Synplicity®

Simply Better Results

Synplify® for Actel

Synplify Reference Manual

September 2004

Synplicity, Inc.
600 West California Avenue
Sunnyvale, CA 94086, USA
(U.S.) +1 408 215-6000 direct
(U.S.) +1 408 990-0263 fax
www.synplicity.com

Disclaimer of Warranty

Synplicity, Inc. makes no representations or warranties, either expressed or implied, by or with respect to anything in this manual, and shall not be liable for any implied warranties of merchantability or fitness for a particular purpose of for any indirect, special or consequential damages.

Copyright Notice

Copyright © 1994-2004 Synplicity, Inc. All Rights Reserved.

Synplicity software products contain certain confidential information of Synplicity, Inc. Use of this copyright notice is precautionary and does not imply publication or disclosure. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language in any form by any means without the prior written permission of Synplicity, Inc. While every precaution has been taken in the preparation of this book, Synplicity, Inc. assumes no responsibility for errors or omissions. This publication and the features described herein are subject to change without notice.

Trademarks

Synplicity, the Synplicity "S" logo, Amplify, Certify, Certify SC, Synplify ASIC, Synplify, Synplify Lite, Synplify Pro, RealPower, HDL Analyst, SCOPE, Behavior Extracting Synthesis Technology, Embedded Synthesis, Simply Better Synthesis, Simply Better Results, and Synthesis Constraint Optimization Environment are registered trademarks, and BEST, DST, Direct Synthesis Technology, MultiPoint, Partition-Driven Synthesis, PowerPlanner, Physical Optimizer, and Total Optimization Physical Synthesis are trademarks of Synplicity, Inc. Verilog is a registered trademark of Cadence Design Systems, Inc. IBM and PC are registered trademarks of International Business Machines Corporation. Microsoft is a registered trademark of Microsoft Corporation. Sun, SPARC, Solaris, and SunOS are trademarks of Sun Microsystems, Inc. UNIX is a registered trademark of X/Open Corporation.

All other product names mentioned herein are the trademarks or registered trademarks of their respective owners.

Restricted Rights Legend

Government Users: Use, reproduction, release, modification, or disclosure of this commercial computer software, or of any related documentation of any kind, is restricted in accordance with FAR 12.212 and DFARS 227.7202, and further restricted by the Synplicity Software License Agreement. Synplicity, Inc., 600 West California Avenue, Sunnyvale, CA 94086, U. S. A.

Printed in the U.S.A
September 2004

Synplicity Software License Agreement

Important! READ CAREFULLY BEFORE PROCEEDING

BY INDICATING YOUR ACCEPTANCE OF THE TERMS OF THIS AGREEMENT, YOU ARE REPRESENTING THAT YOU HAVE THE RIGHT AND AUTHORITY TO LEGALLY BIND YOURSELF OR YOUR COMPANY, AS APPLICABLE, AND CONSENTING TO BE LEGALLY BOUND BY ALL OF THE TERMS OF THIS AGREEMENT. IF YOU DO NOT AGREE TO ALL THESE TERMS DO NOT INSTALL OR USE THE SOFTWARE, AND RETURN THE SOFTWARE TO THE LOCATION OF PURCHASE FOR A REFUND. This is a legal agreement governing use of the software program ("SOFTWARE") provided to you ("Licensee") by Synplicity. The term "SOFTWARE" also includes related documentation (whether in print or electronic form) and any updates or upgrades of the SOFTWARE provided by Synplicity, but does not include certain software licensed by third party licensors and made available to you by Synplicity under the terms of such third party licensor's license (including software licensed under the General Public License (GPL)). If Licensee is a participant in the University Program or has been granted an Evaluation License, then some of the following terms and conditions may not apply (refer to the sections entitled, respectively, **Evaluation License** and **University Program**, below).

Evaluation License. If Licensee has obtained the SOFTWARE pursuant to an evaluation license, then, in addition to all other terms and conditions, the following restrictions apply: (a) The license to the SOFTWARE terminates after 20 days (unless otherwise agreed to in writing by Synplicity); and (b) Licensee may use the SOFTWARE only for the sole purpose of internal testing and evaluation to determine whether Licensee wishes to license the SOFTWARE on a commercial basis. Licensee shall not use the SOFTWARE to design any integrated circuits for production or pre-production purposes or any other commercial use including, but not limited to, for the benefit of Licensee's customers. If Licensee breaches any of the foregoing restrictions, then Licensee shall pay to Synplicity a license fee equal to Synplicity's standard license fee for the commercial version of the SOFTWARE.

License. Synplicity grants to Licensee a non-exclusive right to install the SOFTWARE and to use or authorize use of the SOFTWARE by up to the number of nodes for which Licensee has a license and for which Licensee has the security key(s) or authorization code(s) provided by Synplicity or its agents. If Licensee has obtained the SOFTWARE under a node-locked license, then a "node" refers to a specific machine, and the SOFTWARE may be installed only on the number of "nodes" or machines authorized, must be used only on the machine(s) on which it is installed, and may be accessed only by users who are physically present at that node or machine. A node-locked license may only be used by one user at a time running one instance of the software at a time. If Licensee has obtained the SOFTWARE under a "floating" license, then a "node" refers to a concurrent user or session, and the SOFTWARE may be used concurrently by up to the number of users or sessions indicated. All SOFTWARE must be used within the country for which the systems were licensed and at Licensee's Site (contained within a one kilometer radius); however, if Licensee has a floating license then remote use is permitted by employees who work at the site but are temporarily telecommuting to that same site from less than 50 miles away (for example, an employee who works at a home office on occasion), but the maximum number of concurrent sessions or nodes still applies. In addition, Synplicity grants to Licensee a non-exclusive license to copy and distribute internally the documentation portion of the SOFTWARE in support of its license to use the program portion of the SOFTWARE. For purposes of this Agreement the "Licensee's Site" means the location of the server on which the SOFTWARE resides, or when a server is not required, the location of the client computer for which the license was issued.

Copy Restrictions. This SOFTWARE is protected by United States copyright laws and international treaty provisions and Licensee may copy the SOFTWARE only as follows: (i) to directly support authorized use under the license, and (ii) in order to make a copy of the SOFTWARE for backup purposes. Copies must include all copyright and trademark notices.

Use Restrictions. This SOFTWARE is licensed to Licensee for internal use only. Licensee shall not (and shall not allow any third party to): (i) decompile, disassemble, reverse engineer or attempt to reconstruct, identify or discover any source code, underlying ideas, underlying user interface techniques or algorithms of the SOFTWARE by any means whatever, or disclose any of the foregoing; (ii) provide, lease, lend, or use the SOFTWARE for timesharing or service bureau purposes, on an application service provider basis, or otherwise circumvent the internal use restrictions; (iii) modify, incorporate into or with other software, or create a derivative work of any part of the SOFTWARE; (iv) disclose the results of any benchmarking of the SOFTWARE, or use such results for its own competing software development activities, without the prior written permission of Synplicity; or (v) attempt to circumvent any user limits, maximum gate count limits or other license, timing or use restrictions that are built into the SOFTWARE.

Transfer Restrictions/No Assignment. The SOFTWARE may only be used under this license at the designated locations and designated equipment as set forth in the license grant above, and may not be moved to other locations or equipment or otherwise transferred without the prior written consent of Synplicity. Any permitted transfer of the SOFTWARE will require that Licensee executes a "Software Authorization Transfer Agreement" provided by Synplicity. Further, Licensee shall not sublicense, or assign this Agreement or any of the rights or licenses granted under this Agreement, without the prior written consent of Synplicity.

Security. Licensee agrees to take all appropriate measures to safeguard the SOFTWARE and prevent unauthorized access or use thereof, including without limitation: (i) implementation of firewalls and other security applications, (ii) use of FLEXlm options file that restricts access to the SOFTWARE to identified users; (iii) maintaining and storing license information in paper format only; (iv) changing TCP port numbers every three (3) months; and (v) communicating to all authorized users that use of the SOFTWARE is subject to the restrictions set forth in this Agreement.

Ownership of the SOFTWARE. Synplicity retains all right, title, and interest in the SOFTWARE (including all copies), and all worldwide intellectual property rights therein. Synplicity reserves all rights not expressly granted to Licensee. This License is not a sale of the original SOFTWARE or of any copy.

Ownership of Design Techniques. "Design" means the representation of an electronic circuit or device(s), derived or created by Licensee through the use of the SOFTWARE in its various formats, including, but not limited to, equations, truth tables, schematic diagrams, textual descriptions, hardware description languages, and netlists. "Design Techniques" means the data, circuit and logic elements, libraries, algorithms, search strategies, rule bases, and technical information incorporated in the SOFTWARE and employed in the process of creating Designs. Synplicity retains all right, title and interest in and to Design Techniques incorporated into the SOFTWARE, including all intellectual property rights embodied therein. Licensee acknowledges that Synplicity has an unrestricted, royalty-free right to incorporate any Design Techniques disclosed by Licensee into its software, documentation and other products, and to sublicense third parties to use those incorporated design techniques.

Termination. Synplicity may terminate this Agreement immediately if Licensee breaches any provision, including without limitation, failure by Licensee to implement the Security measures set forth above. Upon notice of termination by Synplicity, all rights granted to Licensee under this Agreement will immediately terminate, and Licensee shall cease using the SOFTWARE and return or destroy all copies (and partial copies) of the SOFTWARE and documentation.

Limited Warranty and Disclaimer. Synplicity warrants that the program portion of the SOFTWARE will perform substantially in accordance with the accompanying documentation for a period of 90 days from the date of receipt. Synplicity's entire liability and Licensee's exclusive remedy for a breach of the preceding limited warranty shall be, at Synplicity's option, either (a) return of the license fee, or (b) providing a fix, patch, workaround, or replacement of the SOFTWARE that does not meet such limited warranty. In either case, Licensee must return the SOFTWARE to Synplicity with a copy of the purchase receipt or similar document. Replacements are warranted for the remainder of the original warranty period or 30 days, whichever is longer. Some states/jurisdictions do not allow limitations, so the above limitation may not apply. EXCEPT AS EXPRESSLY SET FORTH ABOVE, NO OTHER WARRANTIES OR CONDITIONS, EITHER EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, ARE MADE BY SYNPLICITY OR ITS LICENSORS WITH RESPECT TO THE SOFTWARE AND THE ACCOMPANYING DOCUMENTATION, AND SYNPLICITY EXPRESSLY DISCLAIMS ALL WARRANTIES AND CONDITIONS NOT EXPRESSLY STATED HEREIN, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE. SYNPLICITY AND ITS LICENSORS DO NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE SOFTWARE WILL MEET LICENSEE'S REQUIREMENTS, BE UNINTERRUPTED OR ERROR FREE, OR THAT ALL DEFECTS IN THE PROGRAM WILL BE CORRECTED. Licensee assumes the entire risk as to the results and performance of the SOFTWARE. Some states/jurisdictions do not allow the exclusion of implied warranties, so the above exclusion may not apply.

Limitation of Liability. IN NO EVENT SHALL SYNPLICITY OR ITS LICENSORS OR THEIR AGENTS BE LIABLE FOR ANY INDIRECT, SPECIAL, CONSEQUENTIAL OR INCIDENTAL DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTIONS, LOSS OF BUSINESS INFORMATION, OR OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OF OR INABILITY TO USE THE SOFTWARE, EVEN IF SYNPLICITY AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. FURTHER, IN NO EVENT SHALL SYNPLICITY'S LICENSORS BE LIABLE FOR ANY DIRECT DAMAGES ARISING OUT OF LICENSEE'S USE OF THE SOFTWARE. In no event will Synplicity or its licensors be liable to Licensee for damages in an amount greater than the fees paid for the use of the SOFTWARE. Some states/jurisdictions do not allow the limitation or exclusion of incidental or consequential damages, so the above limitations or exclusions may not apply.

Intellectual Property Right Infringement. If a claim alleging infringement of an intellectual property right arises concerning the SOFTWARE (including but not limited to patent, trade secret, copyright or trademark rights), Synplicity in its sole discretion may elect to defend or settle such claim, and/or terminate this Agreement and all rights to use the SOFTWARE, and require the return or destruction of the SOFTWARE, with a refund of the fees paid for use of the SOFTWARE less a reasonable allowance for use and shipping.

Export. Licensee warrants that it is not prohibited from receiving the SOFTWARE under U.S. export laws; that it is not a national of a country subject to U.S. trade sanctions; that it will not use the SOFTWARE in a location that is the subject of U.S. trade sanctions that would cover the SOFTWARE; and that to its knowledge it is not on the U.S. Department of Commerce's table of deny orders or otherwise prohibited from obtaining goods of this sort from the United States.

Miscellaneous. This Agreement is the entire agreement between Licensee and Synplicity with respect to the license to the SOFTWARE, and supersedes any previous oral or written communications or documents (including, if you are obtaining an update, any agreement that may have been included with the initial version of the SOFTWARE). This Agreement is governed by the laws of the State of California, USA excluding its conflicts of laws principals. This Agreement will not be governed by the U. N. Convention on Contracts for the International Sale of Goods and will not be governed by any statute based on or derived from the Uniform Computer Information Transactions Act (UCITA). If any provision, or portion thereof, of this Agreement is found to be invalid or unenforceable, it will be enforced to the extent permissible and the remainder of this Agreement will remain in full force and effect. Failure to prosecute a party's rights with respect to a default hereunder will not constitute a waiver of the right to enforce rights with respect to the same or any other breach.

Government Users. The Software contains commercial computer software and commercial computer software documentation. In accordance with FAR 12.212 and DFARS 227.7202, use, duplication or disclosure is subject to restrictions under paragraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at 252.227-7013, and further restricted by this Agreement. Synplicity, Inc., 600 W. California Avenue, Sunnyvale, CA 94086, U. S. A.

University Program. The following section applies only if Licensee is a participant in Synplicity's University Program; it does not replace the remainder of the Agreement and supersedes only those terms that directly conflict.

University Program: License. Subject to the terms and conditions of this Agreement, Synplicity hereby grants to Licensee (a University) for the License Term (defined below), a non-exclusive license, only for purposes of course work or teaching in connection with a university-sponsored class, or for academic research either sponsored by or conducted under the auspices of Licensee, to (a) install and use the SOFTWARE, and (b) reproduce and distribute copies of the documentation included in the SOFTWARE subject only to payment for those copies (which may be based on the number of users, the number and type of copies, or both). If the SOFTWARE is licensed pursuant to a node-locked license, then the Licensee may install and use the SOFTWARE on the authorized workstations. If the SOFTWARE is licensed pursuant to a floating license, then the Licensee may install the SOFTWARE on the authorized server and use the SOFTWARE on up to the number of nodes for which Licensee has paid license fees and Synplicity has granted authorization.

University Program: License Term and Termination. For purposes of the University Program, "License Term" means one year unless otherwise agreed to in writing. This Agreement will terminate at the end of the License Term, unless earlier terminated in accordance with this Agreement.

University Program: License Restrictions. As Licensee, University may not (i) allow access to the SOFTWARE by any user not registered for a course or participating in an academic research project for which use of the SOFTWARE has been authorized; (ii) use the SOFTWARE to design any commercial products; or (iii) disclose the results of any benchmarking of the SOFTWARE, or use such results for its own competing software development activities, without the prior written permission of Synplicity.

University Program: Technical Liaison. Licensee shall appoint a Technical Liaison who will serve as the single point of contact between Synplicity and Licensee with respect to the subject matter of this Agreement. The Technical Liaison will coordinate installation and maintenance of the SOFTWARE, communicate with Synplicity regarding license procedures, administer Licensee's obligations under this Agreement and respond to inquiries by Synplicity related to the subject matter of this Agreement.

University Program: Technical Support in North America. Unless otherwise agreed in writing, Synplicity will accept calls only from the appointed Technical Liaison. No technical support will be provided other than calls from the Technical Liaison relating to installation of the SOFTWARE. SOFTWARE upgrades may be obtained from the Synplicity Web Site.

University Program: International Technical Support. Technical support is provided through Synplicity's authorized distributors in accordance with their applicable policies.

revised 7/03

Contents

Chapter 7: Timing Constraints

CHAPTER 1

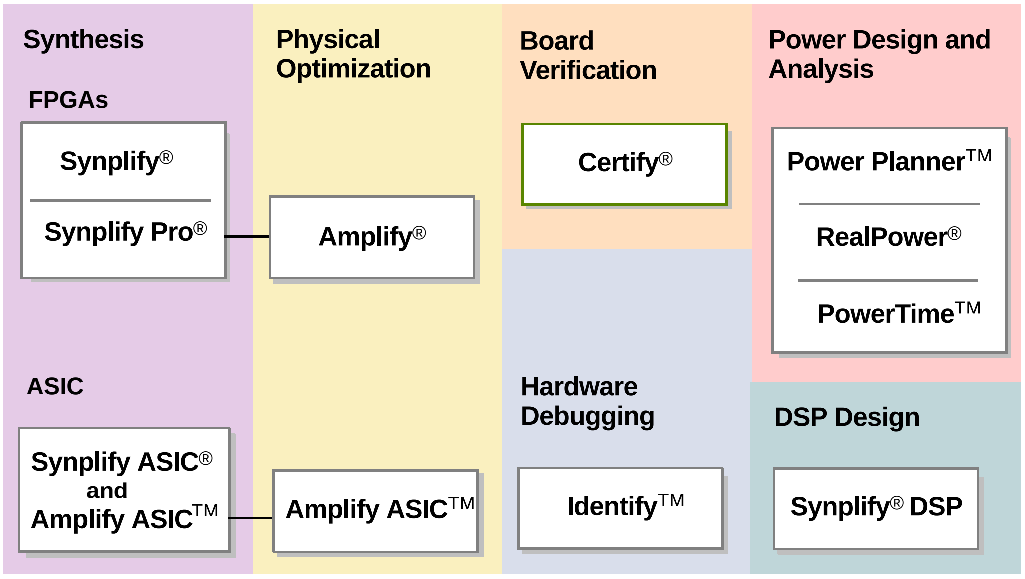
Product Overview

This reference manual is part of a document set that also includes the *Synplify User Guide*. The reference manual details the synthesis tool user interface, commands, and features. The *User Guide* contains "how-to" information, emphasizing user tasks, procedures, design flows, and results analysis. Tutorials are provided to help you become familiar with the tool and includes extended examples.

This chapter provides an introduction to the Synplicity product family.

The Synplicity Product Family

Synplicity products are based on core logic synthesis technology, and share a common look and feel.



- The Synplify and Synplify Pro products are logic synthesis tools for FPGAs (field programmable gate arrays) and CPLDs (complex programmable logic devices). The Synplify Pro tool is an advanced version of the Synplify tool, with many additional features for managing and optimizing complex FPGAs.

  These tools accept high-level input written in industry-standard hardware description languages (Verilog and VHDL) and, using the Synplicity Behavior Extracting Synthesis Technology® (BEST™) algorithms, convert the designs into small, high performance design netlists for popular technology vendors. They can also write VHDL and Verilog netlists after synthesis, which you can then simulate in order to verify functionality.

- The Amplify® Physical Optimizer™ product is the first *physical* synthesis tool for FPGAs, and is a separately licensed option to the Synplify Pro product. You use it to interactively assign physical constraints to a design, by dragging and dropping RTL objects into regions of the design. With this constraint information, the physical optimizer can derive more accurate timing estimates, and use them to perform additional optimizations, producing a more highly optimized circuit in fewer iterations. The Amplify product can also be used in a fully automated flow for an incremental performance improvement.
- The Synplify ASIC and Amplify ASIC software for ASIC designers leverages Synplicity's proven synthesis technology, and delivers high quality of results (QoR) and order-of-magnitude runtime improvements -- up to 15 times faster than traditional synthesis products. This solution offers ASIC designers the ability to perform timing-driven synthesis on any size of a design in a single operation. Because the entire design is optimized in a single operation, with the ASIC software, users typically find significantly better area and power, as well as improved timing performance. The ASIC software includes the unique MultiPoint™ design technology, which delivers a highly memory efficient design methodology for optimizing across and within partitioned boundaries for large ASIC designs. By using MultiPoint synthesis technology, ASIC designers can achieve better QoR and significantly faster runtimes than other synthesis methodologies.

  The Amplify ASIC software is also a physical synthesis tool that assigns physical placement to design objects. Using physical information, the tool can correlate between front and back end environments and provide more accurate estimates than can be achieved through wire load models. Amplify ASIC physical synthesis can floorplan, place, and optimize a design based on timing and physical constraints and technology libraries.
- The Certify® product is an electronic design automation (EDA) tool for system-design, system-on-a-chip (SoC), and ASIC teams that require hardware prototypes of their project early in the design phase. It produces a functional ASIC prototype – partitioned among multiple FPGAs – from RTL code. The Certify tool includes many unique advantages for prototyping, including a feature that combines the gated clocks, which are characteristic of ASIC designs, onto the global clock resources of the FPGAs. Unlike traditional ASIC prototyping techniques, the Certify tool enables prototype delivery before ASIC synthesis to allow a design team to recognize and correct problems earlier in the design flow.

- The Fortify™ family of power solutions is designed to help the ASIC designer manage the power network through every step of the design flow, from concept to GDSII. With every new technology generation, good power grid design becomes even more critical. While the IR drop has a significant impact on the timing delays of the circuit, the increasing impact of electromigration brings serious reliability issues to the forefront.
  - Fortify PowerPlanner™ tool is the only real solution on the market today that permits the ASIC designer to experiment with various power topologies and gain an early insight into the effect power distribution has on the design. Up front power planning has been proven to correlate very well to back-end results, thereby saving the designer costly iterations late in the design flow.
  - Fortify RealPower® tool is the industry's most accurate and complete back-end power solution. It computes the IR drop and performs electromigration analysis on the individual standard cell level using its embedded Dynamic Power Calculator. The result is a lightning fast power engine that can handle multi-million gate designs. Furthermore, with its built-in PowerRoute™ technology, the RealPower software delivers a final, correct by construction power grid that can relieve the designer of having to rely on external, commercially available routers.
- The Identify™ RTL Debugger product helps you debug any design that is implemented by an electronically programmable logic device such as an FPGA or PLD. With the Identify RTL Debugger, you are able to debug live hardware within the target system at target speed with the internal design visibility you need while using intuitive debugging techniques.

  The Identify RTL Debugger is a dual-component system that allows you to probe your HDL design directly in the target environment. The system consists of the Identify Instrumentor software tool that allows you to select your design instrumentation at the HDL level and then create an on-chip hardware probe, and the Identify Debugger software tool that interacts with the on-chip hardware probe and allows you to perform live debugging of your design. The combined system fits easily into your existing design flow with only minor modifications and allows you to debug designs faster, easier, and more efficiently.
- The Synplify® DSP product is a high-level tool for hardware DSP design. It is an add-on to the Simulink® product from The MathWorks®, and provides the designer with an automated path from high-level design and simulation to an optimized, synthesizable, technology-independent,

system-level HDL implementation. This tool provides performance and productivity benefits for designers who are implementing DSP circuits and targeting programmable logic devices. The software achieves significantly higher performance than alternative solutions and provides the designer with a mechanism to make high-level area/performance tradeoffs. The output is synthesizable HDL code ready for use with the Synplicity® Synplify Pro® or Synplify ASIC® software.

Overview of the Synplify Synthesis Tool

This section introduces the technology underlying the synthesis tool, its main features, and the general appearance of its user interface.

Product Audience

The Synplicity products are targeted to FPGA and ASIC system developers. Familiarity with the following is helpful for using the tools:

- Design synthesis
- Register transfer level (RTL) code describing the detailed behavior of a design
- FPGAs or CPLDs
- Window-based user interfaces

Synthesis Tool Features

The Synplify synthesis tool has the following built-in features:

- The HDL Analyst® RTL analysis and debugging environment, a graphical tool for analysis and crossprobing. See Result Analysis, on page 4-1 in the *Synplify User Guide*.
- The Text Editor window, with a language-sensitive editor for writing and editing HDL code. See *Text Editor View*, on page 2-10.
- The SCOPE® (Synthesis Constraint Optimization Environment®) tool, which provides a spreadsheet-like interface for managing timing constraints and design attributes. See *The SCOPE Window*, on page 7-4.
- FSM Compiler, a symbolic compiler that performs advanced finite state machine (FSM) optimizations. See *FSM Compiler*, on page 2-13.
- Other special windows, or *views*, for analyzing your design.

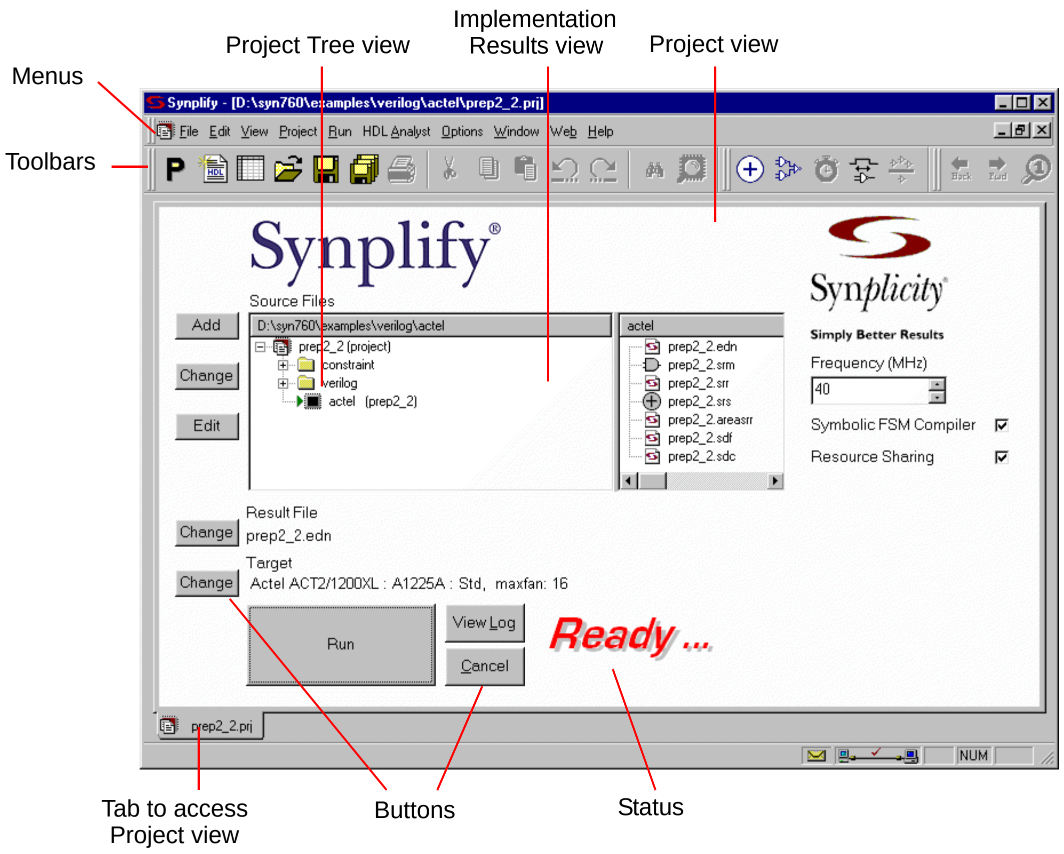
BEST Algorithms

The Behavior Extraction Synthesis Technology (BEST^{TM}) feature is the underlying proprietary technology that the Synplify synthesis tool uses to extract and implement your design structures.

During synthesis, the BEST algorithms recognize high-level abstract structures like RAMs, ROMs, finite state machines (FSMs), and arithmetic operators, and maintain them, instead of converting the design entirely to the gate level. The BEST algorithms automatically map these high-level structures to technology-specific resources using module generators. For example, the algorithms map RAMs to target-specific RAMs, and adders to carry chains. The BEST algorithms also optimize hierarchy automatically.

Graphic User Interface

With the exception of the Synplify tool, the Synplicity family of products share a common graphical user interface (GUI), in order to ensure a cohesive look and feel across the different products. In the Synplify Pro tool, you can choose whether to use this common GUI or to run the tool in Classic mode, which uses the basic Synplify tool GUI (plus the Tcl window and Log Watch window of the common GUI).


Menus
Toolbars
Project Tree view
Implementation Results view
Project view
Synplify - [D:\syn760\examples\verilog\actel\prep2_2.prj]
File Edit View Project Run HDL Analyst Options Window Web Help
Synplify
Synplicity
Simply Better Results
Source Files
Add
Change
Edit
Run
View Log
Cancel
Ready ...
prep2_2.prj
NUM
Tab to access Project view
Buttons
Status

D:\syn760\examples\verilog\actel
prep2_2 (project)
constraint
verilog
actel (prep2_2)

actel
prep2_2.edn
prep2_2.srm
prep2_2.srr
prep2_2.srs
prep2_2.areasrr
prep2_2.sdf
prep2_2.sdc

Frequency (MHz) 40
Symbolic FSM Compiler ✓
Resource Sharing ✓

Result File
prep2_2.edn

Target
Actel ACT2/1200XL : A1225A : Std, maxfan: 16

Figure 1-1: Synplify User Interface

The following table shows where you can find information about different parts of the GUI, some of which are not shown in the above figure. For more information, see the *User Guide*.

Table 1-1: Graphical User Interface References

For information about...	See...
Project window	*Project View*, on page 2-3
RTL view	*RTL View*, on page 2-4
Technology view	*Technology View*, on page 2-6
Text Editor view	*Text Editor View*, on page 2-10

Table 1-1: Graphical User Interface References (Continued)

For information about...	See...
SCOPE spreadsheet	*The SCOPE Spreadsheet*, on page 7-4
Other views and windows	*Project View*, on page 2-3
Menubar menu commands and their dialog boxes	*Menus*, on page 3-2
Toolbars	*Toolbars*, on page 2-21
Buttons	*Buttons and Options*, on page 3-3
Context-sensitive popup menus and their dialog boxes	*Popup Menus*, on page 3-90
Online help	Use the F1 keyboard shortcut or click the Help button in a dialog box. See *Help Command*, on page 3-84, for more information.

Projects, Implementations

Projects contain information about the synthesis run, including the names of design files, constraint files (if used), and other options you have set. A *project file* (`.prj`) is in Tcl format. It points to all the files you need for synthesis and contains the necessary optimization settings. In the Project view, a project appears as a folder.

An *implementation* is one version (also called a revision) of a project, run with certain parameter or option settings. You can synthesize again, with a different set of options, to get a different implementation. In the Project view, an implementation is shown in the folder of its project. The active implementation has a green arrow next to it. The output files generated for the active implementation are displayed in the Implementation Results view on the right.

