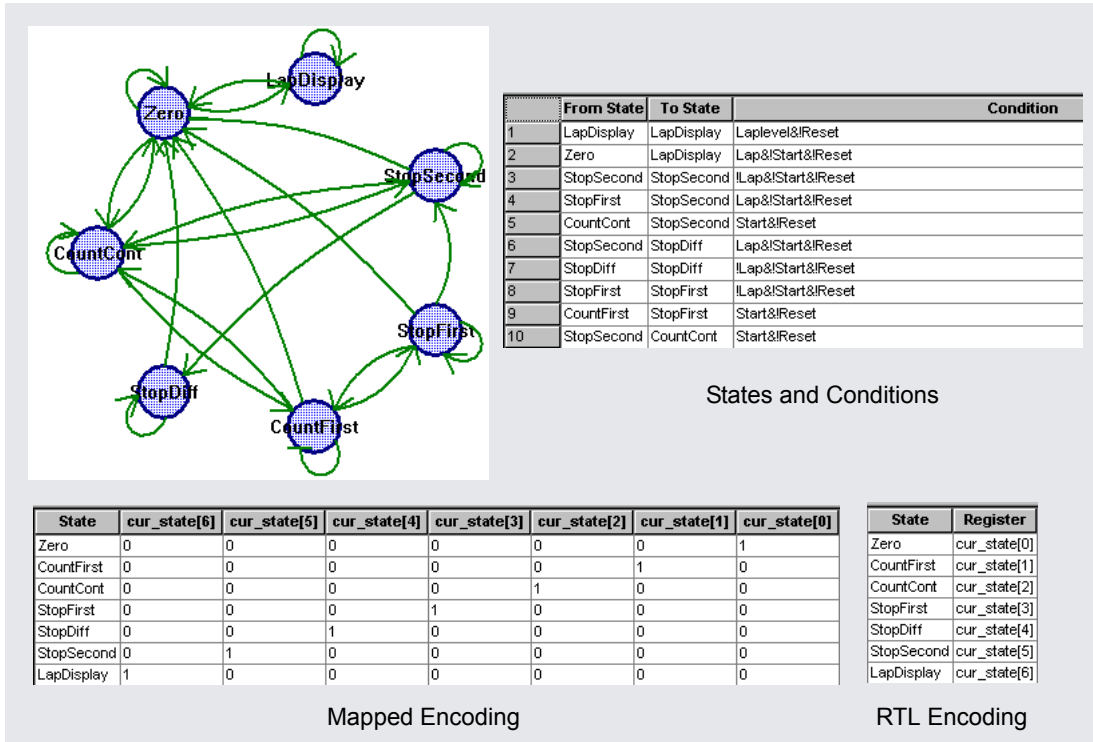
The FSM viewer opens. The viewer consists of a transition bubble diagram and a table for the encodings and transitions. If you used Verilog to define the FSMs, the viewer displays binary values for the state machines if you defined them with the `'define` keyword, and actual names if you used the `parameter` keyword.



2. The following table summarizes basic viewing operations.

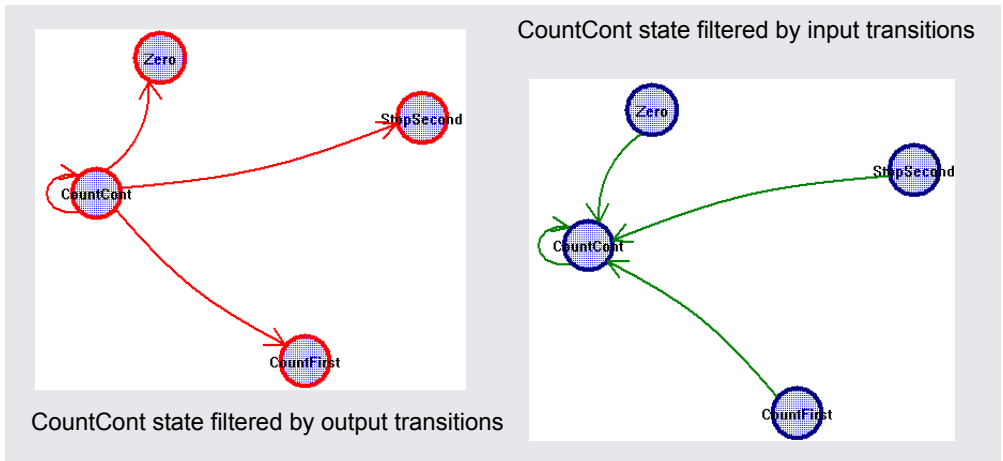
To view...	Do...
from and to states, and conditions for each transition	Click the Transitions tab at the bottom of the table.
the correspondence between the states and the FSM registers in the RTL view	Click the RTL Encoding tab.
the correspondence between the states and the registers in the Technology 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.



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
```

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 10

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 322
- [Generating Custom Timing Reports with STA](#), on page 329
- [Using Analysis Design Constraints](#), on page 332
- [Using Auto Constraints](#), on page 339

Analyzing Timing in Schematic Views

You can use the Timing Analyst and HDL Analyst functionality to analyze timing. This section describes the following:

- [Viewing Timing Information](#), on page 322
- [Annotating Timing Information in the Schematic Views](#), on page 323
- [Analyzing Clock Trees in the RTL View](#), on page 325
- [Viewing Critical Paths](#), on page 325
- [Handling Negative Slack](#), on page 328
- [Generating Custom Timing Reports with STA](#), on page 329

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.

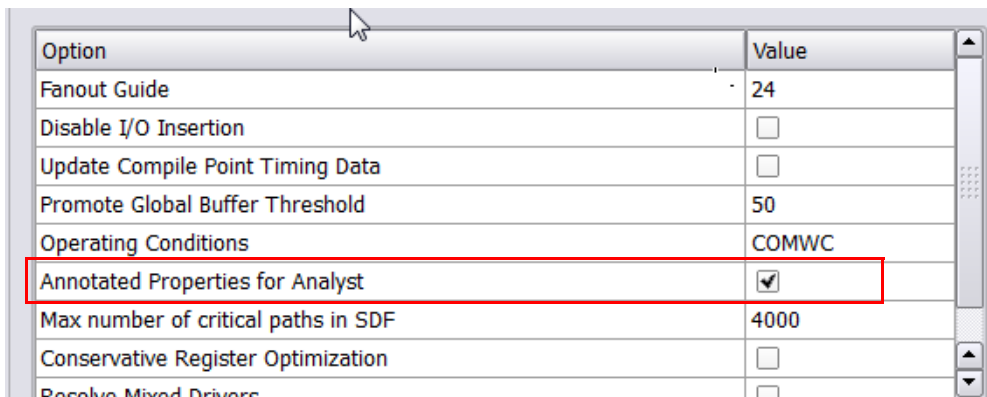
This displays the timing numbers for all instances in a Technology view. It shows the following:

Delay	This is the first number displayed. <ul style="list-style-type: none">• 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.



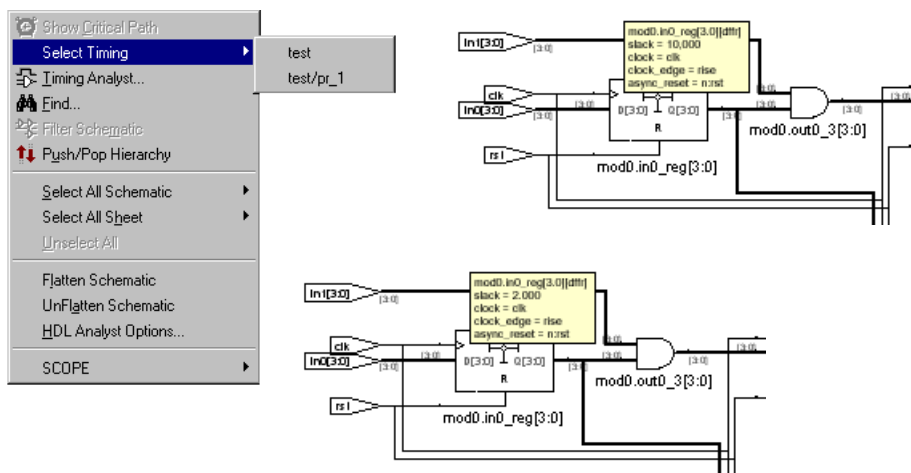
For each synthesis implementation and each place-and-route implementation, the tool generates properties and stores them in two files located in the project folder:

- | | |
|-------|---|
| .sap | Synplify Annotated Properties
Contains the annotated design properties generated after compilation, like clock pins. |
| <hr/> | |
| .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 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.



- 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 [Tcl Find -filter Command, on page 1138](#) in the *Reference Manual* for a list of properties. For information about the find command, see [Tcl find Command, on page 1128](#) in the *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 303](#), [Expanding Pin and Net Logic, on page 305](#) and [Expanding and Viewing Connections, on page 309](#).

3. Check that your defined clock constraints cover the objects in the design.

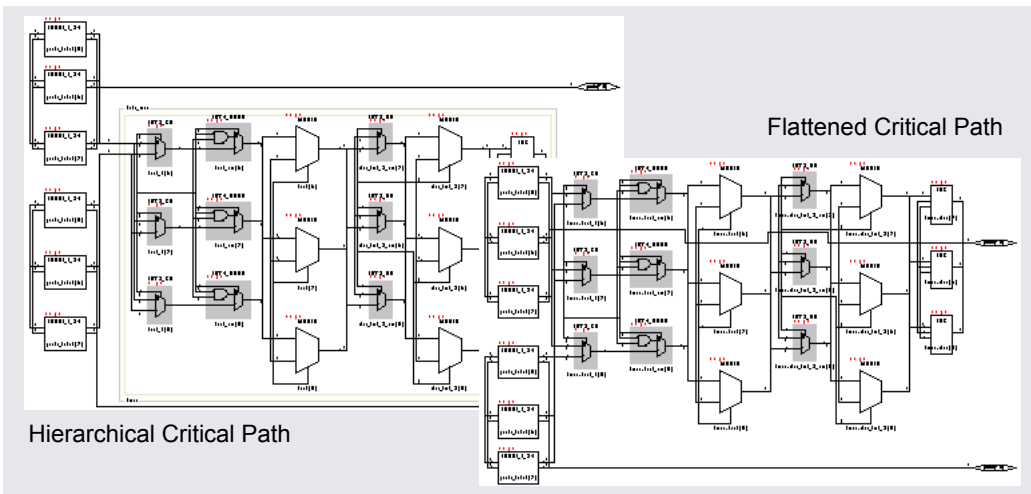
If you do not define your clock constraints accurately, you might not get the best possible synthesis optimizations.

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 329](#) 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.

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.



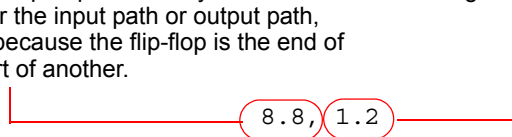
3. Use the timing numbers displayed above each instance to analyze the path. If no numbers are displayed, enable HDL Analyst->Show Timing Information. Interpret the numbers as follows:

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.



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 328](#).

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.

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 305](#) and [Expanding and Viewing Connections, on page 309](#). For more information on filtering schematics, see [Filtering Schematics, on page 303](#).
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 retiming ([Retiming, on page 196](#)), FSM exploration ([Running the FSM Explorer, on page 224](#)), or resource sharing ([Sharing Resources, on page 213](#)).
 - Modifying the source code.
5. Rerun synthesis and check your results.

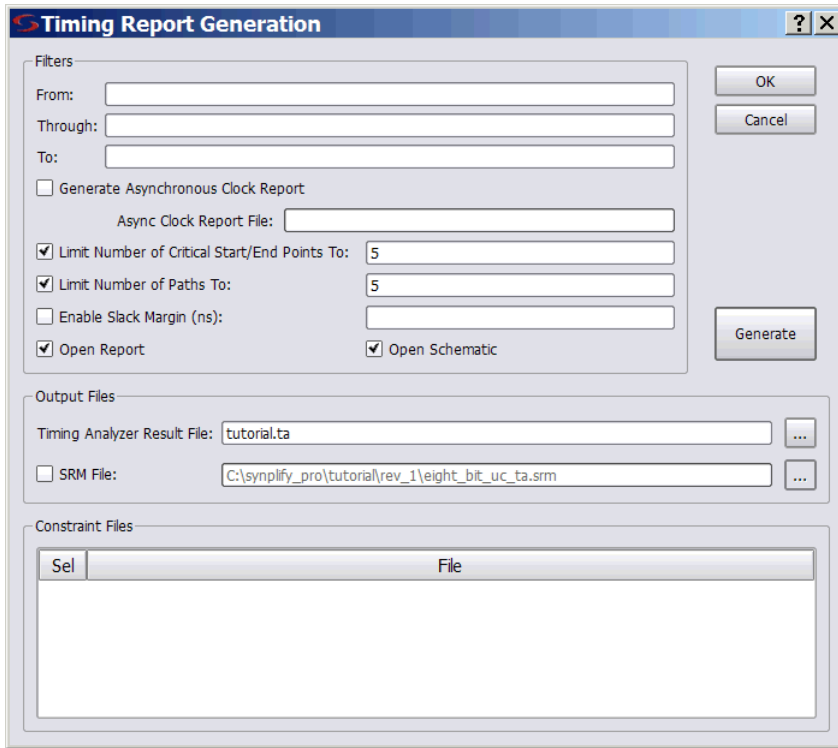
Generating Custom Timing Reports with STA

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 332](#) 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 219](#) 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).



The dialog box is titled "Timing Report Generation" and contains several sections for configuring the report. The "Filters" section includes input fields for "From:", "Through:", and "To:", along with checkboxes for "Generate Asynchronous Clock Report", "Limit Number of Critical Start/End Points To:" (set to 5), "Limit Number of Paths To:" (set to 5), "Enable Slack Margin (ns):", "Open Report", and "Open Schematic". The "Output Files" section has fields for "Timing Analyzer Result File:" (set to "tutorial.ta") and "SRM File:" (set to "C:\synplify_pro\tutorial\rev_1\eight_bit_uc_ta.srm"). The "Constraint Files" section features a table with columns "Sel" and "File".

Sel	File
-----	------

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:           smult2o[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    Delay    Arrival    No. of
Name                    Name    Dir    Time     Time      Fan Out(s)
-----
smult2o[11]              regout Out     0.094    0.094      -
smult2o[11]              Net    -     0.311    -          2
smult1_carry_9           dataf  In     -        0.405      -

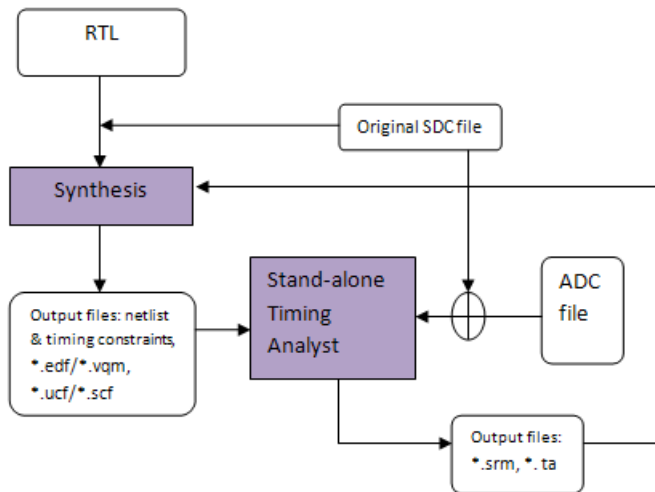
```

- 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.

Using Analysis Design Constraints

Besides generating custom timing reports (see [Generating Custom Timing Reports with STA, on page 329](#)), 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 333
- [Creating an ADC File](#), on page 334
- [Using Object Names Correctly in the adc File](#), on page 338

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 334](#).