A workspace allows you to group related projects together. Although a workspace can contain a set of projects, only one implementation is active at a time. All commands operate on the active project and its implementation. You can open a project independently of the workspace it belongs to, if needed. In the Project view, a workspace is shown as a folder at one level above the project folder.

Starting the Synthesis Tool

This section describes starting the synthesis tool in interactive and batch mode.

Starting the Synthesis Tool in Interactive Mode

Before you can start the Synplify synthesis tool, you must install it and set up the software license appropriately. How you start the tool depends on your environment. For details, see the installation instructions for the tool.

You can start interactive use of the Synplify synthesis tool in any of the following ways:

- To start the synthesis tool from the Microsoft® Windows® operating system, choose

  Start -> Programs -> Synplicity -> Synplify <*version*>

- To start the tool from a DOS command line, specify the executable:

  `synplify_installation_dir\bin\synplify.exe`

  The executable name is the name of the product followed by an `exe` file extension (`.exe`).

- To start the synthesis tool from a UNIX platform, type this command at the UNIX prompt: `synplify`

For information about using the Synplify synthesis tool in batch mode, see *Syntax for the synplify Command*, on page 1-11.

Syntax for the synplify Command

This command lets you run synthesis in batch mode. It starts the Synplify synthesis tool and opens the Project window. It opens the project file or Tcl file, if you specified it.

Syntax

synplify -batch [<*project file*>**]**
[-licensetype <*license feature name*>**]**
[-history <*history log*>**]**
[-compile]
[-log <*log file*>**]**

The following table describes the command options.

Table 1-2: synplify Command Options

Option	Description
-batch	Starts the Synplify synthesis tool in batch mode, without opening the Project window. It opens any project file that you specified.
-licensetype	Lets you specify a license if you work in an environment with multiple Synplicity licenses. Use this in conjunction with the -batch keyword.
-history	Records all Tcl commands and writes them to the specified history log file when the command exits.
-compile	Compiles the design, but does not map it.
-log	Writes all output to the specified log file

Using the Synplify Synthesis Tool

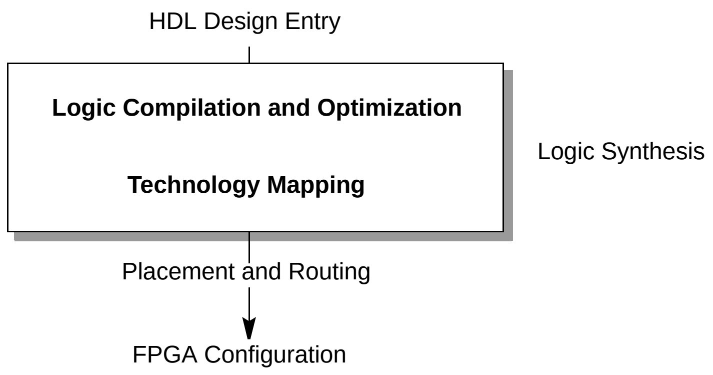
This section provides an overview of how you use the synthesis tool.

Synthesis Software Flow

When you run the Synplifysynthesis tool, it performs *logic synthesis*. This consists of two stages: 1) logic compilation (HDL language synthesis) and optimization, and 2) technology mapping.

Logic compilation and optimization: The synthesis tool first compiles input HDL source code, which describes the design at a high level of abstraction, to known structural elements. Next, it optimizes the design, making it as small as possible and improving circuit performance. These optimizations are technology independent.

Technology mapping: During this stage, the tool optimizes the logic for the target technology, by mapping it to technology-specific components. It uses architecture-specific techniques to perform additional optimizations. Finally, it generates a design netlist for placement and routing.



Synthesizing Your Design

The Synplify synthesis tool accepts high-level designs written in industry-standard hardware description languages (Verilog and VHDL) and uses Behavior Extracting Synthesis Technology® (BEST™) algorithms to keep the design at a high level of abstraction, for better optimization. The tool can also write VHDL and Verilog netlists after synthesis, which you can simulate to verify functionality.

You perform the following actions to synthesize your design. For detailed information, see the Tutorial.

1. Access your design project: open an existing project or create a new one.
2. Specify the input source files to use. Right-click the project name in the Project view, then choose Add Source Files.

   Select the desired Verilog or VHDL file(s), then click OK. (See the examples in the directory `installation_dir/examples`, where `installation_dir` is the directory where the product is installed.)

   You can also add source files in the Project view by dragging and dropping them there from a Windows® Explorer folder (Microsoft® Windows® operating system only).

   Top-level file: The last file compiled is the top-level file. You can designate a new top-level file by moving the desired file to the bottom of the source files list in the Project view, or by using the Implementation Options dialog box.
3. Add design constraints. Use the SCOPE spreadsheet to assign system-level and circuit-path timing constraints that can be forward-annotated to .

   See *Timing Constraints,* on page 7-2, for details on the SCOPE spreadsheet.
4. Choose Project -> Implementation Options, then define the following:
   - Target architecture and technology specifications
   - Optimization options and design constraints
   - Outputs

   For an initial run, use the default options settings for the technology, and no timing goal (Frequency = 0 MHz).

5. Synthesize the design by clicking the Run button.

   This step performs logic synthesis. While synthesizing, the Synplify synthesis tool displays the status (Compiling... or Mapping...). You can monitor messages by checking the log file (View -> View Log File). The log file contains reports with information on timing, usage, and net buffering.

   If synthesis is successful, you see the message Done! or Done (warnings). If processing stops because of syntax errors or other design problems, you see the message Errors! displayed, along with the error status in the log file. If the tool displays Done (warnings), there might be potential design problems to investigate.

6. After synthesis, do one of the following:

   – If there were no synthesis warnings or error messages (Done!), analyze your results in the RTL and Technology views. You can then resynthesize with different implementation options, or use the synthesis results to simulate or place-and-route your design.

   – If there were synthesis warnings (Done (warnings)) or error messages (Errors!), check them in the log file. You can double-click an error or warning message in any of these places, to jump to the corresponding source code: the log file, the HDL file in the Project view, or the Tcl window. Use Next Error and Previous Error from the Run menu to navigate.

     Correct any errors, then rerun synthesis.

Getting Help

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

Table 1-3: Getting Help With Synplicity Products

For help with...	Refer to the...
How to...	*Synplify User Guide,* and various application notes available on the Synplicity support web site
Flow information	*Synplify User Guide,* and various application notes available on the Synplicity support web site
Product updates	Synplicity Technical Resource Center (Web menu commands from within the software)
Licensing	*Copyright and License Agreement,* and the appropriate *License Configuration and Setup* document for your platform
Synthesis features	*Synplify Reference Manual*
Language and syntax	*Synplify Reference Manual*
Attributes and directives	*Synplify Reference Manual*
Tcl language	Online help (Help -> Tcl Help)
Synthesis Tcl commands	*Synplify Reference Manual* or type `help` *<command>* in the Tcl window
Using tool-specific features and attributes	*Synplify User Guide*

Contacting Customer Support

If you have a problem, question, or enhancement request, use one of these methods to contact us:

- E-mail us at support@synplicity.com
- Call us at (U.S.) +1 408 215-6000
- Write to us at this address:

  Synplicity, Inc.
  600 West California Avenue
  Sunnyvale, CA 94086, USA
- FAX us at (U.S.) +1 408 222-0263

CHAPTER 2

User Interface Overview

This chapter presents tools and technologies that are built into the Synplify synthesis software to enhance your productivity.

These are the topics in this chapter:

Windows and Views

The Synplify synthesis tool provides a graphic user interface (GUI) with windows and views that help you manage input and output files, direct the synthesis process, and analyze your design and its results. The following windows and views are presented here, except for the SCOPE spreadsheet. For information on the SCOPE spreadsheet, see SCOPE Constraints, on page 7-9.

- Dockable GUI Entities, on page 2-2
- Project View, on page 2-3
- RTL View, on page 2-4
- Technology View, on page 2-6
- Hierarchy Browser, on page 2-8
- Text Editor View, on page 2-10

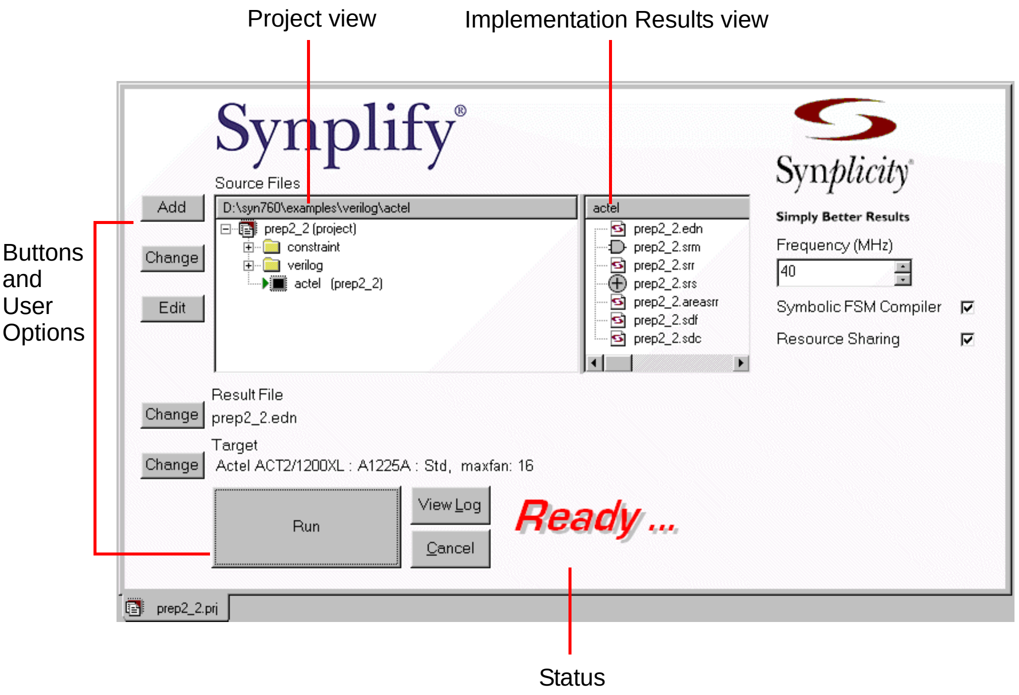
Dockable GUI Entities

Some of the main GUI entities can appear as either independent windows or docked components of the application window. These entities include the menu barand various toolbars. (See the description of each entity for details.) Docked entities function effectively as *panes* of the application window: you can drag the border between two such panes to adjust their relative areas.

Project View

You use the Project view to create or open projects, create new implementations, set device options, and synthesize designs. This is the main synthesis tool view. It contains a list of files you use, and has buttons for quick access to common commands. For information about setting display options in the Project view, see Setting User Interface Preferences, on page 2-19.

Figure 2-1: Project view



Source Files
D:\syn760\examples\verilog\actel
prep2_2 (project)
constraint
verilog
actel (prep2_2)

actel
prep2_2.edn
prep2_2.srm
prep2_2.srr
prep2_2.srs
prep2_2.areasrr
prep2_2.sdf
prep2_2.sdc

Frequency (MHz) 40
Symbolic FSM Compiler ✓
Resource Sharing ✓

Result File
prep2_2.edn
Target
Actel ACT2/1200XL : A1225A : Std, maxfan: 16

prep2_2.prj

The Project view has the following main parts:

- The Status area displays the current status of the synthesis job that is running. Clicking in this area displays additional information about the current job (see Job Status Command, on page 3-46).
- Buttons and options give you immediate access to some of the more common commands. See Buttons and Options, on page 2-36 for details.
- The main Project view lists your projects as well as their associated HDL source files and constraint files. See Projects, Implementations, on page 1-9 for details.

- The Implementation Results view lists the result files for the current implementation. You can only view one set of implementation results at a time. Click an implementation in the Project view to make it active and view its result files.

  The Implementation Results view lists the names and types of the result files, and the dates they were last modified. You can click a column heading to sort the file list accordingly. Click again to reverse the sort direction. An arrow in the column header displays the sorting direction.

RTL View

The RTL view provides a high-level, technology-independent, graphic representation of your design after compilation, using technology-independent components like variable-width adders, registers, large multiplexers, and state machines. RTL views correspond to the `.srs` netlist files generated during compilation. RTL views are only available after your design has been successfully compiled. For information about the other HDL Analyst view (the Technology view generated after mapping), see Technology View, on page 2-6.

To display a RTL view, first compile or synthesize your design, then select HDL Analyst->RTL and choose Hierarchical View or Flattened View, or click the RTL icon (⊕).

An RTL view has two panes: a Hierarchy Browser on the left and an RTL schematic on the right. You can drag the pane divider with the mouse to change the relative pane sizes. For more information about the Hierarchy Browser, see Hierarchy Browser, on page 2-8. Your design is drawn as a set of schematics. The schematic for a design module (or the top level) consists of one or more sheets, only one of which is visible in a given view at any time. The title bar of the window indicates the current hierarchical schematic level, the current sheet, and the total number of sheets for that level.

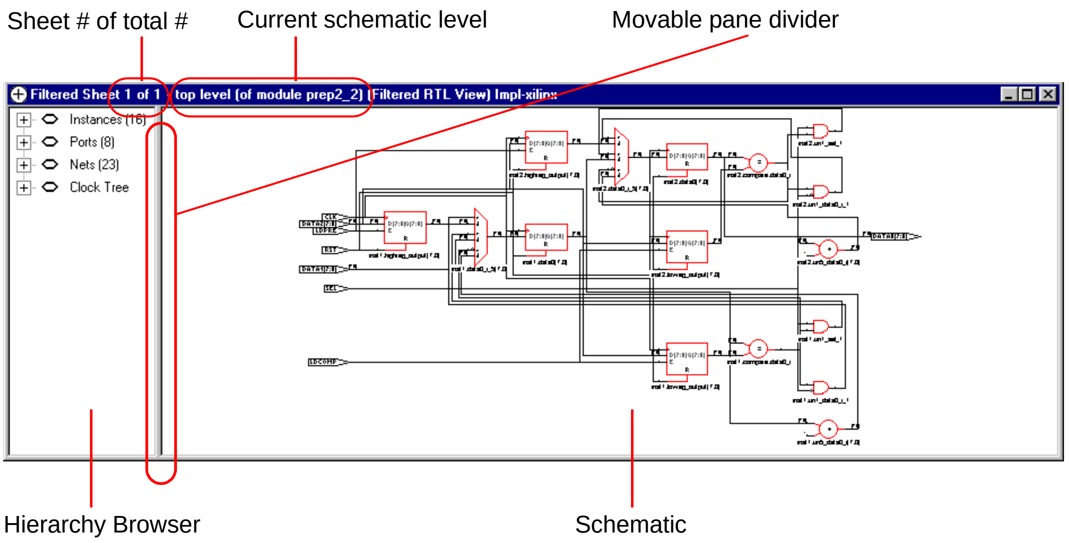


Figure 2-2: RTL view

The design in the RTL schematic can be hierarchical or flattened. Further, the view can consist of the entire design or part of it. Different commands apply, depending on the kind of RTL view.

The following table lists where to find further information about the RTL view:

For information about...	See...
Hierarchy Browser	Hierarchy Browser, on page 2-8
Procedures for RTL view operations like crossprobing, searching, pushing/popping, filtering, flattening, etc.	Basic Operations in the Schematic Views, on page 4-10 of the *Synplify User Guide*.
Explanations or descriptions of features like object display, filtering, flattening, etc.	HDL Analyst Operations, on page 6-1
Commands for RTL view operations like filtering, flattening, etc.	Accessing HDL Analyst Commands, on page 6-3 HDL Analyst Menu, on page 3-49

For information about...	See...
Viewing commands like zooming, panning, etc.	View Menu Commands for Project View, RTL and Technology Views, on page 3-21
History commands: Back and Forward	View Menu Commands for Project View, RTL and Technology Views, on page 3-21
Search command	Find Command (HDL Analyst), on page 3-15

Technology View

A Technology view provides a low-level, technology-specific view of your design after mapping, using components such as look-up tables, cascade and carry chains, multiplexers, and flip-flops. Technology views are only available after your design has been synthesized (compiled and mapped). For information about the other HDL Analyst view (the RTL view generated after compilation), see RTL View, on page 2-4.

To display a Technology view, first synthesize your design, and then either of select a view from the HDL Analyst->Technology menu (Hierarchical View, Flattened View, Flattened to Gates View, Hierarchical Critical Path, or Flattened Critical Path) or select the Technology view icon ( ).

A Technology view has two panes: a Hierarchy Browser on the left and an RTL schematic on the right. You can drag the pane divider with the mouse to change the relative pane sizes. For more information about the Hierarchy Browser, see Hierarchy Browser, on page 2-8. Your design is drawn as a set of schematics at different design levels. The schematic for a design module (or the top level) consists of one or more sheets, only one of which is visible in a given view at any time. The title bar of the window indicates the current schematic level, the current sheet, and the total number of sheets for that level.

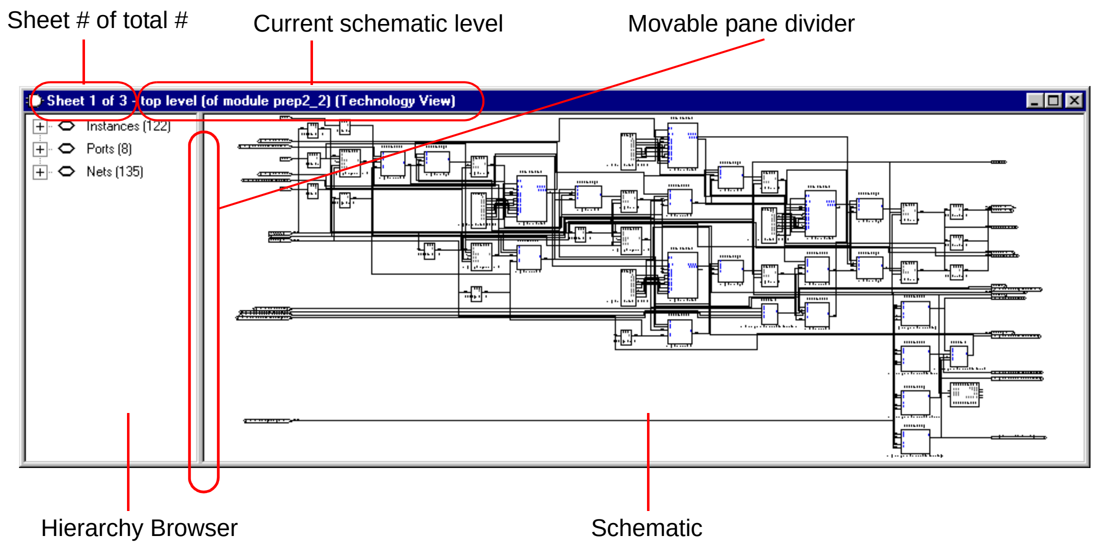


Figure 2-3: Technology View

The schematic design can be hierarchical or flattened. Further, the view can consist of the entire design or a part of it . Different commands apply, depending on the kind of view. In addition to all the features available in RTL views, Technology views have two additional features: critical path filtering and flattening to gates.

The following table lists where to find further information about the Technology view:

For information about...	See...
Hierarchy Browser	Hierarchy Browser, on page 2-8
Procedures for Technology view operations like crossprobing, searching, pushing/popping, filtering, flattening, etc.	Basic Operations in the Schematic Views, on page 4-10 of the *Synplify User Guide*
Explanations or descriptions of features like object display, filtering, flattening, etc.	Chapter 6, *HDL Analyst Operations*

For information about...	See...
Commands for Technology view operations like filtering, flattening, etc.	Accessing HDL Analyst Commands, on page 6-3 HDL Analyst Menu, on page 3-49
Viewing commands like zooming, panning, etc.	View Menu Commands for Project View, RTL and Technology Views, on page 3-21
History commands: Back and Forward	View Menu Commands for Project View, RTL and Technology Views, on page 3-21
Search command	Find Command (HDL Analyst), on page 3-15

Hierarchy Browser

The Hierarchy Browser is the left pane in the RTL and Technology views. (See RTL View, on page 2-4 and Technology View, on page 2-6.) The Hierarchy Browser categorizes the design objects in a series of trees, and lets you browse the design hierarchy or select the objects. Selecting the object in the Browser selects the object in the schematic. The objects are organized as shown in the following table, with a symbol that indicates the object type. See Hierarchy Browser Symbols, on page 2-9 for common symbols.

Instances	Lists all the instances and primitives in the design. In a Technology view, it includes all technology-specific primitives.
Ports	Lists all the ports in the design
Nets	Lists all the nets in the design.
Clock Tree	Lists all the instances and ports that drive clock pins in an RTL view. If you select everything listed under Clock Tree and then use the Filter Schematic command, you see a filtered view of all clock pin drivers in your design. Registers are not shown in the resulting schematic, unless they drive clocks. This view can help you determine what to define as clocks.

A tree node can be expanded or collapsed by clicking the associated icons: the square plus (⊞) or minus (⊟) icons, respectively. You can also expand or collapse all trees at the same time by right-clicking in the Hierarchy Browser and choosing Expand All or Collapse All.

You can use the keyboard arrow keys (left, right, up, down) to move between objects in the Hierarchy Browser, or use the scroll bar. Use the Shift or Ctrl keys to select multiple objects. See Navigating With a Hierarchy Browser, on page 6-24 for more information about using the Hierarchy Browser for navigation and crossprobing.

Hierarchy Browser Symbols

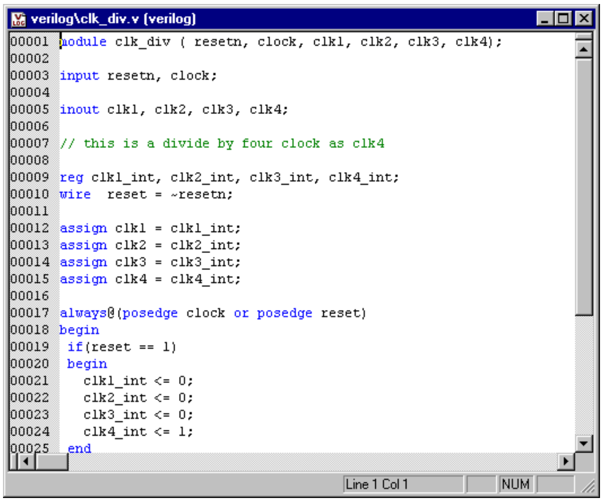
Common symbols used in Hierarchy Browsers are listed in the following table.

Table 2-1: Hierarchy Browser Symbols

Symbol	Description	Symbol	Description
⬭	Folder	⊐∘	NAND gate
■	Input port	⊳	OR gate
▭	Output port	⊳∘	NOR gate
⬬	Bidirectional port	⊳	XOR gate
⊶	Net	⊳∘	XNOR gate
□	Other primitive instance	⊕	Adder
▣	Hierarchical instance	⊛	Multiplier
▫	Technology-specific primitive or inferred ROM	⊜	Equal comparator
▢	Register or inferred state machine	⧀	Less-than comparator
⊳	Multiplexer	≤	Less-than-or-equal comparator
▷	Tristate		
▷∘	Inverter		
▷	Buffer		
⊐	AND gate		

Text Editor View

The Text Editor view displays text files. These can be constraint files, source code files, log files, or other informational or report files. You can enter and edit text in the window. You use this window to update source code and fix syntax or synthesis errors. You can also use it to crossprobe the design. For information about using the Text Editor, see Editing Source Files with the Built-in Text Editor, on page 2-5 in the *Synplify User Guide*.



```
module clk_div ( resetn, clock, clk1, clk2, clk3, clk4);

input resetn, clock;

inout clk1, clk2, clk3, clk4;

// this is a divide by four clock as clk4

reg clk1_int, clk2_int, clk3_int, clk4_int;
wire  reset = ~resetn;

assign clk1 = clk1_int;
assign clk2 = clk2_int;
assign clk3 = clk3_int;
assign clk4 = clk4_int;

always@(posedge clock or posedge reset)
begin
 if(reset == 1)
 begin
   clk1_int <= 0;
   clk2_int <= 0;
   clk3_int <= 0;
   clk4_int <= 1;
 end
```

Figure 2-4: Text Editor viewText Editor View

Opening the Text Editor

To open the Text Editor to edit an existing file, do one of the following:

- Double-click a source code file (.v or .vhd) in the Project view.
- Choose File -> Open. In the dialog box displayed, double-click a file to open it.

  With the Microsoft® Windows® operating system, you can instead drag and drop a source file from a Windows folder into the gray background area of the GUI (*not* into any particular view).

To open the Text Editor on a new file, do one of the following:

- Choose File -> New, then specify the kind of text file you want to create.
- Click the HDL icon ( ) to create and edit an HDL source file.

The Text Editor colors HDL source code keywords such as module and output blue, and comments green.

Text Editor Features

The Text Editor has the features listed in the following table.

Table 2-2: Text Editor Features

Feature	Description
Color coding	Keywords are blue, comments green, and strings red. All other text is black.
Editing text	You can use the Edit menu or keyboard shortcuts for basic editing operations like Cut, Copy, Paste, Find, Replace, and Goto.
Completing keywords	To complete a keyword, type enough characters to determine it uniquely, then press the Esc key.
Indenting a block of text	The Tab key indents a selected block of text to the right. Shift-Tab indents text to the left.
Inserting a bookmark	Click the line you want to bookmark. Choose Edit -> Toggle Bookmark, type Ctrl-F2, or click the Toggle Bookmark icon ( ) on the Edit toolbar. The line number is highlighted to indicate that there is a bookmark at the beginning of the line.
Deleting a bookmark	Click the line with the bookmark. Choose Edit -> Toggle Bookmark, type Ctrl-F2, or click the Toggle Bookmark icon ( ) on the Edit toolbar. The line number is no longer highlighted, after the bookmark is deleted.
Deleting all bookmarks	Choose Edit -> Delete all Bookmarks, type Ctrl-Shift-F2, or click the Clear All Bookmarks icon ( ) on the Edit toolbar. The line numbers are no longer highlighted, after the bookmarks are deleted.
Editing columns	Press and hold Alt, then drag the mouse down a column of text, to select it. On some platforms, Alt is replaced by a different key, such as Meta.

Table 2-2: Text Editor Features (Continued)

Feature	Description
Commenting out code	Choose Edit -> Advanced -> Comment Code. The rest of the current line is commented out: the appropriate comment prefix is inserted at the current text cursor position.
Checking syntax	Use Run -> Syntax Check to highlight syntax errors, such as incorrect keywords and punctuation, in source code. If the active window shows an HDL file, then only that file is checked. Otherwise, the entire project is checked.
Checking synthesis	Use Run -> Synthesis Check to highlight hardware-related errors in source code, like incorrectly coded flip-flops. If the active window shows an HDL file, then only that file is checked. Otherwise, the entire project is checked.

See also:

- Editor Options Command, on page 3-71, for information on setting Text Editor preferences.
- File Menu, on page 3-4, for information on printing setup operations.
- Edit Menu Commands for the Text Editor, on page 3-12, for information on Text Editor editing commands.
- Text Editor Popup Menu, on page 3-96, for information on the Text Editor popup menu.
- Project View Popup Menu, on page 3-91, for information on the command Open as Text, which opens a file in the Text Editor even if there is a different default method of opening it. An example is opening a constraint (`.sdc`) file.
- Edit Toolbar, on page 2-27, for information on bookmark icons of the Edit toolbar.
- Keyboard Shortcuts, on page 2-28, for information on keyboard short-cuts that can be used in the Text Editor.

FSM Compiler

This section describes the FSM Compiler, what it does, and in what contexts it is used.

What FSM Compiler Does

Other synthesis tools treat state machines as regular logic, but the FSM Compiler performs proprietary state machine optimization techniques. You enable the FSM compiler to take advantage of these techniques; you do not need special directives or attributes to locate the state machines in your design. You can also, however, enable the FSM compiler selectively for individual state machines, using synthesis directives in the HDL description.

When you enable the FSM compiler, state machines are discovered and extracted into symbolic graph form, then special optimizations are performed. These include re-encoding state representations, which generates a better starting point for logic optimization.

The FSM compiler examines your design for state machines. It looks for registers with feedback that is controlled by the current value of the register, such as case or if-then-else statements that test the current value of a state register. It converts state machines to a symbolic form that provides a better starting point for logic optimization. Several proprietary optimizations are performed on each symbolic state machine.

Converting from an encoded state machine to a one-hot state machine often produces better results. However, one-hot implementations are not always the best choice, even in FPGAs and CPLDs. For example, one-hot state machines might result in higher speeds in CPLDs, but cause fitting problems because of the larger number of global signals. An example where the one-hot implementation can be detrimental in an FPGA is a state machine that drives a large decoder, generating many output signals. In a 16-state state machine, for instance, the output decoder logic might reference eight signals in a one-hot implementation, but only four signals in an encoded representation.

During synthesis, a state encoding for an FSM is determined, based on certain predefined characteristics of the state machine. The optional FSM Explorer feature enhances this capability by automatically determining and using the best encoding styles for the state machines based on the design constraints and the area/delay requirements. You can force the use of a particular encoding style for a state machine by including the appropriate directive in the HDL description.

The log file contains a description of each state machine extracted, including a list of the reachable states and the state encoding used.

When to Use FSM Compiler

Use the symbolic FSM compiler to generate better results for state machines, or to debug state machines. If you do not want to use the symbolic FSM compiler on the final circuit, you can use it only during initial synthesis to check that the state machines are described correctly. Many common state machine description errors result in unreachable states, which are optimized away during synthesis, resulting in a smaller number of states than you expect. Reachable states are reported in the log file.

To view a textual description of a state machine in terms of inputs, states, and transitions, select the state machine in the RTL view, right-click, then choose View FSM Info File in the popup menu.

See also:

- Log File, on page 4-7, for information on the log file.
- RTL View and Technology View Popup Menu Commands, on page 3-97, for information on the command View FSM Info File.

Where to Use FSM Compiler (Global and Local Use)

Enable the FSM Compiler check box in the Project view to turn on FSM synthesis. This allows the tool to recognize, extract and optimize the state machines in the design.

The following table summarizes the operations you can carry out. For more information, see *Choosing When to Use the FSM Compiler, on page 5-17* of the *Synplify User Guide.*

Table 2-3: FSM Compiler Operations

To...	Do this...
Globally enable (disable) the FSM Compiler	Enable (disable) the FSM Compiler check box in the Project view.
Enable (disable) the FSM compiler for a specific register	*Dis*able (*en*able) the FSM Compiler check box and set the Verilog syn_state_machine directive to 1 (0), or the VHDL syn_state_machine directive to true (false), for that instance of the state register.

Using the Mouse

The mouse button operations in Synplicity products are standard. This section describes the following:

- Using Mouse Strokes, on page 2-15
- Using the Mouse Buttons, on page 2-17
- Using The Mouse Wheel, on page 2-19

Using Mouse Strokes

As a shortcut tool or several often-used commands, use mouse strokes to draw simple patterns with the mouse button while pressing the appropriate keyboard keys at the same, as specified in the table below.

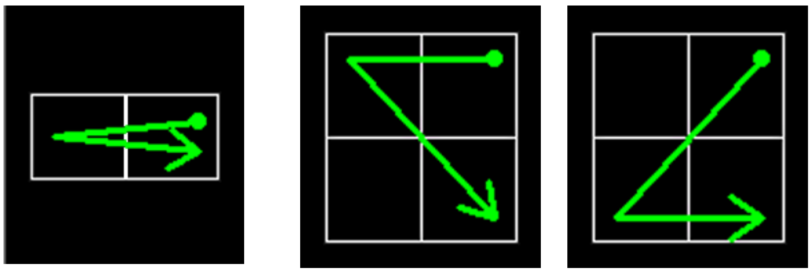
Table 2-4: Mouse Stroke Button and Key Combinations

Operating System	Mouse Button + Keyboard Key(s)
Microsoft® Windows®	left mouse button with Alt keyboard key
UNIX Linux with UNIX-style keyboard	left mouse button with Meta or diamond keyboard key
Linux with PC-style keyboard	left mouse button with both Ctrl and Alt keyboard keys

Note: If you use a three-button mouse on the UNIX platform, you only need to use the middle button to draw the shortcut stroke. No modifier key is necessary.

For information on each of the available mouse strokes, consult the mouse stroke tutor using the Help -> Mouse Stroke Tutor menu.

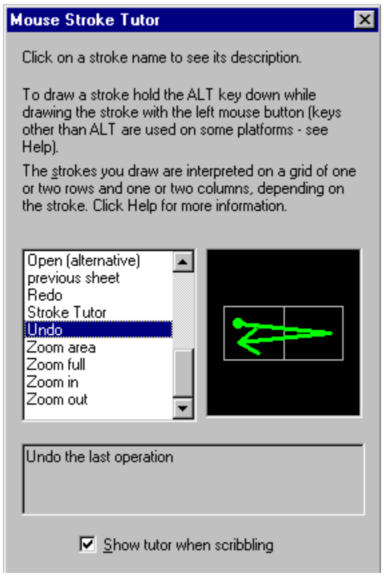
The strokes you draw are interpreted on a grid of one or two rows and one or two columns, depending on the stroke. Some strokes are similar, differing only in the number of columns or rows, so it may take a little practice to draw them correctly. For example, the strokes for Redo and Back differ in that the Redo stroke is back and forth horizontally, within a single-row grid, while the Back stroke involves vertical movement as well:



Redo stroke: one row

Back strokes: two rows

Figure 2-5: Redo and Back mouse strokes


Mouse Stroke Tutor

Click on a stroke name to see its description.

To draw a stroke hold the ALT key down while drawing the stroke with the left mouse button (keys other than ALT are used on some platforms - see Help).

The strokes you draw are interpreted on a grid of one or two rows and one or two columns, depending on the stroke. Click Help for more information.

Open (alternative)
previous sheet
Redo
Stroke Tutor
Undo
Zoom area
Zoom full
Zoom in
Zoom out

Undo the last operation

☑ Show tutor when scribbling

Figure 2-6: Mouse Stroke Tutor

To use the tutor, select a command or action in the list; its mouse stroke is then displayed. When Show tutor when scribbling is enabled, you can scribble an unrecognized stroke at any time to display the mouse stroke tutor.

Using the Mouse Buttons

The operations you can perform using mouse buttons include the following:

- You can select an (unselected) object by clicking it. You can deselect a (selected) object by clicking it. Selecting an object by clicking it deselects all previously selected objects.
- You can select and deselect multiple objects by pressing and holding the Control key (Ctrl) while clicking each of the objects.
- You can select a range of objects in a Hierarchy Browser, as follows:
  - select the first object in the range
  - scroll the tree of objects, if necessary, to display the last object in the range

- press and hold the Shift key while clicking the last object in the range

  Selecting a range of objects in a Hierarchy Browser crossprobes to the corresponding schematic, where the same objects are automatically selected.

- You can select all of the objects in a region by tracing a selection rectangle around them (lassoing).
- You can select text by dragging the mouse over it. You can alternatively select text containing no whitespace (such as spaces) by double-clicking it.
- Double-clicking sometimes selects an object and immediately initiates a default action associated with it. For example, double-clicking a source file in the Project view opens the file in a Text Editor window.
- You can access a contextual popup menu by right-clicking or pressing and holding the right mouse button. The menu displayed is specific to the current context, including the object or window under the mouse.

  For example, right-clicking a project name in the Project view displays a popup menu with operations appropriate to the project file. Right-clicking a source (HDL) file in the Project view displays a popup menu with operations applicable to source files.

  Right-clicking a selectable object in an HDL Analyst schematic also *selects* it, and deselects anything that was selected. The resulting popup menu applies only to the selected object. See RTL View, on page 2-4, and Technology View, on page 2-6, for information on HDL Analyst views.

Most of the mouse button operations involve selecting and unselecting objects. To use the mouse in this way in an HDL Analyst schematic, the mouse pointer must be the cross-hairs symbol: ┼. If the cross-hairs pointer is not displayed, right-click the schematic background to display it.

Using The Mouse Wheel

If your mouse has a wheel and you are using a Microsoft Windows platform, you can use it to scroll and zoom, as follows:

- Whenever only a horizontal scroll bar is visible, rotating the wheel scrolls the window horizontally.
- Whenever a vertical scroll bar is visible, rotating the wheel scrolls the window vertically.
- Whenever both horizontal and vertical scroll bars are visible, rotating the wheel while pressing and holding the Shift key scrolls the window horizontally.
- In a window that can be zoomed, such as a graphics window, rotating the wheel while pressing and holding the Ctrl key zooms the window.

Setting User Interface Preferences

This section summarizes some user preferences.

Customizing the User Interface

The following table lists the commands with which you can set preferences and customize the user interface. For detailed procedures, see the *Synplify User Guide*.

Table 2-5: Setting User Interface Preferences

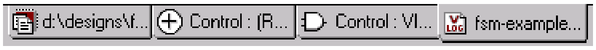
Preferences	Description	For option descriptions, see...
Text Editor	Fonts and colors	Editor Options Command
HDL Analyst tool (RTL/Technology views)	HDL Analyst options	HDL Analyst Menu
Project view	Organization and display of project files	Project View Options Command

Managing Views

As you work on a project, you move between different views of the design. The following guidelines can help you manage the different views you have open.

1. Enable the option View -> Workbook Mode.

   Below the Project view are tabs, one for each open view. The icon accompanying the view name on a tab indicates the type of view. This example, shows tabs for four views: the Project view, an RTL view, a Technology view, and a Verilog Text Editor view.

2. To bring an open view to the front and make it the current (active) view, click any visible part of the window, or click the tab of the view.

   If you previously minimized the view, it will be activated but will remain minimized. To display it, double-click the minimized view.

3. To activate the next view and bring it to the front, type Ctrl-F6. Repeating this keyboard shortcut cycles through all open views. If the next view was minimized it remains minimized, but it is brought to the front so that you can restore it.

4. To close a view, type Ctrl-F4 in the view, or choose File -> Close.

5. You can rearrange open windows using the Window menu: you can cascade them (stack them, slightly offset), or tile them horizontally or vertically.

Toolbars

Toolbars provide a quick way to access common menu commands by clicking their icons. The following standard toolbars are available:

- Project Toolbar — Project control and file manipulation.
- Analyst Toolbar — Manipulation of RTL and Technology views.
- View Toolbar — Viewing and hierarchy navigation.
- Edit Toolbar — Text Editor bookmarking operations.

You can enable and disable the display of individual toolbars – see Toolbar Command, on page 3-22.

You can customize these standard toolbars or create your own. Customization includes choosing the icons to include, their size (large or small), their appearance, and whether or not to associate them with tooltips. For information on customizing toolbars, see Customize Dialog Box, on page 3-24.

By dragging a toolbar, you can move it anywhere on the screen: you can make it float in its own window or dock it at a docking area (an edge) of the application window. To move the menu bar to a docking area without docking it there (that is, to leave it floating), press and hold the Ctrl or Shift key while dragging it.

Right-clicking the window *title bar* when a toolbar is floating displays a popup menu with commands Hide and Move. Hide removes the window. Move lets you position the window using either the arrow keys or the mouse.

Project Toolbar

The Project toolbar provides the following icons, by default:

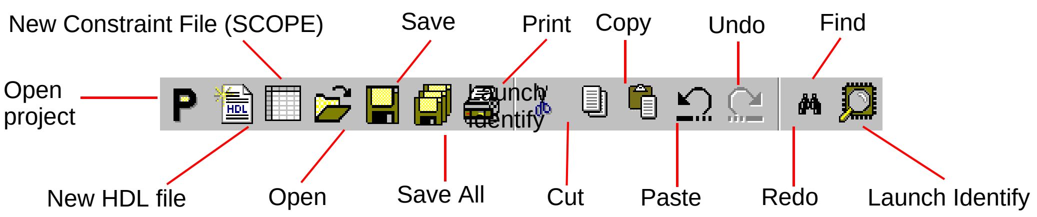


Figure 2-7: Project toolbar

The following table describes the default Project icons. Each is equivalent to a File or Edit menu command; for more information, see the following:

- File Menu, on page 3-4
- Edit Menu, on page 3-11

Table 2-6: Project Icons

Icon	Description
Open Project	Displays the Open Project dialog box to create a new project or open an existing project. A Project wizard can help you specify and arrange project files. Same as File -> Open Project.
New HDL file	Opens the Text Editor window with a new, empty source file. Same as File -> New, Verilog File or VHDL File.
New Constraint File (SCOPE)	Opens the SCOPE spreadsheet with a new, empty constraint file. Same as File -> New, Constraint File (SCOPE).
Open	Displays the Open dialog box, to open a file. Same as File -> Open.

Table 2-6: Project Icons (Continued)

Icon	Description
Save	Saves the current file. If the file has not yet been saved, this displays the Save As dialog box, where you specify the filename. The kind of file depends on the active view. Same as File -> Save.
Save All	Saves all files associated with the current design. Same as File -> Save As.
Print	Prints the active view (RTL view, Technology view, or Text Editor). Same as File -> Print.
Cut	Cuts text or graphics from the active view, making it available to Paste. Same as Edit -> Cut.
Paste	Pastes previously cut or copied text or graphics to the active view. Same as Edit -> Paste.
Undo	Undoes the last action taken. Same as Edit -> Undo.
Redo	Performs the action undone by Undo. Same as Edit -> Redo.
Find	Finds text in the Text Editor or objects in an RTL view or Technology view. Same as Edit -> Find.
Launch Identify Instrumentor	Launches the Synplicity Identify Instrumentor product.

Analyst Toolbar

The Analyst toolbar becomes active after a design has been compiled. It provides the following icons, by default:

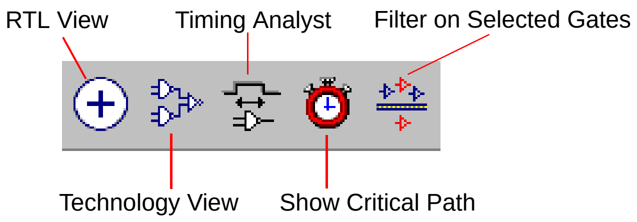


Figure 2-8: Analyst toolbar

The following table describes the default Analyst icons. Each is equivalent to an HDL Analyst menu command – see HDL Analyst Menu, on page 3-49, for more information.

Table 2-7: Analyst Icons

Icon	Description
⊕ RTL View	Opens a new, hierarchical RTL view: a register transfer-level schematic of the compiled design, together with the associated Hierarchy Browser. Same as HDL Analyst -> RTL -> Hierarchical View.

Table 2-7: Analyst Icons (Continued)

Icon	Description
Technology View	Opens a new, hierarchical Technology view: a technology-level schematic of the mapped (synthesized) design, together with the associated Hierarchy Browser. Same as HDL Analyst -> Technology -> Hierarchical View.
Show Critical Path	Filters your design to show only the instances (and their paths) whose slack times are within the slack margin of the worst slack time of the design (see HDL Analyst -> Set Slack Margin). The result is flat if the entire design was already flat. Icon Show Critical Path also enables HDL Analyst -> Show Timing Information. Available only in a Technology view. Not available in a Timing view. Same as HDL Analyst -> Show Critical Path.
Filter on Selected Gates	Filters your entire design to show only the selected objects. The result is a *filtered* schematic. Same as HDL Analyst -> Filter Schematic.

View Toolbar

The View toolbar lets you zoom in and out of your schematic, and traverse its hierarchy in various ways. It provides the following icons, by default:

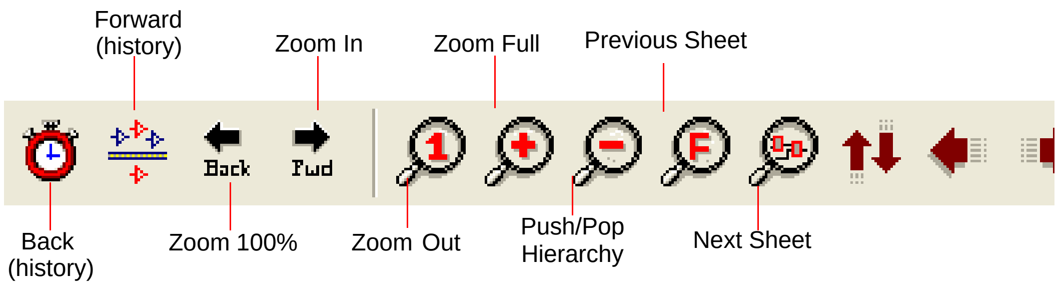


Figure 2-9: View toolbar

The following table describes the default View icons. Each is available in HDL Analyst views, and each is equivalent to a View menu command available there – see View Menu Commands for Project View, RTL and Technology Views, on page 3-21, for more information. Zoom icons are also available in some other views.

Table 2-8: View Icons

Icon	Description
Back	Goes backward in the history of displayed sheets of the current HDL Analyst view. Same as View -> Back.
Forward	Goes forward in the history of displayed sheets of the current HDL Analyst view. Same as View -> Forward.
Zoom 100%	Zooms in at a 1:1 ratio and centers the active view where you click. If the view is already normal size, it re-centers the view at the new click location. Same as View -> Normal View.[a]
Zoom In Zoom Out	Zooms the view in or out. Buttons stay active until deselected. Same as View -> Zoom In or View -> Zoom Out.[a]
Zoom Full	Zoom that reduces the active view to display the entire design. Same as View -> Full View.[b]
Push/Pop Hierarchy	Toggles traversing the hierarchy using the push/pop mode. Same as View -> Push/Pop Hierarchy.
Previous Sheet	Displays the previous sheet of a multiple-sheet schematic. Same as View -> Previous Sheet.
Next Sheet	Displays the next sheet of a multiple-sheet schematic. Same as View -> Previous Sheet.

a. Available only in the SCOPE spreadsheet, RTL views, and Technology views.
b. Available only in RTL views and Technology views.

Edit Toolbar

The Edit toolbar is active whenever the Text Editor is active. You use it to edit *bookmarks* in the file. (Other editing operations are located on the Project toolbar – see Project Toolbar, on page 2-22.) The Edit toolbar provides the following icons, by default:

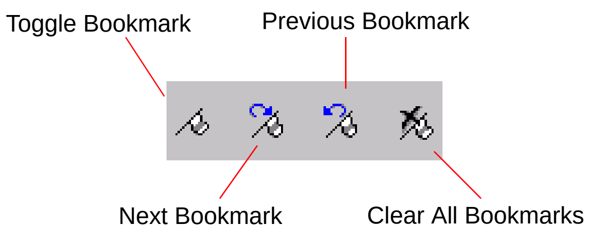


Figure 2-10: Edit toolbar

The following table describes the default Edit icons. Each is available in the Text Editor, and each is equivalent to an Edit menu command there – see Edit Menu Commands for the Text Editor, on page 3-12, for more information.

Table 2-9: Edit Icons

Icon	Description
Toggle Bookmark	Alternately inserts and removes a bookmark at the line that contains the text cursor. Same as Edit -> Toggle bookmark.
Next Bookmark	Takes you to the next bookmark. Same as Edit -> Next bookmark.
Previous Bookmark	Takes you to the previous bookmark. Same as Edit -> Previous bookmark.
Clear All Bookmarks	Removes all bookmarks from the Text Editor window. Same as Edit -> Delete all bookmarks.

Keyboard Shortcuts

Keyboard shortcuts are key sequences that you type in order to run a command. Menus list keyboard shortcuts next to the corresponding commands. For example, to check syntax, you can press and hold the Shift key while you type the F7 key, instead of using the menu command Run -> Syntax Check.

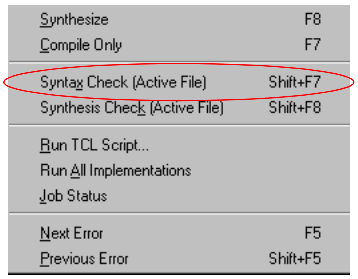


Figure 2-11: Keyboard shortcut indicated in menu

The following table describes the keyboard shortcuts.

Table 2-10: Keyboard Shortcuts

Keyboard Shortcut	**Description**
b	In an RTL or Technology view, shows all logic between two or more selected objects (instances, pins, ports). The result is a *filtered* schematic. Limited to the current schematic. Same as HDL Analyst -> Current Level -> Expand Paths (see HDL Analyst Menu: Filtering and Flattening Commands, on page 3-53).
Ctrl-1	In an RTL or Technology view, zooms the active view, when you click, to full (normal) size. Same as View -> Normal View .
Ctrl-a	Centers the window on the design. Same as View -> Pan Center.

Table 2-10: Keyboard Shortcuts (Continued)

Keyboard Shortcut	Description
Ctrl-Alt-h	In an RTL or Technology view, shows all pins on selected *transparent* hierarchical (non-primitive) instances. Pins on primitives are always shown. Available only in a filtered schematic. Same as HDL Analyst -> Show All Hier Pins (see HDL Analyst Menu: Analysis Commands, on page 3-58).
Ctrl-b	In an RTL or Technology view, shows all logic between two or more selected objects (instances, pins, ports). The result is a *filtered* schematic. Operates hierarchically, on lower levels as well as the current schematic. Same as HDL Analyst -> Hierarchical -> Expand Paths (see HDL Analyst Menu: Hierarchical and Current Level Submenus, on page 3-51).
Ctrl-c	Copies the selected object. Same as Edit -> Copy. This shortcut is sometimes available even when Edit -> Copy is not. See, for instance, Find Command (HDL Analyst), on page 3-15.)
Ctrl-d	In an RTL or Technology view, selects the driver for the selected net. Operates hierarchically, on lower levels as well as the current schematic. Same as HDL Analyst -> Hierarchical -> Select Net Driver (see HDL Analyst Menu: Hierarchical and Current Level Submenus, on page 3-51).
Ctrl-e	In an RTL or Technology view, expands along the paths from selected pins or ports, according to their directions, to the nearest objects (no farther). The result is a *filtered* schematic. Operates hierarchically, on lower levels as well as the current schematic. Same as HDL Analyst -> Hierarchical -> Expand (see HDL Analyst Menu: Hierarchical and Current Level Submenus, on page 3-51).
Ctrl-f	Finds the selected object. Same as Edit -> Find.
Ctrl-F2	Alternately inserts and removes a bookmark to the line that contains the text cursor. Same as Edit -> Toggle bookmark (see Edit Menu Commands for the Text Editor, on page 3-12).
Ctrl-F4	Closes the current window. Same as File -> Close.

Table 2-10: Keyboard Shortcuts (Continued)

Keyboard Shortcut	Description
Ctrl-F5	Opens the first project source file that has errors.
Ctrl-F6	Activates the next window.
Ctrl-g	In the Text Editor, jumps to the specified line. Same as Edit -> Goto (see Edit Menu Commands for the Text Editor, on page 3-12). In an RTL or Technology view, selects the sheet number in a multiple-page schematic. Same as View -> View Sheets (see View Menu Commands for Project View, RTL and Technology Views, on page 3-21).
Ctrl-h	In the Text Editor, replaces text. Same as Edit -> Replace (see Edit Menu Commands for the Text Editor, on page 3-12). In an RTL or Technology view, enables/disables crossprobing to the Visual Elite design tool. Same as HDL Analyst -> Visual Elite Crossprobing -> Connect to Visual Elite or Disconnect from Visual Elite (see HDL Analyst Menu: Crossprobing Commands, on page 3-61).
Ctrl-i	In an RTL or Technology view, selects instances connected to the selected net. Operates hierarchically, on lower levels as well as the current schematic. Same as HDL Analyst -> Hierarchical -> Select Net Instances (see HDL Analyst Menu: Hierarchical and Current Level Submenus, on page 3-51).
Ctrl-j	In an RTL or Technology view, displays the unfiltered schematic sheet that contains the net driver for the selected net. Operates hierarchically, on lower levels as well as the current schematic. Same as HDL Analyst -> Hierarchical -> Goto Net Driver (see HDL Analyst Menu: Hierarchical and Current Level Submenus, on page 3-51).
Ctrl-k	In an RTL or Technology view, enables crossprobing to windows in other tools. Same as HDL Analyst -> External Crossprobing Engaged (see HDL Analyst Menu: Crossprobing Commands, on page 3-61).

Table 2-10: Keyboard Shortcuts (Continued)

Keyboard Shortcut	Description
Ctrl-l	In an RTL or Technology view, toggles zoom locking. When locking is enabled, if you resize the window the displayed schematic is resized proportionately, so that it occupies the same portion of the window. Same as View -> Zoom Lock (see View Menu Commands for All Views, on page 3-20).
Ctrl-n	In an RTL or Technology view, expands inside the subdesign, from the lower-level port that corresponds to the selected pin, to the nearest objects (no farther). Same as HDL Analyst -> Hierarchical -> Expand Inwards (see HDL Analyst Menu: Hierarchical and Current Level Submenus, on page 3-51). In other contexts, creates a new file or project. Same as File -> New .
Ctrl-o	Opens an existing file or project. Same as File -> Open.
Ctrl-p	Prints the current view. Same as File -> Print .
Ctrl-r	In an RTL or Technology view, expands along the paths from selected pins or ports, according to their directions, until registers, ports, or black boxes are reached. The result is a *filtered* schematic. Operates hierarchically, on lower levels as well as the current schematic. Same as HDL Analyst -> Hierarchical -> Expand to Register/Port (see HDL Analyst Menu: Hierarchical and Current Level Submenus, on page 3-51).
Ctrl-s	In the Project View, saves the file. Same as File -> Save.
Ctrl-u	In the Text Editor, changes the selected text to lower case. Same as Edit -> Advanced -> Lowercase (see Edit Menu Commands for the Text Editor, on page 3-12).
Ctrl-v	Pastes the last object copied or cut. Same as Edit -> Paste.
Ctrl-w	Opens a wizard to help you define constraints for a particular SCOPE spreadsheet panel, or attributes for an object. Same as Edit -> Insert Wizard (see Edit Menu Commands for the SCOPE Spreadsheet, on page 3-12).

Table 2-10: Keyboard Shortcuts (Continued)

Keyboard Shortcut	Description
Ctrl-x	Cuts the selected object(s), making it available to Paste. Same as Edit -> Cut.
Ctrl-y	In an RTL or Technology view, goes forward in the history of displayed sheets for the current HDL Analyst view. Same as View -> Forward (see View Menu Commands for Project View, RTL and Technology Views, on page 3-21). In other contexts, performs the action undone by Undo. Same as Edit -> Redo.
Ctrl-z	In an RTL or Technology view, goes backward in the history of displayed sheets for the current HDL Analyst view. Same as View -> Back (see View Menu Commands for Project View, RTL and Technology Views, on page 3-21). In other contexts, undoes the last action. Same as Edit -> Undo.
Ctrl-Shift-F2	Removes all bookmarks from the Text Editor window. Same as Edit -> Delete all bookmarks (see Edit Menu Commands for the Text Editor, on page 3-12).
Ctrl-Shift-i	In an RTL or Technology view, selects all instances on the current schematic level (all sheets). This does *not* select instances on other levels. Same as HDL Analyst -> Select All Schematic -> Instances (see HDL Analyst Menu, on page 3-49).
Ctrl-Shift-p	In an RTL or Technology view, selects all ports on the current schematic level (all sheets). This does *not* select ports on other levels. Same as HDL Analyst -> Select All Schematic -> Ports (see HDL Analyst Menu, on page 3-49).
Ctrl-Shift-u	In the Text Editor, changes the selected text to lower case. Same as Edit -> Advanced -> Uppercase (see Edit Menu Commands for the Text Editor, on page 3-12).
d	In an RTL or Technology view, selects the driver for the selected net. Limited to the current schematic. Same as HDL Analyst -> Current Level -> Select Net Driver (see HDL Analyst Menu, on page 3-49).
Delete (DEL)	Removes the selected files from the project. Same as Project -> Remove Files From Project.

Table 2-10: Keyboard Shortcuts (Continued)

Keyboard Shortcut	Description
e	In an RTL or Technology view, expands along the paths from selected pins or ports, according to their directions, to the nearest objects (no farther). Limited to the current schematic. Same as HDL Analyst -> Current Level -> Expand (see HDL Analyst Menu, on page 3-49).
F1	Provides context-sensitive help. Same as Help -> Help.
F2	In an RTL or Technology view, toggles traversing the hierarchy using the push/pop mode. Same as View -> Push/Pop Hierarchy (see View Menu Commands for Project View, RTL and Technology Views, on page 3-21). In the Text Editor, takes you to the next bookmark. Same as Edit -> Next bookmark (see Edit Menu Commands for the Text Editor, on page 3-12).
F3	In an RTL or Technology view, toggles the display of visual properties of instances, pins, nets, and ports in a design.
F4	In the Project view, adds a file to the project. Same as Project -> Add Source File (see Build Project Command, on page 3-6). In an RTL or Technology view, zooms the view so that it shows the entire design. Same as View -> Full View (see View Menu Commands for Project View, RTL and Technology Views, on page 3-21).
F5	Displays the next source file error. Same as Run -> Next Error/Warning (see Run Menu, on page 3-43).
F7	Compiles your design, without mapping it. Same as Run -> Compile Only (see Run Menu, on page 3-43).
F8	Synthesizes (compiles and maps) your design. Same as Run -> Synthesize (see Run Menu, on page 3-43).
F10	In an RTL or Technology view, lets you pan (scroll) the schematic by dragging it with the mouse. Same as View -> Pan (see View Menu Commands for Project View, RTL and Technology Views, on page 3-21).
F11	Toggles zooming in. Same as View -> Zoom In (see View Menu Commands for Project View, RTL and Technology Views, on page 3-21).

Table 2-10: Keyboard Shortcuts (Continued)

Keyboard Shortcut	Description
F12	In an RTL or Technology view, filters your entire design to show only the selected objects. Same as HDL Analyst -> Filter Schematic – see HDL Analyst Menu: Filtering and Flattening Commands, on page 3-53.
i	In an RTL or Technology view, selects instances connected to the selected net. Limited to the current schematic. Same as HDL Analyst -> Current Level -> Select Net Instances (see HDL Analyst Menu, on page 3-49).
j	In an RTL or Technology view, displays the unfiltered schematic sheet that contains the net driver for the selected net. Same as HDL Analyst -> Current Level -> Goto Net Driver (see HDL Analyst Menu, on page 3-49).
r	In an RTL or Technology view, expands along the paths from selected pins or ports, according to their directions, until registers, ports, or black boxes are reached. The result is a *filtered* schematic. Limited to the current schematic. Same as HDL Analyst -> Current Level -> Expand to Register/Port (see HDL Analyst Menu, on page 3-49).
Shift-F2	In the Text Editor, takes you to the previous bookmark.
Shift-F5	Displays the previous source file error. Same as Run -> Previous Error/Warning (see Run Menu, on page 3-43).
Shift-F7	Checks source file syntax. Same as Run -> Syntax Check (see Run Menu, on page 3-43).
Shift-F8	Checks synthesis. Same as Run -> Synthesis Check (see Run Menu, on page 3-43).
Shift-F11	Toggles zooming out. Same as View -> Zoom Out (see View Menu, on page 3-20).

Table 2-10: Keyboard Shortcuts (Continued)

Keyboard Shortcut	Description
Shift-Left Arrow	Displays the previous sheet of a multiple-sheet schematic.
Shift-Right Arrow	Displays the next sheet of a multiple-sheet schematic.
Shift-s	Dissolves the selected instances, showing their lower-level details. Dissolving an instance one level replaces it, in the current sheet, by what you would see if you pushed into it using the push/pop mode. The rest of the sheet (not selected) remains unchanged. The number of levels dissolved is the Dissolve Levels value in the Schematic Options dialog box. The type (filtered or unfiltered) of the resulting schematic is unchanged from that of the current schematic. However, the effect of the command is different in filtered and unfiltered schematics. Same as HDL Analyst -> Dissolve Instances – see Dissolve Instances, on page 3-59.

Buttons and Options

The Project view contains several buttons and a few additional features that give you immediate access to some of the more common commands and user options.

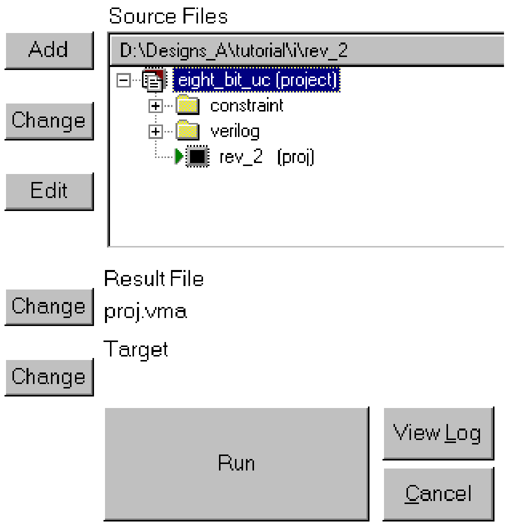


Figure 2-12: Project view buttons and options

The following table describes the Project View buttons and options.