- **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
```

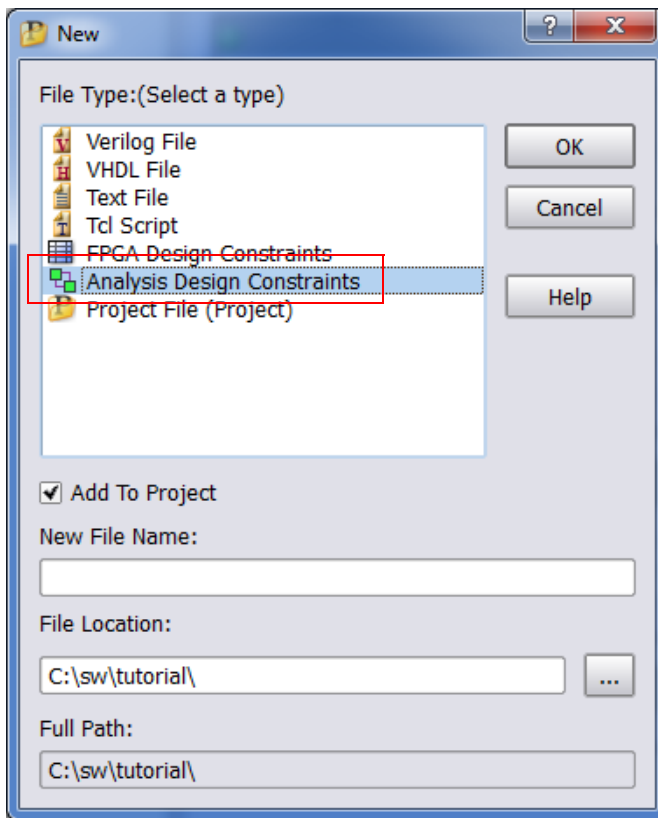
- **Adding additional timing exceptions**
When you analyze the results of the first synthesis run, you often find functional or clock-to-clock timing exceptions, and you can handle these with adc constraints. For example:
 - Applying false paths on synchronization circuitry
 - Adding false paths between 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 332](#), step 3.

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.

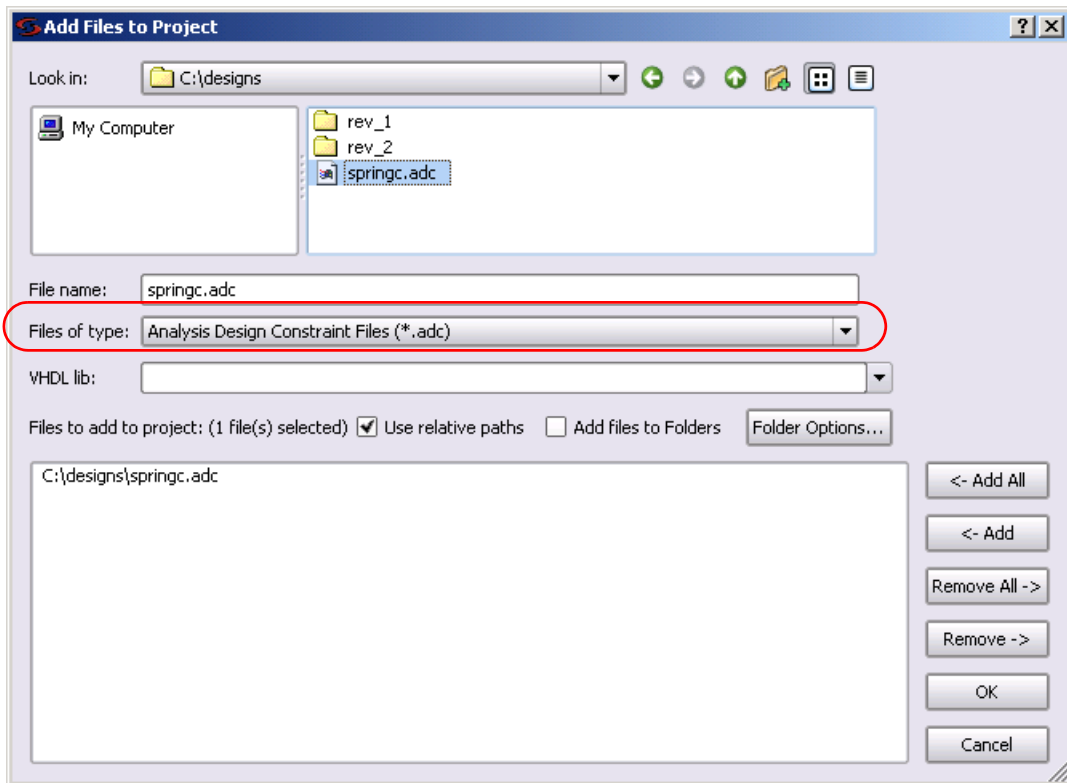


- Type a name and location for the file. The tool automatically assigns the adc extension to the filename.
 - Enable Add to Project, and click OK. This opens the text editor where you can specify the new constraints.
3. Type in the constraints you want and save the file. Remember the following when you 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 393](#) 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 338](#).
- 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

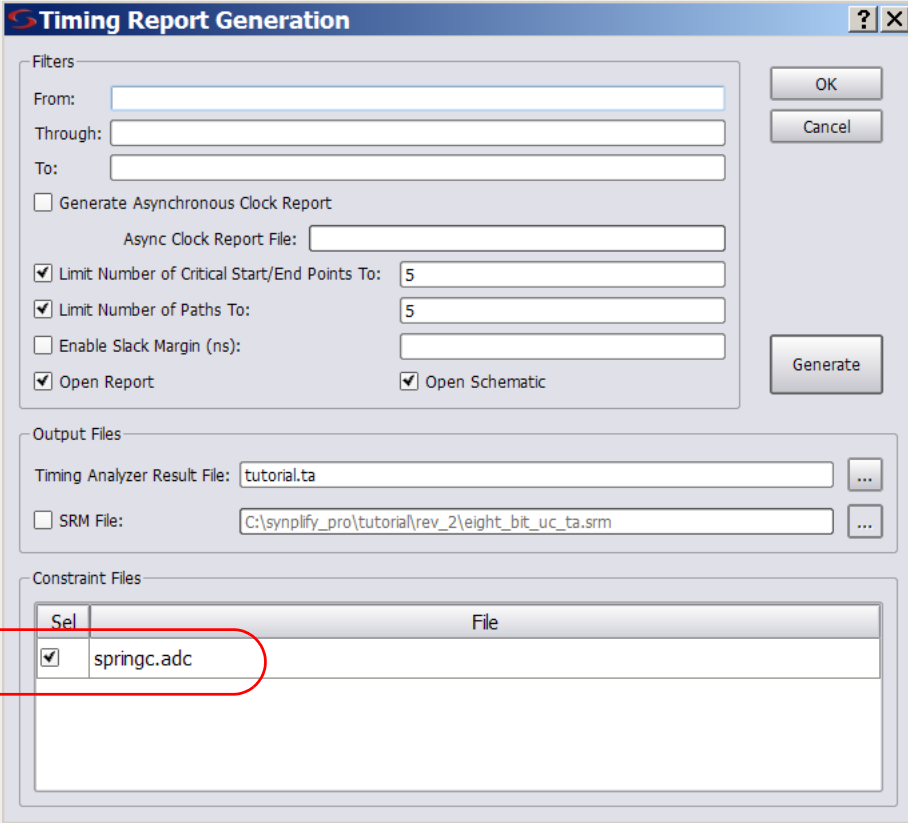
- When you are done, save and close the file. This adds the file to your project.



- 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.



The image shows a 'Timing Report Generation' dialog box. It has several sections: 'Filters', 'Output Files', and 'Constraint Files'. In the 'Filters' section, there are input fields for 'From:', 'Through:', and 'To:'. Below these are checkboxes for 'Generate Asynchronous Clock Report', 'Limit Number of Critical Start/End Points To:' (set to 5), 'Limit Number of Paths To:' (set to 5), 'Enable Slack Margin (ns):', 'Open Report', and 'Open Schematic'. In the 'Output Files' section, there is a 'Timing Analyzer Result File:' field set to 'tutorial.ta' and an 'SRM File:' field set to 'C:\synplify_pro\tutorial\rev_2\eight_bit_uc_ta.srm'. In the 'Constraint Files' section, there is a table with two columns: 'Sel' and 'File'. The first row has a checked checkbox in the 'Sel' column and 'springc.adc' in the 'File' column. This row is highlighted with a red oval. There are 'OK', 'Cancel', and 'Generate' buttons on the right side of the dialog.

Sel	File
<input checked="" type="checkbox"/>	springc.adc

- 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.

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 334](#) 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]}}
```

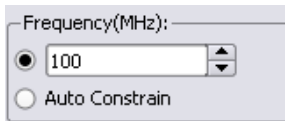
or

```
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]}}
```

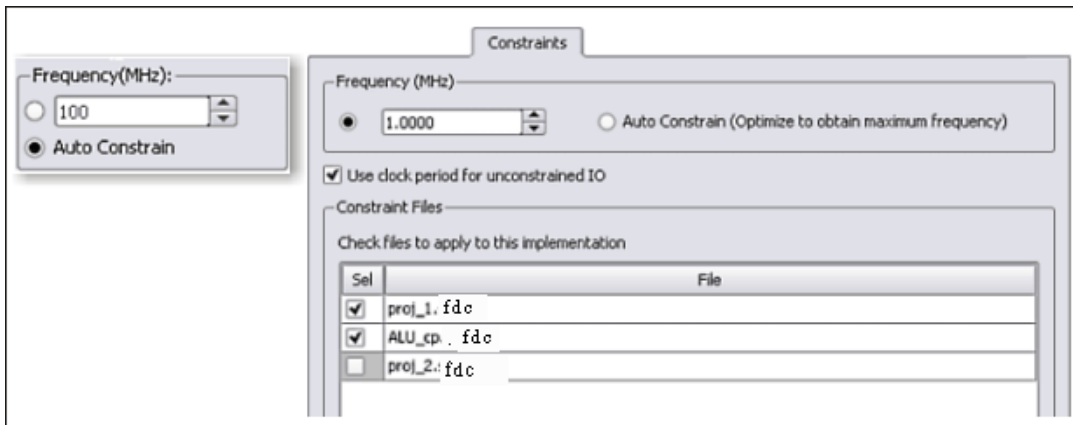
Using Auto Constraints

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 341](#).

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.



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 341
- [I/O Constraints and Timing Exceptions](#), on page 342
- [Reports and Forward-annotation](#), on page 342
- [Repeatability of Results](#), on page 343

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 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 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.fdc`, 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.

CHAPTER 11

Optimizing for Microsemi Designs

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 6, *Inferring High-Level Objects*](#).

Refer to the design tips for the following Microsemi-specific procedures:

- [Using Predefined Microsemi Black Boxes](#), on page 346
- [Using Smartgen Macros](#), on page 347
- [Working with Radhard Designs](#), on page 347
- [Specifying `syn_radhardlevel` in the Source Code](#), on page 348

For additional Microsemi-specific information, see [Passing Information to the P&R Tools](#), on page 352 and [Generating Vendor-Specific Output](#), on page 354.

Optimizing Microsemi Designs

The Synplify Pro synthesis tool supports Microsemi designs. The following procedures Microsemi-specific design tips.

- [Using Predefined Microsemi Black Boxes](#), on page 346
- [Using Smartgen Macros](#), on page 347
- [Working with Radhard Designs](#), on page 347
- [Specifying syn_radhardlevel in the Source Code](#), on page 348

For additional Microsemi-specific information, see [Passing Information to the P&R Tools](#), on page 352 and [Generating Vendor-Specific Output](#), on page 354.

Using Predefined Microsemi Black Boxes

The Microsemi macro libraries contain predefined black boxes for Microsemi macros so that you can manually instantiate them in your design. For information about using ACTGen macros, see [Using Smartgen Macros](#), on page 347. For general information about working with black boxes, see [Defining Black Boxes for Synthesis](#), on page 166.

To instantiate an Microsemi macro, use the following procedure.

1. Locate the Microsemi macro library file appropriate to your technology and language (v or vhd) in one of these subdirectories under *installDirectory/lib*.

proasic	ProASIC3/3E/3L, Fusion/SmartFusion/SmartFusion2 and IGLOO/IGLOO+/IGLOOe/IGLOO2 macros
microsemi	Macros for all other Microsemi technologies.

Use the macro file that corresponds to your target architecture.

2. Add the Microsemi macro library *at the top* of the source file list for your synthesis project. Make sure that the library file is first in the list.

3. For VHDL, also add the appropriate library and use clauses to the top of the files that instantiate the macros:

```
library family;  
use family.components.all ;
```

Specify the appropriate technology in *family*.

Using Smartgen Macros

The Smartgen macros replace the ACTgen macros, which were available in the previous Designer 6.x place-and-route tool. The following procedure shows you how to include Smartgen macros in your design. For information about using Microsemi macro libraries, see [Using Predefined Microsemi Black Boxes, on page 346](#). For general information about working with black boxes, see [Defining Black Boxes for Synthesis, on page 166](#).

1. In Smartgen, generate the function you want to include.
2. For Verilog macros, do the following:
 - Include the appropriate Microsemi macro library file for your target architecture in your the source files list for your project.
 - Include the Verilog version of the Smartgen result in your source file list. Make sure that the Microsemi macro library is first in the source files list, followed by the Smartgen Verilog files, followed by the other source files.
3. Synthesize your design as usual.

Working with Radhard Designs

The following procedure outlines how to specify radhard values for a design with the `syn_radhardlevel` attribute. Remember that the attribute is not recursive. It only applies to all registers at the level where it is set and does not affect lower-level registers.

You specify radhard values in modules and architecture in both the Attributes panel in SCOPE and in the source code. However, for registers, it must be specified in the source code only.

To set a global or default `syn_radhardlevel` attribute, set the value in the source file for the module. The following sets all registers of `module_b` to `tmr`:

VHDL

```
library synplify;
use synplify.attributes.all;
attribute syn_radhardlevel of
  behav: architecture is "tmr";
```

Verilog

```
module module_b (a, b, sub,
  clk, rst) /*synthesis
  syn_radhardlevel="tmr"*/;
```

Specifying `syn_radhardlevel` in the Source Code

For a module, you can attach the `syn_radhardlevel` attribute either in the Attributes panel of the SCOPE window or in the source code. For a register, you can only apply this attribute in the source code.

To set attributes in SCOPE, see [How Attributes and Directives are Specified, on page 904](#). The following procedure outlines how to set this attribute in the source code.

1. To set a `syn_radhardlevel` value for all the registers of a module, do the following:
 - Set the value in the source file. The following sets all registers of `module_b` to `tmr`:

VHDL

```
library synplify;
use synplify.attributes.all;
attribute syn_radhardlevel of
  behav: architecture is "tmr";
```

Verilog

```
module module_b (a, b, sub,
  clk, rst) /*synthesis
  syn_radhardlevel="tmr"*/;
```

The attribute is not recursive. When used at the module or architecture level, it only applies to the registers at that level, and does not affect lower-level registers.

2. To set a `syn_radhardlevel` value on a per-register basis, set the value on the register in the source file for the module. For example, to set the value of register `bl_int` to `tmr`, enter the following in the module source file:

VHDL

```
library synplify;
use synplify.attributes.all;
attribute syn_radhardlevel of
    bl_int: signal is "tmr"
```

Verilog

```
reg [15:0] a1_int, b1_int
/* synthesis syn_radhardlevel
= "tmr" */;
```

Use a register-level attribute to override a default value with another value, or set it to `none` to ensure that a global default value is not applied to the register.

3. To prevent a default from being applied to a register or module/entity, set `syn_radhardlevel` to `none` for that register, module, or entity.

CHAPTER 12

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 6, *Inferring High-Level Objects*](#).

This chapter describes the following:

- [Passing Information to the P&R Tools](#), on page 352
- [Generating Vendor-Specific Output](#), on page 354

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 352
- [Specifying Locations for Microsemi Bus Ports](#), on page 353
- [Specifying Macro and Register Placement](#), on page 353

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. For information about other placement properties, see [Specifying Macro and Register Placement, on page 353](#).

1. Start with a design using one of the following vendors and technologies: Microsemi families.
2. Add the appropriate attribute to the port. For a bus, list all the bus pins, separated by commas. To specify Microsemi bus port locations, see [Specifying Locations for Microsemi Bus Ports, on page 353](#).
 - 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 *Reference Manual* for syntax details.

Family	Attribute and Value
Microsemi	<code>syn_loc {pin_number}</code> or <code>alspin {pin_number}</code>

Specifying Locations for Microsemi Bus Ports

You can specify pin locations for Microsemi bus ports. To assign pin numbers to a bus port, or to a single- or multiple-bit slice of a bus port, do the following:

1. Open the constraint file and add these attributes to the design.
2. Specify the `syn_noarrayports` attribute globally to bit blast all bus ports in the design.

```
define_global_attribute syn_noarrayports {1};
```

3. Use the `alspin` attribute to specify pin locations for individual bus bits. This example shows locations specified for individual bits of bus `ADDRESS0`.

```
define_attribute {ADDRESS0[4]} alspin {26}
define_attribute {ADDRESS0[3]} alspin {30}
define_attribute {ADDRESS0[2]} alspin {33}
define_attribute {ADDRESS0[1]} alspin {38}
define_attribute {ADDRESS0[0]} alspin {40}
```

The software forward-annotates these pin locations to the place-and-route software.

Specifying Macro and Register Placement

You can use attributes to specify macro and register placement in Microsemi designs. The information here supplements the pin placement information described in [Specifying Pin Locations, on page 352](#) and bus pin placement information described in [Specifying Locations for Microsemi Bus Ports, on page 353](#).

For...

Use...

Relative placement of Microsemi macros and IP blocks

alsloc
`define_attribute {u1} alsloc {R15C6}`

Generating Vendor-Specific Output

The following topics describe generating vendor-specific output in the synthesis tools.

- [Targeting Output to Your Vendor](#), on page 354
- [Customizing Netlist Formats](#), on page 355

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
Microsemi	EDIF (.edn or .edf) *_sdc.sdc	Libero SoC Libero IDE

3. To generate mapped Verilog/VHDL netlists and constraint files, check the appropriate boxes and click OK.

See [Specifying Result Options, on page 135](#) 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, on page 50](#).

Customizing Netlist Formats

The following table lists some attributes for customizing your Microsemi output netlists:

For...	Use...
Netlist formatting	<i>syn_netlist_hierarchy</i> (Microsemi) define_global_attribute syn_netlist_hierarchy {0}
Bus specification	<i>syn_noarrayports</i> (Microsemi) define_global_attribute syn_noarrayports {1}

CHAPTER 13

Running Post-Synthesis Operations

The following topics describe how to run post-synthesis operations:

- [Running P&R Automatically after Synthesis](#), on page 358
- [Working with the Identify Tools](#), on page 359
- [Simulating with the VCS Tool](#), on page 366

Running P&R Automatically after Synthesis

You can run place-and-route from within the tool or in batch mode.

Note: To run place and route successfully, first set the environment variable PATH for the place-and-route tool.

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 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.

Working with the Identify Tools

The Synopsys Identify tool set is a dual-component system that is a valuable part of the HDL design flow process. The system consists of the Identify instrumentor and Identify debugger.

- The Identify instrumentor allows you to select your design instrumentation at the HDL level and then create an on-chip hardware probe.
- The Identify debugger interacts with the on-chip hardware probe and lets you do live debugging of the design.

The combination of these tools allows you to probe your HDL design in the target environment. The combined system allows you to debug your design faster, easier, and more efficiently.

The Synplify Pro tool has integrated the Identify instrumentor into the synthesis user interface. This section describes how to take advantage of this integration and use the Identify instrumentor:

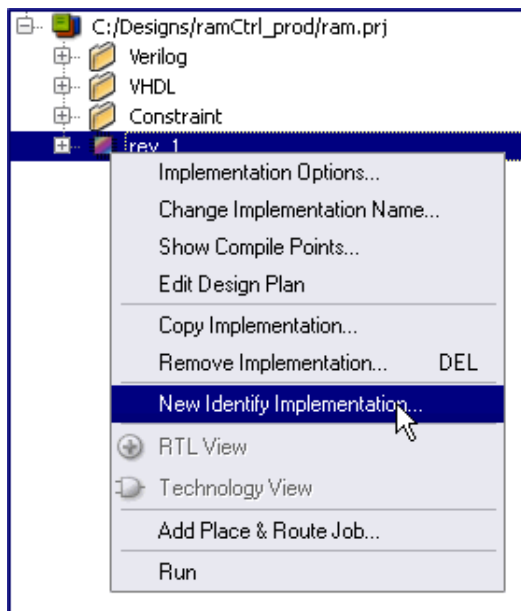
- [Launching from the Synplify Pro Tool](#), on page 359
- [Handling Problems with Launching Identify](#), on page 361
- [Using the Identify Tool](#), on page 362
- [Using Compile Points with the Identify Tool](#), on page 364

Launching from the Synplify Pro Tool

Define a Synplify Pro project that you can pass to and launch in the Identify instrumentor. For the Synplify Pro tool, you must create an Identify implementation in order to run the Identify instrumentor. If you already have an Identify implementation, open it and use the Identify tool as described in [Using the Identify Tool, on page 362](#).

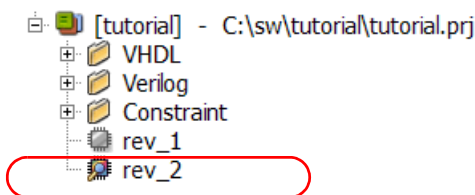
Do the following to add an Identify implementation:


1. In the synthesis interface, open the design you want to debug.
2. Do one of the following tasks to add an Identify implementation:
 - With the project implementation selected, right-click and select New Identify Implementation from the pop-up menu.



- Select Project->New Identify Implementation.

An Implementation Options dialog box appears where you can set the options for your implementation. An Identify implementation is created.




3. To run Identify instrumentor, select the Launch Identify Instrumentor icon () in the toolbar or select Run->Identify Instrumentor.