Table 2-11: Project View Buttons and Options

Button/Option	Action
Frequency (Mhz)	Sets the global frequency, which you can override locally with attributes. Same as enabling the Frequency (Mhz) option on the Constraints panel of the Options for implementation dialog box.
FSM Compiler	Turning this on enables special FSM optimizations. Same as enabling the FSM Compiler option on the Options panel of the Options for implementation dialog box (see FSM Compiler, on page 2-13 and Using the Symbolic FSM Compiler, on page 5-17 in the *Synplify User Guide*).

Table 2-11: Project View Buttons and Options (Continued)

Button/Option	Action
Resource Sharing	When enabled, makes the synthesis use resource sharing techniques. This produces the resource sharing report in the log file (see Resource Usage Report, on page 4-10). Same as enabling the Resource Sharing option on the Options panel of the Options for implementation dialog box (see Sharing Resources, on page 5-5 in the *Synplify User Guide*).
Run	Runs synthesis (compilation and mapping). Same as the Run -> Synthesize command (see Run Menu, on page 3-43).
✉ Technical Resource Center	Goes to the web page for the Synplicity Technical Resource Center, which contains up-to-date product and technical information.

CHAPTER 3

User Interface Commands

This chapter describes the different commands and ways to access these commands in the graphical user interface (GUI).

These are the chapter topics:

Command Access

The product interface provides access to the commands through the following:

- Menus
- Context-sensitive Popup Menus
- Toolbars
- Keyboard Shortcuts
- Buttons and Options
- Tcl Command Equivalents

Menus

The set of commands on the pulldown menus in the menu bar varies depending on the view, design status, task to perform, and selected object(s). For example, the File menu commands in the Project view differ slightly from those in the RTL view. Menu commands that are not available for the current context are grayed out.

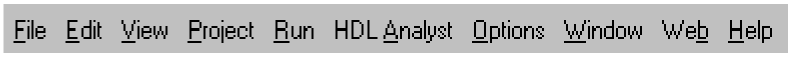


Figure 3-1: Project View Menu Bar

The menu can be moved by dragging it any where on the screen. You can either dock it to the edge of the application window or let it float it in its own window. To move the menu bar to the edge of the application window without docking it, hold the Ctrl or Shift key when dragging it.

The individual menus, their commands, and the associated dialog boxes are described in the following sections:

- File Menu, on page 3-4
- Edit Menu, on page 3-11
- View Menu, on page 3-20
- Project Menu, on page 3-28
- Run Menu, on page 3-43

- HDL Analyst Menu, on page 3-49
- Format Menu, on page 3-65
- Options Menu, on page 3-68
- Web Menu, on page 3-80
- Help Menu, on page 3-83

Context-sensitive Popup Menus

Popup menus, available by right-clicking, offer access to commonly used commands that are specific to the current context. See Popup Menus, on page 3-90, for information on individual popup menus.

Toolbars

Toolbars contain icons associated with commonly used commands. For more information about toolbars and customizing them, see Toolbars, on page 2-21 and Customize Dialog Box, on page 3-24.

Keyboard Shortcuts

Keyboard shortcuts are available for commonly used commands. The shortcut appears next to the command in the menu. See Keyboard Shortcuts, on page 2-28 for details.

Buttons and Options

The Project view has buttons for quick access to commonly used commands and options. See Buttons and Options, on page 2-36 for details.

Tcl Command Equivalents

Tcl (Tool Command Language) commands can be included in Tcl scripts that you can run in batch mode. For information about Tcl commands, see Chapter 5, *Tcl Commands and Scripts*.

File Menu

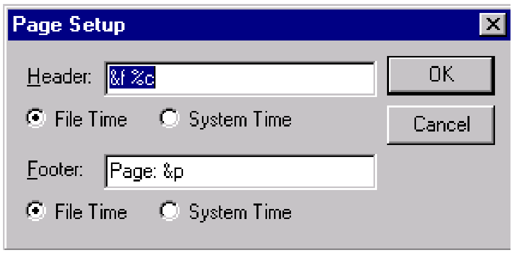
Use the File menu for opening, creating, saving, and closing projects and files. The following table describes the File menu commands.

Command	Description
New...	Creates one of the following types of files: Tcl script, VHDL, Verilog, constraint, and project. This command is specific to your operating system. See the operating system documentation for details.
Open...	Opens projects or file. This command is specific to your operating system. See the operating system documentation for details.
Close	Closes projects. This command is specific to your operating system. See the operating system documentation for details.
Save	Saves a project or a file. This command is specific to your operating system. See the operating system documentation for details.
Save As...	Saves a project or a file to a specified name. This command is specific to your operating system. See the operating system documentation for details.
Save All	Saves all projects or files. This command is specific to your operating system. See the operating system documentation for details.
Print...	Prints a file. For more information about printing, see the appropriate *License Configuration and Set Up* document for your platform and the operating system documentation.
Print Preview	Previews the file to be printed. In the Text Editor, only the lines of selected text are previewed for printing. This command is specific to your operating system. See the operating system documentation for details.
Print Setup...	Lets you specify print options. This command is specific to your operating system. See the operating system documentation for details.
Print Image	Available only in the *RTL* and *Technology views*. Prints an image of a region you define in the view. A camera pointer ( ) appears. You drag a selection rectangle around the region to print, then release the mouse button.

Command	Description
Page Setup...	Displays the Page Setup dialog box. Available only in the *Text Editor*, however settings apply to all views that support printing. Sets header and footer information. See Page Setup Command, on page 3-5.
Build Project...	Creates a new project based on the file open in the Text Editor (if active), or lets you choose files to add to a new project. See Build Project Command, on page 3-6.
Open Project...	Opens a project. See Open Project Command, on page 3-7.
New Project	Creates a new project. If a project is already open, it prompts you to save it before creating a new one. If you want to open multiple projects, select Allow multiple projects to be opened in the Project View dialog box. See Project View Options Command, on page 3-69.
Close Project	Closes the current project.
Recent Projects ->	Lists recently accessed projects. Choose a project listed in the submenu to open it.
Recent files (listed as separate menu items)	The last six files you opened, as separate menu items. Choose a file to open it.
Exit	Exits the session.

Page Setup Command

Select File -> Page Setup to display the Page Setup dialog box, where you enter Header and/or Footer variables. Choose from two Date and time variable formats: File Time, the date/time the file was last modified, or System Time the current date/time.



Header: &f %c
(● File Time) ○ System Time

Footer: Page: &p
(● File Time) ○ System Time

Figure 3-2: Page Setup dialog box (File -> Page Setup)

The following table contains the variable syntax to enter in the header and footer fields.

Variable	Description
&p	Page number.
&f	File name, including full path.
%c	Date (mo/day/yr) and time (hr:min:sec).
%#c	Long form of date and time (Sunday, October 31, 1999, 12:42:30).
%x	Date (mo/day/yr).
%X	Time (hr:min:sec).
%#x	Long form of date (Sunday, October 31, 1999).

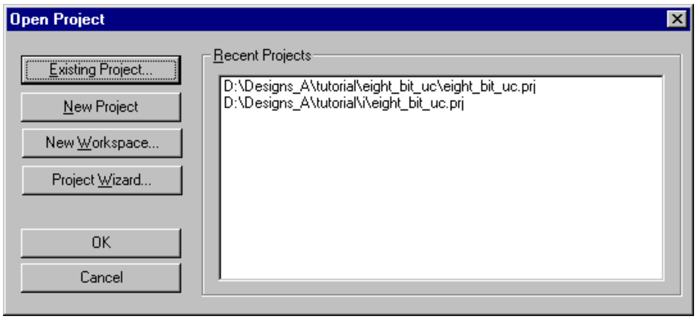
Build Project Command

Select File -> Build Project to build a new project. This command behaves differently if an HDL file is open in the Text Editor.

- When an active Text Editor window with an HDL file is open, File -> Build Project creates a project with the same name as the open file.
- If no file is open, File -> Build Project prompts you to add files to the project using the Select Files to Add to Project dialog box. The name of the new project is the name of the first HDL file added. See Add Source File Command, on page 3-40.

Open Project Command

Select File -> Open Project to open an existing project or create a new project or workspace.



D:\Designs_A\tutorial\eight_bit_uc\eight_bit_uc.prj
D:\Designs_A\tutorial\i\eight_bit_uc.prj

Field/Option	Description
Existing Project...	Displays the Open dialog box for opening an existing project.
New Project	Creates a new project and places it in the Project view.
Project Wizard...	Calls up the Project wizard, which helps you set up a new project. See Project Wizard, on page 3-7.

Project Wizard

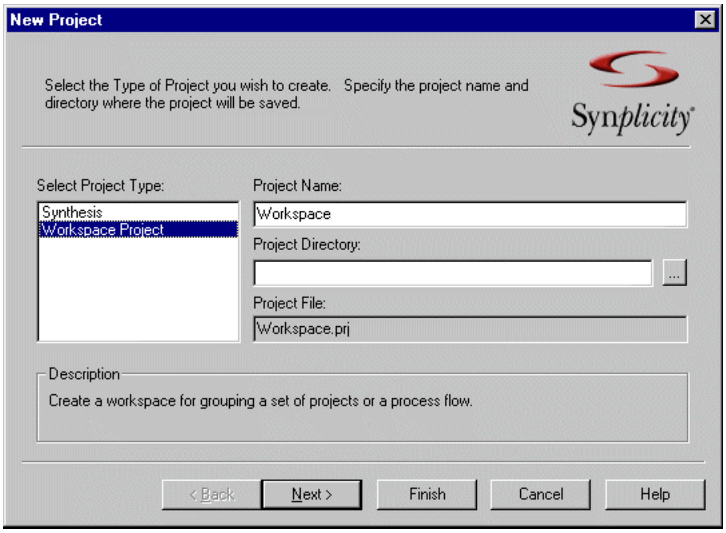
The Project wizard walks you through the steps needed to create a project. You can use the wizard to just create a project file, or use it to set implementation options and constraints as well. You can exit the wizard at any point with the Finish button. The wizard helps you do the following:

- Set up the project. See Set up the Project, on page 3-8.
- Add files. See Add Files in the Project Wizard, on page 3-8.
- Set Device Options. The features of this dialog box are the same as the Device panel of the Options for Implementation dialog box. The options available depend on the technology. Press F1 for technology-specific details.
- Set synthesis options. See Set Synthesis Options in the Project Wizard, on page 3-9.

- Set constraints. See Set Constraints in the Project Wizard, on page 3-9
- Set Verilog/VHDL source file options. See Set Language Options in the Project Wizard, on page 3-9.

Set up the Project

In the first dialog box select the type of project (Synthesis), give the project a name, then click Next.


New Project

Select the Type of Project you wish to create. Specify the project name and directory where the project will be saved.

Select Project Type:
Synthesis
Workspace Project

Project Name: Workspace

Project Directory:

Project File: Workspace.prj

Description
Create a workspace for grouping a set of projects or a process flow.

Feature	Description
Select Project Type	Select Synthesis to create a single project .
Project Name	Name of the Project or Workspace.

Add Files in the Project Wizard

Add files to the project. If you click Finish after you add the files, you will have a project set up. If you click Next, you can continue and set options and constraints.

Set Synthesis Options in the Project Wizard

This dialog box lets you set options for synthesis. The features of this dialog box are the same as the Options panel of the Options for Implementation dialog box and vary with the technology.

For more information about the options, see the following:

- Setting Implementation Options, on page 3-2 in the *Synplify User Guide*
- FSM Compiler, on page 2-13 and Using the Symbolic FSM Compiler, on page 5-17 in the *Synplify User Guide*

Set Constraints in the Project Wizard

Lets you set some timing constraints for the design. The features of this dialog are a subset of the Constraints panel of the Options for Implementation dialog box.

Frequency	Sets the global frequency. To set clock and other timing constraints, use the SCOPE interface.
Use clock period for unconstrained IO	To use only explicitly defined I/O constraints for synthesis, enable this option (default). To synthesize with all the constraints, using the clock period for all I/O paths that do not have an explicit constraint, disable the option.

Set Language Options in the Project Wizard

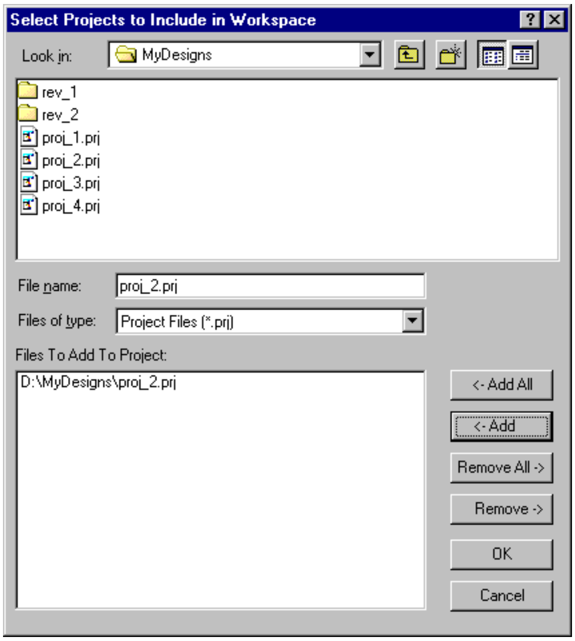
The features of this dialog box are a subset of the Verilog or VHDL panel of the Options for Implementation dialog box.

Option	Description
Top Level Entity/Module	The name of the top-level entity/module of your design.
Push Tristates	When enabled (default), tristates are pushed across process/block boundaries. For details, see Push Tristates Option, on page 3-38.
Use Verilog 2001 (Verilog only)	When enabled, the default Verilog standard for the project is Verilog 2001. When disabled, the default standard is Verilog 95. See Verilog Panel, on page 3-37 for more information

Option	Description
Include Path Order (Verilog only)	Specifies the search paths for the include commands in the Verilog design files of your project. Use the buttons in the upper right corner of the box to add, delete, or reorder the paths.
Default Enum Encoding (VHDL only)	The default enumeration encoding to use. See VHDL Panel, on page 3-35 for further details.

New Workspace Command

Select File -> New Workspace to display the Select Projects to Include in Workspace dialog box.



File name: proj_2.prj
Files of type: Project Files (*.prj)
Files To Add To Project: D:\MyDesigns\proj_2.prj

Figure 3-3: Select Projects to Include in Workspace dialog box

Field/Option	Description
File name	The name of a project file to add to the workspace. If you enter a name using the keyboard, then you must include the file extension.
Files of type	A description of file types, with wildcard expressions to match the file to add. Only project files (`.prj`) can be added to a workspace.
Files To Add To Project	Displays the project files to add to the workspace. You add files to this list with the <- Add and <- Add All buttons. You remove files from this list with the Remove -> and Remove All -> buttons.
<- Add All	Adds all of the files currently displayed in the directory to the Files to Add to Project list.
<- Add	Adds the file named in the File name field to the Files to Add to Project list.
Remove ->	Removes a selected file from the Files to Add to Project list.
Remove All ->	Removes all of the files from the Files to Add to Project list.

Edit Menu

You use the Edit menu to edit text files (such as HDL source files) in your project. This includes cutting, copying, pasting, finding, and replacing text; manipulating bookmarks; and commenting-out code lines. The Edit menu commands available at any time depend on the active window or view (Project, Text Editor, SCOPE spreadsheet, RTL, or Technology).

The available Edit menu commands vary, depending on your current view. The following table describes all the Edit menu commands:

Command	Description
Basic Edit Menu Commands	
Undo	Cancels the last action.
Redo	Performs the action undone by Undo.

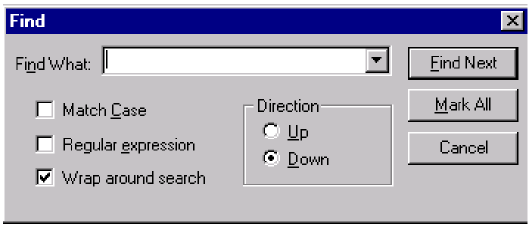
Command	Description
Cut	Removes the selected text and makes it available to Paste.
Copy	Duplicates the selected text and makes it available to Paste.
Paste	Pastes text that was cut (Cut) or copied (Copy).
Find...	Searches the file for text matching a given search string. See Find Command (Text), on page 3-14.
Find Again	Continues the search initiated by the last Find.
Edit Menu Commands for the Text Editor	
Select All	Selects all text in the file.
Replace...	Finds and replaces text. See Replace Command, on page 3-18.
Goto	Goes to a specific line number. See Goto Command, on page 3-19.
Advanced -> Comment Code	Inserts the appropriate comment prefix at the current text cursor location.
Advanced -> Uppercase	Makes the selected string all upper case.
Advanced -> Lowercase	Makes the selected string all lower case.
Toggle bookmark	Toggles between inserting and removing a bookmark on the line that contains the text cursor.
Next bookmark	Takes you to the next bookmark.
Previous bookmark	Takes you to the previous bookmark.
Delete all bookmarks	Removes all bookmarks from the Text Editor window.
Edit Menu Commands for the SCOPE Spreadsheet	
Cut/Disable	Removes the selected text and makes it available to Paste. When used in the Enabled column, it disables the checkbox.
Clear/Enable	Clears a row of information. When used in the Enabled column, it enables the checkbox.
Fill Down	Copies the information from a cell to selected cells below it in the same column.

Command	Description
Remove Rows	Removes selected rows.
Insert Row	Inserts a new row above the selected row. When no row is selected, inserts a new row at the bottom of the spreadsheet.
Insert Wizard...	Opens an Insert wizard to help you define constraints for a particular SCOPE panel, or attributes for an object. See SCOPE Constraint Wizards, on page 7-6, for descriptions of the constraint wizards, and SCOPE Attribute Wizard, on page 8-10, for information about the attributes wizard.
Insert Quick	Automatically inserts constraints for all objects that apply to the current SCOPE panel (for example all clocks in the Clocks panel). See Insert Quick Mode, on page 7-8.
Properties	Displays the Scope Properties dialog box, showing the name and path of the constraint and netlist files.
Edit Menu Commands for the RTL and Technology Views	
Copy Image	Lets you copy part or all of your schematic in the RTL or Technology view to the clipboard for subsequent pasting into a graphics program. When selected, a camera pointer ( ) appears. Click to copy the whole schematic, or drag a selection rectangle around the portion to copy.
Find...	Displays the Object Query dialog box, which lets you search your design for instances, symbols, nets, and ports, by name. See Find Command (HDL Analyst), on page 3-15.

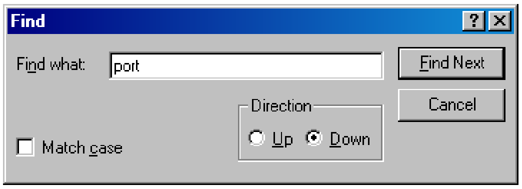
Find Command (Text)

Select Edit -> Find to display the Find dialog box. In the SCOPE window and the Text Editor window, the command has basic text-based search capabilities. Some search features, like wrap-around searching and line-number highlighting, are available only in the Text Editor.

The HDL Analyst Find command is different; see Find Command (HDL Analyst), on page 3-15 for details.



In Text Editor



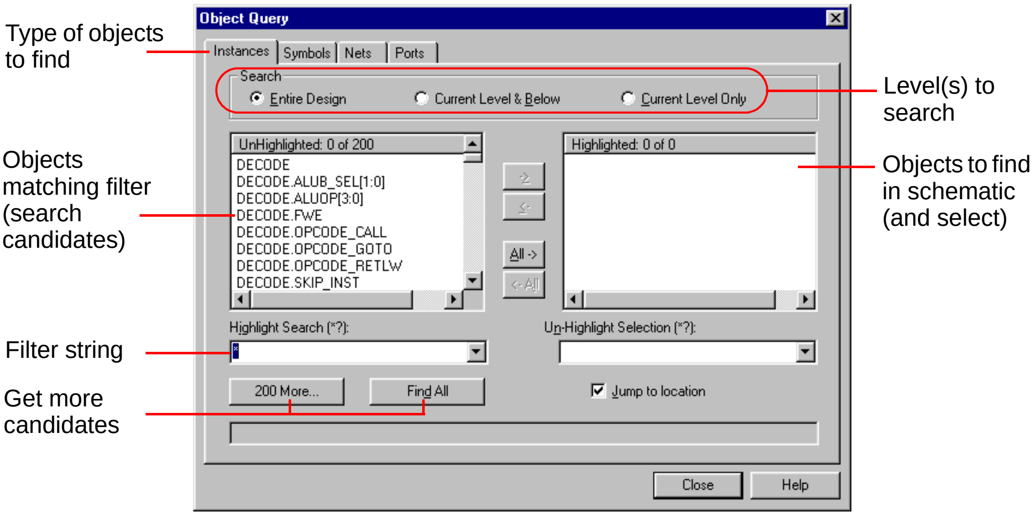
In SCOPE and FSM Viewer

Figure 3-4: Find dialog box (Edit -> Find)

Field/Option	Description
Find What	Search string matching the text to find. You can use the pulldown list to view and reuse search strings used previously in the current session.
Match Case	When enabled, searching is case sensitive.
Direction	Buttons for choosing the search direction (Up or Down).
Find Next	Initiates a search for the search string (see Find What).
Regular expression (Text editor only)	When enabled, wildcard characters (* and ?) can be used in the search string: ? matches any single character; * matches any string of characters, including the empty string.
Wrap around search (Text editor only)	When enabled, searching starts again after reaching the end (Down) or the beginning (Up) of the file.
Mark All (Text editor only)	Highlights the line numbers of the text matching the search string and closes the Find dialog box.

Find Command (HDL Analyst)

In the RTL or Technology view, use Edit -> Find to display the Object Query dialog box. For a detailed procedure about using this command, see see Using Find for Hierarchical and Restricted Searches, on page 4-32 of the *Synplify User Guide*.



Search: [✓ Entire Design] Current Level & Below Current Level Only

UnHighlighted: 0 of 200

```
DECODE
DECODE.ALUB_SEL[1:0]
DECODE.ALUOP[3:0]
DECODE.FWE
DECODE.OPCODE_CALL
DECODE.OPCODE_GOTO
DECODE.OPCODE_RETLW
DECODE.SKIP_INST
```

Highlighted: 0 of 0

Highlight Search (*?): *

[✓] Jump to location

Figure 3-5: Object Query dialog box (Edit -> Find)

Field/Option	Description
Instances, Symbols, Nets, Ports	Tabbed panels for finding different kinds of objects. Choose a panel for a given object type by clicking its tab. In terms of memory consumption, searching for Instances is most efficient, and searching for Nets is least efficient.
Search	Where to search: Entire Design, Current Level & Below or Current Level Only. See Using Find for Hierarchical and Restricted Searches, on page 4-32 of the *Synplify User Guide* for details.

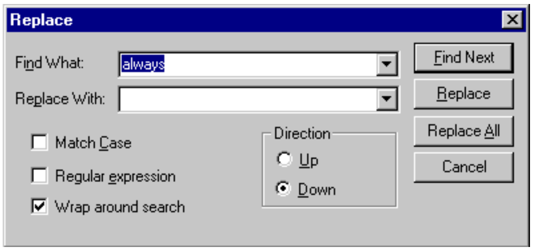
Field/Option	Description
UnHighlighted	Names of all objects of the current panel type, in the level(s) chosen to Search, that match the Highlight Search (*?) filter. This list is populated by the Find 200 and Find All buttons. To select an object as a candidate for highlighting, click its name in this list. The complete name of the selected object appears near the bottom of the dialog box. You can select part or all of this complete name, then use the Ctrl-C keyboard shortcut to copy it for pasting. You can select multiple objects by pressing the Ctrl or Shift key while clicking; press Ctrl and click a selection to deselect it. The number of objects selected, and the total number listed, are displayed above the list, after the UnHighlighted: label: # selected of # total. To confirm a selection for highlighting and move the selected objects to the Highlighted list, click the -> button.
Highlight Search (*?)	Determines which object names appear in the UnHighlighted area, based on the case-sensitive filter string that you enter. For tips about using this field, see Using Wildcards with the Find Command, on page 4-35 of the *Synplify User Guide.* The filter string can contain the following wildcard characters: • * (asterisk) – matches any sequence of characters; • ? (question mark) – matches any single character; • . (period) – does not match any characters, but indicates a change in hierarchical level. Wildcards * and ? only match characters within the current hierarchy level; a*b*, for example, will not cross levels to match alpha.beta (where the period indicates a change in hierarchy). If you must match a period character occurring in a name, use \. (backslash period) in the filter string. The backslash prevents interpreting the period as a wildcard. The filter string is matched at each searched level of the hierarchy (the Search levels are described above). Use filter strings that are as specific as possible to limit the number of unwanted matches. Unnecessarily extensive search can be costly in terms of memory performance.
->	Moves the selected names from the UnHighlighted area to the Highlighted area, and highlights their objects in the RTL and Technology views.
<-	Moves the selected names from the Highlighted area to the UnHighlighted area, and unhighlights their objects in the RTL and Technology views.

Field/Option	Description
All ->	Moves all names from the UnHighlighted to the Highlighted area, and highlights their objects in the RTL and Technology views.
<- All	Moves all names from the Highlighted to the UnHighlighted area, and unhighlights their objects in the RTL and Technology views.
Highlighted	Complementary and analogous to the UnHighlighted area. You select object names here as candidates for moving to the UnHighlighted list. (You move names to the UnHighlighted list by clicking the <- button; this unselects and unhighlights the corresponding objects.) When you select a name in the Highlighted list, the view is changed to show the (original, unfiltered) schematic sheet containing the object.
Un-Highlight Selection (*?)	Complementary and analogous to the Highlight Search area: selects names in the Highlighted area, based on the filter string you input here.
Jump to location	When enabled, jumps to another sheet if necessary to show target objects.
Find 200	Adds up to 200 more objects that match the filter string to the UnHighlighted list. This button becomes available after you enter a Highlight Search (*?) filter string. This button does not find objects in HDL Analyst views. It matches names of design objects against the Highlight Search (*?) filter and provides the candidates listed in the UnHighlighted list, from which you select the objects to find. Using the Enter (Return) key when the cursor is in the Highlight Search (*?) field is equivalent to clicking the Find 200 button. *Usage note:* Click Find 200 before Find All to prevent unwanted matches in case the Highlight Search (*?) string is less selective than you expect.
Find All	Places all objects that match the Highlight Search (*?) filter string in the UnHighlighted list. This button does not find objects in HDL Analyst views. It matches names of design objects against the Highlight Search (*?) filter and provides the candidates listed in the UnHighlighted list , from which you select the objects to find. (Enter a filter string before clicking this button.) See *Usage Note* for Find 200, above.

For more information on using the Object Query dialog box, see Using Find for Hierarchical and Restricted Searches, on page 4-32 of the *Synplify User Guide*.

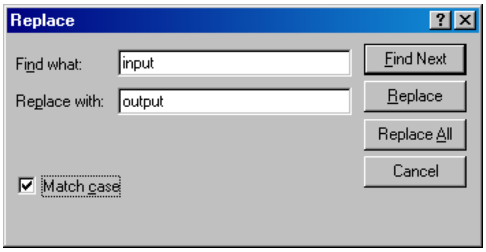
Replace Command

Use Edit -> Replace to find and optionally replace text in the Text Editor or SCOPE spreadsheet. Some options are only available in the Text Editor.



Find What: always
Replace With:
☐ Match Case
☐ Regular expression
☑ Wrap around search
Direction: ○ Up [✓ ● Down]

In Text Editor



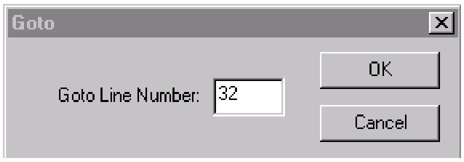
Find what: input
Replace with: output
☑ Match case

In SCOPE spreadsheet

Feature	Description
Find What	Search string matching the text to find. You can use the pulldown list to view and reuse search strings used previously in the current session.
Replace With	The text that replaces the found text. You can use the pulldown list to view and reuse replacement text used previously in the current session.
Match Case	When enabled, searching is case sensitive.
Find Next	Initiates a search for the search string (see Find What).
Replace	Replaces the found text with the replacement text, and locates the next match.
Replace All	Replaces all text that matches the search string.
Regular expression (Text editor only)	When enabled, wildcard characters (* and ?) can be used in the search string: ? matches any single character; * matches any string of characters, including the empty string.
Wrap around search (Text editor only)	When enabled, searching starts again after reaching the end (Down) or the beginning (Up) of the file.
Direction (Text editor only)	Buttons for choosing the search direction (Up or Down).

Goto Command

Use Edit -> Goto to go to a specified line number in the Text Editor.



Goto Line Number: 32

View Menu

Use the View menu to set the display and viewing options, choose toolbars, and display result files. The commands in the View menu vary with the active view. The following tables describe the View menu commands in various views.

Command	Description
View Menu Commands for All Views	
Toolbars...	Displays the Toolbars dialog box, where you specify the toolbars to display. See Toolbar Command, on page 3-22.
Status Bar	When enabled, displays context-sensitive information in the lower left corner of the main window as you move the mouse pointer over design elements. This information includes element identification.
Workbook Mode	When enabled, tabs appear near the bottom of the Project window allowing you to access open process views, such as an RTL view or SCOPE spreadsheet.
View Log File	Displays a log file that includes a net report and a performance summary on your design speed.
View Result File	Displays a detailed netlist report.
View Zoom Commands	
Zoom In Zoom Out	Lets you Zoom in or out. When selected, a Z-shaped mouse pointer ( ) appears. Zoom in or out on the view by clicking or dragging a box around (lassoing) the region. Clicking zooms in or out on the center of the view; lassoing zooms in or out on the lassoed region. Right-click to exit zooming mode. In the SCOPE spreadsheet, selecting these commands increases or decreases the view in small increments.
Pan	Lets you pan (scroll) a schematic or FSM view using the mouse.
Pan Center	Centers a clicked area of a schematic or FSM view in the window.
Full View	Zooms the active view so that it shows the entire design.