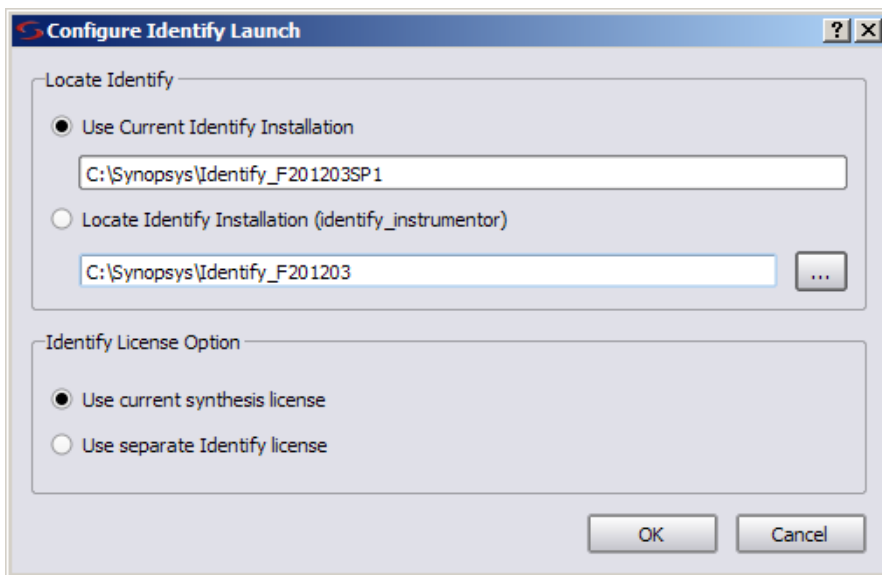
The Identify interface opens. You can now use the Identify tool as described in [Using the Identify Tool, on page 362](#) For complete details, consult the Identify documentation.

If you run into problems while launching the Identify instrumentor, refer to [Handling Problems with Launching Identify, on page 361](#).

Handling Problems with Launching Identify

If you have not installed Identify correctly, you might run into problems when you try to launch it from the synthesis tools. The following describe some situations:

- If the Launch Identify Instrumentor icon () and the Run->Identify Instrumentor menu command are inaccessible, you are either on an unsupported platform or you are using a technology that does not support this feature.
- If you have the Identify software installed but the synthesis application cannot find it, select Options->Configure Identify Launch.



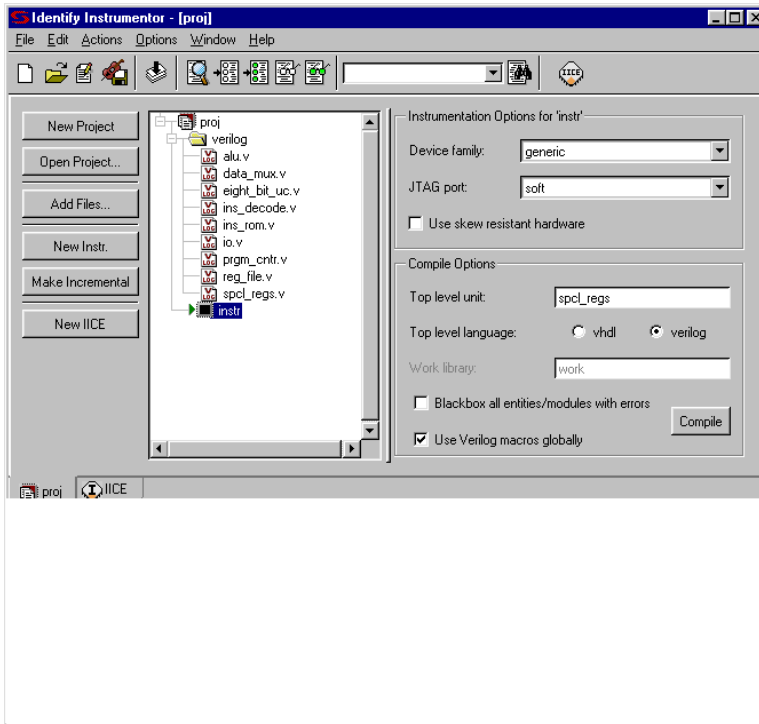
In the resulting dialog box, either:

- check the Use Current Identify Installation entry. This entry is set by the SYN_IDENTIFY_EXE environment variable to point to the Identify installation. If this path is incorrect, change the environment variable setting and restart the synthesis tool. button and specify the correct location in the Locate Identify Installation field. You can use the Browse button to open the Select Identify Installation Directory dialog box and navigate to your current Identify installation directory.
- click the Locate Identify Installation button and specify the correct location in the corresponding field. Use the browse button to open the Select Identify Installation Directory dialog box and navigate to your current Identify installation directory.

Using the Identify Tool

This procedure provides an overview of how to use the Identify instrumentor. For detailed information about the tool, refer to the Identify RTL debugger documentation.

1. The Identify instrumentor software interface opens, with an Identify project automatically set up for the design to be instrumented and debugged (IICE tab). The following figure shows the main project window.



2. Do the following in the Identify instrumentor interface:

- Instrument the design. For details of using the Identify instrumentor, refer to the Identify RTL debugger documentation.
- Save the instrumented design.

The Identify instrumentor tool exports the instrumented design to the synthesis software. It creates an instrumentation subdirectory under your synthesis working directory called *designName_instr*, which contains the following:

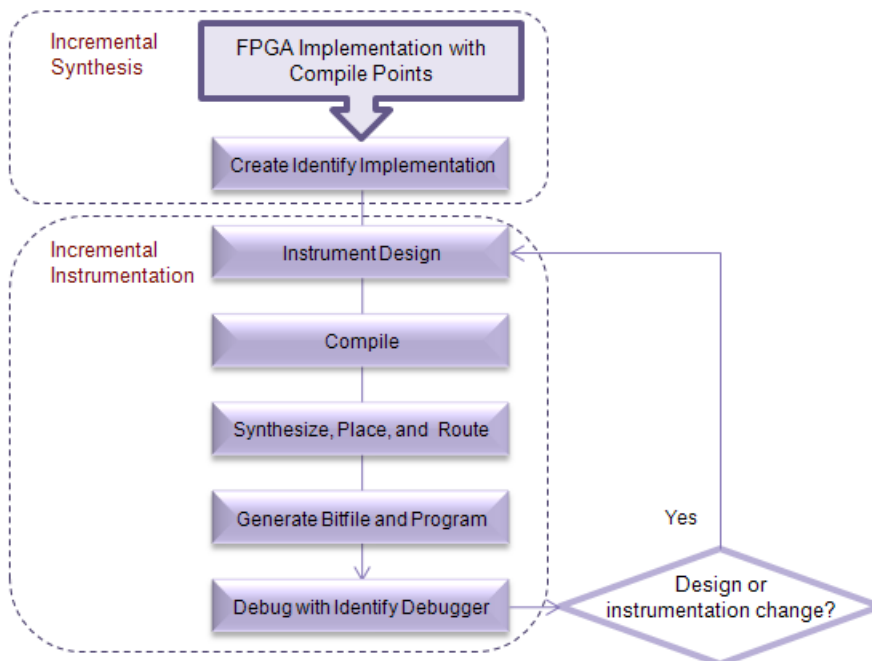
- A synthesis project file
- An *instr_sources* subdirectory for the instrumented HDL files
- Tcl scripts for loading the instrumented design

3. Return to the synthesis interface and view the instrumented design that contains the debugging logic.

- In the synthesis interface, open the project file for the instrumented design, which is in the `instr_sources` subdirectory listed in the Implementations Results view for your original synthesis project.
 - Synthesize the design.
 - Open the RTL view to see the inserted debugging logic.
4. Place and route the instrumented design after synthesis.
 5. Use the Identify debugger tool to debug the instrumented design.

Using Compile Points with the Identify Tool

You can use compile points to run incrementally. This can reduce runtime while running synthesis, and also while running the Identify flow. The following figure illustrates this:



When you use Identify instrumentation, the tool creates extra IICE logic at the top level of the design and the corresponding interface to the signals that need to be debugged. If you define compile points, the tool need only rerun the compile points that have changed because of the insertion of this logic. On subsequent runs, it can incrementally re-instrument only those compile points where there are instrumentation changes or design modifications. The following procedure describes the steps to follow to implement the flow and take advantage of incremental synthesis and instrumentation:

1. Create a synthesis implementation with compile points.
2. Set up the Identify implementation:
 - Generate the Identify implementation by right-clicking the FPGA synthesis implementation and selecting New Identify Implementation from the popup menu.
 - Copy the compile point subdirectories manually to the new Identify implementation directory.
3. Run the tools.
 - Run synthesis.
 - Before running the Identify tool, enable the top-level constraint file and all compile point constraint files in the Identify implementation.
 - Instrument the design. The tool inserts additional logic for instrumentation.

4. Resynthesize the design.


The tool runs incrementally, only resynthesizing the compile points affected by the inserted instrumentation logic. If you make any other design changes, the tool incrementally synthesizes the affected compile points.

5. Rerun instrumentation.

The tool runs incrementally, and only re-instruments the affected compile points.

Simulating with the VCS Tool

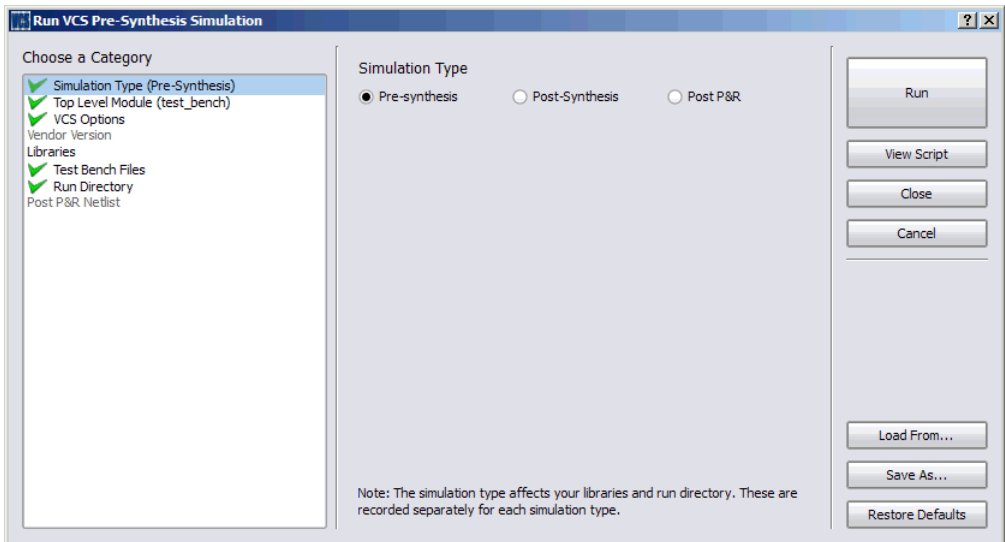
The Synopsys VCS® tool is a high-performance, high-capacity Verilog simulator that incorporates advanced, high-level abstraction verification technologies into a single, open, native platform. You can launch this simulation tool from the synthesis tools on Linux and Unix platforms by following the steps 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 208](#) 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

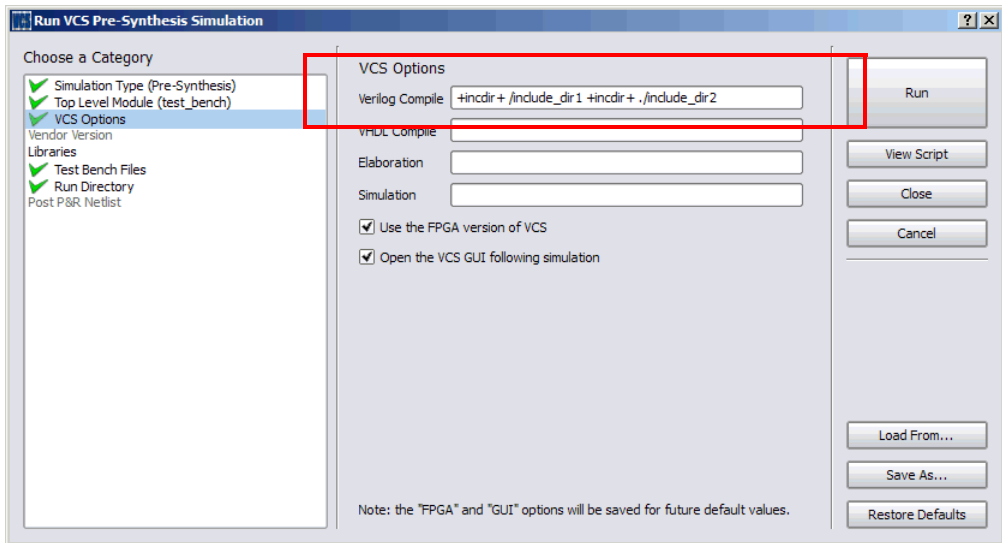


- Choose the category VCS Options in the dialog box to set options such as the following VCS commands.

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:



Example Verilog File:

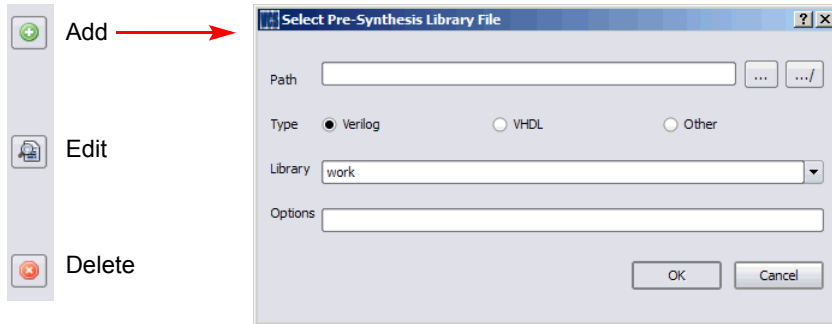
```
`include "component.v"
module Top (input a, output x);
...
endmodule
```

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.
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  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 `vlog` commands.

CHAPTER 14

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 372
- [The Synopsys FPGA IP Encryption Flow](#), on page 410
- [Working with Encrypted IP](#), on page 416
- [Using Hyper Source](#), on page 426

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.

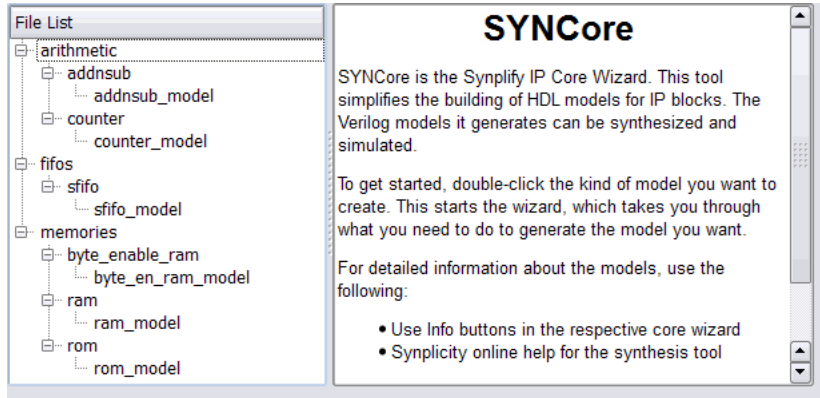
- [Specifying FIFOs with SYNCore](#), on page 372
- [Specifying RAMs with SYNCore](#), on page 378
- [Specifying Byte-Enable RAMs with SYNCore](#), on page 386
- [Specifying ROMs with SYNCore](#), on page 392
- [Specifying Adder/Subtractors with SYNCore](#), on page 397
- [Specifying Counters with SYNCore](#), on page 404

Specifying FIFOs with SYNCore

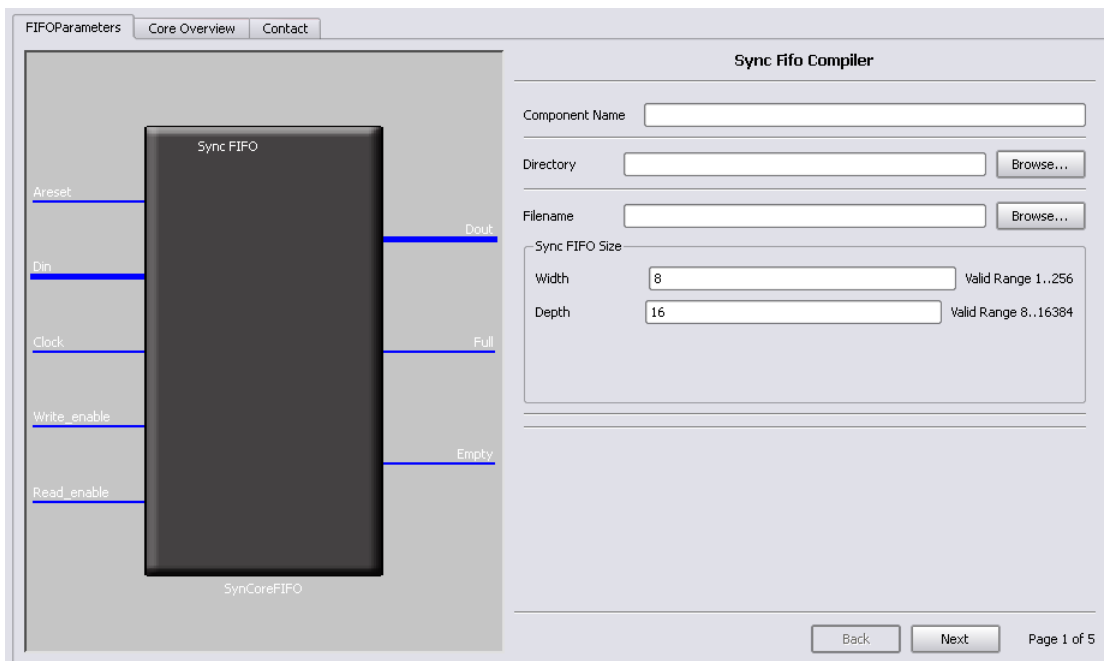
The SYNCore IP Wizard helps you generate Verilog code for your FIFO implementations. The following procedure shows you how to generate Verilog code for a FIFO using the SYNCore IP wizard.

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.



- Specify the parameters you need in the five pages of the wizard. For details, refer to [Specifying SYNCore FIFO Parameters, on page 376](#).

The FIFO symbol on the left reflects the parameters you set.

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

- 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 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 181](#) in the *Reference Manual*. For timing diagrams, see [Synplicity Archive Utility, on page 766](#) in the *Reference Manual*.

1. Start the SYNCORE wizard, as described in [Specifying FIFOs with SYNCORE, on page 372](#).
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 372](#).
 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 778](#) 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 781](#) in the *Reference Manual* for details.

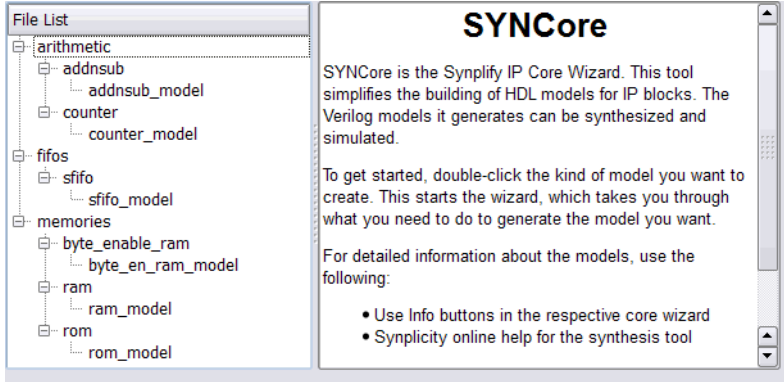
You can now generate the FIFO and add it to the design, as described in [Specifying FIFOs with SYNCORE, on page 372](#).

Specifying RAMs with SYNCORE

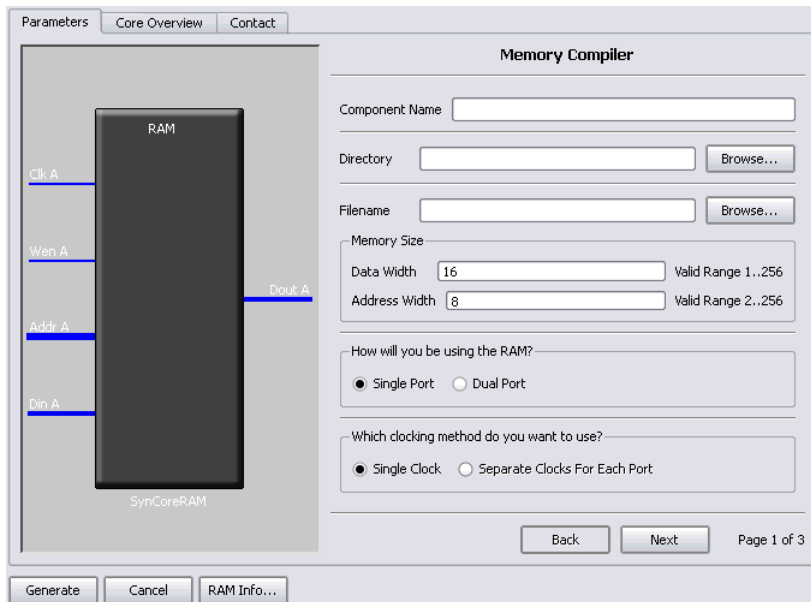
The SYNCORE IP wizard helps you generate Verilog code for your RAM implementation requirements. The following procedure shows you how to generate Verilog code for a RAM using the SYNCORE IP wizard.

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 382](#).

- For details about the parameters for a dual-port RAM, see [Specifying Parameters for Dual-Port RAM, on page 383](#). 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.

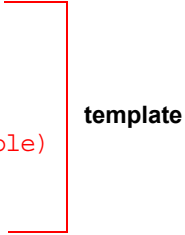
4. Edit the RAM files if necessary.

- 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.

```
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
```



- 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 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 190](#) in the *Reference Manual*.

1. Start the SYNCORE RAM wizard, as described in [Specifying RAMs with SYNCORE, on page 378](#).
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 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 378](#). 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 383](#). For read/write timing diagrams, see [Read/Write Timing Sequences, on page 791](#) 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 382](#). 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 791](#) of the *Reference Manual*.

1. Start the SYNCORE RAM wizard, as described in [Generating IP with SYNCORE, on page 372](#).
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.
- 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.

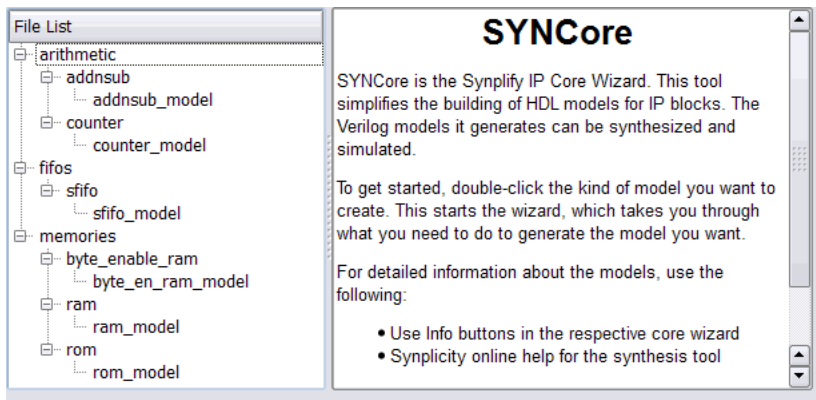
You can now generate the RAM by clicking Generate, as described in [Generating IP with SYNCORE, on page 372](#), and add it to your design.

Specifying Byte-Enable RAMs with SYNCore

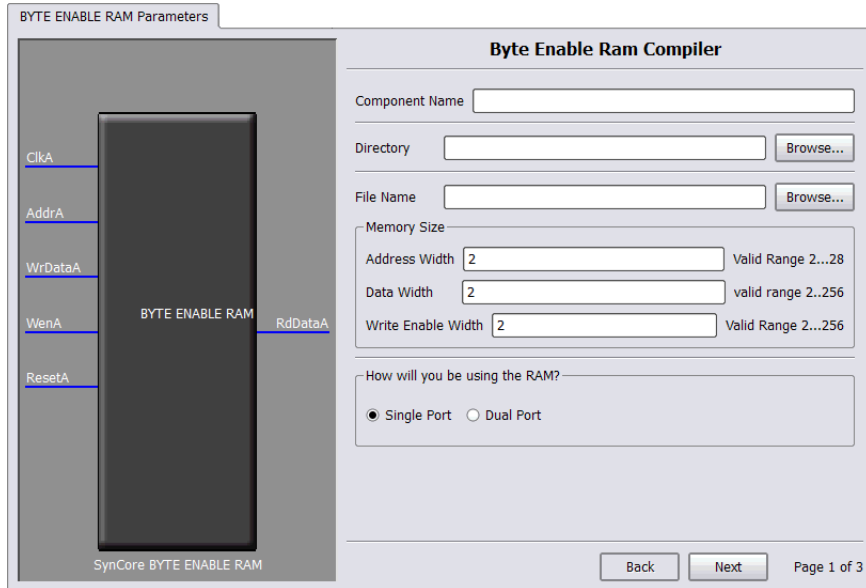
The SYNCore IP wizard helps you generate SystemVerilog code for your byte-enable RAM implementation requirements. The following procedure shows you how to generate SystemVerilog code for a byte-enable RAM using the SYNCore IP wizard.

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.



2. Specify the parameters you need in the wizard. For details about the parameters, see [Specifying Byte-Enable RAM Parameters, on page 390](#). 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.

```

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.

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 798](#) in the Reference Manual.

1. Start the SYNCORE byte-enable RAM wizard as described in [Specifying Byte-Enable RAMs with SYNCORE, on page 386](#).
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.

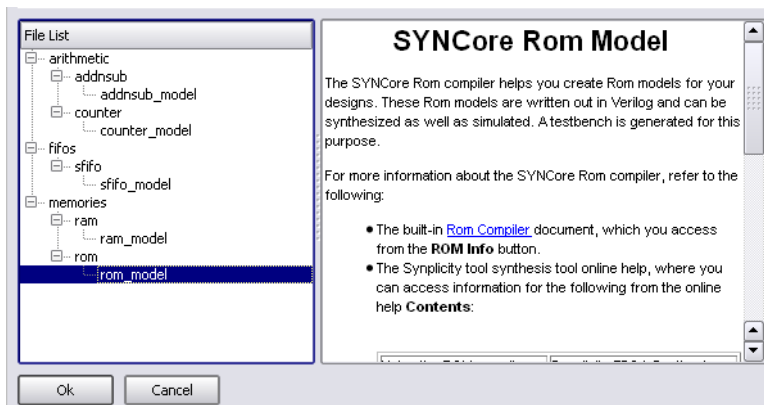
5. Generate the byte-enable RAM by clicking **Generate**. Add the file to your project and edit the template file as described in [Specifying Byte-Enable RAMs with SYNCORE, on page 386](#). For read/write timing diagrams, see [Read/Write Timing Sequences, on page 795](#) of the Reference Manual.

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.

The screenshot shows the 'Rom Compiler' wizard interface. On the left, a block diagram of a ROM is shown with three inputs: 'ClkA' (clock), 'AddrA' (address), and 'DataA' (data). The ROM is labeled 'SynCore ROM'. On the right, the configuration fields are as follows:

- Component Name:** A text input field.
- Directory:** A text input field with a 'Browse...' button.
- File Name:** A text input field with a 'Browse...' button.
- ROM Size:** A section containing:
 - Read Data width:** A text input field with the value '8' and a 'Valid Range 1..256' label.
 - ROM address width:** A text input field with the value '10' and a 'Valid Range 2..256' label.
- Configuring the ROM:** A section with two radio buttons:
 - ☒ Single Port Rom
 - ☐ Dual Port Rom

At the bottom right, there are 'Back' and 'Next' buttons, and a page indicator 'Page 1 of 4'.

- Specify the parameters you need in the wizard. For details about the parameters, see [Specifying ROM Parameters, on page 396](#). The ROM symbol on the left reflects any parameters you set.
- 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.

- 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 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

- 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

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 190](#) in the *Reference Manual*.

1. Start the SYNCore ROM wizard, as described in [Specifying ROMs with SYNCore, on page 392](#).
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 392](#) and add it to your design. All output files are in the directory you specified on page 1 of the wizard.

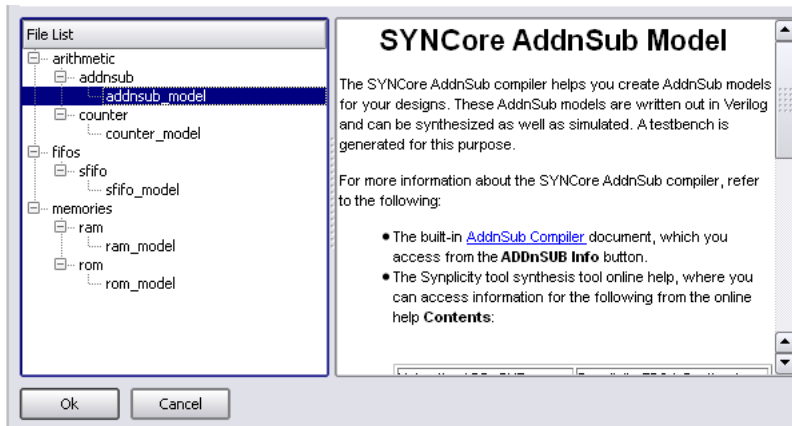
For read/write timing diagrams, see [Read/Write Timing Sequences, on page 791](#) of the *Reference Manual*.

Specifying Adder/Subtractors with SYNCORE

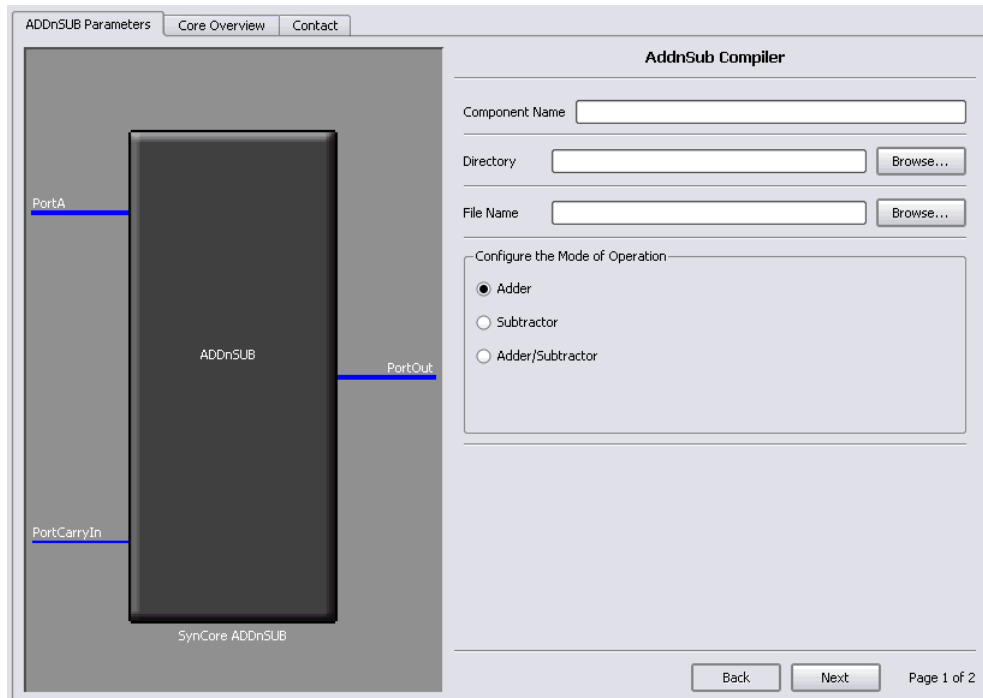
The SYNCORE IP wizard helps you generate Verilog code for your adder/subtractor implementation requirements. The following procedure shows you how to generate Verilog code for an adder/subtractor using the SYNCORE IP wizard.

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.



- In the window that opens, select `addnsub_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 Adder/Subtractor Parameters, on page 402](#). 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.

```

.....
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

- 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 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

```

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

Port Name	Description	Required/Optional
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 201](#) in the *Reference Manual*.

1. Start the SYNCORE adder/subtractor wizard as described in [Specifying Adder/Subtractors with SYNCORE, on page 397](#).
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.
 - 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.

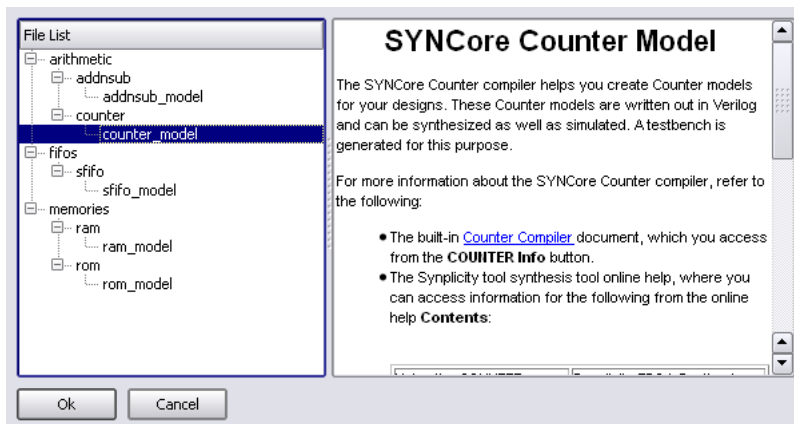
7. Generate the adder/subtractor by clicking the Generate button as described in [Specifying Adder/Subtractors with SYNCore, on page 397](#) and add it to your design. All output files are in the directory you specified on page 1 of the wizard.

Specifying Counters with SYNCore

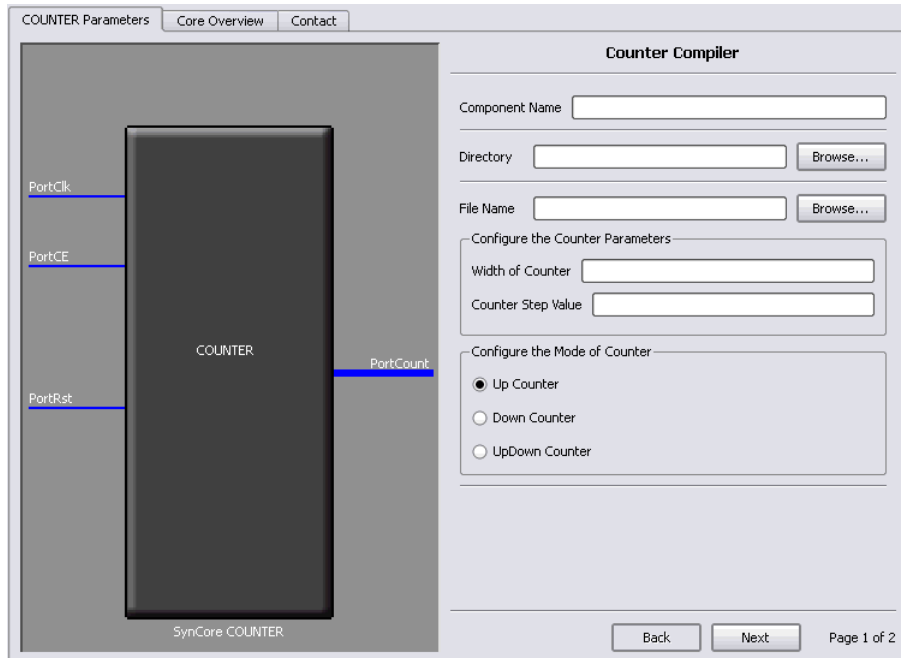
The SYNCore IP wizard helps you generate Verilog code for your counter implementation requirements. The following procedure shows you how to generate Verilog code for a counter using the SYNCore IP wizard.

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 page 1 of the wizard.



2. Specify the parameters you need in the wizard. For details about the parameters, see [Specifying Counter Parameters, on page 408](#). The COUNTER 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 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

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 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

```

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
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 205](#) of the *Reference Manual*.

1. Start the SYNCORE counter wizard, as described in [Specifying Counters with SYNCORE, on page 404](#).

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 page 1 of the wizard.

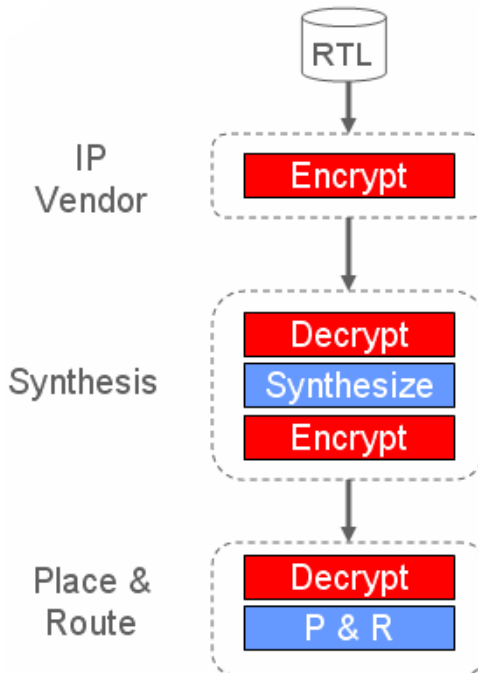
The Synopsys FPGA IP Encryption Flow

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:

- P1735 with key-block embedded rights information (Version 1)
- OpenIP

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.

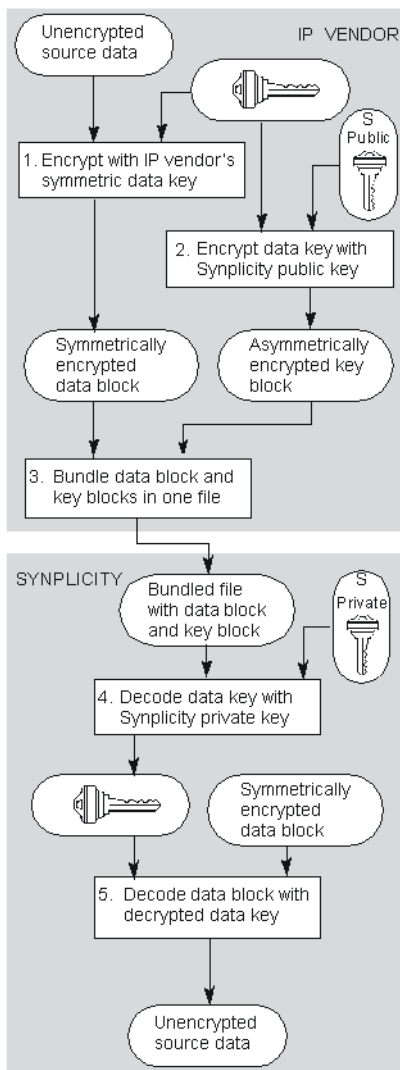


For further details of the hand-offs between vendors and how encryption and decryption are handled, see [Encryption and Decryption, on page 411](#).

Encryption and Decryption

There are two major classes of encryption/decryption algorithms: symmetric, and asymmetric (see [Encryption and Decryption Methodologies, on page 822](#) 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 handoffs. 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.