Command	Description
Normal View	Zooms the active view to normal size and centers it where you click. If the view is already normal size, clicking just centers the view.
Zoom Lock	When enabled, the displayed schematic or FSM viewer will resize proportionally with the window.
View Menu Commands for Project View, RTL and Technology Views	
Push/Pop Hierarchy	Traverses design hierarchy using the push/pop mode – see Traversing Design Hierarchy with the Hierarchy Browser, on page 4-24 of the *Synplify User Guide.*
Previous Sheet	Displays the previous sheet of a multiple-sheet schematic.
Next Sheet	Displays the next sheet of a multiple-sheet schematic.
View Sheets...	Displays the Goto Sheet dialog box where you can select a sheet to display from a list of all sheets. See View Sheets Command, on page 3-25.
Visual Properties	Toggles the display of information for nets, instances, pins, and ports in the HDL Analyst view. To customize the information that displays, set the values with Options->Schematic Options->Visual Properties. See Visual Properties Panel, on page 3-78.
Back	Goes backward in the history of displayed sheets for the current HDL Analyst view.
Forward	Goes forward in the history of displayed sheets for the current HDL Analyst view.
View Menu Commands for the Text Editor	
Show Line Numbers	Displays consecutive line numbers for lines of code in the file.
View Menu Commands for the SCOPE Spreadsheet	
ReadOnly	Locks a constraint panel in the SCOPE spreadsheet to prevent changes.
Properties...	Displays the Display Settings dialog box, where you define the SCOPE spreadsheet display settings. See Display Settings dialog box (View -> Properties, in SCOPE view), on page 3-26.

Toolbar Command

Select View -> Toolbars to display the Toolbars dialog box, where you choose the toolbars to display, customize their appearance and content, or create a new custom toolbar using the Customize dialog box.

You can edit icons in any of the existing toolbars, or you can create additional, custom toolbars with your own choice of icons.

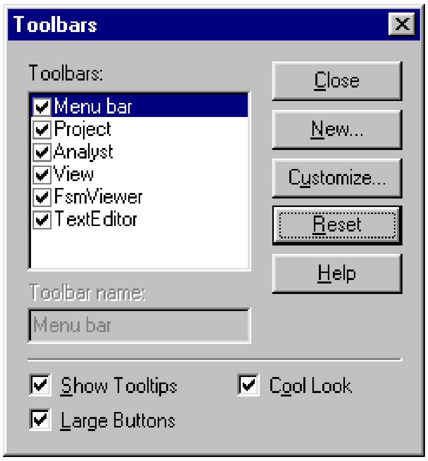


Figure 3-6: Toolbars dialog box (View -> Toolbars)

Table 3-1: Toolbars Dialog Box Features

Feature	Description
Toolbars	Lists the available toolbars (standard and custom). Select the toolbars that you want to display.
New...	Prompts for the name of a new custom toolbar. After you enter a name and click OK, the Customize dialog box appears where you add icons to the newly created toolbar. See Customize Dialog Box, on page 3-24.
Customize...	Displays the Customize dialog box, where you can choose the icons for each toolbar (standard or custom). See Customize Dialog Box, on page 3-24.
Reset	Resets a selected standard (predefined) toolbar to its default set of icons.

Table 3-1: Toolbars Dialog Box Features (Continued)

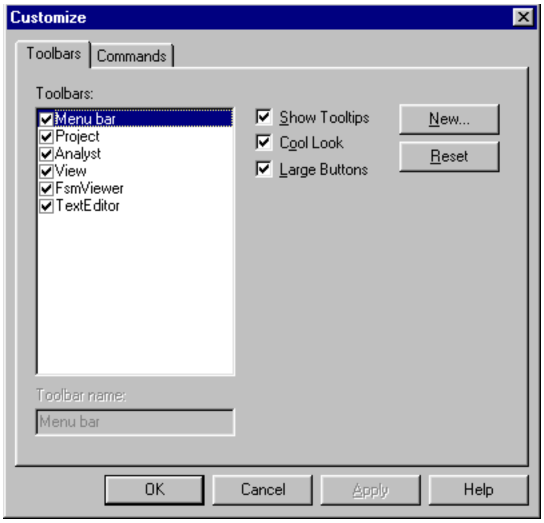
Feature	Description
Delete	Deletes a selected custom toolbar.
Toolbar name	Displays the name of the selected toolbar. You can edit the names of custom toolbars.
Show Tooltips	When selected, a descriptive tooltip appears whenever you position the pointer over an icon.
Cool Look	When selected, icons are outlined only when you pass the mouse pointer over them; otherwise, they blend with the background. When disabled, icons are always outlined.
Large Buttons	When selected, large icons are used.

Customize Dialog Box

Do one of the following to display the Customize dialog box:

- Select View -> Toolbars and click the Customize button in the dialog box
- Enter a custom toolbar name after clicking New in the Toolbars dialog box
- Select Options->Customize and select the Toolbars tab.

To choose icons for a toolbar, select the Commands tab of the Customize dialog box. Drag and drop individual icons to or from any displayed toolbars (standard or custom). The Toolbars panel of the dialog box offers the same operations as the Toolbars dialog box (see Toolbar Command, on page 3-22).

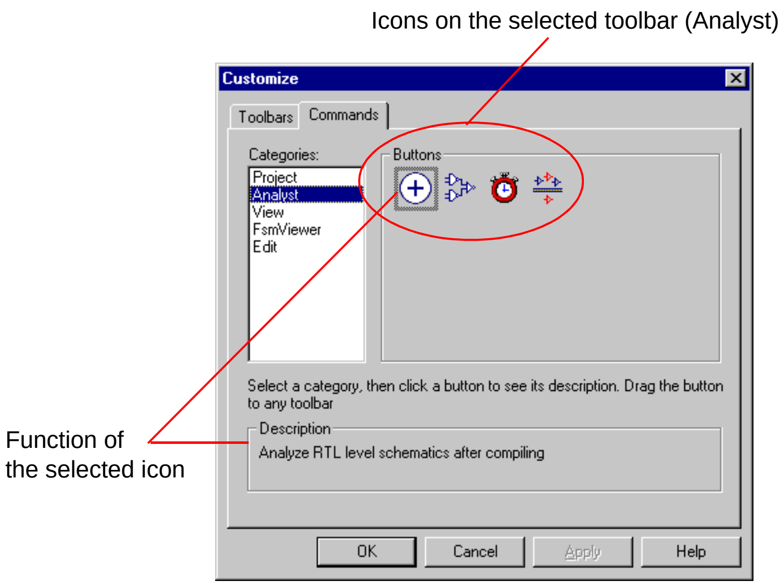


Toolbars:
- ✓ Menu bar
- ✓ Project
- ✓ Analyst
- ✓ View
- ✓ FsmViewer
- ✓ TextEditor

✓ Show Tooltips
✓ Cool Look
✓ Large Buttons

Toolbar name: Menu bar

Figure 3-7: Toolbars panel, Customize dialog box



Select a category, then click a button to see its description. Drag the button to any toolbar

Analyze RTL level schematics after compiling

Figure 3-8: Commands panel, Customize dialog box

View Sheets Command

Select View -> View Sheets to display the Goto Sheet dialog box and select a sheet to display. The Goto Sheet dialog box is only available in an RTL or Technology view, and only when a multiple-sheet design is present.

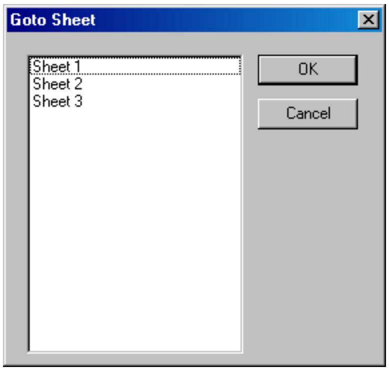
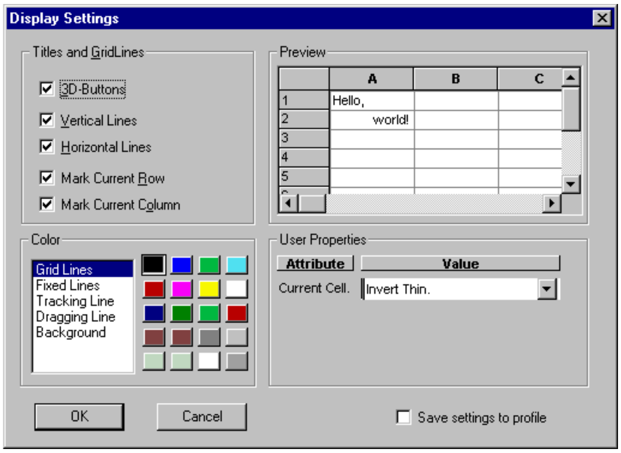


Figure 3-9: Goto Sheet dialog box (View -> View Sheets)

To see if your design has multiple sheets, check the sheet count display at the top of the schematic window.

Properties Command (SCOPE)

Select View -> Properties in the SCOPE spreadsheet to show the Display Settings dialog box, where you can edit row, line, button settings, and colors for table lines. The Preview window allows you to see the design changes before you commit to them.



Display Settings

Titles and GridLines

- ✓ 3D-Buttons
- ✓ Vertical Lines
- ✓ Horizontal Lines
- ✓ Mark Current Row
- ✓ Mark Current Column

Preview

	A	B	C
1	Hello,		
2	world!		
3			
4			
5			

Color

- Grid Lines
- Fixed Lines
- Tracking Line
- Dragging Line
- Background

User Properties

Attribute	Value
Current Cell.	Invert Thin.

☐ Save settings to profile

Figure 3-10: Display Settings dialog box (View -> Properties, in SCOPE view)

The following table describes the fields and options.

Feature	Description
Titles and Gridlines	3D-Buttons: When enabled, row numbers and column headings are displayed as buttons that appear 3-dimensional. Vertical Lines: When enabled, vertical lines are shown between columns. Horizontal Lines: When enabled, horizontal lines are shown between rows. Mark Current Row: When enabled, the selected row number is highlighted; for example, its 3D button appears depressed. Mark Current Column: When enabled, the selected column heading is highlighted; for example, its 3D button appears depressed.
Color	Sets the color for the selected spreadsheet feature. For example, if Grid Lines is selected, then horizontal and vertical grid lines are set to appear in the selected color.
User Properties	Sets the value of the indicated Attribute to the Value you choose in the pulldown list.
Save settings to profile	When enabled, saves any settings you made in the dialog box, for subsequent synthesis tool sessions.

Project Menu

You use the Project menu to set implementation options, add or remove files from a project, change project filenames, and create new implementations. The Project menu commands are the same in all views.

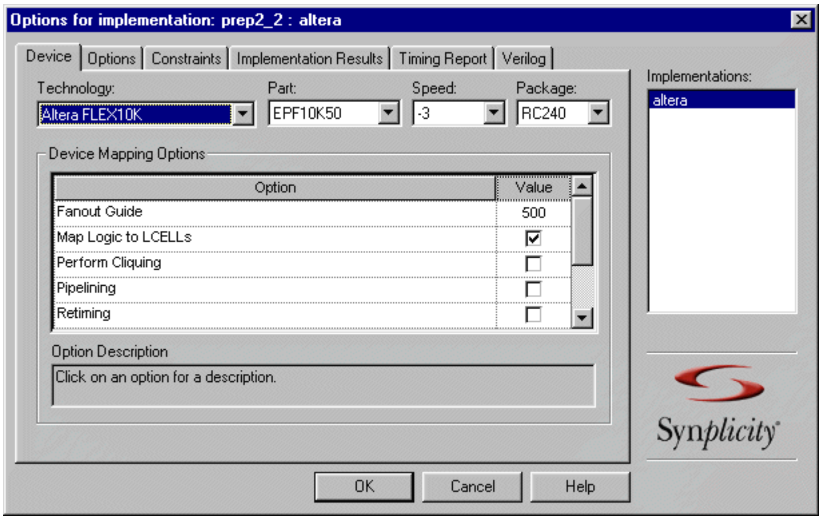
The following table describes the Project menu commands.

Table 3-2: Project Menu Commands

Command	Description
Implementation Options...	Displays the Options for implementation dialog box, where you set options for implementing your design. See Implementation Options Command, on page 3-29.
Add Source File...	Displays the Select Files to Add to Project dialog box. See Add Source File Command, on page 3-40.
Remove Files From Project	Removes selected files from your project.
Change File...	Replaces the selected file in your project with another that you choose. See Change File Command, on page 3-42.
Set VHDL Library... (Set Library...)	Displays the File Options dialog box, where you choose the library (Library Name) for synthesizing VHDL files. The default library is called work. See Set VHDL Library Command, on page 3-43.
New Implementation...	Creates a new implementation for a current design. Each implementation pertains to the same design, but it can have different options settings and/or constraints for synthesis runs. See New Implementation Command, on page 3-43).

Implementation Options Command

You use the Options for implementation dialog box to define the implementation options for the current project. You can access this dialog box from Project -> Implementation Options, by clicking the button in the Project view, or by clicking the text in the Project view that lists the current technology options. If you use the Project wizard, it brings up a subset of these options.



Technology: Altera FLEX10K
Part: EPF10K50
Speed: -3
Package: RC240

Implementations: altera

Device Mapping Options

Option	Value
Fanout Guide	500
Map Logic to LCELLs	✓
Perform Cliquing	☐
Pipelining	☐
Retiming	☐

Option Description

Click on an option for a description.

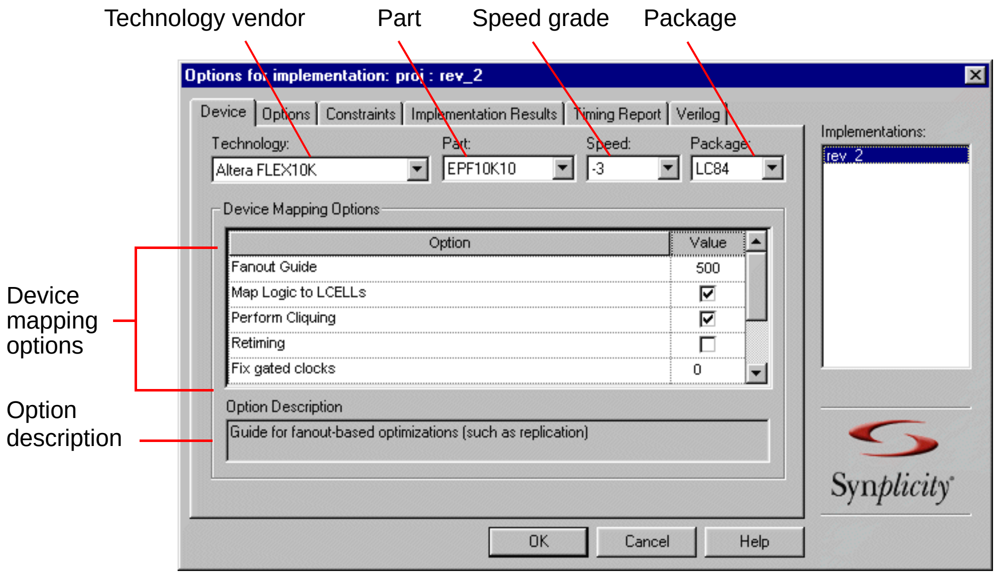
Figure 3-11: Options for implementation dialog box

The dialog box includes the following panels:

- Device Panel, on page 3-30
- Options Panel, on page 3-30
- Constraints Panel, on page 3-31
- Implementation Results Panel, on page 3-32
- Timing Report Panel, on page 3-34
- VHDL Panel, on page 3-35
- Verilog Panel, on page 3-37

Device Panel

You use the Device panel to set mapping options for the selected technology. The options vary, depending on the technology. See Setting Device Options, on page 3-2 in the *Synplify User Guide* and the vendor sections in this reference manual for details.



Technology: Altera FLEX10K
Part: EPF10K10
Speed: -3
Package: LC84

Implementations: rev_2

Device Mapping Options

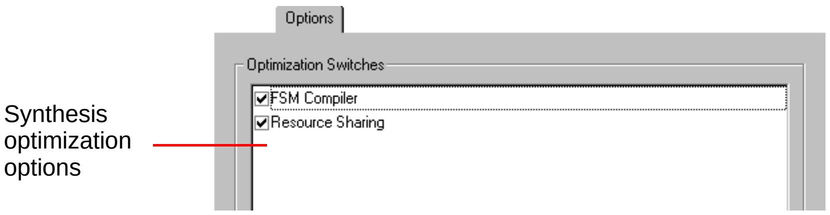
Option	Value
Fanout Guide	500
Map Logic to LCELLs	✓
Perform Cliquing	✓
Retiming	☐
Fix gated clocks	0

Option Description
Guide for fanout-based optimizations (such as replication)

Figure 3-12: Device panel, Options for Implementation dialog box

Options Panel

You use the Options panel of the Options for implementation dialog box to define general options for synthesis optimization. See Setting Constraint and Optimization Options, on page 3-5 of the *Synplify User Guide* for details.



Optimization Switches
☑ FSM Compiler
☑ Resource Sharing

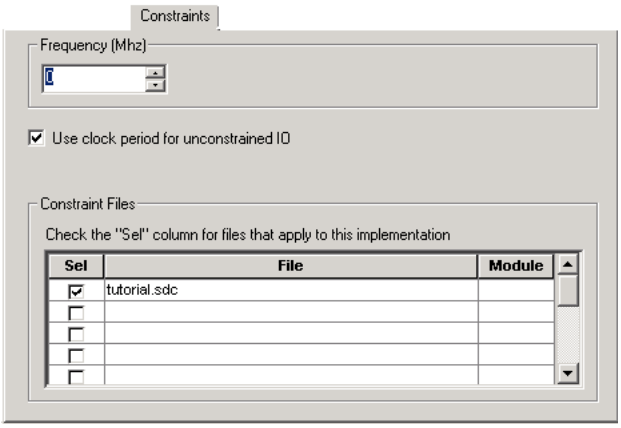
Figure 3-13: Options panel, Options for Implementation dialog box

The following table shows you where to find more information for each option:

Option	See...
FSM Compiler	FSM Compiler, on page 2-13 Using the Symbolic FSM Compiler, on page 5-17 in the *Synplify User Guide*
Resource Sharing	Sharing Resources, on page 5-5 in the *Synplify User Guide*

Constraints Panel

You use the Constraints panel of the Options for implementation dialog box to specify target frequency and timing constraint files for design synthesis. See Setting Constraint and Optimization Options, on page 3-5 of the *Synplify User Guide* for details.



0

☑ Use clock period for unconstrained IO

Check the "Sel" column for files that apply to this implementation

Sel	File	Module
☑	tutorial.sdc	
☐		
☐		
☐		
☐		

Option	Description
Frequency	Sets the default global frequency. You can either set the global frequency here or in the Project view. To override the default you set here, set individual clock constraints from the SCOPE interface.

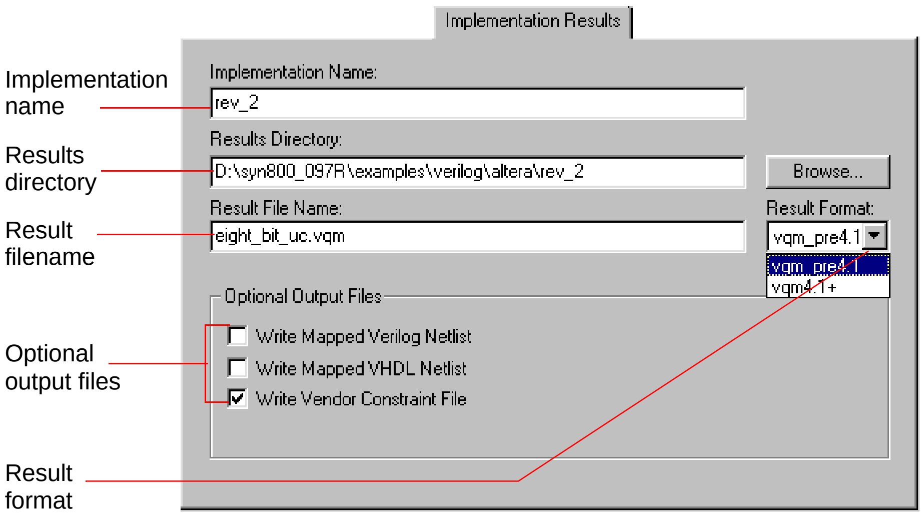
Option	Description
Use clock period for unconstrained IO	Determines whether default constraints are used for I/O ports that do not have user-defined constraints. When disabled, only define_input_delay or define_output_delay constraints are considered during synthesis or forward-annotated after synthesis. When enabled, the software considers any explicit define_input_delay or define_output_delay constraints, as before. In addition, for all ports without explicit constraints, it uses constraints based on the clock period of the attached registers. Both the explicit and implicit constraints are used for synthesis and forward-annotation. The default is off (disabled) for new designs. For designs created with older versions of the software (before Synplify 7.3) the default is on, to maintain backwards compatibility.
Constraint Files	Specifies which constraint files to use for the implementation. Enable the checkbox to select a file.

Implementation Results Panel

You use the Implementation Results panel to specify the implementation name (default: rev_1), the results directory, and the name and format of the top-level output netlist file (Result File). You can also specify output constraint and netlist files. See Specifying Result Options, on page 3-6 of the *Synplify User Guide* for details.

The results directory is a subdirectory of the project file directory. Clicking the Browse button brings up the Select Run Directory dialog box to allow you to browse for the results directory. You can change the location of the results directory, but its name must be identical to the implementation name.

Enable optional output file check boxes to generate the corresponding Verilog netlist, VHDL netlist, or vendor constraint files.



Implementation Name: rev_2

Results Directory: D:\syn800_097R\examples\verilog\altera\rev_2

Result File Name: eight_bit_uc.vqm

Result Format: vqm_pre4.1 (options: vqm_pre4.1, vqm4.1+)

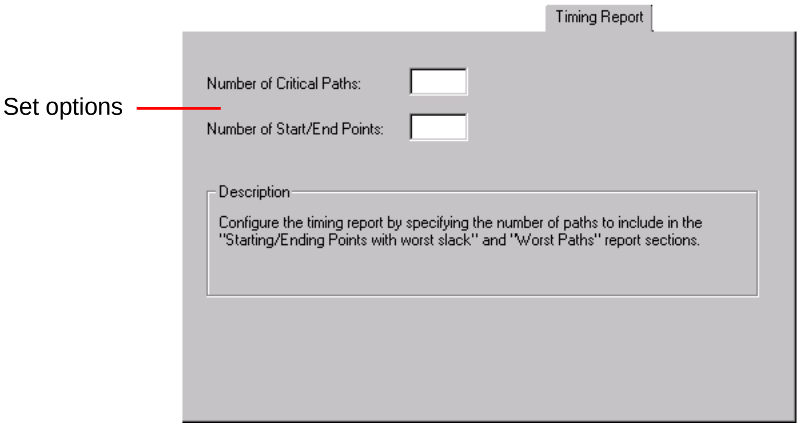
Optional Output Files

- [ ] Write Mapped Verilog Netlist
- [ ] Write Mapped VHDL Netlist
- [x] Write Vendor Constraint File

Figure 3-14: Implementation Results panel, Options for Implementation dialog box

Timing Report Panel

You use the Timing Report panel to set criteria for the (default) output timing report. You specify the number of critical paths and the number of start and end points to appear in the timing report. See Specifying Timing Report Output, on page 3-8 of the *Synplify User Guide* for details.



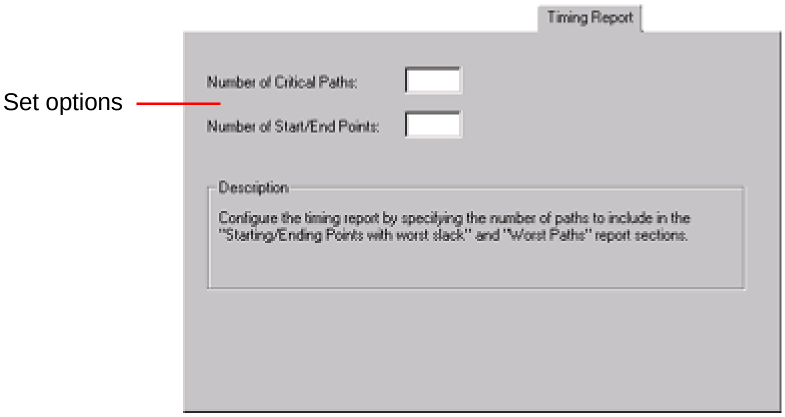
Number of Critical Paths:

Number of Start/End Points:

Description

Configure the timing report by specifying the number of paths to include in the "Starting/Ending Points with worst slack" and "Worst Paths" report sections.

Figure 3-15: Timing Report panel, Options for implementation dialog box



Number of Critical Paths:

Number of Start/End Points:

Description

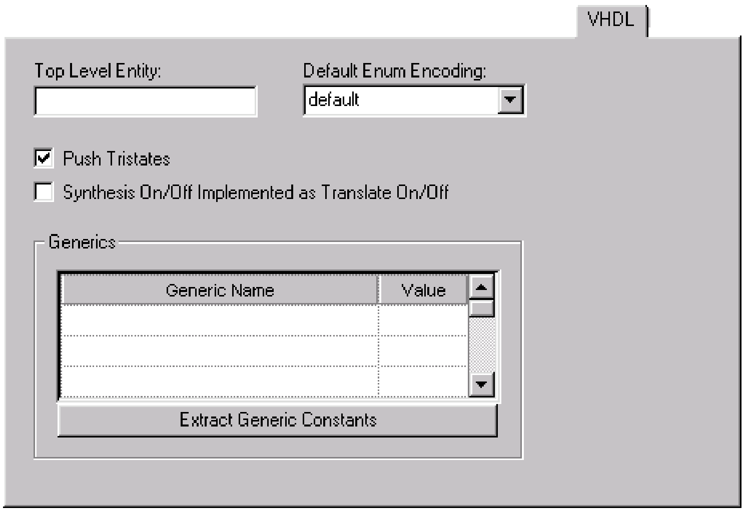
Configure the timing report by specifying the number of paths to include in the "Starting/Ending Points with worst slack" and "Worst Paths" report sections.

See also:

- Timing Report, on page 7-47, for more information on the default timing report, which is affected by the Timing Report panel settings.

VHDL Panel

You use the VHDL panel in the Options for implementation dialog box to specify various language-related options. With mixed HDL designs, the VHDL and Verilog panels are both available. See Setting Verilog and VHDL Options, on page 3-9 of the *Synplify User Guide* for details.

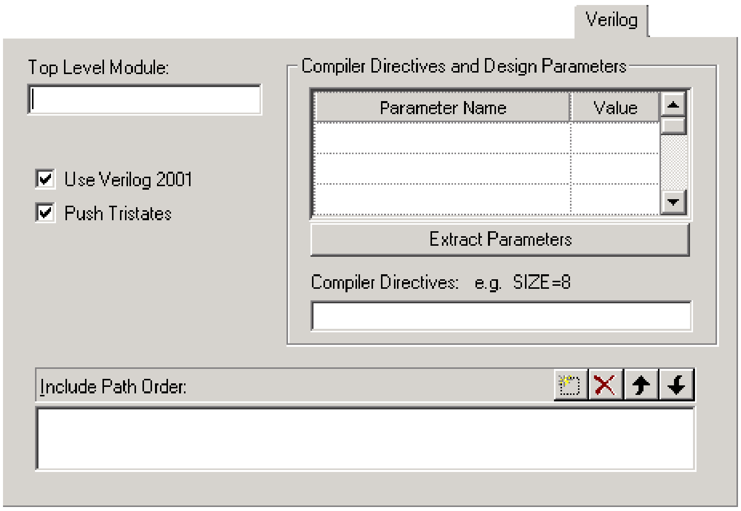


The following table describes the options available:

Feature	Description
Top Level Entity	The name of the top-level entity of your design.
Default Enum Encoding	The default enumeration encoding to use. This is only for enumerated types; the FSM compiler automatically determines the state-machine encoding, or you can specify the encoding using the syn_encoding attribute.
Push Tristates	When enabled (default), tristates are pushed across process/block boundaries. For more information, see Push Tristates Option, on page 3-38.
Synthesis On/Off Implemented as Translate On/Off	When enabled, the software ignores any VHDL code between synthesis_on and synthesis_off directives. It treats these third-party directives like translate_on/translate_off directives (see translate_off/translate_on, on page 8-103 for details).
Generics	Shows generics extracted with Extract Generic Constants. You can override the default and set a new value for the generic constant. The value is valid for the current implementation.
Extract Generic Constants	Extracts generics from the top-level entity and displays them in the table.

Verilog Panel

You use the Verilog panel in the Options for implementation dialog box to specify various language-related options. With mixed HDL designs, the VHDL and Verilog panels are both available. See Setting Verilog and VHDL Options, on page 3-9 of the *Synplify User Guide* for details.



Feature	Description
Top Level Module	The name of the top-level module of your design.
Compiler Directives and Design Parameters	Shows design parameters extracted with Extract Design Parameters. You can override the default and set a new value for the parameter. The value is valid for the current implementation.
Extract Design Parameters	Extracts design parameters from the top-level module and displays them in the table.

Compiler Directives	Provides an interface where you can enter compiler directives that you would normally enter in the code with `‘ifdef` and `‘define` statements. Use spaces to separate the statements. The directives you enter are stored in the project file. For example, if you enter
	`size=32 test_impl`
	The software writes the following statements to the project file:
	`set_option -hdl_define -set "size=32 test_impl"`

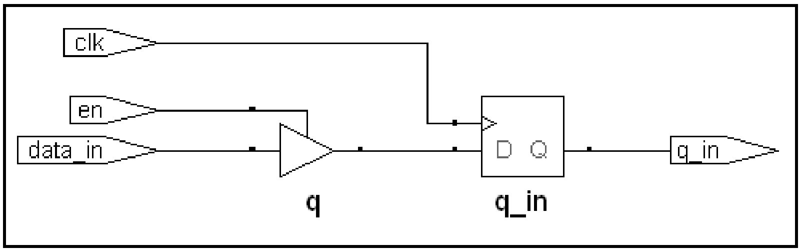
Feature	Description
Use Verilog 2001	When enabled, the default Verilog standard for the project is Verilog 2001. When disabled, the default standard is Verilog 95. For information about Verilog 2001, see Verilog 2001 Support, on page 9-4. You can override the default project standard on a per file basis, by selecting the file, right-clicking, and selecting the File Options command (see File Options Command, on page 3-93).
Push Tristates	When enabled (default), tristates are pushed across process/block boundaries. For details, see Push Tristates Option, on page 3-38.
Include Path Order	Specifies the search paths for the include commands in the Verilog design files of your project. Use the buttons in the upper right corner of the box to add, delete, or reorder the paths.