The following figure illustrates the steps in this encryption/decryption methodology, showing the handoff from an IP vendor to a Synopsys FPGA synthesis tool.

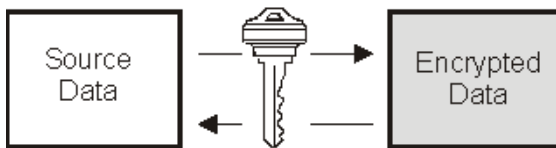


The following describes each of the phases shown in the figure. Note that Synopsys provides the following scripts to simplify and automate the process of encrypting data for the IP vendor.

- P1735
- OpenIP

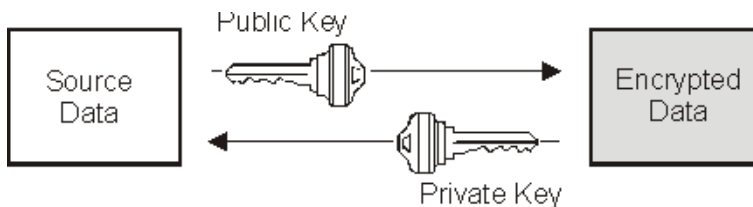
Data Encryption - Step 1

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.



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 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.

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 833](#) 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.

Working with Encrypted IP

The Synopsys FPGA IP encryption schemes available to Synplify Pro include:

- P1735
- 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 416
- [Preparing the IP Package](#), on page 422

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. All of the Synopsys encryption methodologies support RSA encryption.

Synopsys provides scripts to simplify this process. See the following procedures for details on script usage.

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.

3. 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 418](#)
- [Encrypting IP with the encryptIP Script, on page 419.](#)

4. Package your IP, as described in [Preparing the IP Package, on page 422.](#)

5. 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 System Designer™ 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 P1735 proposed standard with limited interoperability and rights information embedded in the key block. The encryptP1735.pl script accepts inputs from three sources: command line arguments, the RTL input file containing one or more encryption envelopes, and a file containing the public keys.

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 ([Using Hyper Source, on page 426](#)). 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. Synplify currently supports RSA encryption.

The encryptP1735.pl script is located in the *installDir/lib* directory and requires the installation of Perl on your machine. You cannot run the script if you do not have Perl installed. The following examples show typical script applications. For more information on the script and the command line arguments, see [Syntax for Running encryptP1735, on page 823](#) in the Reference Manual.

Example 1

To encrypt a file using a random key:

```
perl encryptP1735.pl -input plain_ip.v -output protected_ip.v  
-output_method plaintext
```

Example 2

To encrypt a file using a user-specified key:

```
perl encryptP1735.pl -input plain_ip.v -output protected_ip.v  
-output_method plaintext -key mySpecifiedKey
```

In the above examples, the `-input` (or `-i`) argument specifies the name of the file to be encrypted, and the `-output` (or `-o`) argument specifies the name of the resultant encrypted file. The `-output_method` (or `-om`) argument accepts the following values:

- `plaintext` indicates that the synthesis output netlist for the IP will be in plaintext format
- `encrypted` indicates that the synthesis output netlist for the IP will be encrypted using the same session key as input
- `blackbox` indicates that the synthesis output netlist for the IP will not be written (that is, the IP will be black-boxed).

The `-key` (or `-k`) argument is optional and specifies the session key in text format.

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 ([Using Hyper Source, on page 426](#)).

- 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. Synplify currently supports RSA encryption.

1. Install the `encryptIP` Perl script.

- You can download the `encryptIP` Perl script from SolvNet. See the article published at:

<https://solvnet.synopsys.com/retrieve/032343.html>

- 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 for Running encryptIP, on page 832](#) 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 `plaintext`, 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 421](#) 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 411](#).

All other output files from synthesis, including `srm` 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 for Running encryptIP, on page 832](#) 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 for Running encryptIP, on page 832](#) or [Syntax for Running encryptP1735, on page 823](#) in the *Reference Manual*.

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 encryptIP script, set `-om` to `persistent_key` 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 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 view, 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.

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 416](#).
 - Make sure your package includes the files listed in [IP Package File List, on page 423](#).
 - Structure the files.
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 424](#) 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.
SPIRIT IP-XACT v1.4 models	System-level models for your IP. This allows the synthesis tools to include your IP in a system-level design by stitching the IP together using bus architectures.

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.

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.

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 more information:

- [Using Hyper Source for Prototyping](#), on page 426
- [Using Hyper Source for IP Designs](#), on page 426
- [Threading Signals Through the Design Hierarchy of an IP](#), on page 427

Using Hyper Source for Prototyping

For prototyping, use hyper source to address the following issues:

- Use it to efficiently thread nets across multiple modules to the top-level design to support Time Domain Multiplexing (TDM).
- 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 the RTL. After the RTL has been fully tested with complete QoR results, use hyper source to debug. For example:

- 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.

- 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 428](#).

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
@W: : 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)
```

```
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;

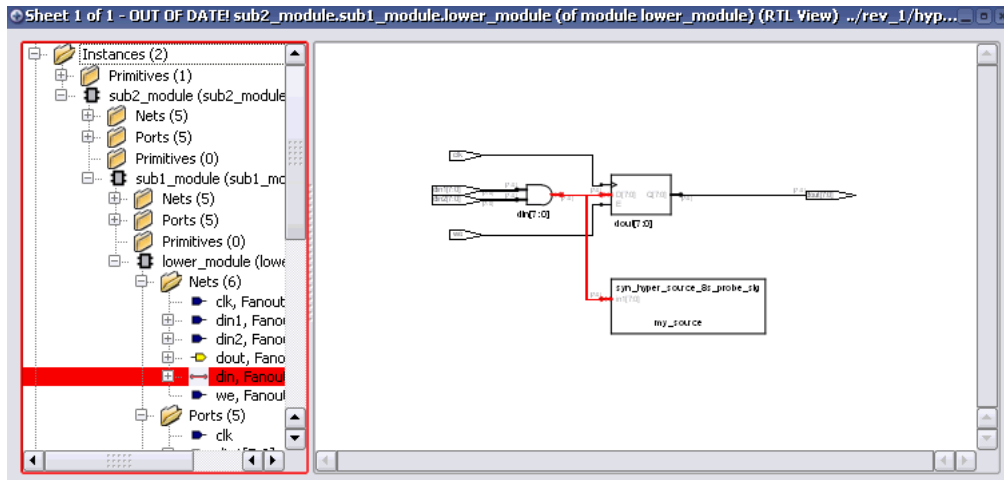
syn_hyper_connect connect_block(probe);
defparam connect_block.tag = "probe_sig"; /* to thread the signal this
tag_name must match to name used in the hyper connect block */
defparam connect_block.w = 8;

sub2_module sub2_module (clk, dout, din1, din2, we);

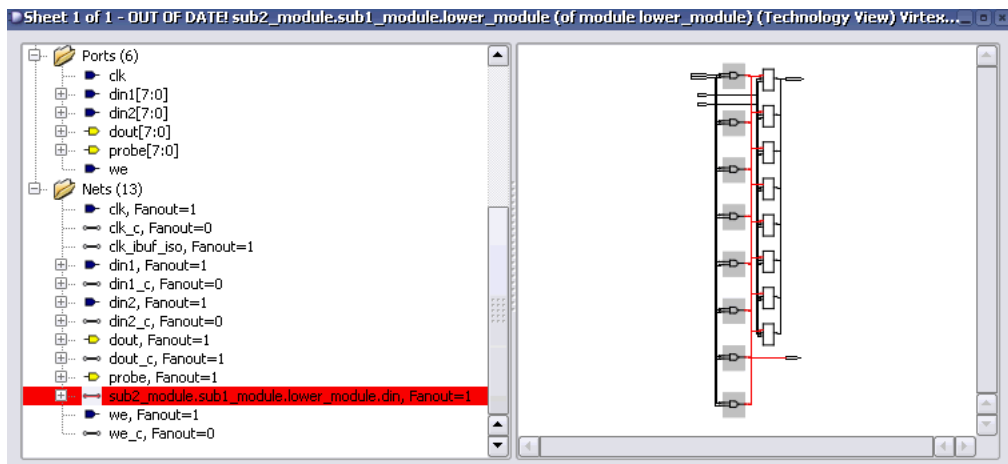
endmodule
```

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



CHAPTER 15

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 432
- [Compile Point Synthesis Basics](#), on page 441
- [Synthesizing Compile Points](#), on page 451
- [Using Compile Points with Other Features](#), on page 462
- [Resynthesizing Incrementally](#), on page 463

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](#), next
- [Nested Compile Points](#), on page 435
- [Compile Point Types](#), on page 436

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 433
- [Design Preservation](#), on page 433

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 447](#) 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. See [Combining Compile Points with Multiprocessing, on page 462](#).
- 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 463](#).

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.

- 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

Manual compile points require more 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

Determine the kind of compile point to use based on what the design requires. 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.
-

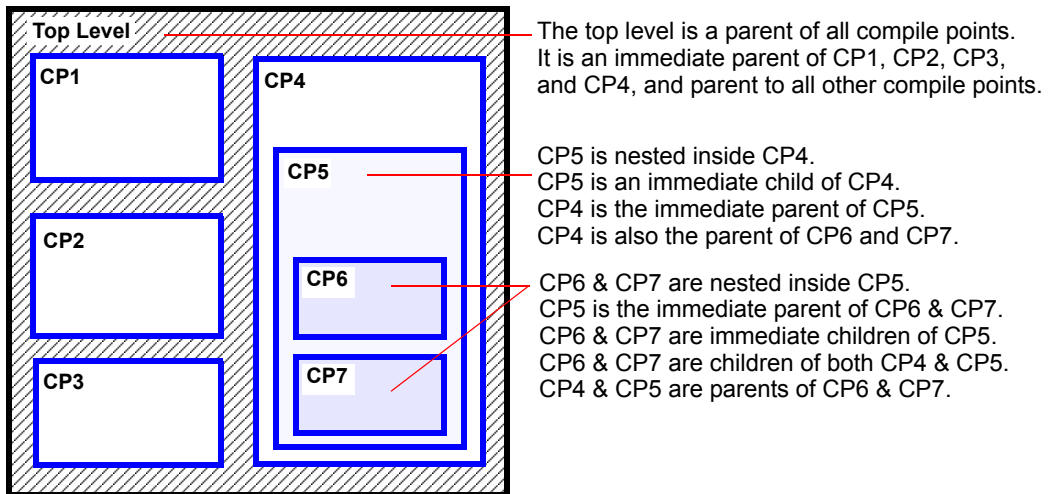
When you do not want further optimizations to a completed compile point.

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 point type determines whether boundary optimizations are allowed. For manual compile points, you define the type. See [Defining the Compile Point Type, on page 457](#) for details.

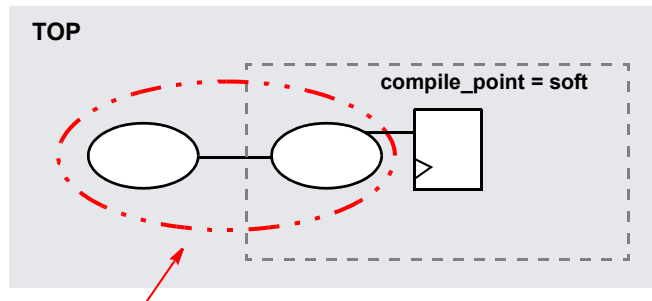
These are descriptions of the **soft**, **hard**, and **locked** compile types:

- **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.

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.

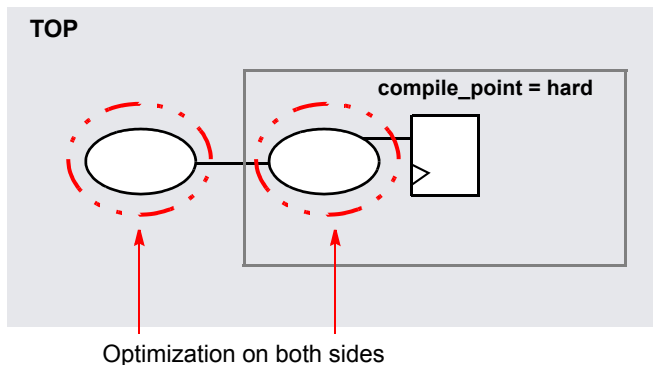


Optimization of entire logic cone across boundary

- 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.



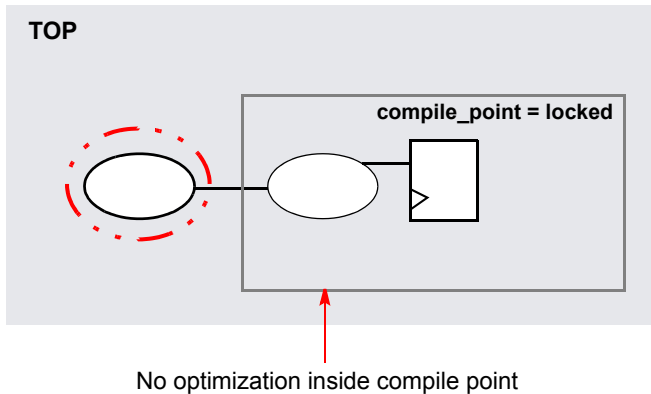
The hard compile point type allows for optimizations on both sides of the boundary without changing the boundary. There is a trade-off in quality of results to keep the boundaries. Using hard also allows for hierarchical equivalence checking for the compile point module.

- Locked

This is the default compile point type. 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 444](#)) 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 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
Allows for obtaining stable results for a completed part of the design.	
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

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 441
- [Interface Logic Models](#), on page 444
- [Interface Timing for Compile Points](#), on page 444
- [Compile Point Synthesis](#), on page 447
- [Incremental Compile Point Synthesis](#), on page 450
- [Forward-annotation of Compile Point Timing Constraints](#), on page 451

For step-by-step information about how to use compile points, see [Synthesizing Compile Points](#), on page 451.

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

```

- Constraint 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 458](#) 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 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.pgrm_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"
```

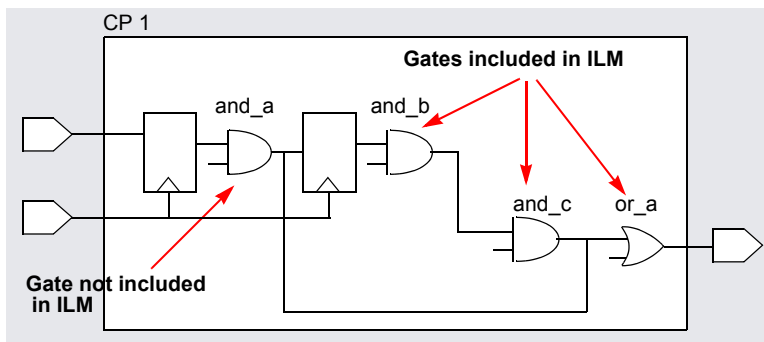
If your design has some compile points with their own constraint files and others without them, the tool uses the defined compile point constraints when it synthesizes those compile points. For the other compile points without defined constraints, it uses automatic interface timing, as described in [Interface Timing for Compile Points](#), on page 444.

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.



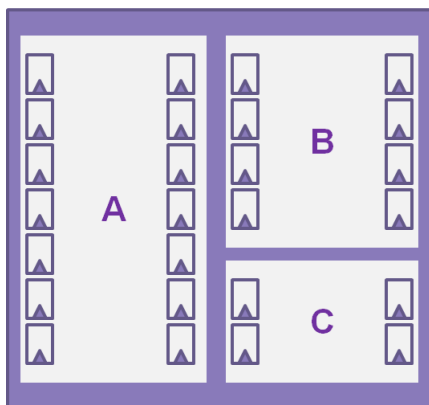
Interface Timing for Compile Points

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.

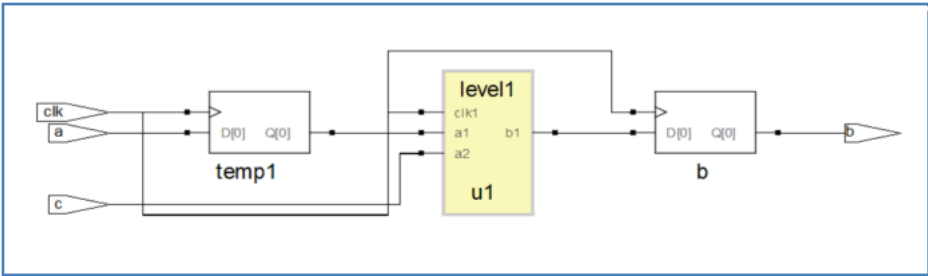
- When there are compile point constraint files, the tool first synthesizes the compile point using the constraints in the compile point constraints file and then synthesizes the top level using the top-level constraints.

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.

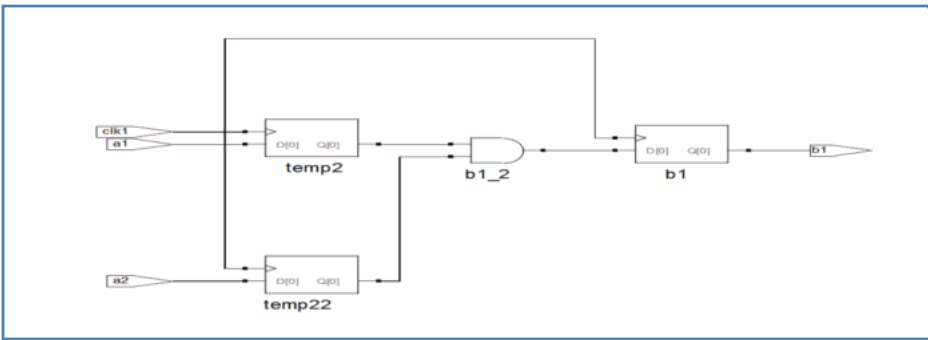


Interface Timing Example

The design below shows how the interface timing works on compile points.



Contents of level1 Module

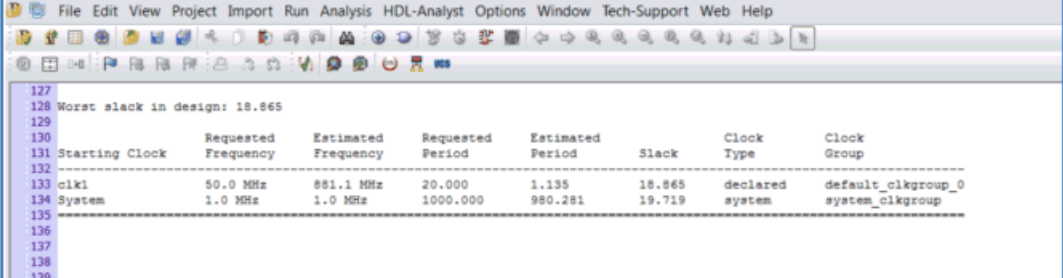


Interface Timing Off

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.

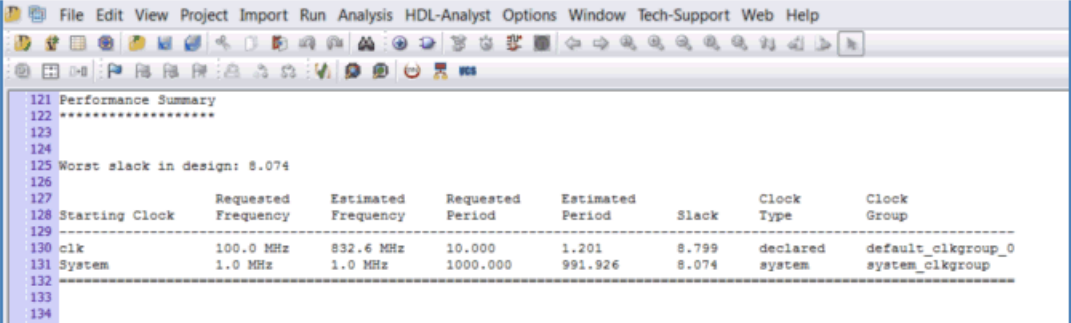


127 Worst slack in design: 18.865

Starting Clock	Requested Frequency	Estimated Frequency	Requested Period	Estimated Period	Slack	Clock Type	Clock Group
clk1	50.0 MHz	881.1 MHz	20.000	1.135	18.865	declared	default_clkgroup_0
System	1.0 MHz	1.0 MHz	1000.000	980.281	19.719	system	system_clkgroup

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:



121 Performance Summary

122 *****

123

124

125 Worst slack in design: 8.074

Starting Clock	Requested Frequency	Estimated Frequency	Requested Period	Estimated Period	Slack	Clock Type	Clock Group
clk	100.0 MHz	832.6 MHz	10.000	1.201	8.799	declared	default_clkgroup_0
System	1.0 MHz	1.0 MHz	1000.000	991.926	8.074	system	system_clkgroup

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 451](#).

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. You can specify that the tool ignore the error and continue synthesizing other compile points. See [Continue on Error Mode, on page 449](#).

Stage 2: Top-Level Synthesis

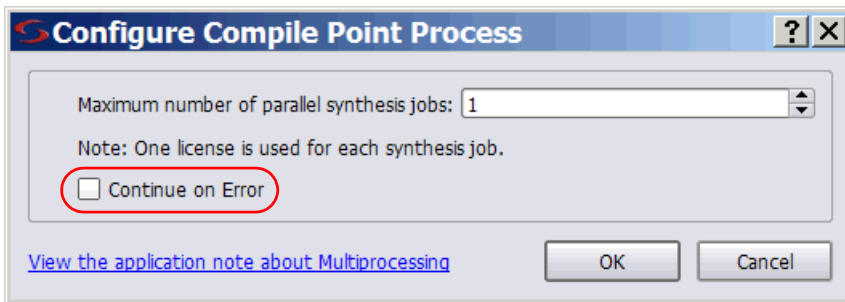
Once all the compile points have been synthesized, the tool synthesizes the entire design from the top down, using the model information generated for each compile point and constraints defined in the top-level constraints file. You do not need to duplicate compile point constraints at a higher level, 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 451](#) for a description of the constraints that are forward-annotated.

Continue on Error Mode

By default, the tool stops the synthesis process if it encounters an error within a compile point. If you want to be able to continue compile-point synthesis with another compile point, do any of the following:

- Select Options->Configure Compile Point Process from the top menu and enable the Continue on Error checkbox
- Select the Options panel on the Implementation Options dialog box and enable the Continue on Error checkbox
- Check the Continue on Error checkbox on the left side of the Project view
- Enter a `set_option -continue_on_error` option with a value of 1 at the Tcl script prompt



With this setting, when the tool encounters a mapper error it black boxes the affected compile point and continues to synthesize other compile points. The log file report after synthesis contains warnings like the following for the ignored errors:

```
@W:: m1.v(1) | Mapping of compile point m1 - Unsuccessful
```

Note the following about the scope of this setting:

- The setting applies to both mapper errors. All compiler errors, however, must be corrected before synthesis can proceed.
- In Synplify Pro, the setting only applies to compile-point mapper errors. Top-level design errors terminate synthesis, and the tool does not generate an output netlist.

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:

- 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.

Synthesizing Compile Points

This section describes the synthesis process with manual compile points in your design:

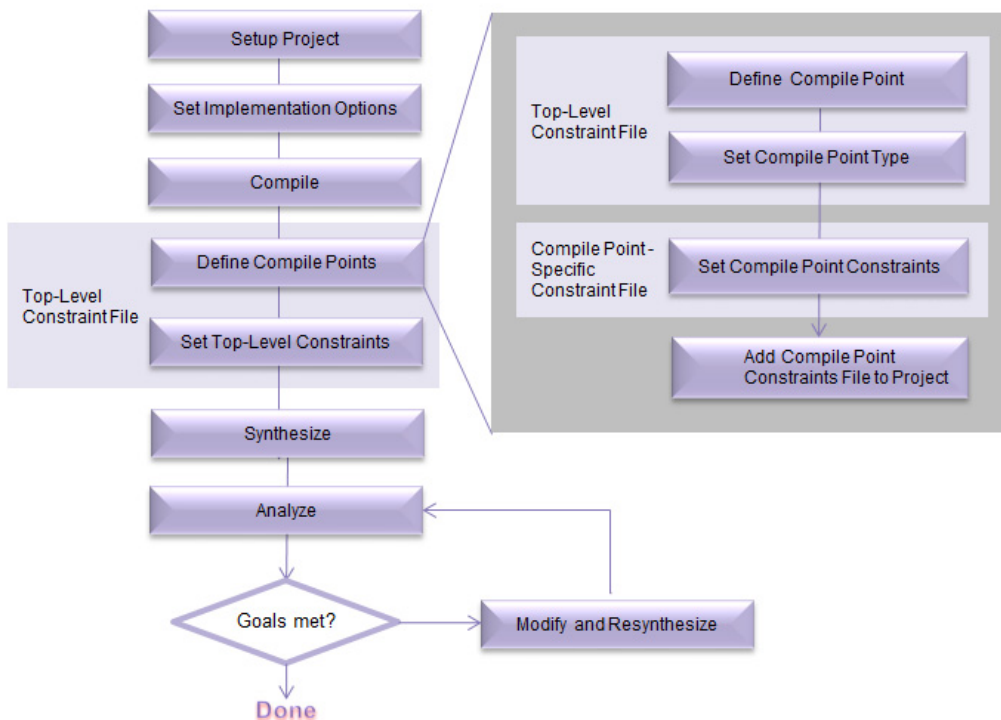
- [The Manual Compile Point Flow](#), on page 452
- [Creating a Top-Level Constraints File for Compile Points](#), on page 454
- [Defining Manual Compile Points](#), on page 455
- [Setting Constraints at the Compile Point Level](#), on page 458
- [Analyzing Compile Point Results](#), on page 460

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. This includes most of the newer Microsemi device families.
 - 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 454](#). Note that by default, the tool automatically calculates the interface timing for all compile points.
 - Set timing constraints and attributes in the top-level constraint file:

Constraint	Apply to...	Example
Clock	All clocks in the design.	<code>create_clock {p:clk} -name clk -period 100 -clockgroup cg1</code>
I/O constraints	All top-level port constraints. Register the compile point I/O boundaries to improve timing.	<code>set_input_delay {p:a} {1} -clock {clk:r}</code>
Timing exceptions	All timing exceptions that are outside the compile point module, or that might be partially in the compile point modules.	<code>set_false_path -from {i:reg1} -to {i:reg2}</code>
Attributes	All attributes that are applicable to the rest of the design, not within the compile points.	<code>define_attribute {i:statemachine_1} syn_encoding {sequential}</code>

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 458](#) for a step-by-step procedure. After setting the compile point constraints, add the compile point constraint file to the project.
5. If you do not want to interrupt synthesis for compiler errors, select Options->Configure Compile Point Process and enable the Continue on Error option.

With this option enabled, the tool black boxes any compile points that have mapper errors and continues to synthesize the rest of the design. See [Continue on Error Mode, on page 449](#) for more information about this mode.

6. Synthesize the design.

The tool synthesizes the compile points separately and then synthesizes the top level. See [Compile Point Synthesis, on page 447](#) 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.

7. Analyze the synthesis results using the top-level srr log file.

See [Analyzing Compile Point Results, on page 460](#) for details.

8. 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 450](#). See [Resynthesizing Compile Points Incrementally, on page 463](#) for a detailed procedure.

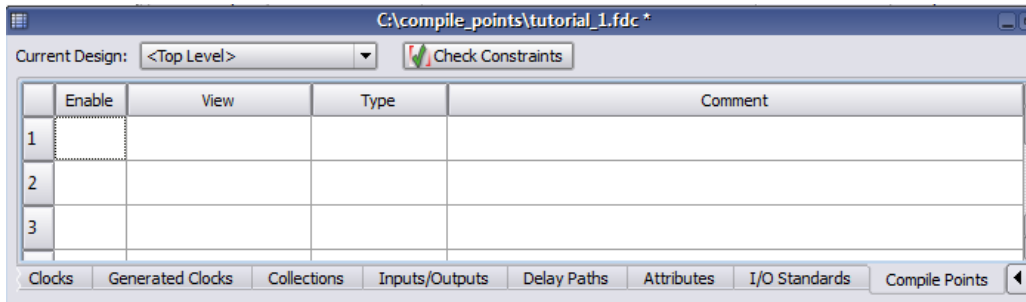
Creating a Top-Level Constraints File for Compile Points

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 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.

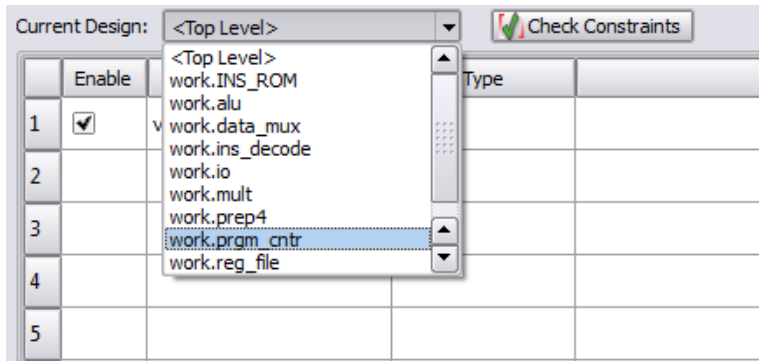
See [Defining Manual Compile Points, on page 455](#) for details.

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 454](#) 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 {soft}
```

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_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 457](#) 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 458](#) for details.

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 436](#). 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
---------------------	--------

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}
```

	Enabled	Module	Type
1	<input checked="" type="checkbox"/>	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
------------------------------------	--------

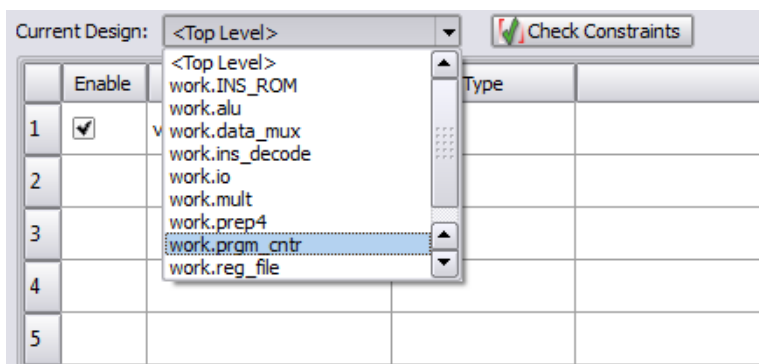
3. If your goal is design preservation, set the compile point you want to preserve to locked.

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 441 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.

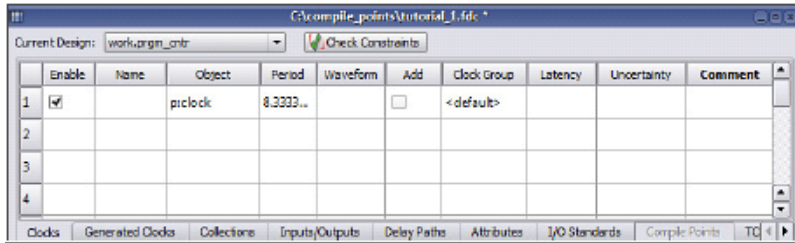
When you specify compile point constraints, the tool synthesizes the compile point using the compile point timing models instead of automatic interface timing from the top level. This procedure explains how to create a (compile point constraint file, and set constraints for the compile point:

1. In an open project, click the SCOPE icon (). Click 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.



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 459](#) 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 :						

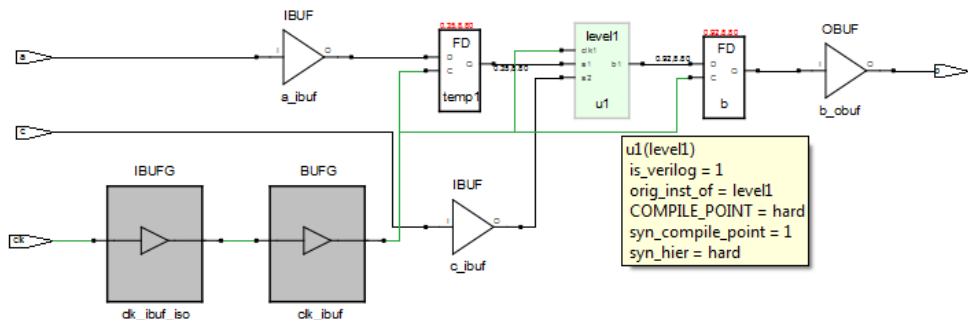
Name	Status	Reason	Start Time	End Time	Realtime	CPU Time
user_top	Remapped	Mapping options changed	Mon Mar 14 04:43:42 2011	Mon Mar 14 04:44:27 2011	0h:00m:44s	0h:00m:33s

Note that this section reports black box compile points as Not Mapped, and lists the reason as Black Box.

- Review all warnings and determine which should be addressed and which can be ignored.

- 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 point of type {hard} is 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.

Using Compile Points with Other Features

You can effectively combine compile points with other synthesis features for better runtime. The following sections describe how you can use compile points with multiprocessing:

- [Combining Compile Points with Multiprocessing](#), on page 462

Combining Compile Points with Multiprocessing

To use compile points with multiprocessing, do the following.

1. Set up the project with compile points.
2. Specify the number of parallel jobs to run with the Options->Configure Compile Point Process command.

Alternatively, you can set this with the `set_option -max_parallel_jobs` Tcl command, or in the ini file.

3. Run synthesis.

The software synthesizes the compile points as separate processor jobs. Parallel processing reduces runtime.

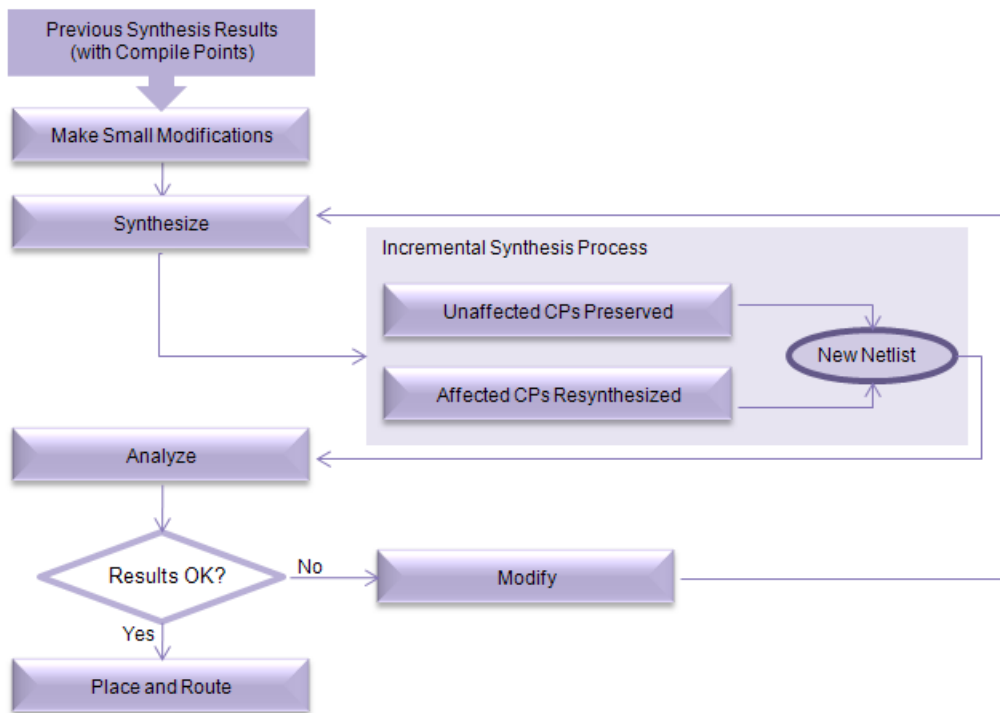
Resynthesizing Incrementally

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 450
- [Resynthesizing Compile Points Incrementally](#), on page 463

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.

- 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 450](#) 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.

First Run Log Summary

Summary of Compile Points		
Name	Status	Reason
mult	Mapped	No database
comb_logic	Mapped	No database
alu	Mapped	No database
eight_bit_uc	Mapped	No database
=====		

Syntax changes only; not resynthesized

Logic changes; compile
point resynthesized

Incremental Run Log Summary

Summary of Compile Points		
Name	Status	Reason
mult	Unchanged	-
comb_logic	Remapped	Design changed
alu	Unchanged	-
eight_bit_uc	Unchanged	-
=====		

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.

CHAPTER 16

Optimizing Processes for Productivity

This chapter covers topics that can help the advanced user improve productivity and inter operability with other tools. It includes the following:

- [Using Batch Mode](#), on page 468
- [Working with Tcl Scripts and Commands](#), on page 474
- [Automating Flows with synhooks.tcl](#), on page 481

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 474](#).

This section describes the following operations:

- [Running Batch Mode on a Project File](#), on page 468
- [Running Batch Mode with a Tcl Script](#), on page 469
- [License Queuing](#), on page 470

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 469](#).

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 476](#).
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.

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.

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 468](#).

1. Create a Tcl batch script. See [Creating a Tcl Synthesis Script, on page 476](#) 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 the following:

```
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 `result-File.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.ssr* 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

License Queuing

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 not immediately available. You can use the following types of license queuing:

- [Blocking-style Queuing](#)
- [Non Blocking-style Queuing](#)

Blocking-style Queuing

When blocking-style queuing is enabled and the requested license feature is on the server but not available, the tool waits until the license becomes available.

To enable blocking-style license queuing, either:

- Set environment variable `toolName_LICENSE_WAIT=1` (*toolName* is the name of the FPGA synthesis tool)
- Include a `-license_wait` command-line argument when launching batch mode as shown in the following example:

```
synplify_pro -batch -license_wait Tcl_script.tcl
```

For blocking-style license queuing, the following message is generated to `stdout.log` or the Tcl window:

```
Waiting for license: toolName
```

For example:

```
Waiting for license: synplifypro
```

Queuing Considerations

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.

Non Blocking-style Queuing

When non blocking-style queuing is enabled and the requested license feature is on the server but not available, the tool waits up to the maximum wait time specified in seconds for the license to become available.

To enable non blocking-style license queuing, you can:

- 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
```

- 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
```

For non blocking-style license queuing, the following message is generated to stdout.log or the Tcl window:

```
Waiting up to n seconds for license: toolName
```

For example:

```
Waiting up to 230 seconds for license: synplifypro
```

Non blocking-style license queuing behavior for the environment variable `toolName_LICENSE_WAIT=waitTime` or `-license_wait waitTime` command-line argument is shown in the following tables.