Push Tristates Option

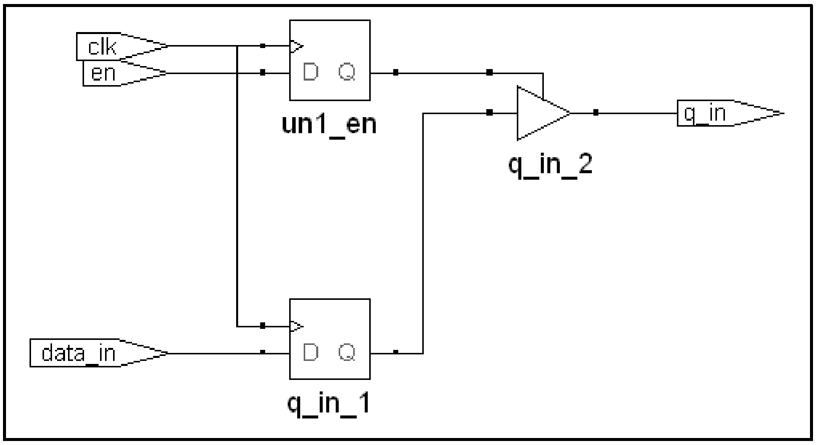
Pushing tristates is a synthesis optimization option you set with Project -> Implementation Options -> Verilog or VHDL.

Description

When the Push Tristates option is enabled, the Synplify compiler pushes tristates through objects such as muxes, registers, latches, buffers, nets, and tristate buffers, and propagates the high impedance state. The high-impedance states are not pushed through combinational gates such as ANDs or ORs.



Push Tristates off: tristate is not pushed through the flip-flop



Push Tristates on: tristate is pushed through the flip-flop so that the result matches RTL simulation

If there are multiple tristates, the software muxes them into one tristate and pushes it through. The software pushes tristates through loops and stores the high impedance across multiple cycles in the register.

Advantages and Disadvantages

The advantage to pushing tristates to the periphery of the design is that you get better timing results because the software uses tristate output buffers.

The Synplify software approach to tristate inference matches the simulation approach. Simulation languages are defined to store and propagate 0, 1, and Z (high impedance) states. Like the simulation tools, the Synplify synthesis tool propagates the high impedance states instead of producing tristate drivers at the outputs of process (VHDL) or always (Verilog) blocks.

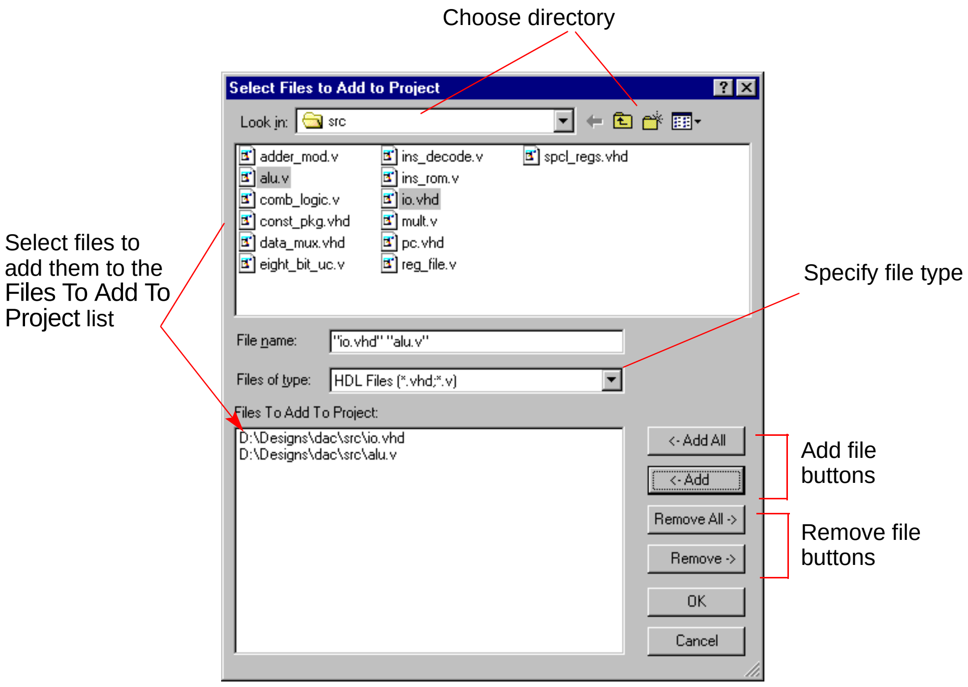
The disadvantage to pushing tristates is that you might use more design resources.

Effect on Other Synthesis Options

Tristate pushing has no effect on the syn_tristatetomux attribute. This is because tristate pushing is a compiler directive, while the syn_tristatetomux attribute is used during mapping.

Add Source File Command

Select Project -> Add Source File to add source files to your project. This displays the Select Files to Add to Project dialog box. You can also access this dialog box as part of the project wizard.



File name: "io.vhd" "alu.v"

Files of type: HDL Files (*.vhd;*.v)

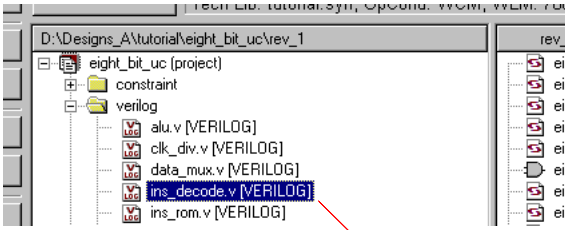
Files To Add To Project:

```
D:\Designs\dac\src\io.vhd
D:\Designs\dac\src\alu.v
```

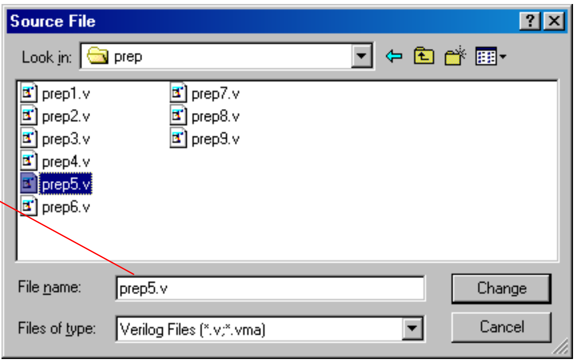
Feature	Description
Look in	The directory of the file to add. You can use the pulldown directory list or the Up One Level button to choose the directory.
File name	The name of a file to add to the project. If you enter a name using the keyboard, then you must include the file-type extension.
Files of type	A description of file types, with wildcard expressions to match the file to add. Only files in the chosen directory that match the expressions are displayed in the list of files. You can use the pulldown list to choose the file types to match.
Files To Add To Project	The files to add to the project. You add files to this list with the <- Add and <- Add All buttons. You remove files from this list with the Remove -> and Remove All -> buttons.
<- Add All	Adds all of the files currently displayed in the directory to the Files to Add to Project list.
<- Add	Adds the file named in the File name field to the Files to Add to Project list.
Remove ->	Removes a selected file from the Files to Add to Project list.
Remove All ->	Removes all of the files from the Files to Add to Project list.

Change File Command

Select Project -> Change File to replace a file in the project files list with another of the same type. This displays the Source File dialog box, where you specify the replacement file. You must first select the file to replace, in the Project view, before you can use this command.



First, select a file in the Project view



File name: prep5.v

Files of type: Verilog Files (*.v;*.vma)

Then, choose the replacement file

Set VHDL Library Command

Select Project -> Set VHDL Library to display the File Options dialog box, where you view VHDL file properties and specify the VHDL library name. See File Options Command, on page 3-93. This is the same dialog box as that displayed by right-clicking a VHDL filename in the Project view and choosing File Options.

New Implementation Command

Select Project -> New Implementation to create a new implementation for the selected project. This displays the Options for implementation dialog box, where you define the implementation options for the project – see Implementation Options Command, on page 3-29. This is the same dialog box as that displayed by Project -> Implementation Options, except that there is no list of Implementations to the right of the tabbed panels.

Run Menu

You use the Run menu to perform tasks such as the following:

- Compile a design, without mapping it.
- Synthesize (compile and map) or resynthesize a design
- Check design syntax and synthesis.code, and check source code errors
- Run Tcl scripts.
- Check the status of the current job.
- Launch the Identify Instrumentor tool.

The following table describes the Run menu commands.

Table 3-3: Run Menu Commands

Command	Description
Synthesize	Synthesizes (compiles and maps) the design. You can view the result of design synthesis in the RTL and Technology views. Same as clicking the Run button in the Project view.
Resynthesize All	Resynthesizes (compiles and maps) the entire design, including the top level and *all compile points*, whether or not their constraints, implementation options or source code changed since the last synthesis. If you do *not* want to force a *recompilation of all compile points*, then use Run -> Synthesize instead.
Compile Only	Compiles the design into technology-independent high-level structures. You can view the result in the RTL view.
Syntax Check	Runs a syntax check on design code. The status bar at the bottom of the window displays any error messages. If the active window shows an HDL file, then the command checks only that file; otherwise, it checks all project source code files.
Synthesis Check	Runs a synthesis check on your design code. This includes a syntax check and a check to see if the synthesis tool could map the design to the hardware. No optimizations are carried out. The status bar at the bottom of the window displays any error messages. If the active window shows an HDL file, then the command checks only that file; otherwise, it checks all project source code files.
Arrange VHDL files	Reorders the source files so you don't have to manually specify the top-level entry file.
Run Tcl Script...	Displays the Open dialog box, where you choose a Tcl script to run. See Run Tcl Script Command, on page 3-45.
Job Status	During compilation, tells you the name of the current job, and gives you the runtime and directory location of your design. This option is enabled during synthesis. See Job Status Command, on page 3-46. Clicking in the status area of the Project view is a shortcut for this command.

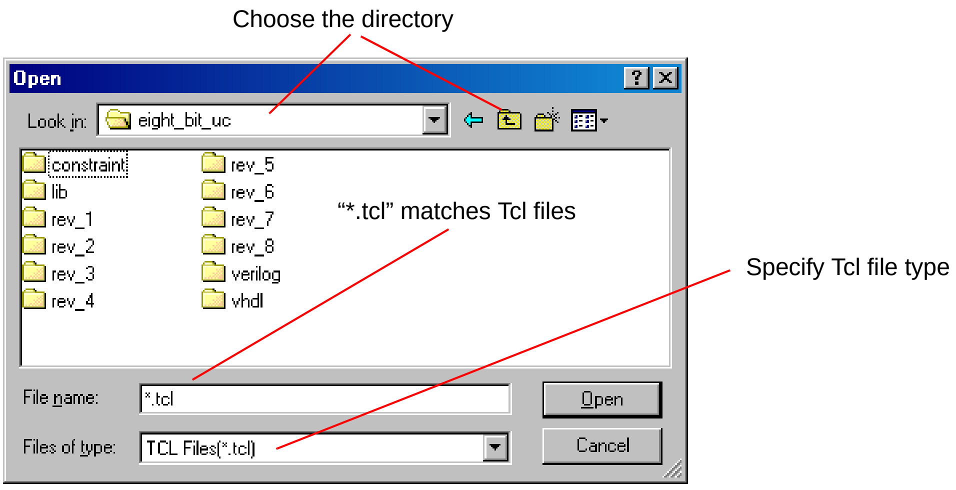
Table 3-3: Run Menu Commands (Continued)

Command	Description
Next Error/Warning	Shows the next error or warning in your source code file.
Previous Error/Warning	Shows the previous error or warning in your source code file.
Launch Identify Instrumentor	Launches the Identify Instrumentor software. See Launch Identify Instrumentor Command, on page 3-46.

Run Tcl Script Command

Select Run -> Run Tcl Script to display the Open dialog box, where you specify the Tcl script file to execute. The File name area is filled automatically with the wildcard string "*.tcl", corresponding to Tcl files.

This dialog box is the same as that displayed with File -> Open, except that no Open as read-only check box is present. See Open Project Command, on page 3-7, for an explanation of the features in the Open dialog box.



File name: *.tcl

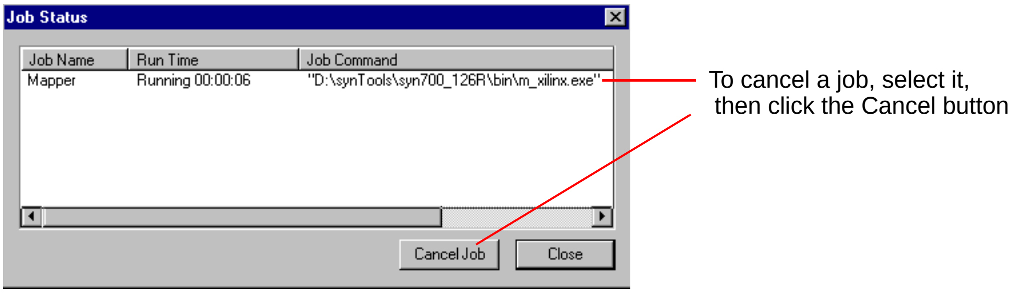
Files of type: TCL Files(*.tcl)

Figure 3-16: Open dialog box (Run -> Run Tcl Script)

Job Status Command

Select Run -> Job Status to monitor the synthesis jobs that are running, their run times, and their associated commands. This information appears in the Job Status dialog box. This dialog box is also displayed when you click in the status area of the Project view (see Project View, on page 2-3).

You can cancel a displayed job by selecting it in the dialog box and clicking Cancel Job.



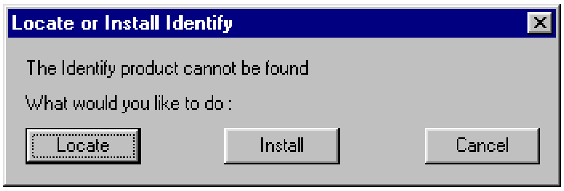
Job Name	Run Time	Job Command
Mapper	Running 00:00:06	"D:\synTools\syn700_126R\bin\m_xilinx.exe"

Figure 3-17: Job Status dialog box

Launch Identify Instrumentor Command

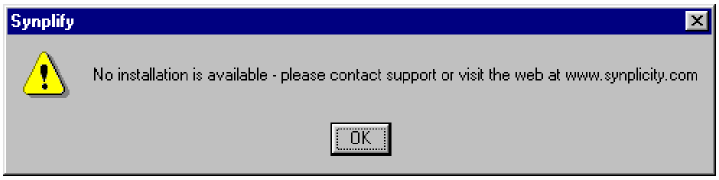
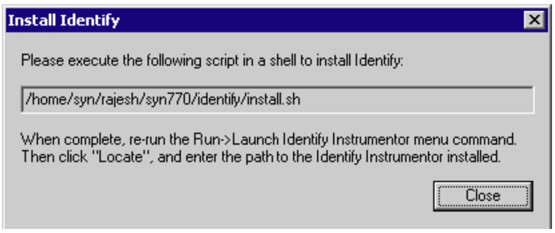
This command lets you start the Identify software from within the synthesis interface. For a description of the work flow using the Identify software, see Working with the Identify RTL Debugger, on page 7-16 in the *User Guide.*

- If the Identify software is installed, this command opens the Launch Identify dialog box, which lets you start the Identify software. See Launch Identify Dialog Box, on page 3-48.
- If the software is not installed, the command opens the Locate or Install Identify dialog box, where you can locate or install the Identify software.

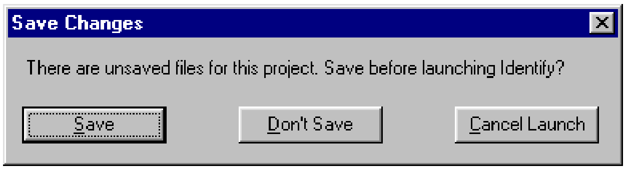


The Identify product cannot be found

What would you like to do :

Command	Description
Locate	Opens the Launch Identify dialog box. See Launch Identify Dialog Box, on page 3-48. Use this if you have an Identify installation available. If you use this option when there is no Identify installation available, a dialog box asks you to contact support to download an evaluation copy.
Install	For the PC, opens an informational dialog directing you to the Synplicity website and technical support.  Synplify — No installation is available - please contact support or visit the web at www.synplicity.com — OK. For UNIX, opens the Install Identify dialog box displaying an installation script to run in a shell window.  Install Identify — Please execute the following script in a shell to install Identify: /home/syn/rajesh/syn770/identify/install.sh — When complete, re-run the Run->Launch Identify Instrumentor menu command. Then click "Locate", and enter the path to the Identify Instrumentor installed. — Close

- If the software is installed but you have unsaved changes in your synthesis project, a Save Changes dialog box appears.

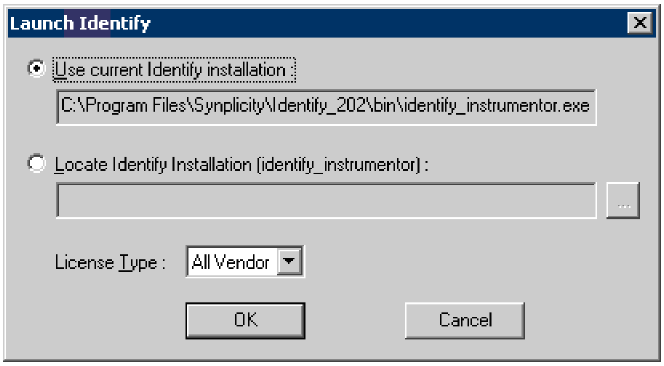

Save Changes

There are unsaved files for this project. Save before launching Identify?

Command	Description
Save	Saves the modified files in your synthesis project and launches the Identify tool.
Don’t Save	Launches the Identify tool without saving your synthesis project changes.
Cancel Launch	Cancels the Launch Identify command and takes you back to your synthesis project.

Launch Identify Dialog Box

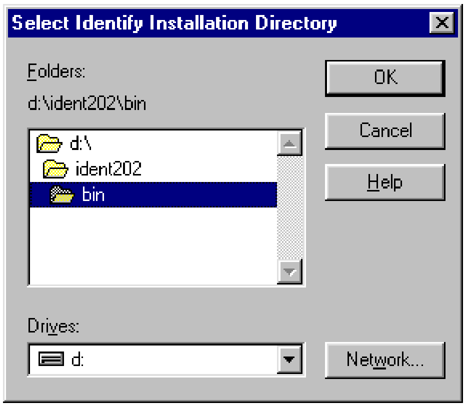
The Launch Identify dialog box lets you specify the location of the Identify software.



(● Use current Identify installation :) C:\Program Files\Synplicity\Identify_202\bin\identify_instrumentor.exe

○ Locate Identify Installation (identify_instrumentor) :

License Type : All Vendor

Command	Description
Use current Identify installation:	This command is activated when the application locates an existing installation of Identify.
Locate Identify Installation (identify_instrumentor)	Enable this command and select the (...) button to find the bin directory in the Identify installation which contains the instrumentor executable. 
License Type	This pull-down menu lists the vendors available to license with the Identify product.
Run	Launches the Identify Instrumentor software.

HDL Analyst Menu

The HDL Analyst menu provides you with commands for working in the HDL Analyst schematic views of your project: RTL views (RTL View, on page 2-4) and Technology views (Technology View, on page 2-6). You can access commands that affect these views from

- The HDL Analyst menu, which is described in HDL Analyst Options Command, on page 3-73.
- Context-sensitive right-click popup menus within the views (see RTL and Technology Views Popup Menus, on page 3-97 for details).

The table below describes the HDL Analyst menu commands in different views. Commands may be disabled (grayed out), depending on the current context. Generally, the commands enabled in any context reflect those available in the corresponding popup menus. The descriptions in the table indicate when commands are context-dependent. For explanations about the terms used in the table, like filtered and unfiltered, transparent and opaque, see Filtered and Unfiltered Schematic Views, on page 6-2 and Transparent and Opaque Display of Hierarchical Instances, on page 6-7. For procedures on using the HDL Analyst tool, see Result Analysis, on page 4-1 of the *Synplify User Guide*.

For ease of use, the commands have been divided into sections that correspond to the divisions in the HDL Analyst menu.

- HDL Analyst Menu: RTL and Technology Submenus, on page 3-50
- HDL Analyst Menu: Hierarchical and Current Level Submenus, on page 3-51
- HDL Analyst Menu: Filtering and Flattening Commands, on page 3-53
- HDL Analyst Menu: Timing Commands, on page 3-57
- HDL Analyst Menu: Analysis Commands, on page 3-58
- HDL Analyst Menu: Selection Commands, on page 3-61
- HDL Analyst Menu: FSM Commands, on page 3-61
- HDL Analyst Menu: Crossprobing Commands, on page 3-61

HDL Analyst Menu: RTL and Technology Submenus

This table describes the commands on the HDL Analyst->RTL and HDL Analyst ->Technology submenus. For procedures on using these commands, see Result Analysis, on page 4-1 of the *Synplify User Guide*.

HDL Analyst Command	Description
RTL -> ⊕ Hierarchical View	Opens a new, hierarchical RTL view. The schematic is unfiltered.

HDL Analyst Command	Description
RTL -> Flattened View	Opens a new RTL view of your entire design, with a flattened, unfiltered schematic at the level of generic logic cells. See Usage Notes for Flattening, on page 3-55 for some usage tips.
Technology -> Hierarchical View	Opens a new, hierarchical Technology view. The schematic is unfiltered.
Technology -> Flattened View	Creates a new Technology view of your entire design, with a flattened, unfiltered schematic at the level of technology cells. See Usage Notes for Flattening, on page 3-55 for tips about flattening.
Technology -> Flattened to Gates View	Creates a new Technology view of your entire design, with a flattened, unfiltered schematic at the level of Boolean logic gates. See Usage Notes for Flattening, on page 3-55 for tips about flattening
Technology -> Hierarchical Critical Path	Creates a new Technology view of your design, with a hierarchical, *filtered* schematic showing only the instances and paths whose slack times are within the slack margin you specified in the Slack Margin dialog. This command automatically enables HDL Analyst -> Show Timing Information.
Technology -> Flattened Critical Path	Creates a new Technology view of your design, with a flattened, *filtered* schematic showing only the instances and paths whose slack times are within the slack margin you specified in the Slack Margin dialog. This command automatically enables HDL Analyst -> Show Timing Information. See Usage Notes for Flattening, on page 3-55 for tips about flattening.

HDL Analyst Menu: Hierarchical and Current Level Submenus

This table describes the commands on the HDL Analyst->Hierarchical and HDL Analyst->Current Level submenus. For procedures on using these commands, see Result Analysis, on page 4-1 of the *Synplify User Guide.*

HDL Analyst Command	Description
Hierarchical -> Expand	Expands paths from selected pins and/or ports up to the nearest objects on any hierarchical level, according to pin/port directions. The result is a *filtered* schematic. Operates hierarchically, on lower schematic levels as well as the current level. Successive Expand commands expand the paths further, based on the new current selection.
Hierarchical -> Expand to Register/Port	Expands paths from selected pins and/or ports, in the port/pin direction, up to the next register, port, or black box. The result is a *filtered* schematic. Operates hierarchically, on lower schematic levels as well as the current level.
Hierarchical -> Expand Paths	Shows all logic, on any hierarchical level, between two or more selected instances, pins, or ports. The result is a *filtered* schematic. Operates hierarchically, on lower schematic levels as well as the current level.
Hierarchical -> Expand Inwards	Expands within the hierarchy of an instance, from the lower-level ports that correspond to the selected pins, to the nearest objects and no further. The result is a *filtered* schematic. Operates hierarchically, on lower schematic levels as well as the current level.
Hierarchical -> Goto Net Driver	Displays the unfiltered schematic sheet that contains the net driver for the selected net. Operates hierarchically, on lower schematic levels as well as the current level.
Hierarchical ->Select Net Driver	Selects the driver for the selected net. The result is a *filtered* schematic. Operates hierarchically, on lower schematic levels as well as the current level.
Hierarchical ->Select Net Instances	Selects instances connected to the selected net. The result is a *filtered* schematic. Operates hierarchically, on lower schematic levels as well as the current level.
Current Level ->Expand	Expands paths from selected pins and/or ports up to the nearest objects on the current level, according to pin/port directions. The result is a *filtered* schematic. Limited to all sheets on the current schematic level. This command is only available if a HDL Analyst view is open. Successive Expand commands expand the paths further, based on the new current selection.

HDL Analyst Command	Description
Current Level -> Expand to Register/Port	Expands paths from selected pins and/or ports, according to the pin/port direction, up to the next register, ports, or black box on the current level. The result is a *filtered* schematic. Limited to all sheets on the current schematic level.
Current Level-> Expand Paths	Shows all logic on the current level between two or more selected instances, pins, or ports. The result is a *filtered* schematic. Limited to the current schematic level (all sheets).
Current Level-> Goto Net Driver	Displays the unfiltered schematic sheet that contains the net driver for the selected net. Limited to all sheets on the current schematic level.
Current Level-> Select Net Driver	Selects the driver for the selected net. The result is a *filtered* schematic. Limited to all sheets on the current schematic level.
Current Level-> Select Net Instances	Selects instances on the current level that are connected to the selected net. The result is a *filtered* schematic. Limited to all sheets on the current schematic level.

HDL Analyst Menu: Filtering and Flattening Commands

This table describes the filtering and flattening commands on the HDL Analyst menu. For procedures on filtering and flattening, see Result Analysis, on page 4-1 of the *Synplify User Guide.*

HDL Analyst Command	Description
Filter Schematic	Filters your entire design to show only the selected objects. The result is a *filtered* schematic. For more information about using this command, see Filtering Schematics, on page 4-48 of the *Synplify User Guide*. This command is only available with an open HDL Analyst view.

HDL Analyst Command	Description
Flatten Current Schematic (Unfiltered Schematic)	In an unfiltered schematic, the command flattens the current schematic, at the current level and all levels below. In an RTL view, the result is at the generic logic level. In a Technology view, the result is at the technology-cell level. See the next table entry for information about flattening a filtered schematic. This command does not do the following: • Flatten your entire design (unless the current level is the top level) • Open a new view window • Take into account the number of Dissolve Levels defined in the Schematic Options dialog box. See Usage Notes for Flattening, on page 3-55 for tips.

HDL Analyst Command	Description
Flatten Current Schematic (Filtered Schematic)	In a filtered schematic, flattening is a two-step process: • Only unhidden transparent instances (including nested ones) are flattened in place, in the context of the entire design.Opaque and hidden hierarchical instances remain hierarchical. The effect of this command is that all hollow boxes with pale yellow borders are removed from the schematic, leaving only what was displayed inside them. • The original filtering is restored. In an RTL view, the result is at the generic logic level. In a Technology view, the result is at the technology-cell level. This command does not do the following: • Flatten everything inside a transparent instance. It only flattens transparent instances and any nested transparent instances they contain. • Open a new view window • Take into account the number of Dissolve Levels defined in the Schematic Options dialog box. See Usage Notes for Flattening, on page 3-55 for usage tips.
Unflatten Current Schematic	Undoes any flattening operations and returns you to the original schematic, as it was before flattening and any filtering. This command is available only if you have explicitly flattened a hierarchical schematic using HDL Analyst -> Flatten Current Schematic, for example. It is not available for flattened schematics created directly with the RTL and Technology submenus of the HDL Analyst menu.

Usage Notes for Flattening

It is usually more memory-efficient to flatten only parts of your design, as needed. The following are a few tips for flattening designs with different commands. For detailed procedures, see Flattening Schematic Hierarchy, on page 4-56 of the *Synplify User Guide*.

RTL/Technology -> Flattened View Commands

- Use Flatten Current Schematic to flatten only the current hierarchical level and below.
- Flatten selected hierarchical instances with Dissolve Instances (followed by Flatten Current Schematic, if the schematic is filtered).
- To make hierarchical instances transparent without flattening them, use Dissolve Instances in a filtered schematic. This shows their details nested inside the instances.

Flatten Current Schematic Command (Unfiltered View)

- Flatten selected hierarchical instances with Dissolve Instances.
- To see the lower-level logic inside a hierarchical instance, push into it instead of flattening.
- Selectively flatten your design by hiding the instances you do not need, flattening, and then unhiding the instances.
- Flattening erases the history of displayed sheets for the current view. You can no longer use View -> Back. You can, however, use UnFlatten Schematic to get an unflattened view of the design.

Flatten Current Schematic Command (Filtered View)

- Flatten selected hierarchical instances with Dissolve Instances, followed by Flatten Current Schematic.
- Selectively flatten your design by hiding the instances you do not need, flattening, and then unhiding the instances.
- Flattening erases the history of displayed sheets for the current view. You can no longer use View -> Back. You can do the following:
- Use View -> Back for a view of the transparent instance flattened in the context of the entire design. This is the view generated after step 1 of the two-step flattening process described above. Use UnFlatten Schematic to get an unflattened view of the design.

HDL Analyst Menu: Timing Commands

This table describes the timing commands on the HDL Analyst menu. For procedures on using the timing commands, see Result Analysis, on page 4-1 of the *Synplify User Guide.*

HDL Analyst Command	Description
Set Slack Margin...	Displays the Slack Margin dialog box, where you set the slack margin. HDL Analyst -> Show Critical Path displays only those instances whose slack times are worse than the limit set here. Available only in a Technology view.
Show Critical Path	Filters your entire design to show only the instances and paths whose slack times exceed the slack margin set with Set Slack Margin, above. The result is flat if the entire design was already flat. This command also enables Show Timing Information (see below). Available only in a Technology view. Not available in a Timing view created with the Timing Analyst command.
Show Timing Information	When enabled, Technology view schematics are annotated with timing numbers above each instance. The first number is the cumulative path delay; the second is the slack time of the worst path through the instance. Negative slack indicates that timing has not met requirements. Available only in a Technology view. For more information, see Annotated Timing Information, on page 6-34.
Timing Analyst...	Not available in Synplify.