Environment Variable	Queuing Behavior
SYNPLIFYPRO_LICENSE_WAIT (undefined)	Queuing off
SYNPLIFYPRO_LICENSE_WAIT=0	Queuing off
SYNPLIFYPRO_LICENSE_WAIT=1	Queuing on; wait indefinitely
SYNPLIFYPRO_LICENSE_WAIT= <i>n</i> (<i>n</i> >1)	Queuing on; wait up to <i>n</i> seconds

Command Line Argument	Queuing Behavior
<code>synplifypro -license_wait arguments</code>	Queuing on; wait indefinitely
<code>synplifypro -license_wait 1 arguments</code>	Queuing on; wait indefinitely
<code>synplifypro license_wait n (n>1)</code>	Queuing on; wait up to <i>n</i> seconds
<code>synplifypro arguments</code>	Queuing off
<code>synplifypro -license_wait 0 arguments</code>	Queuing off

- Specify a list of features to wait for using batch mode:

```
synplify_pro -batch -license_wait -licensetype
synplifypro:synplifypro_microsemi name_of_project.prj
```

- Specify feature names to wait for using the `toolName_LICENSE_TYPE` environment variable:

```
SYNPLIFYPRO_LICENSE_TYPE=synplifypro:synplifypro_microsemi
```

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 475
- [Setting Number of Parallel Jobs](#), on page 475
- [Creating a Tcl Synthesis Script](#), on page 476
- [Using Tcl Variables to Try Different Clock Frequencies](#), on page 478
- [Using Tcl Variables to Try Several Target Technologies](#), on page 479
- [Running Bottom-up Synthesis with a Script](#), on page 480

You can also use synhooks Tcl scripts, as described in [Automating Flows with synhooks.tcl](#), on page 481.

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 *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 475](#) and [Creating a Tcl Synthesis Script, on page 476](#).
- 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 1093](#) in the *Reference manual*.

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 416](#).

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 PROASIC3
    set_option -part A3P400
    set_option -package FBGA144
    set_option -speed_grade -Std

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 1068 in the *Reference Manual*). 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 [i:statemod.statereg\[*\]](#), on page 49 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 *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.

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 diagram illustrates a Tcl script snippet with annotations. On the left, a grey box contains the text "Set of frequencies to try" with a red arrow pointing to the list of frequencies in the script. Below it, another grey box contains the text "Foreach loop" with a red arrow pointing to the `foreach` loop in the script. On the right, a grey box contains the text "Tcl commands that set the frequency, create separate log files for each run, and run synthesis" with a red arrow pointing to the commands inside the `foreach` loop.

```
set try_freq {  
    85.0  
    90.0  
    92.0  
    95.0  
    97.0  
    100.0  
}  
  
foreach frequency $try_freq {  
    set_option -frequency $frequency  
    project -log_file $frequency.srr  
    project -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 478](#). 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 {

    PROASIC3 PROASIC3E # 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
}
```

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
    IGLOO IGLOOE FUSION # 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 476](#) for details.
2. Create a top-level script that reads the block scripts. Create the script 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 249](#)), 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 formal verification with the Cadence Conformal tool. 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 location of the `synhooks.tcl` file.
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 [synhooks File Syntax, on page 1118](#) in the *Reference Manual*.

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 483 .	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 483 .	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]  
    }  
}
```


CHAPTER 17

Using Multiprocessing

The following sections describe how to use multiprocessing to run parallel synthesis jobs and improve runtime:

- [Multiprocessing With Compile Points](#), on page 486
 - [Setting Maximum Parallel Jobs](#), on page 486
 - [License Utilization](#), on page 487

Multiprocessing With Compile Points

Use the Configure Compile Point Process command to run multiprocessing with compile points. This option allows the synthesis software to run multiple, independent compile point jobs simultaneously, providing additional runtime improvements for the logical compile point synthesis flows. On the Configure Compile Point Process dialog box, specify the maximum number of synthesis jobs you can run in parallel. Note, one license is used for each job. For a description of how to set the maximum number of parallel synthesis jobs, see [Setting Maximum Parallel Jobs, on page 486](#).

To use multiprocessing in the Logical Compile Point Synthesis flows for the Synplify Pro tool, see [Chapter 15, Working with Compile Points](#).

Setting Maximum Parallel Jobs

You can set maximum number of parallel jobs in the following ways:

- [INI Variable — MaxParallelJobs](#)
- [Tcl Variable — max_parallel_jobs](#)

INI Variable — MaxParallelJobs

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. For a description of the dialog box, see [Configure Compile Point Process Command, on page 243](#).

Tcl Variable — max_parallel_jobs

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.

License Utilization

When you decide to run parallel synthesis jobs, a license is used for each compile point job that runs. For example, if you set the Maximum number of parallel synthesis jobs to 4, then the synthesis tool consumes one license and three additional licenses are utilized to run the 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:

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 [Specifying License Types, on page 488](#).

Factors 1 and 3 above can change during a single synthesis run. The number of jobs equals the number of licenses; which then equates the lowest value of these three factors.

Specifying License Types

You can specify multiple license types to increase the total number of licenses available for multiprocessing. To do this, you can either:

- Use the `-licensetype` command line option when you execute your tool.

For example, suppose you have two `synplifypro` licenses, two `synplifypro_allvendor` licenses, and three `synplifypro_microsemi` licenses. Type the following at the command line:

```
synplify_pro.exe -licensetype  
"synplifypro:synplifypro_allvendor:synplifypro_microsemi"
```

- Use the following environment variables specified with the license type:
 - `SYNPLIFYPRO_LICENSE_TYPE` (Synplify Pro tool)

```
setenv SYNPLIFYPRO_LICENSE_TYPE=  
"synplifypro:synplifypro_allvendor:synplifypro_microsemi"
```

Multiprocessing can access any of these license types for additional licenses.

Index

Symbols

.adc file [334](#)
.fdc file [53](#)
.ini file
 parallel jobs [475](#)

A

ACTgen macros [347](#)
adc constraints [334](#)
adc file
 creating [334](#)
 object names [338](#)
adc file, using [332](#)
adders
 SYNCore [397](#)
alspin
 bus port pin numbers [353](#)
Alt key
 column editing [36](#)
 mapping [295](#)
analysis design constraint file (.adc) [334](#)
analysis design constraints
 design scenarios [333](#)
analysis design constraints (adc) [332](#)
analysis design constraints (adc), using
 with fdc [335](#)
archive utility
 using [153](#)
archiving projects [153](#)
area, optimizing [193](#)
asterisk wildcard
 Find command [283](#)
attributes
 adding [142](#)
 adding in constraint files [47](#)
 adding in SCOPE [146](#)

 adding in Verilog [145](#)
 adding in VHDL [143](#)
 effects of retiming [200](#)
 for FSMs [177](#), [224](#)
 syn_hier (on compile points) [459](#)
 VHDL package [143](#)

audience for the document [21](#)
auto constraints, using [339](#)
AutoConstraint_design_name.fdc [342](#)

B

B.E.S.T [299](#)
backslash
 escaping dot wildcard in Find
 command [283](#)
batch mode [468](#)
 using find and expand [80](#)
Behavior Extracting Synthesis
 Technology. See B.E.S.T
black boxes [166](#)
 adding constraints [170](#)
 adding constraints in SCOPE [173](#)
 adding constraints in Verilog [172](#)
 adding constraints in VHDL [171](#)
 instantiating in Verilog [166](#)
 instantiating in VHDL [168](#)
 passing VHDL boolean generics [44](#)
 passing VHDL integer generics [45](#)
 pin attributes [174](#)
 timing constraints [170](#)
blocking-style license queuing [471](#)
bookmarks
 in source files [36](#)
 using in log files [241](#)
bottom-up design flow
 compile point advantages [432](#)
browsers [270](#)
buffering
 controlling [211](#)

byte-enable RAMs
SYNCore 386

C

c_diff command, examples 87
c_intersect command, examples 87
c_list command
 different from c_print 89
 example 91
 using 91
c_print command
 different from c_list 89
 using 90
c_syndiff command, examples 88
c_union command, examples 87
callback functions, customizing flow 481
clock constraints
 edge-to-edge delay 98
 false paths 109
 setting 59, 98
clock domains
 clock enables 105
 setting up 106
clock enables
 defining with multicycle path
 constraints 104
 negative slack 104
clock groups
 effect on false path constraints 73, 109
 for global frequency clocks 100
clock trees 325
clocks
 asymmetrical 101
 defining 100
 frequency 101
 gated. *See* gated clocks
 implicit false path 73, 109
 limited resources 106
 overriding false paths 109
 start and end points for clock enables 105
collections
 adding objects 86
 concatenating 86
 constraints 85
 copying 90

 creating from common objects 86
 creating from other collections 84
 creating in SCOPE 83
 creating in Tcl 85
 crossprobing objects 84
 definition 82
 diffing 86
 highlighting in HDL Analyst views 89
 listing objects 90
 listing objects and properties 89
 listing objects in a file 90
 listing objects in columnar format 89
 listing objects with c_list 89
 special characters 88
 Tcl window and SCOPE comparison 82
 using Tcl expand command 78
 using Tcl find command 76
 viewing 88
column editing 36
comments
 source files 36
compile point types
 hard 437
 locked 438
compile points
 advantages 432
 analyzing results 460
 automatic timing budgeting 444
 child 435
 constraint files 441
 constraints for forward-annotation 451
 constraints, internal 451
 creating constraint file 458
 defined 432
 defining in constraint files 455
 feature summary 440
 Identify flow 364
 incremental synthesis 463
 manual compile point flow 452
 multiprocessing 462
 nested 435
 optimization 448
 order of synthesis 448
 parent 435
 preserving with syn_hier 459
 resynthesis 450
 setting constraints 458
 setting type 457
 syn_hier 459
 synthesis process 447

- synthesizing 451
 - types 436
 - using `syn_allowed_resources` attribute 460
 - compile-point synthesis
 - interface logic models 444
 - compile-point synthesis flow
 - defining compile points 455
 - setting constraints 458
 - compiler directives 32
 - using 32
 - compiler directives (Verilog)
 - specifying 139
 - constants
 - extracting from VHDL source code 141
 - constraint files 46
 - applying to a collection 85
 - black box 170
 - compile point 441, 451
 - creating in a text editor 47
 - defining clocks 93
 - defining register delays 94
 - editing 65
 - effects of retiming 200
 - opening 53
 - options 135
 - setting for compile points 458
 - specifying through points 69
 - types of 95
 - vendor-specific 50
 - when to use 46
 - context
 - for object in filtered view 302
 - context help editor 32
 - SystemVerilog 32
 - continue on error 133
 - counters
 - SYNCore 404
 - critical paths
 - delay 326
 - flat view 326
 - hierarchical view 326
 - negative slack on clock enables 104
 - slack time 326
 - using `-route` 195
 - viewing 325
 - crossprobing 291
 - and retiming 200
 - collection objects 84
 - filtering text objects for 296
 - from FSM viewer 298
 - from log file 241
 - from message viewer 246
 - from text files 294
 - Hierarchy Browser 291
 - importance of encoding style 298
 - paths 295
 - RTL view 292
 - Technology view 292
 - Text Editor view 292
 - text file example 295
 - to FSM Viewer 298
 - to place-and-route file 267
 - Verilog file 292
 - VHDL file 292
 - within RTL and Technology views 291
 - current level
 - expanding logic from net 306
 - expanding logic from pin 306
 - searching current level and below 280
 - custom folders
 - creating 120
 - hierarchy management 120
 - customization
 - callback functions 481
- ## D
- data block 413
 - data key 413
 - default enum encoding 141
 - `define_attribute` 149
 - `define_clock` constraint 93
 - `define_false_paths` constraint 94
 - `define_input_delay` constraint 94
 - `define_multicycle_path` constraint 94
 - `define_output_delay` constraint 94
 - `define_reg_input_delay` constraint 94
 - `define_reg_output_delay` constraint 94
 - design flow
 - customizing with callback functions 481
 - design guidelines 192
 - design hierarchy

- viewing [300](#)
- design size
 - amount displayed on a sheet [267](#)
- design views
 - moving between views [266](#)
- device options
 - See also implementation options
- directives
 - adding [142](#)
 - adding in Verilog [145](#)
 - adding in VHDL [143](#)
 - black box [171](#), [172](#)
 - for FSMs [177](#)
 - specifying for Verilog compiler [139](#)
 - syn_state_machine [222](#)
 - syn_tco [172](#)
 - adding black box constraints [171](#)
 - syn_tpd [172](#)
 - adding black box constraints [171](#)
 - syn_tsu [172](#)
 - adding black box constraints [171](#)
- dissolving instances for flattening hierarchy [313](#)
- dot wildcard
 - Find command [283](#)
- drivers
 - preserving duplicates with syn_keep [203](#)
 - selecting [309](#)
- dual-port RAMs
 - SYNCore parameters [383](#)

E

- Editing window [34](#)
- editor
 - compiler directives [32](#)
- editor view
 - context help [32](#)
- emacs text editor [40](#)
- encoding styles
 - and crossprobing [298](#)
 - default VHDL [141](#)
 - FSM Compiler [221](#)
- encryption flow. See ReadyIP, encryptIP
- encryptip output constraints [422](#)

- encryptip output method
 - effect on output netlists [422](#)
- encryptIP script
 - controlling output [421](#)
 - encrypting IP [419](#)
 - output methods [421](#)
- environment variables
 - SYN_TCL_HOOKS [481](#)
- error codes [468](#)
- errors
 - continuing [133](#)
 - definition [34](#)
 - filtering [245](#)
 - sorting [245](#)
 - source files [34](#)
 - Verilog [34](#)
 - VHDL [34](#)
- expand
 - batch mode [80](#)
- Expand command
 - connection logic [309](#)
 - pin and net logic [305](#)
 - using [306](#)
- expand command
 - different from Tcl search [287](#)
- expand command (Tcl). See Tcl expand command
- Expand Inwards command
 - using [306](#)
- Expand Paths command
 - different from Isolate Paths [309](#)
- Expand to Register/Port command
 - using [306](#)
- expanding
 - connections [309](#)
 - pin and net logic [305](#)

F

- false paths
 - defining between clocks [109](#)
 - I/O paths [73](#), [109](#)
 - impact of clock group assignments [73](#), [109](#)
 - overriding [109](#)
 - ports [73](#), [108](#)
 - registers [73](#), [108](#)

- setting constraints 73, 108
- fanouts
 - buffering vs replication 211
 - hard limits 210
 - soft global limit 209
 - soft module-level limit 210
 - using syn_keep for replication 204
 - using syn_maxfan 209
- feature comparison
 - FPGA tools 18
- FIFOs
 - compiling with SYNCORE 372
- files
 - .fdc 53
 - .prf file 248
 - filtered messages 249
 - fsm.info 222
 - log 237
 - message filter (prf) 248
 - output 354
 - rom.info 273
 - searching 150
 - statemachine.info 319
 - synhooks.tcl 481
 - Tcl 474
 - See also Tcl commands
 - Tcl batch script 469
- Filter Schematic command, using 303
- Filter Schematic icon, using 303
- filtering 303
 - advantages over flattening 303
 - using to restrict search 280
- Find command
 - 280
 - browsing with 280
 - hierarchical search 281
 - long names 280
 - message viewer 245
 - reading long names 283
 - search scope, effect of 284
 - search scope, setting 281
 - searching the mapped database 282
 - searching the output netlist 288
 - setting limit for results 282
 - using in RTL and Technology views 280
 - using wildcards 283
 - wildcard examples 285
- find command
 - different from Tcl search 287
 - hierarchy 287
 - nuances and differences 287
- Find command (Tcl)
 - See also Tcl find command
- finding information
 - information organization 22
- Flatten Current Schematic command
 - transparent instances 311
 - using 311
- Flatten Schematic command
 - using 311
- flattening 311
 - See also dissolving
 - compared to filtering 303
 - dissolving instances 313
 - hidden instances 312
 - transparent instances 311
 - using syn_hier 207
 - using syn_netlist_hierarchy 207
- forward annotation
 - vendor-specific constraint files 50
- forward-annotation
 - compile point constraints 451
- FPGA Design Constraints Editor
 - using TCL View 62
- frequency
 - clocks 101
 - defining for non-clock signals 102
 - internal clocks 102
 - setting global 133
- from constraints
 - specifying 69
- FSM Compiler
 - advantages 219
 - enabling 220
- FSM encoding
 - user-defined 179
 - using syn_enum_encoding 178
- FSM Explorer 219
 - running 224
 - when to use 219
- FSM view
 - crossprobing from source file 294
- FSM Viewer 315
 - crossprobing 298

fsm.info file [222](#)

FSMs

See also FSM Compiler, FSM Explorer
attributes and directives [177](#)
defining in Verilog [175](#)
defining in VHDL [176](#)
definition [175](#)
optimizing with FSM Compiler [218](#)
properties [319](#)
state encodings [318](#)
transition diagram [316](#)
viewing [316](#)

G

gated clocks
defining [106](#)

generics

extracting from VHDL source code [141](#)
passing boolean [44](#)
passing integer [45](#)

global optimization options [132](#)

H

HDL Analyst

See also RTL view, Technology view
critical paths [325](#)
crossprobing [291](#)
filtering schematics [303](#)
Push/Pop mode [273](#), [276](#)
traversing hierarchy with mouse strokes [271](#)
traversing hierarchy with Push/Pop mode [273](#)
using [299](#)

HDL Analyst tool

deselecting objects [264](#)
selecting/deselecting objects [263](#)

HDL Analyst views

highlighting collections [89](#)

HDL views, annotating timing information [323](#)

help

information organization [22](#)

hidden instances

consequences of saving [301](#)
flattening [312](#)
restricting search by hiding [281](#)

specifying [301](#)

status in other views [301](#)

hierarchical design

expanding logic from nets [306](#)
expanding logic from pins [306](#)

hierarchical instances

dissolving [313](#)
hiding. *See* hidden instances, Hide Instances command
multiple sheets for internal logic [302](#)
pin name display [304](#)
viewing internal logic [301](#)

hierarchical objects

pushing into with mouse stroke [272](#)
traversing with Push/Pop mode [273](#)

hierarchical search [280](#)

hierarchy

flattening [311](#)
traversing [270](#)

hierarchy browser

clock trees [325](#)
controlling display [267](#)
crossprobing from [291](#)
defined [270](#)
finding objects [278](#)
traversing hierarchy [270](#)

hierarchy management (custom folders)
[120](#)

hyper source

example [428](#)
for IPs [426](#)
for prototyping [426](#)
IP design hierarchy [426](#)
threading signals [427](#)

I

I/O insertion [218](#)

I/O pads

specifying I/O standards [62](#)

I/O paths

false path constraint [73](#), [109](#)

I/O standards [62](#)

I/Os

auto-constraining [340](#)
constraining [61](#), [107](#)
Verilog black boxes [166](#)

- VHDL black boxes 168
- Identify
 - compile points 364
- implementation options 129
 - device 129
 - global frequency 133
 - global optimization 132
 - part selection 129
 - specifying results 135
- implementations
 - copying 127
 - deleting 127
 - multiple. *See* multiple implementations.
 - overwriting 127
 - renaming 127
- incremental synthesis
 - compile points 463
- input constraints, setting 60, 107
- instances
 - preserving with `syn_noprune` 203
 - properties 258
 - properties of pins 259
- ILM *See* interface logic models
- interface logic models 444
- interface timing 444
- IP
 - encryption-decryption flow 411
 - re-encryption 415
- IP design hierarchy
 - hyper source 426
- IP encryption flow overview 410
- IP encryption scheme 416
- IP vendors
 - encrypting IP 416
 - package file list for encrypted IP flow 423
 - packaging for evaluation 422
 - supplying vendor information 424
- IPs
 - encrypting 416
 - encryption flow 410
 - SYNCore 372
 - SYNCore byte-enable RAMs 386
 - SYNCore counters 404
 - SYNCore FIFOs 372
 - SYNCore RAMs 378
 - SYNCore ROMs 392
 - SYNCore subtractors 397
 - using hyper source for debug 426
- Isolate Paths command
 - different from Expand Paths 309, 310
- J**
- job management
 - up-to-date checking 232
- K**
- key assignments
 - customizing 482
- key block 413
- keywords
 - completing words in Text Editor 35
- L**
- license queuing 470
 - blocking-style 471
 - non blocking-style 472
- log files
 - checking FSM descriptions 225
 - checking information 237
 - retiming report 199
 - setting default display 237
 - state machine descriptions 221
 - viewing 237
- logic
 - expanding between objects 309
 - expanding from net 306
 - expanding from pin 306
- logic preservation
 - `syn_hier` 207
 - `syn_keep` for nets 203
 - `syn_keep` for registers 203
 - `syn_noprune` 203
 - `syn_preserve` 203
- logical folders
 - creating 120
- M**
- manual compile points
 - flow 452

- max_parallel_jobs variable [476](#)
- maximum parallel jobs [475](#), [486](#)
- MaxParallelJobs variable [475](#)
- memory usage
 - maximizing with HDL Analyst [315](#)
- Message viewer
 - filtering messages [246](#)
 - keyboard shortcuts [245](#)
 - saving filter expressions [248](#)
 - searching [245](#)
 - using [244](#)
 - using the F3 key to search forward [245](#)
 - using the Shift-F3 key to search backward [245](#)
- messages
 - demoting [251](#)
 - filtering [246](#)
 - promoting [251](#)
 - saving filter information from command line [248](#)
 - saving filter information from GUI [248](#)
 - severity levels [252](#)
 - suppressing [251](#)
 - writing messages to file [249](#)
- Microsemi
 - ACTgen macros [347](#)
 - output netlist [354](#)
 - pin numbers for bus ports [353](#)
- Microsemil
 - macro libraries [346](#)
- mixed designs
 - troubleshooting [44](#)
- mixed language files [41](#)
- mouse strokes
 - pushing/popping objects [271](#)
- multicycle constraints
 - clock enables [104](#)
- multicycle paths
 - setting constraints [59](#), [99](#)
- multiple implementations [126](#)
 - running from project [127](#)
- multiprocessing
 - compile points [462](#)
 - maximum parallel jobs [475](#), [486](#)
- multisheet schematics [265](#)
 - for nested internal logic [302](#)
 - searching just one sheet [280](#)

- transparent instances [265](#)

N

- name spaces
 - output netlist [288](#)
 - technology view [282](#)
- navigating among design views [266](#)
- netlists for different vendors [354](#)
- nets
 - expanding logic from [306](#)
 - preserving for probing with syn_probe [203](#)
 - preserving with syn_keep [203](#)
 - properties [258](#)
 - selecting drivers [309](#)
- New property [261](#)
- non blocking-style license queuing [472](#)
- notes
 - filtering [245](#)
 - sorting [245](#)
- notes, definition [34](#)

O

- objects
 - finding on current sheet [280](#)
 - flagging by property [259](#)
 - selecting/deselecting [263](#)
- open_design
 - with find and expand [80](#)
- optimization
 - for area [193](#)
 - for timing [194](#)
 - logic preservation. *See* logic preservation.
 - preserving hierarchy [207](#)
 - preserving objects [203](#)
 - tips for [192](#)
- OR [71](#)
- orig_inst_of property [262](#)
- output constraints, setting [60](#), [107](#)
- output files [354](#)
 - specifying [135](#)
- output netlists
 - finding objects [288](#)

P

- package library, adding [114](#)
- pad types
 - industry standards [62](#)
- parallel jobs [475](#)
- parameter passing [45](#)
 - boolean generics [44](#)
- parameters
 - extracting from Verilog source code [138](#)
- part selection options [129](#)
- path constraints
 - false paths [73](#), [108](#)
- pathnames
 - using wildcards for long names (Find) [283](#)
- paths
 - crossprobing [295](#)
 - tracing between objects [309](#)
 - tracing from net [306](#)
 - tracing from pin [306](#)
- pattern searching [150](#)
- PDF
 - cutting from [36](#)
- pin names, displaying [305](#)
- pins
 - expanding logic from [306](#)
 - properties [258](#)
- ports
 - false path constraint [73](#), [108](#)
 - properties [258](#)
- POS interface
 - using [69](#)
- post-synthesis constraints with adc [333](#)
- preferences
 - crossprobing to place-and-route file [267](#)
 - displaying Hierarchy Browser [267](#)
 - displaying labels [268](#)
 - RTL and Technology views [267](#)
 - sheet size (UI) [267](#)
- primitives
 - pin name display [304](#)
 - pushing into with mouse stroke [272](#)
 - viewing internal hierarchy [300](#)
- probes
 - adding in source code [227](#)
 - definition [227](#)
 - retiming [201](#)
- Product of Sums interface. *See* POS interface
- project command
 - archiving projects [153](#)
 - copying projects [160](#)
 - unarchiving projects [157](#)
- project file hierarchy [120](#)
- project files
 - adding files [116](#)
 - adding source files [112](#)
 - batch mode [468](#)
 - creating [112](#)
 - definition [112](#)
 - deleting files from [116](#)
 - opening [115](#)
 - replacing files in [116](#)
 - updating include paths [119](#)
 - VHDL file order [115](#)
 - VHDL library [114](#)
- projects
 - archiving [153](#)
 - copying [160](#)
 - restoring archives [157](#)
- properties
 - displaying with tooltip [258](#)
 - finding objects with Tcl find -filter [74](#)
 - orig_inst_of [262](#)
 - reporting for collections [89](#)
 - viewing for individual objects [259](#)
- prototyping
 - using hyper source threading [426](#)
- Push/Pop mode
 - HDL Analyst [271](#)
 - keyboard shortcut [273](#)
 - using [271](#), [273](#)

Q

- question mark wildcard, Find command [283](#)

R

- RAMs
 - compiling with SYNCore [378](#)

- initializing [186](#)
 - SYNCore [378](#)
 - SYNCore, byte-enable [386](#)
- RAMs, inferring [180](#)
 - advantages [180](#)
- register balancing. *See* retiming
- register constraints, setting [98](#)
- registers
 - false path constraint [73](#), [108](#)
- Registers panel
 - using SCOPE [58](#)
- replication
 - controlling [211](#)
- resource sharing
 - optimization technique [193](#)
 - overriding option with syn_sharing [213](#)
 - results example [213](#)
 - using [213](#)
- resynthesis
 - compile points [450](#)
 - forcing with Resynthesize All [450](#)
 - forcing with Update Compile Point Timing Data [450](#)
- retiming
 - effect on attributes and constraints [200](#)
 - example [198](#)
 - overview [196](#)
 - probes [201](#)
 - report [199](#)
 - simulation behavior [201](#)
- return codes [468](#)
- rom.info file [273](#)
- ROMs
 - SYNCore [392](#)
 - viewing data table [273](#)
- RTL view
 - See also* HDL Analyst
 - analyzing clock trees [325](#)
 - crossprobing collection objects [84](#)
 - crossprobing description [291](#)
 - crossprobing from [292](#)
 - crossprobing from Text Editor [294](#)
 - defined [257](#)
 - description [256](#)
 - filtering [303](#)
 - finding objects with Find [280](#)

- finding objects with Hierarchy Browser [278](#)
- flattening hierarchy [311](#)
- highlighting collections [89](#)
- opening [258](#)
- selecting/deselecting objects [263](#)
- setting preferences [267](#)
- state machine implementation [222](#)
- traversing hierarchy [270](#)

S

- schematics
 - multisheet. *See* multisheet schematics
 - page size [267](#)
 - selecting/deselecting objects [263](#)
- SCOPE
 - adding attributes [146](#)
 - adding probe insertion attribute [228](#)
 - collections compared to Tcl script window [82](#)
 - creating compile-point constraint file [458](#)
 - defining compile points [454](#)
 - drag and drop [66](#)
 - editing operations [67](#)
 - I/O pad type [62](#)
 - multicycle paths [72](#)
 - Registers panel [58](#)
 - setting compile point constraints [458](#)
 - setting constraints (FDC) [52](#)
 - state machine attributes [177](#)
- scope of the document [21](#)
- search
 - browsing objects with the Find command [280](#)
 - browsing with the Hierarchy Browser [278](#)
 - finding objects on current sheet [280](#)
 - setting limit for results [282](#)
 - setting scope [281](#)
 - using the Find command in HDL Analyst views [280](#)
- See also* search
- set command
 - collections [90](#)
- set_option command [131](#)
- sheet connectors
 - navigating with [266](#)

- sheet size
 - setting number of objects 267
- Shift-F3 key
 - Message Viewer 245
- Show Cell Interior option 300
- Show Context command
 - different from Expand 302
 - using 302
- signals
 - threading with hyper source. *See* hyper source
- simulation, effect of retiming 201
- single-port RAMs
 - SYNCore parameters 382
- slack 328
 - setting margins 325
- slack time display 322
- Slow property 260
- source code
 - commenting with synthesis on/off 142
 - crossprobing from Tcl window 297
 - defining FSMs 175
 - fixing errors 37
 - opening automatically to crossprobe 293
 - optimizing 192
 - when to use for constraints 46
- source files
 - See also* Verilog, VHDL.
 - adding comments 36
 - adding files 112
 - checking 33
 - column editing 36
 - copying examples from PDF 36
 - creating 30
 - crossprobing 294
 - editing 35
 - editing operations 35
 - mixed language 41
 - specifying default encoding style 141
 - specifying top level file for mixed language projects 42
 - specifying top level in Project view 115
 - specifying top-level file 141
 - state machine attributes 177
 - using bookmarks 36
- special characters
 - Tcl collections 88
- STA 329
- STA, generating custom timing reports 329
- STA, using analysis design constraints (adc) 332
- stand-alone timing analyst. *See* STA
- state machines
 - See also* FSM Compiler, FSM Explorer, FSM viewer, FSMs.
 - attributes 177
 - descriptions in log file 221
 - implementation 222
 - parameter and 'define comparison 176
- statemachine.info file 319
- subtractors
 - SYNCore 397
- syn_allow_retiming
 - compile points 460
 - using for retiming 197
- syn_allowed_resources
 - compile points 460
- syn_encoding attribute 178
- syn_enum_encoding directive
 - FSM encoding 178
- syn_hier attribute
 - controlling flattening 207
 - preserving hierarchy 207
 - using with compile points 459
- syn_isclock
 - black box clock pins 174
- syn_keep
 - replicating redundant logic 204
- syn_keep attribute
 - preserving nets 203
 - preserving shared registers 203
- syn_keep directive
 - effect on buffering 211
- syn_macro
 - specifying encrypted IP as white box 421
- syn_maxfan attribute
 - setting fanout limits 209
- syn_noarrayports attribute
 - use with alsin 353
- syn_noprune directive
 - preserving instances 203

- syn_preserve
 - effect on buffering [211](#)
 - preserving power-on for retiming [197](#)
- syn_preserve directive
 - preserving FSMs from optimization [177](#)
 - preserving logic [203](#)
- syn_probe attribute [227](#)
 - inserting probes [227](#)
 - preserving nets [203](#)
- syn_ramstyle attribute
 - multi-port RAM inference [184](#)
- syn_reference_clock
 - defining non-clock signal frequencies [102](#)
- syn_reference_clock constraint [93](#)
- syn_replicate attribute
 - using buffering [212](#)
- syn_sharing directive
 - overriding default [213](#)
- syn_state_machine directive
 - using with value=0 [222](#)
- SYN_TCL_HOOKS environment variable [481](#)
- syn_tco attribute
 - adding in SCOPE [173](#)
- syn_tco directive [172](#)
 - adding black box constraints [171](#)
- syn_tpd attribute
 - adding in SCOPE [173](#)
- syn_tpd directive [172](#)
 - adding black box constraints [171](#)
- syn_tsu attribute
 - adding in SCOPE [173](#)
- syn_tsu directive [172](#)
 - adding black box constraints [171](#)
- syn_useioff
 - preventing flops from moving during retiming [198](#)
- SYNCore
 - adders [397](#)
 - counters [404](#)
 - FIFO compiler [372](#)
 - RAMs [378](#)
 - RAMs, byte-enable [386](#)
 - RAMs, dual-port parameters [383](#)
 - RAMs, single-port parameters [382](#)

- ROMs [392](#)
- ROMs, parameters [396](#)
- subtractors [397](#)
- synhooks
 - automating message filtering [249](#)
- synhooks.tcl file [481](#)
- Synopsys
 - FPGA product family [16](#)
- Synplify Pro synthesis tool
 - overview [16](#)
- synplify_pro UNIX command [22](#)
- syntax
 - checking source files [33](#)
- syntax check [34](#)
- synthesis check [34](#)
- synthesis_on/off
 - using [142](#)
- SystemVerilog keywords
 - context help [32](#)

T

- Tcl
 - max_parallel_jobs variable [476](#)
- tcl callbacks
 - customizing key assignments [482](#)
- Tcl commands
 - batch script [469](#)
 - entering in SCOPE [99](#)
 - running [474](#)
- Tcl expand
 - using [74](#)
- Tcl expand command
 - crossprobing objects [84](#)
 - usage tips [78](#)
 - using in SCOPE [83](#)
- Tcl files [474](#)
 - creating [476](#)
 - for bottom-up synthesis [480](#)
 - guidelines [48](#)
 - naming conventions [48](#)
 - recording from commands [475](#)
 - synhooks.tcl [481](#)
 - using variables [478](#)
 - wildcards [49](#)
- Tcl find

- batch mode [80](#)
 - filtering results by property [74](#)
 - search patterns [74](#)
 - using [74](#)
 - Tcl find command
 - annotating properties [74](#)
 - crossprobing objects [84](#)
 - database differences [84](#)
 - Tcl window vs SCOPE [82](#)
 - usage tips [76](#)
 - useful -filter examples [76](#)
 - using in SCOPE [83](#)
 - Tcl Script window
 - crossprobing [297](#)
 - message viewer [244](#)
 - Tcl script window
 - collections compared to SCOPE [82](#)
 - Tcl scripts
 - See Tcl files.
 - TCL View [62](#)
 - Technology view
 - See also HDL Analyst
 - critical paths [325](#)
 - crossprobing [291](#), [292](#)
 - crossprobing collection objects [84](#)
 - crossprobing from source file [294](#)
 - filtering [303](#)
 - finding objects [282](#)
 - finding objects with Find [280](#)
 - finding objects with Hierarchy Browser [278](#)
 - flattening hierarchy [311](#)
 - general description [256](#)
 - highlighting collections [89](#)
 - opening [258](#)
 - selecting/deselecting objects [263](#)
 - setting preferences [267](#)
 - state machine implementation in [222](#)
 - traversing hierarchy [270](#)
 - text editor
 - built-in [34](#)
 - external [40](#)
 - using [34](#)
 - Text Editor view
 - crossprobing [292](#)
 - Text Editor window
 - colors [38](#)
 - crossprobing [37](#)
 - fonts [38](#)
 - text files
 - crossprobing [294](#)
 - The Synopsys FPGA Product Family [16](#)
 - through constraints [69](#)
 - AND lists [70](#)
 - OR lists [69](#)
 - time stamp, checking on files [117](#)
 - timing analysis [322](#)
 - timing analysis using STA [329](#)
 - timing budgeting
 - compile points [444](#)
 - timing constraints [47](#), [93](#)
 - timing exceptions, adding constraints after synthesis [333](#)
 - timing exceptions, modifying with adc [333](#)
 - timing failures [328](#)
 - timing information commands [322](#)
 - timing information in HDL views [323](#)
 - timing information, critical paths [326](#)
 - timing optimization [194](#)
 - timing reports
 - specifying format options [137](#)
 - timing reports, custom [329](#)
 - tips
 - memory usage [315](#)
 - to constraints
 - specifying [69](#)
 - top level
 - specifying [140](#)
 - top-down design flow
 - compile point advantages [432](#)
 - transparent instances
 - flattening [311](#)
 - lower-level logic on multiple sheets [265](#)
- ## U
- UNIX commands
 - synplify_pro [22](#)
 - up-to-date checking [232](#)
 - copying job logs to log file [234](#)
 - limitations [236](#)

V

- vendor-specific netlists [354](#)
- Verilog
 - 'define statements [139](#)
 - adding attributes and directives [145](#)
 - adding probes [227](#)
 - black boxes [166](#)
 - black boxes, instantiating [166](#)
 - checking source files [33](#)
 - choosing a compiler [138](#)
 - creating source files [30](#)
 - crossprobing from HDL Analyst view [292](#)
 - defining FSMs [175](#)
 - defining state machines with parameter and 'define [176](#)
 - editing operations [35](#)
 - extracting parameters [138](#)
 - include paths, updating [119](#)
 - initializing RAMs [186](#)
 - Microsemi ACTgen macros [347](#)
 - mixed language files [41](#)
 - RAM structures for inference [181](#)
 - specifying compiler directives [139](#)
 - specifying top-level module [140](#)
- Verilog 2001
 - setting global option from the Project view [138](#)
 - setting option per file [138](#)
- Verilog macro libraries
 - Microsemi [346](#)
- Verilog model (.vmd) [444](#)
- VHDL
 - adding attributes and directives [143](#)
 - adding probes [227](#)
 - black boxes [168](#)
 - black boxes, instantiating [168](#)
 - checking source file [33](#)
 - constants [141](#)
 - creating source files [30](#)
 - crossprobing from HDL Analyst view [292](#)
 - defining FSMs [176](#)
 - editing operations [35](#)
 - extracting generics [141](#)
 - file order in mixed designs [44](#)
 - global signals in mixed designs [44](#)
 - initializing RAMs with variable declarations [189](#)
 - initializing with signal declarations [187](#)
 - macro libraries, Microsemi [346](#)
 - mixed language files [41](#)
 - RAM structures for inference [181](#)
 - specifying top-level entity [140](#)
- VHDL files
 - adding library [114](#)
 - adding third-party package library [114](#)
 - order in project file [115](#)
 - ordering automatically [115](#)
- vi text editor [40](#)
- virtual clock, setting [98](#)

W

- warning messages
 - definition [34](#)
- warnings
 - feedback muxes [195](#)
 - filtering [245](#)
 - handling [254](#)
 - sorting [245](#)
- Watch window [242](#)
 - moving [242](#), [244](#)
 - multiple implementations [127](#)
 - resizing [242](#), [244](#)
- wildcards
 - effect of search scope [284](#)
 - message filter [247](#)
- wildcards (Find)
 - examples [285](#)
 - how they work [283](#)