HDL Analyst Menu: Analysis Commands

This table describes the analysis commands on the HDL Analyst menu. For procedures on using the analysis commands, see Result Analysis, on page 4-1 of the *Synplify User Guide.*

HDL Analyst Command	Description
Isolate Paths	Filters the current schematic to display only paths associated with all the pins of the selected instances. The paths follow the pin direction (from output to input pins), up to the next register, black box, port, or hierarchical instance. If the selected objects include ports and/or pins on unselected instances, the result also includes paths associated with those selected objects. The range of the operation is all sheets of a filtered schematic or just the current sheet of an unfiltered schematic. The result is always a filtered schematic. In contrast to the Expand operations, which add to what you see, Isolate Paths can only remove objects from the display. While Isolate Paths is similar to Expand to Register/Port, Isolate Paths reduces the display while Expand to Register/Port augments it.
Show Context	Shows the original, unfiltered schematic sheet that contains the selected instance. Available only in a filtered schematic.
Hide Instances	Hides the logic inside the selected hierarchical (non-primitive) instances. This affects only the active HDL Analyst view; the instances are not hidden in other HDL Analyst views. The logic inside hidden instances is not loaded (saving dynamic memory), and it is unrecognized by searching, dissolving, flattening, expansion, and push/pop operations. (Crossprobing does recognize logic inside hidden instances, however.) See Usage Notes for Hiding Instances, on page 3-60 for tips.
Unhide Instances	Undoes the effect of Hide Instances: the selected hidden hierarchical instances become visible (susceptible to loading, searching, dissolving, flattening, expansion, and push/pop operations). This affects only the current HDL Analyst view; the instances are not hidden in other HDL Analyst views.

HDL Analyst Command	Description
Show All Hier Pins	Shows all pins on the selected transparent, non-primitive instances. Available only in a filtered schematic. Normally, transparent instance pins that are connected to logic that has been filtered out are not displayed. This command lets you display these pins that connected to logic that has been filtered out. Pins on primitives are always shown.
Dissolve Instances	Shows the lower-level details of the selected non-hidden hierarchical instances. The number of levels dissolved is determined by the Dissolve Levels value in the HDL Analyst Options dialog box (HDL Analyst Options Command, on page 3-73). For usage tips, see Usage Notes for Dissolving Instances, on page 3-60.
Dissolve to Gates	Dissolves the selected instances by flattening them to the gate level. This command displays the lower-level hierarchy of selected instances, but it dissolves technology primitives as well as hierarchical instances. Technology primitives are dissolved to generic synthesis symbols. The command is only available in the Technology view. The number of levels dissolved is determined by the Dissolve Levels value in the HDL Analyst Options dialog box (HDL Analyst Options Command, on page 3-73). Dissolving an instance one level redraws the current sheet, replacing the hierarchical dissolved instance with the logic you would see if you pushed into it using Push/pop mode. Unselected objects or selected hidden instances are not dissolved. The effect of the command varies: • In an unfiltered schematic, this command *flattens* the selected instances. This means the history of displayed sheets is removed. The resulting schematic is unfiltered. • In a filtered schematic, this command makes the selected instances *transparent*, displaying their internal, lower-level logic inside hollow boxes. History is retained. You can use Flatten Schematic to flatten the transparent instances, if necessary. The resulting schematic if filtered.

Usage Notes for Hiding Instances

The following are a few tips for hiding instances. For detailed procedures, see Flattening Schematic Hierarchy, on page 4-56of the *Synplify User Guide*.

- Hiding hierarchical instances soon after startup can often save memory. After the interior of an instance has been examined (by searching or displaying), it is too late for this savings.
- You can save memory by creating small, temporary working files: File -> Save As `.srs` or `.srm` files does not save the hidden logic (hidden instances are saved as black boxes). Restarting the synthesis tool and loading such a saved file can often result in significant memory savings.
- You can selectively flatten instances by temporarily hiding all the others, flattening, then unhiding.
- You can limit the range of Edit -> Find (see Find Command (HDL Analyst), on page 3-15) to prevent it looking inside given instances, by temporarily hiding them.

Usage Notes for Dissolving Instances

Dissolving an instance one level redraws the current sheet, replacing the hierarchical dissolved instance with the logic you would see if you pushed into it using Push/pop mode. Unselected objects or selected hidden instances are not dissolved. For additional information about dissolving instances, see Flattening Schematic Hierarchy, on page 4-56 of the *Synplify User Guide*.

The type (filtered or unfiltered) of the resulting schematic is unchanged from that of the current schematic. However, the effect of the command is different in filtered and unfiltered schematics:

- In an unfiltered schematic, this command flattens the selected instances. This means the history of displayed sheets is removed.
- In a filtered schematic, this command makes the selected instances transparent, displaying their internal, lower-level logic inside hollow boxes. History is retained. You can use Flatten Schematic to flatten the transparent instances, if necessary. This command is only available if an HDL Analyst view is open.

HDL Analyst Menu: Selection Commands

This table describes the selection commands on the HDL Analyst menu.

HDL Analyst Command	Description
Select All Schematic -> Instances -> Ports	Selects all Instances or Ports, respectively, on all sheets of the current schematic. All other objects are unselected. This does not select objects on other schematics.
Select All Sheet -> Instances -> Ports	Selects all Instances or Ports, respectively, on the current schematic sheet. All other objects are unselected.
Unselect All	Unselects all objects in all HDL Analyst views.

HDL Analyst Menu: FSM Commands

This table describes the FSM commands on the HDL Analyst menu.

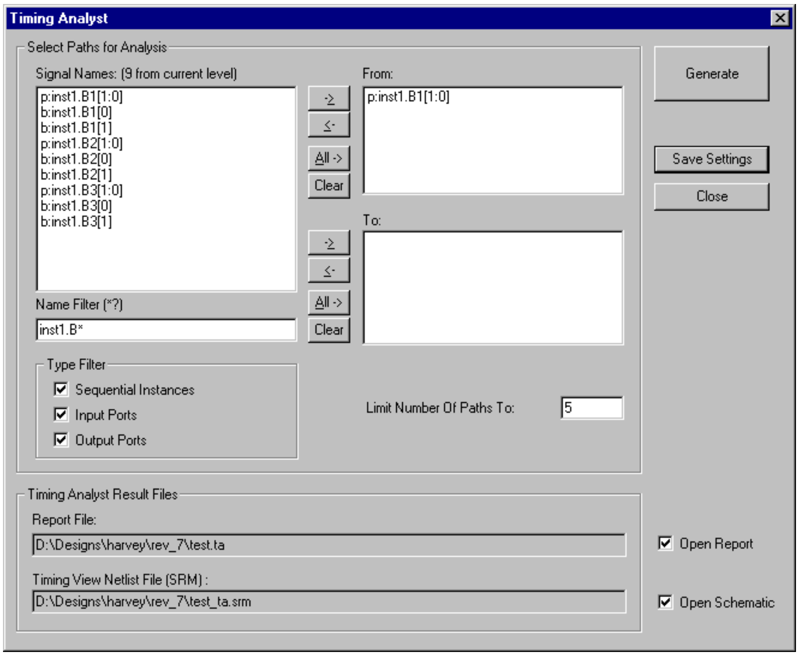
HDL Analyst Command	Description
View FSM Info File	Displays information about the selected finite state machine module, including the number of states, the number of inputs, and a table of the states and transitions. Available only in an RTL view.

HDL Analyst Menu: Crossprobing Commands

This table describes the crossprobing commands on the HDL Analyst menu.

HDL Analyst Command	Description
Visual Elite Cross Probing -> Connect to Visual Elite -> Disconnect from Visual Elite	Connects/disconnects the Synplify synthesis tool to/from the Visual Elite™ design tool. When a connection is established, the Cross Probing Enabled command becomes available. To enable the connection, the Visual Elite tool must be set up correctly. See Working with Visual Elite, on page 7-12 *of the* Synplify *User Guide* and Crossprobing Between Different Views, on page 6-15 for details.

HDL Analyst Command	Description
Visual Elite Cross Probing -> Cross Probing Enabled	Enables and disables crossprobing between the synthesis tool and the Visual Elite design tool. Available only when the tools are connected (see previous command). See Working with Visual Elite, on page 7-12 *of the* Synplify *User Guide* and Crossprobing Between Different Views, on page 6-15 for details.
External Cross Probing Engaged	Enables crossprobing to any ToolNet-enabled application, like the ModelSim® HDL simulator or the Visual Elite™ design tool. See Integrating with ModelSim, on page 7-10 of the *Synplify User Guide* for details.



Signal Names: (9 from current level)

```
p:inst1.B1[1:0]
b:inst1.B1[0]
b:inst1.B1[1]
p:inst1.B2[1:0]
b:inst1.B2[0]
b:inst1.B2[1]
p:inst1.B3[1:0]
b:inst1.B3[0]
b:inst1.B3[1]
```

From: p:inst1.B1[1:0]

To:

Name Filter (*?): inst1.B*

☑ Sequential Instances
☑ Input Ports
☑ Output Ports

Limit Number Of Paths To: 5

Report File: D:\Designs\harvey\rev_7\test.ta

Timing View Netlist File (SRM): D:\Designs\harvey\rev_7\test_ta.srm

☑ Open Report
☑ Open Schematic

The following table describes the Timing Analyst dialog box options.

Field/Option	Description
Signal Names	Names of all signal objects that correspond to the selected Type Filter. By default, the names listed are those selected in the Technology views of the current design or, if none are selected, all signal objects in the design. You select objects in this list, then use the right-arrow buttons to copy them to the From and To fields. Select multiple names by pressing the Ctrl or Shift key while clicking; press Ctrl and click a selection to deselect it.
Name Filter (*?)	Lists signal object names in the Signal Names area, based on the (case-insensitive) match string that you input here (you use the keyboard Enter/Return key to enter the string). The string can contain the wildcard characters * (asterisk), which matches any sequence of characters, and ? (question mark), which matches any single character.
Type Filter	The types of signal object to match against the Name Filter (*?) match string (see Sequential Instances, Input Ports, Output Ports, below).
Sequential Instances	When enabled, the instances of the design are included as candidates to match against the Name Filter (*?) match string.
Input Ports	When enabled, the input ports of the design are included as candidates to match against the Name Filter (*?) match string.
Output Ports	When enabled, the output ports of the design are included as candidates to match against the Name Filter (*?) match string.
->	Click this button to copy the selected objects from the Signal Names field to the corresponding From or To field.
<-	Click this button to move the selected objects from the corresponding From or To field to the Signal Names field.
All ->	Click this button to copy all of the objects from the Signal Names field to the corresponding From or To field.
<- All	Click this button to move all of the objects from the corresponding From or To field to the Signal Names field.
Clear	Click this button to clear (empty) the associated From and To field.
From	Start points of the paths to include in the report. More than one start point can be used.

Field/Option	Description
To	End points of the paths to include in the report. More than one end point can be used.
Limit Number Of Paths To	The maximum number of paths to report. The default is 5.
Report File	The name and path of the output report file (*resultsfile*.ta). The file is located in the project results area of the Project view. You cannot modify the file name here. The same name is used for subsequent runs of the same implementation, so the file is then overwritten. To change the file name, specify it on the Implementation Options dialog box (Implementation Results Panel, on page 3-32). To view the report in a Text editor window, enable Open Report, below.
Timing View Netlist File (SRM)	The name and path of the netlist file (*resultsfile*_ta.srm) with the timing results. The file is in the implementation results directory. You cannot modify the file name here. The same name is used for subsequent runs of the same implementation, so the file is then overwritten. To view the results in a Timing view, enable Open Schematic, below.
Save Settings	Saves the current dialog box entries (except for Signal Names) to the project file (.prj) for subsequent use. This does *not* generate a timing report; for that, click Generate.
Generate	Clicking this button generates the specified timing Report File and Timing View Netlist File (SRM), saves the current dialog box entries for subsequent use, then closes the dialog box. While the report is being generated, Analyzing Timing appears in the large status area near the top of the application window.
Open Report	When enabled, clicking the Generate button opens the Text Editor on the generated custom timing report specified in Report File.
Open Schematic	When enabled, clicking the Generate button opens a Technology view showing the netlist specified in Timing View Netlist File (SRM).

Format Menu

The Format menu is available only when the SCOPE view is active. You can use it to configure SCOPE spreadsheet cells, rows, and columns, and to set text alignment and style.

Command	Description
Cells...	Specifies the font and color to use. Adds borders to cells. See Cells Command, on page 3-65.
Resize Rows	Changes the row height to fit the text.
Resize Columns	Changes the column width to fit the text.
Cover Cells	Temporarily hides selected cells from view.
Remove Covering	Removes the covering previously placed on cells by Cover Cells, so they become visible again.
Styles...	Changes the styles of columns, rows, and warnings. See Styles Command, on page 3-67.
Align	Aligns text in columns and rows to the Left, Center, or Right.
Style	Makes text Bold, Italic, or Underlined.

Cells Command

Select Format -> Cells to open the Cells dialog box, where you set font, color, border, and alignment options for the SCOPE spreadsheet cells. The dialog box has panels for the different options.

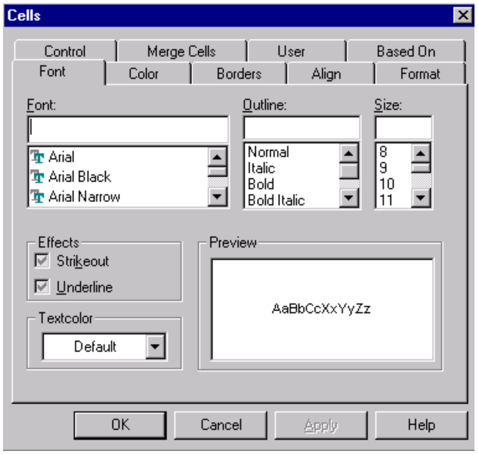


Figure 3-18: Cells dialog box (Format -> Cells)

Panel	Description
Font	Set the font face, style, size, and color for the text that displays in the cells.
Color	Set the foreground and background color and pattern for the cell. You can also set 3-D visual effects such as raised or inset.
Borders	Set the border line type, line width, and color.
Align	Set the vertical and horizontal alignment of the text in the cells. Select other options such as Wrap Text, Allow Enter, and Auto Size.
Format	Set the format for the type of content such as date, decimal, and percent. Enter the precision value.
Control	Lets you add simple user interface objects such as radio buttons, check boxes, and popup menus.
Merge Cells	Lets you merge cells horizontally or vertically.
User	Lets you enable user attributes to control the usability and appearance of the spreadsheet interface.
Based On	Lets you format cells based on other cell styles that you create and edit in the Format Styles dialog box.

Styles Command

Select Format -> Styles to access the Styles dialog box and change the styles of your columns, rows, and warnings. Select an item that you want to change, then click the Change button to display the Cells dialog box (see Cells Command, on page 3-65). If you would like to create a new style, type the name of the new style, then click the Add >> button. Edit the style in the Cells dialog box.

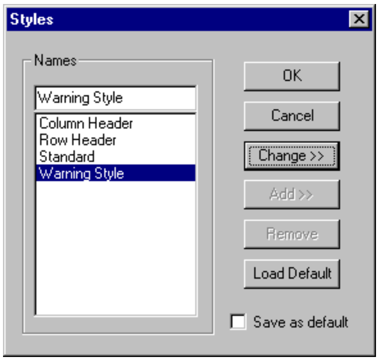


Figure 3-19: Styles dialog box (Format -> Styles)

Options Menu

Use the Options menu to configure the VHDL and Verilog compilers, customize toolbars, and set options for the Project view, Text Editor, and HDL Analyst schematics. When using certain technologies, additional menu commands let you run technology-vendor software from this menu.

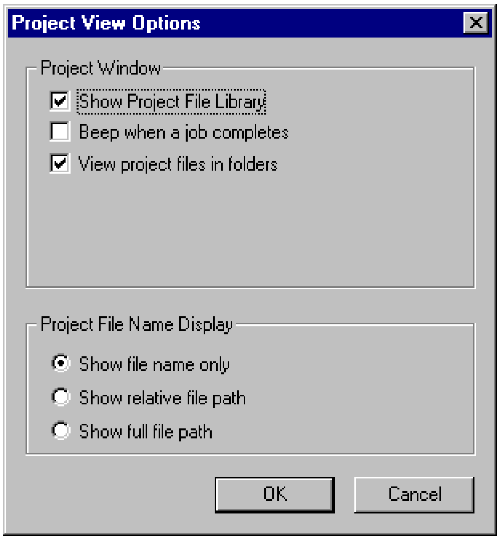
The following table describes the Options menu commands.

Table 3-4: Options Menu Commands

Command	Description
Configure VHDL Compiler...	Opens the Implementation Options dialog box where you can set the top-level entity and the encoding method for enumerated types. State-machine encoding is automatically determined by the FSM compiler, or you can specify it explicitly using the syn_encoding attribute. See Implementation Options Command, on page 3-29 for details.
Configure Verilog Compiler...	Opens the Implementation Options dialog box where you can specify the top-level module and the ‘include search path. See Implementation Options Command, on page 3-29.
Toolbars...	Lets you customize your toolbars. See Customize Dialog Box, on page 3-24 for details.
Project View Options...	Sets options for organizing files in the Project view. See Project View Options Command, on page 3-69.
Editor Options...	Sets your Text Editor syntax coloring, font, and tabs. See Editor Options Command, on page 3-71.
HDL Analyst Options...	Sets display preferences for HDL Analyst schematics (RTL and Technology views). See HDL Analyst Options Command, on page 3-73.
Configure External Programs	Lets you set browser and Acrobat Reader options on UNIX and Linux platforms. See Configure External Programs Command, on page 3-79 for details.

Project View Options Command

Select Options -> Project View Options to display the Project View Options dialog box, where you define how projects appear and are organized in the Project view.



[✓] Show Project File Library
[ ] Beep when a job completes
[✓] View project files in folders

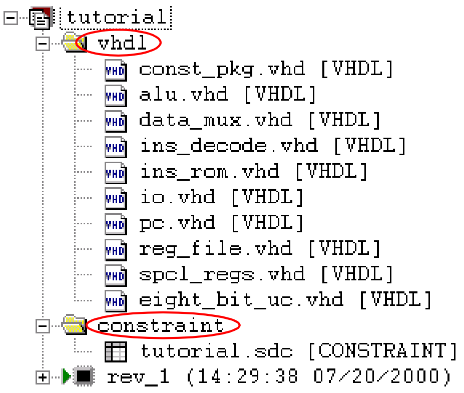
[✓ Show file name only]
Show relative file path
Show full file path

The following table describes the Project View Options dialog box features.

Field/Option	Description
Show Project File Library	When enabled, displays the corresponding VHDL library next to each source VHDL filename, in the Project Tree view of the Project view. For example, with library dune, file pc.vhd is listed as [dune] pc.vhd if this option is enabled, and as pc.vhd if it is disabled. (See also Set VHDL Library Command, on page 3-43, for how to change the library of a file.)
Beep when a job completes	When enabled, sounds an audible signal whenever a project finishes running.
View project files in folders	When enabled, organizes project files into separate folders, by type. See View Project Files in Folders Option, on page 3-70.

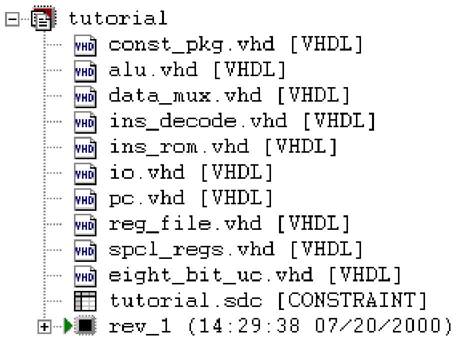
Field/Option	Description
Show file name only	When enabled, displays only filenames.
Show relative file path	When enabled, displays relative paths, as well as filenames.
Show full file path	When enabled, displays full paths and filenames.

View Project Files in Folders Option



```
tutorial
  vhdl
    const_pkg.vhd [VHDL]
    alu.vhd [VHDL]
    data_mux.vhd [VHDL]
    ins_decode.vhd [VHDL]
    ins_rom.vhd [VHDL]
    io.vhd [VHDL]
    pc.vhd [VHDL]
    reg_file.vhd [VHDL]
    spcl_regs.vhd [VHDL]
    eight_bit_uc.vhd [VHDL]
  constraint
    tutorial.sdc [CONSTRAINT]
  rev_1 (14:29:38 07/20/2000)
```

View project files in folders *enabled*



```
tutorial
  const_pkg.vhd [VHDL]
  alu.vhd [VHDL]
  data_mux.vhd [VHDL]
  ins_decode.vhd [VHDL]
  ins_rom.vhd [VHDL]
  io.vhd [VHDL]
  pc.vhd [VHDL]
  reg_file.vhd [VHDL]
  spcl_regs.vhd [VHDL]
  eight_bit_uc.vhd [VHDL]
  tutorial.sdc [CONSTRAINT]
  rev_1 (14:29:38 07/20/2000)
```

View project files in folders *disabled*

Editor Options Command

Select Options -> Editor Options to display the Editor Options dialog box, where you select either the Internal Text Editor, or an External Editor.

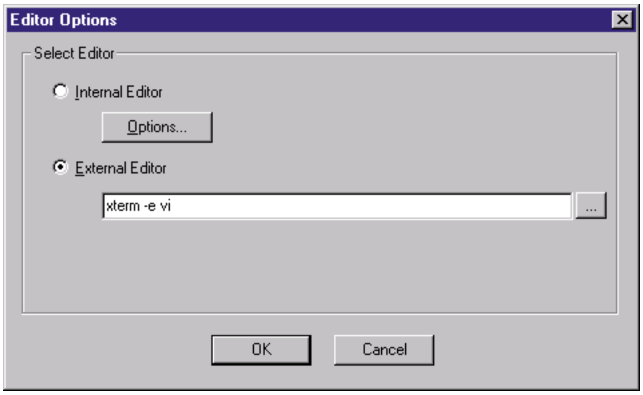


Figure 3-20: Editor Options dialog box (Options -> Editor Options)

The following table describes the Editor Options dialog box features.

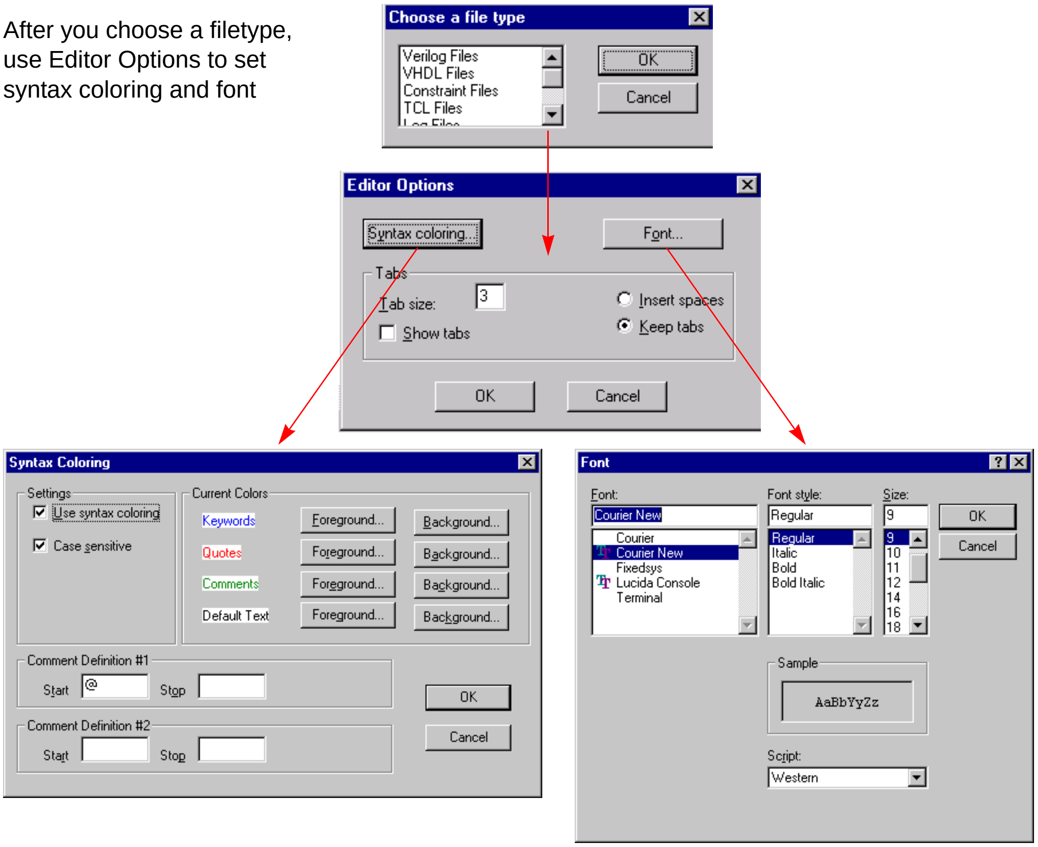
Field/Option	Description
Internal Editor	Sets the Synplicity Text Editor as the default text editor.
Options	Lets you define whitespace, syntax coloring, and font to use with the internal text editor. See Setting Editing Window Preferences, on page 2-9 of the *Synplify User Guide*. (Only available when Internal Editor is enabled.)
External Editor	Uses the specified external text editor program to view text files from within the Synplicity tool. The executable specified must open its own window for text editing. See Using an External Text Editor, on page 2-8 of the *Synplify User Guide* for a procedure. *Note:* Files opened with an external editor *cannot* be crossprobed.

Editor Color, Font, and Tab Options

The Options button of the Editor Options dialog box (Internal Editor only) lets you define various text editing preferences for the built-in Synplicity Text Editor. Clicking this button displays the Choose a File Type dialog box, where you select a type of file whose preferences you want to define. (The Default Files choice means “other”: all file types other than the explicit choices.)

Clicking OK displays another dialog box (also named Editor Options), where you define the preferences for the specified file type. These include the whitespace format, syntax coloring, and font. See Setting Editing Window Preferences, on page 2-9 of the *Synplify User Guide* for more information.

After you choose a filetype, use Editor Options to set syntax coloring and font



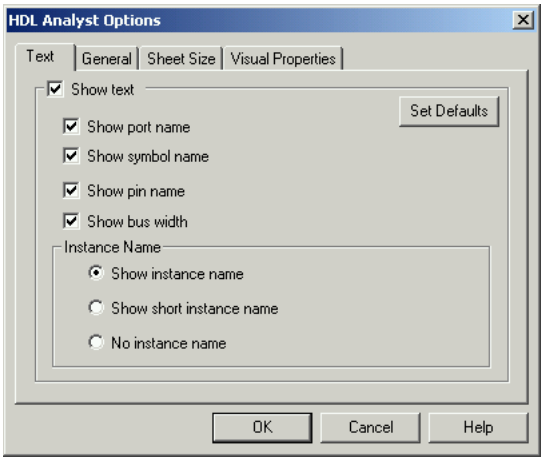
HDL Analyst Options Command

Select Options -> HDL Analyst Options to display the HDL Analyst Options dialog box, where you define preferences for the HDL Analyst schematic views (RTL and Technology views). Some preferences take effect immediately; others only take effect in the next view that you open. For details see Setting Schematic View Preferences, on page 4-20 of the *Synplify User Guide*.

For information about the options, see the following, which correspond to the tabs on the dialog box:

- Text Panel, on page 3-73
- General Panel, on page 3-74
- Sheet Size Panel, on page 3-76
- Visual Properties Panel, on page 3-78

Text Panel


HDL Analyst Options

Text | General | Sheet Size | Visual Properties

☑ Show text

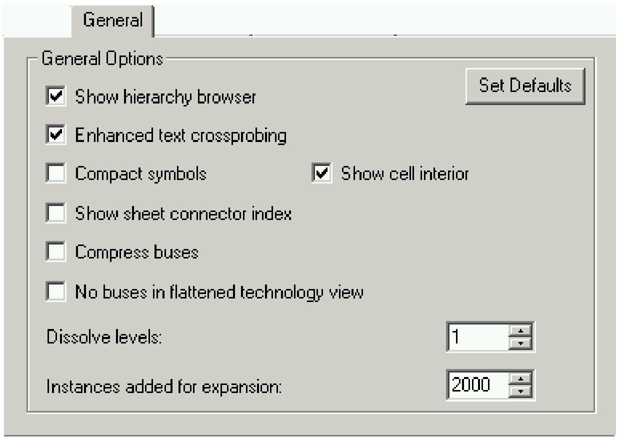
☑ Show port name
☑ Show symbol name
☑ Show pin name
☑ Show bus width

Instance Name
● Show instance name
○ Show short instance name
○ No instance name

The following options are in the Text panel.

Field/Option	Description
Show Text	Enables the selective display of schematic labels. Which labels are displayed is governed by the other Show * features and Instance Name, described below.
Show Port Name	When enabled, port names are displayed.
Show Symbol Name	When enabled, symbol names are displayed.
Show Pin Name	When enabled, pin names are displayed.
Show Bus Width	When enabled, connectivity bit ranges are displayed near pins (in square brackets: []), indicating the bits used for each bus connection.
Instance Name	Determines how to display instance names: • Show instance name • Short short instance Name • No instance name
Set Defaults	Set the dialog box to display the default values.

General Panel


General

General Options

[✓] Show hierarchy browser

[✓] Enhanced text crossprobing

[ ] Compact symbols [✓] Show cell interior

[ ] Show sheet connector index

[ ] Compress buses

[ ] No buses in flattened technology view

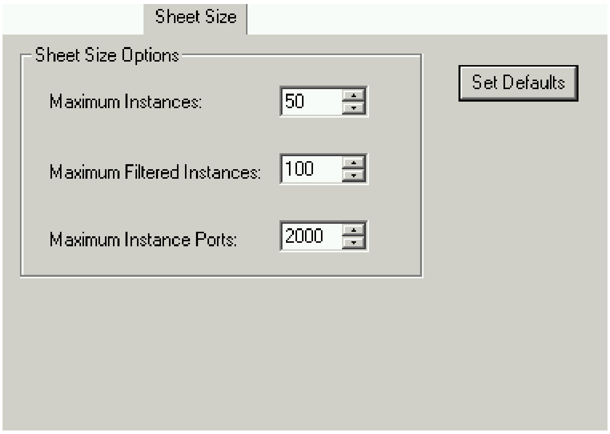
Dissolve levels: 1

Instances added for expansion: 2000

The following options are in the General panel.

Field/Option	Description
Show Hierarchy Browser	When enabled, a Hierarchy Browser is present as the left pane of RTL and Technology views.
Enhanced Text Crossprobing	When enabled, you can crossprobe from any text file, including non-HDL files (you can always crossprobe to and from HDL source files).
Compact Symbols	When enabled, symbols are displayed in a slightly more compact manner, to save space in schematics. When this is enabled, Show Cell Interior is disabled.
Show Cell Interior	When enabled, the internal logic of cells that are technology-specific primitives (such as LUTs) is shown in Technology views. This is not available if Compact Symbols is enabled.
Show Sheet Connector Index	When enabled, sheet connectors show connecting sheet numbers – see Sheet Connectors, on page 6-5.
Compress Buses	When enabled, buses having the same source and destination instances are displayed as bundles, to reduce clutter. A single bundle can connect to more than one pin on a given instance. The display of a bundle of buses is similar to that of a single bus.
No Buses in Flattened Technology View	When enabled, buses are not displayed; they are only indicated as bits in a Technology View. This applies only to flattened views created by HDL Analyst -> Technology -> Flattened View (or Flattened to Gates View), not to hierarchical views that you have flattened (using, for example, HDL Analyst -> Flatten Current Schematic).
Dissolve Levels	The number of levels to dissolve, during HDL Analyst -> Dissolve Instances. See Dissolve Instances, on page 3-59
Instances added for expansion	The maximum number of instances to add during any operation (such as HDL Analyst -> Hierarchical -> Expand) that results in a *filtered* schematic. When this limit is reached, you are prompted to continue adding more instances.

Sheet Size Panel

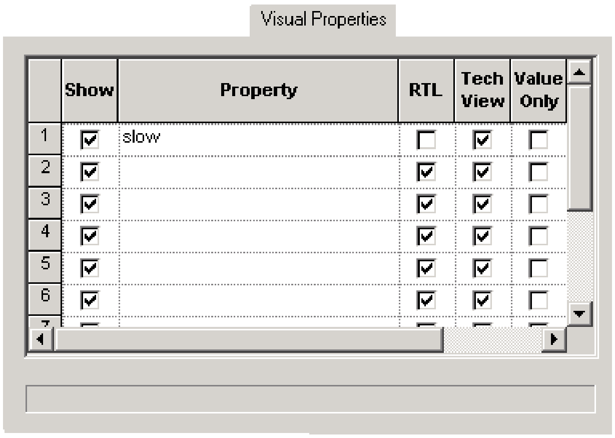


Maximum Instances: 50
Maximum Filtered Instances: 100
Maximum Instance Ports: 2000

The following options are in the Sheet Size panel.

Maximum Instances	Defines the maximum number of instances to display on a single sheet of an unfiltered schematic. If a given hierarchical level has more than this number of instances, then it will be partitioned into multiple sheets. See Multiple-sheet Schematics, on page 6-18.
Maximum Filtered Instances	Defines the maximum number of instances to display on a filtered schematic sheet, at any visible hierarchical level. This limit is applied recursively, at each visible *level*, when • the sheet itself is a level, and • each transparent instance is a level (even if inside another transparent instance). Whenever a given level has more child instances inside it than the value of Filtered Instances, it is divided into multiple sheets. (Only children are counted, not grandchildren or below. Instance A is a *child* of instance B if it is inside no other instance that is inside B.) In fact, at each level except the sheet itself, an additional margin of allowable child instances is added to the Filtered Instances value, increasing its effective value. This means that you can see more child instances than Filtered Instances itself implies. The Filtered Instances value must be at least the Instances value. See Multiple-sheet Schematics, on page 6-18.
Maximum Instance Ports	Defines the maximum number of instance pins to display on a schematic sheet.

Visual Properties Panel



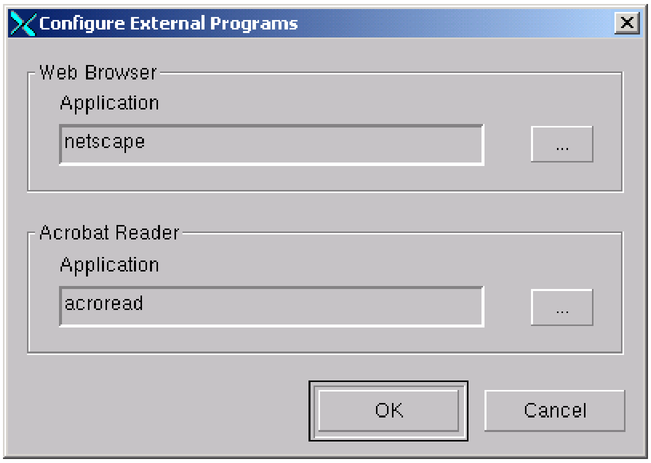
	Show	Property	RTL	Tech View	Value Only
1	✓	slow	☐	✓	☐
2	✓		✓	✓	☐
3	✓		✓	✓	☐
4	✓		✓	✓	☐
5	✓		✓	✓	☐
6	✓		✓	✓	☐

The following options are in the Visual Properties panel.

Show	Controls the display of the selected property in open HDL Analyst views. The properties are displayed as colored boxes on the relevant objects. To display these properties, the View->Visual Properties command must also be enabled. For more information about properties, see Viewing Object Properties, on page 4-14 in the *Synplify User Guide*. The property name and value is displayed in a color-coded box on the object.
Property	Sets the properties to display.
RTL	Enables or disables the display of visual properties in the RTL view.
Tech View	Enables or disables the display of visual properties of in the Technology view.
Value Only	Displays only the value of an item and not its property name.

Configure External Programs Command

This command is for UNIX and Linux platforms only. It lets you specify the web browser and PDF reader for accessing the Technical Resource Center (see Web Menu, on page 3-80 for details about the TRC) and online documents.



Web Browser Application: netscape

Acrobat Reader Application: acroread

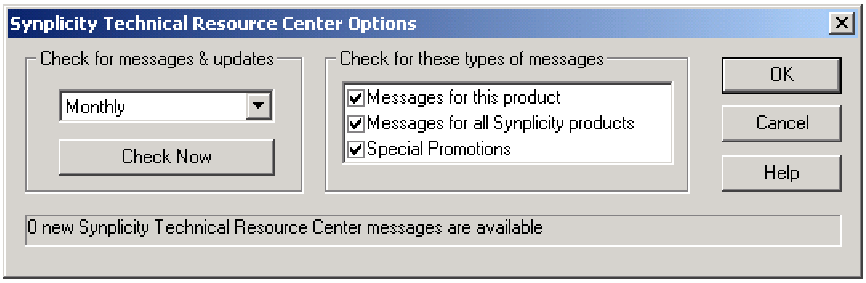
Field/Option	Description
Web Browser	Specify your web browser as an absolute path. You can use the Browse button to locate the browser you need. The default is netscape. If your browser requires additional environment settings, you must do so outside the synthesis tool.
Acrobat Reader	Specify your PDF reader as an absolute path. You can use the Browse button to locate the reader you need. The default is acroread.

Web Menu

This menu contains commands that access up-to-date information from Synplicity.

Command	Description
Go to Resource Center	Opens the home page for the Synplicity Technical Resource Center (TRC). This is a website that contains links to useful technical information, like application notes, white papers, updates, and other user-oriented documentation.
✉ Check Resource Center Messages	Opens a web page that contains update messages, customized according to the options you set with Set Resource Center Options. You can also access this page by clicking the Message (envelope) icon in the status bar at the bottom of the application window. See Resource Center Messages, on page 3-81 for additional information.
Configure Resource Center	Lets you set options for Resource Center updates. See Configure Resource Center Command, on page 3-82 for details.
Synplicity Home	Opens the Synplicity corporate home page.
Synplicity Products	Opens the main page for Synplicity products. You can find information about the full line of Synplicity products here.
Synplicity Support	Opens the Synplicity support page. From this page, there are links to product-specific information like application notes.

1. To go to the TRC, select Web->Go To Resource Center.
2. To set preferences for accessing the Resource Center, select Web->Configure Resource Center and do the following:
   - Set the frequency at which you wish to be notified.
   - Select the type of information you want to receive.
   - To go to the resource center immediately, click Check Now.
   - Click OK.


Synplicity Technical Resource Center Options

Check for messages & updates: Monthly

Check Now

Check for these types of messages:
- [✓] Messages for this product
- [✓] Messages for all Synplicity products
- [✓] Special Promotions

OK | Cancel | Help

0 new Synplicity Technical Resource Center messages are available

If the TRC has never been configured, the envelope icon in the lower right of the application window will be yellow (to indicate available messages). Once you have configured TRC access, a yellow blinking envelope indicates new messages, and a gray envelope indicates there are no new messages.

3. To check updates, do one of the following. If you have not set your TRC preferences, the software opens the Resource Center Options dialog box described in the previous step.
   - Click on the yellow envelope icon in the status bar (lower right of the application window).
   - Select Web->Check Resource Center Messages.
   - Select Web->Go To Technical Resource Center.
4. For other Synplicity product information, select the following:

Web->Synplicity Home page	Synplicity home page
Web->Synplicity Products	Synplicity product information
Web->Synplicity Support	Synplicity technical support

Resource Center Messages

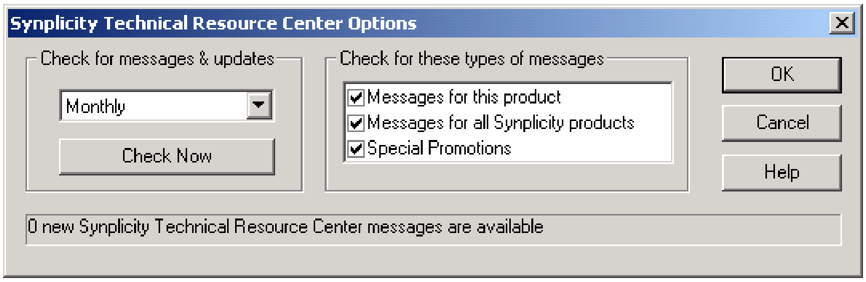
The envelope icon in the status bar (lower right of the application window) indicates when new messages are available:

Yellow or blinking	New messages available
Gray	No new messages, or you have elected not to check for new messages.

Click on the icon to go to the Messages page of the Resource Center.

Configure Resource Center Command

Selecting this command displays the Resource Center Options dialog box, where you set preferences for the kinds of update information you want to receive, and the frequency at which it is received. You also see this dialog box if you try to check for Resource Center messages without first configuring your preferences.


Synplicity Technical Resource Center Options
Check for messages & updates
Check for these types of messages
OK
Cancel
Help
Check Now

Monthly

☑ Messages for this product
☑ Messages for all Synplicity products
☑ Special Promotions

0 new Synplicity Technical Resource Center messages are available

Option	Description
Check for messages and updates	Determines the frequency at which the software checks for updates. Select a setting from the menu. At the specified frequency, the software checks for new messages, and then sets the envelope icon to indicate message status.
Check Now	Goes to the Technical Resource Center. The page shows both old and new messages.
Check for these types of messages	Determines the kinds of information you want to receive. Check the boxes for the information you want. • Messages for this product Available updates and critical bulletins for the product • Messages for all Synplicity products Available updates and critical bulletins for all Synplicity products • Special promotions Promotional packages and pricing, occasional surveys, and other related information

Help Menu

The following table describes the Help menu commands. Some commands are only available in certain views.

Command	Description
Help	Displays hyperlinked online help for the product.
Additional Products	Displays a form that you can fill out, then fax or e-mail to Synplicity, to request product information.
How to Use Help	Displays online help on how to use the online help (!).
Online Documents	Displays hyperlinked PDF documentation on the product: release notes, user guide, reference manual, tutorial, and help for configuration and setup. You need Adobe Acrobat Reader® to view the PDF files.
Synplify vs. Synplify Pro	Details the differences between these products. Only available in the Project view.
Error Messages	Displays help for notes, errors, and warning messages that appear in the Log file and Tcl window during synthesis.
TCL	Displays help for Tcl commands.
Mouse Stroke Tutor	Displays the Mouse Stroke Tutor dialog box, providing information on the available mouse strokes – see Using Mouse Strokes, on page 2-15.
License Agreement	Displays the Synplify software license agreement.
License Wizard	Opens the License wizard, to help you request or install your license. See License Wizard Command, on page 3-86 for details.
License Editor	Lets you edit your current (node-locked) license information. See License Editor Command, on page 3-88 for details. *Available only in the Project view.*
Floating License Usage	Specifies the number of floating licenses currently being used and their users.

Command	Description
Preferred License Selection	Displays the floating licenses that are available for your selection. See Preferred License Selection Command, on page 3-89.
Tip of the Day...	Displays a daily tip on how to use the Synplify synthesis tools better. See Tip of the Day Command, on page 3-90.
About this program...	Displays the About Synplify dialog box, showing the synthesis tool product name, license expiration date, customer identification number, version number, and copyright. The dialog box also provides links to the Synplicity web site. Clicking the Versions button in the About Synplify dialog box displays the Version Information dialog box, listing the installation directory and the versions of all the synthesis tool compiler and mapper programs.

Help Command

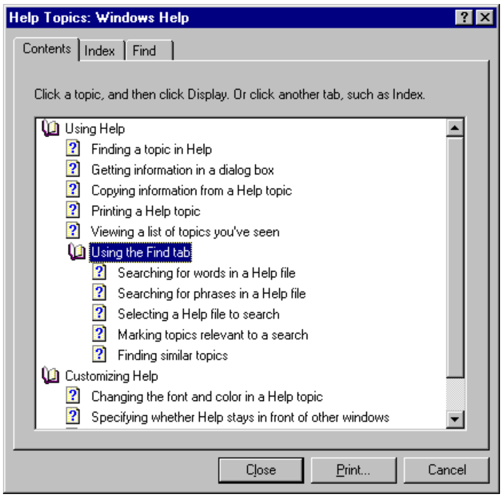
There are four help systems accessible from the Help menu:

- Help on the Synplify synthesis tool (Help -> Help)
- Help on standard Tcl commands (Help -> TCL Help)
- Help on error messages (Help->Error Messages)
- Help on using online help (Help -> How to Use Help)

In all four systems, the Help Topics dialog box has three panels which are described below: Contents Panel, on page 3-84, Index Panel, on page 3-85, and Find Panel, on page 3-86.

Contents Panel

Select the Contents panel to display the table of contents for the online help. Double-click a closed book ( ) icon to list its contents. Double-click an open book ( ) icon to close it. Double-click a document icon ( ) to display the corresponding document.



Help Topics: Windows Help

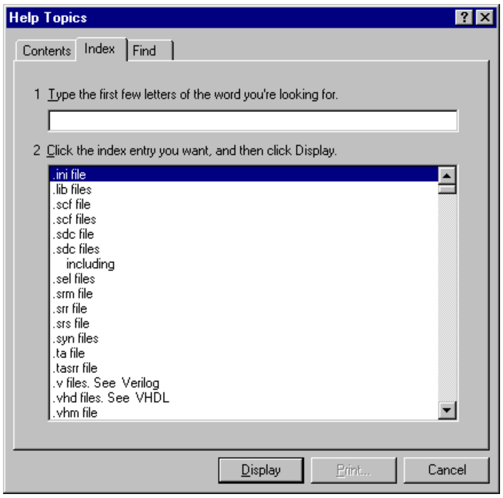
Contents | Index | Find

Click a topic, and then click Display. Or click another tab, such as Index.

- Using Help
  - Finding a topic in Help
  - Getting information in a dialog box
  - Copying information from a Help topic
  - Printing a Help topic
  - Viewing a list of topics you've seen
  - Using the Find tab
    - Searching for words in a Help file
    - Searching for phrases in a Help file
    - Selecting a Help file to search
    - Marking topics relevant to a search
    - Finding similar topics
- Customizing Help
  - Changing the font and color in a Help topic
  - Specifying whether Help stays in front of other windows

Index Panel

Select the Index panel to display an alphabetical index of principal topics. Type the first few letters of the topic you are searching for or scroll through the index to find a topic of interest. Double-click a topic to display it.



Help Topics

Contents | Index | Find

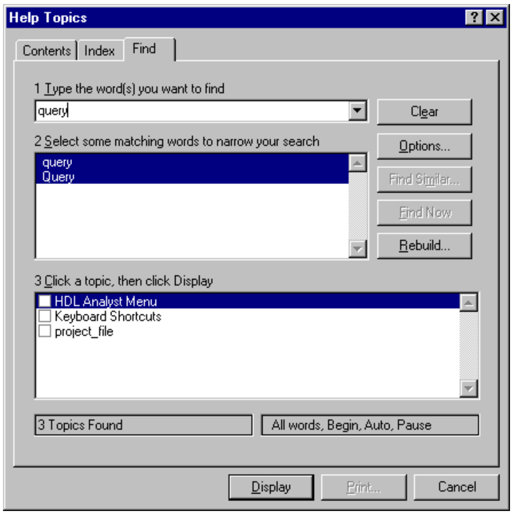
1 Type the first few letters of the word you're looking for.

2 Click the index entry you want, and then click Display.

.ini file
.lib files
.scf file
.scf files
.sdc file
.sdc files
  including
.sel files
.srm file
.srr file
.srs file
.syn files
.ta file
.tasrr file
.v files. See Verilog
.vhd files. See VHDL
.vhm file

Find Panel

You use the Find panel to search all of the text in the help database for a string. Enter a search string, then select a topic from the lower box to display the information.



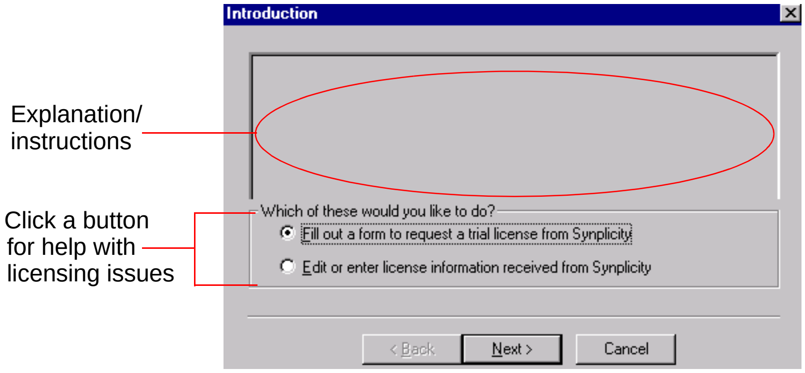
The Options button on the Find panel displays the Find Options dialog box, useful for multiple-word searches and searches based on word roots. In particular, it lets you perform boolean searches for all or at least one of several search terms, or search for occurrences that match the search string in different positions (start, end, containing).

License Wizard Command

You use the License wizard to do the following:

- Fill out a form to request a trial license from Synplicity
- Edit or enter license information received from Synplicity

Choose one of these actions in the License wizard Introduction dialog box, accessed by Help -> License Wizard, then follow the directions in subsequently displayed License wizard dialog boxes.

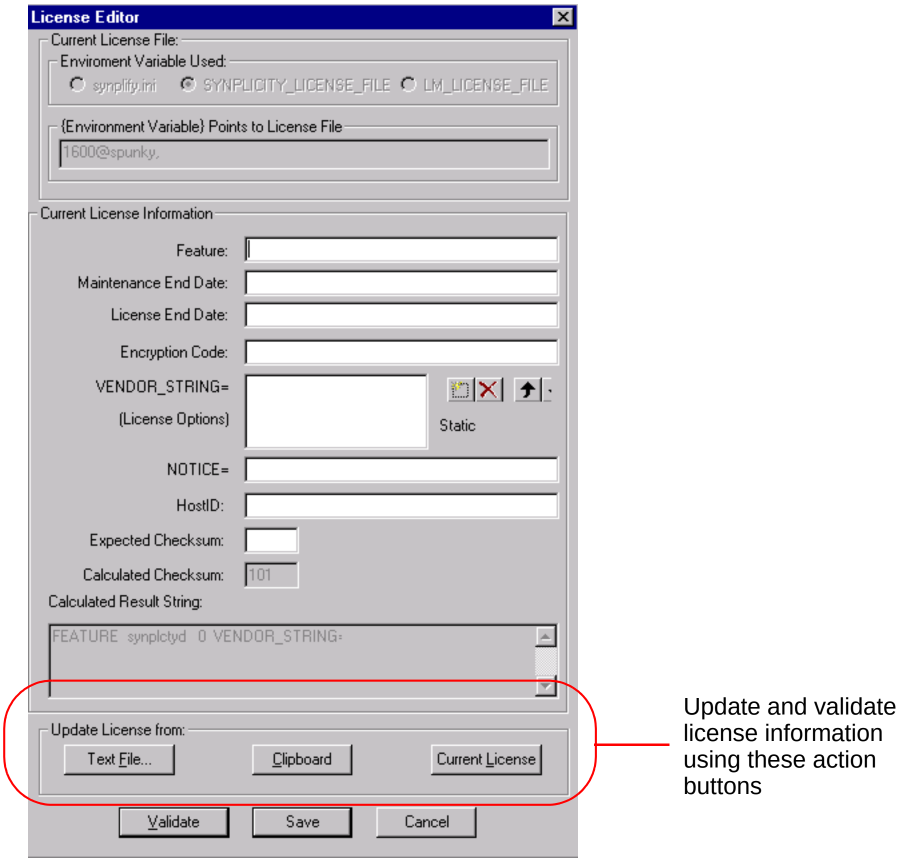

Introduction
Explanation/ instructions
Click a button for help with licensing issues
Which of these would you like to do?
Fill out a form to request a trial license from Synplicity
Edit or enter license information received from Synplicity
< Back
Next >
Cancel

If you choose Fill out a form to request a trial license from Synplicity, you will be asked for additional information regarding your context, such as name, company, host IDs, platforms, vendors, and languages. You can either fax or e-mail the request to Synplicity.

If you choose Edit or enter license information received from Synplicity, then the License Editor dialog box appears, where you can view and edit the license information. See License Editor Command, on page 3-88, for details.

License Editor Command

Select Help -> License Editor to view and edit your current license information (node-locked license only), using the License Editor dialog box. Refer to the product installation notes for the most current information.



License Editor

Current License File:

Enviroment Variable Used:

synplify.ini [✓ SYNPLICITY_LICENSE_FILE] LM_LICENSE_FILE

{Environment Variable} Points to License File

1600@spunky,

Current License Information

Feature:	
Maintenance End Date:	
License End Date:	
Encryption Code:	
VENDOR_STRING= (License Options)	
NOTICE=	
HostID:	
Expected Checksum:	
Calculated Checksum:	101

Calculated Result String:

```
FEATURE synplctyd 0 VENDOR_STRING=
```

Follow these steps to edit your current Synplicity license information:

1. If you are using the Microsoft® Windows® operating system and your license file is not in the standard license directory, specify the location with the variable SYNPLICITY_LICENSE_FILE or LM_LICENSE_FILE (Environment Variable Used option).

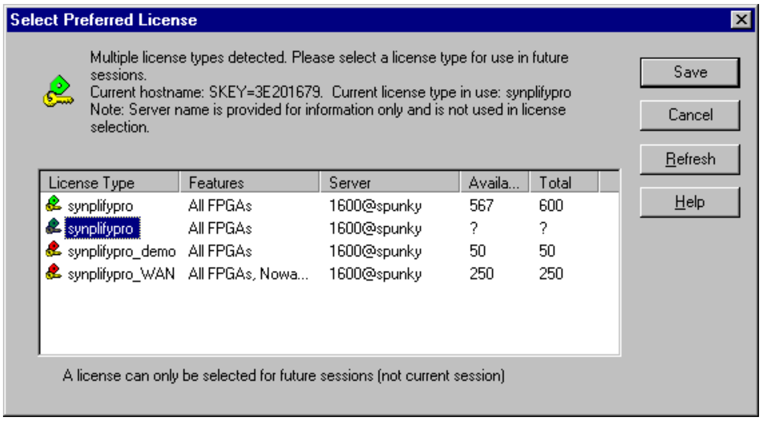
   If you are using a UNIX system, then specify the location of the license file with the variable LM_LICENSE_FILE.

2. Depending on whether your licensing information (Synplicity Software Authorization form) is in the form of an e-mail or a text file, enter the information as follows:

Format	Do this...
E-mail	Copy the licensing information. Click the Clipboard button. The licensing information from the copied document is automatically pasted into the dialog box.
Text file	Click the Text File button, then specify the name of the file. The licensing information in the file is automatically pasted into the dialog box.

Preferred License Selection Command

Select Help -> Preferred License to display the Select Preferred License dialog box, listing the available licenses for you to choose from. Select a license from the License Type column and click Save. Close and restart the Synplify synthesis tool. The new session uses the preferred license you selected.



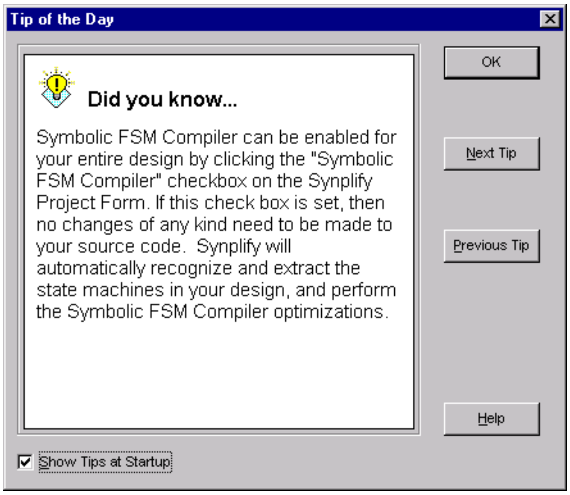
Multiple license types detected. Please select a license type for use in future sessions.
Current hostname: SKEY=3E201679. Current license type in use: synplifypro
Note: Server name is provided for information only and is not used in license selection.

License Type	Features	Server	Availa...	Total
synplifypro	All FPGAs	1600@spunky	567	600
synplifypro	All FPGAs	1600@spunky	?	?
synplifypro_demo	All FPGAs	1600@spunky	50	50
synplifypro_WAN	All FPGAs, Nowa...	1600@spunky	250	250

A license can only be selected for future sessions (not current session)

Tip of the Day Command

Select Help -> Tip of the Day to display the Tip of the Day dialog box, with a daily tip on how to best use the Synplify synthesis tool. This dialog box also displays automatically when you first start the tool. To prevent it from redisplaying at product startup, deselect Show Tips at Startup.



Did you know...

Symbolic FSM Compiler can be enabled for your entire design by clicking the "Symbolic FSM Compiler" checkbox on the Synplify Project Form. If this check box is set, then no changes of any kind need to be made to your source code. Synplify will automatically recognize and extract the state machines in your design, and perform the Symbolic FSM Compiler optimizations.

☑ Show Tips at Startup

Popup Menus

Popup menus, available by clicking the right mouse button, offer quick ways to access commonly used menu commands that are specific to the view where you click. Commands shown grayed out (dimmed) are currently inaccessible. Popup menu commands generally duplicate commands available from the regular menus, but sometimes have commands that are only available from the popup menu. The following table lists the popup menus:

Popup Menu	Description
Project view	See Project View Popup Menu, on page 3-91 for details
SCOPE window	Contains commonly used commands from the Edit menu. For information about these commands, see Edit Menu Commands for the SCOPE Spreadsheet, on page 3-12.
Text Editor window	See Text Editor Popup Menu, on page 3-96 for more information.
RTL and Technology views	See RTL and Technology Views Popup Menus, on page 3-97.

Project View Popup Menu

The popup menu commands available in the Project view are context-sensitive, depending on what is currently selected and where in the view you open the popup menu. Most commands duplicate commands from the File, Project, Run, and Options menus. The following table describes the Project view popup menu commands that are not described in other menus.

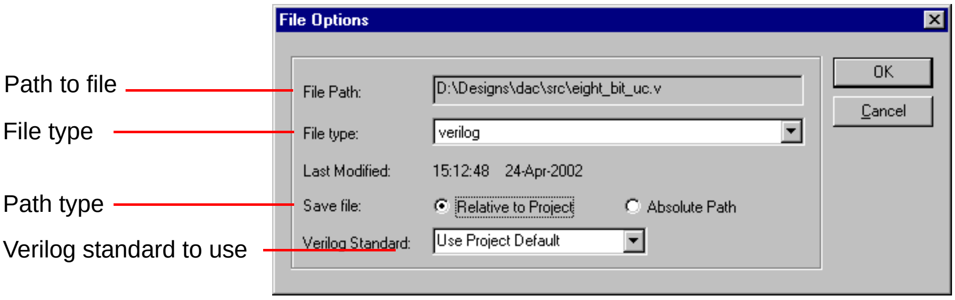
Command	Description
File Options...	When a source file or constraint file is selected, it displays the File Options dialog box that shows general properties of the selected file, such as its modification date, path, and type. See File Options Command, on page 3-93.
Copy File...	Displays the Copy File dialog box, where you copy the selected file and add it to the current project. You specify a new name for the file. See Copy File Command, on page 3-94.

Command	Description
Project Options...	Displays project properties, such as name and location. You can click a selected implementation to make it current. See Project Options Popup Menu Command, on page 3-95.
Build Workspace...	Creates a project workspace. In the Project view, select existing projects that you want to include in the project workspace. See Text Editor Popup Menu, on page 3-96.
Open as Text	Opens the selected file in the Text Editor, even if the file is of a type, such as `.sdc` or `.info`, that is normally viewed otherwise.
Change Implementation Name...	Displays the Implementation Name dialog box, where you rename the selected implementation. (See Change Implementation Name/Copy Implementation Command, on page 3-94.)
Copy Implementation	Copies the selected implementation and adds it to the current project with the name you specify in the dialog box. (See Change Implementation Name/Copy Implementation Command, on page 3-94.)
Remove Implementation	Removes the selected implementation from the project.

File Options Command

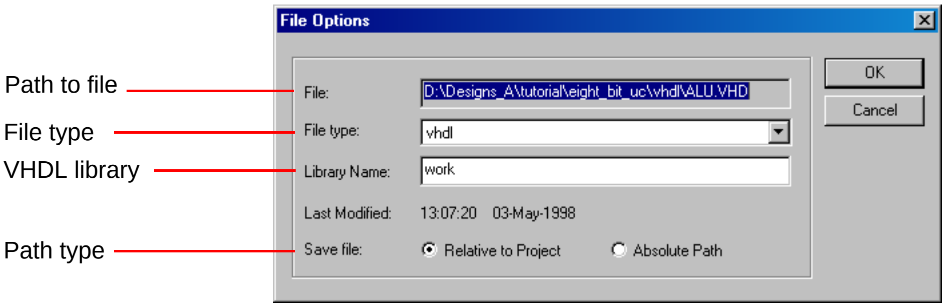
To display the File Options dialog box, right-click on a project file and select File Options from the popup menu. Specify the File type and the path type to use when listing the file in the project file (`.prj`).

Field/Option	Description
File Path	Path to the file.
File Type	The kind of file. The following choices are available: • `.v` and `.vhd` (Verilog and VHDL HDL files) • `.sdc` SCOPE constraint file • `.txt` (Text file) • `.tcl` (Tcl Script file) • `.prj` (Project file) Changing the file type does *not* change the file extension. It simply places the file in the corresponding Project view folder. For example, if you change the file type of a file from vhdl (`foo.vhd`) to verilog (`foo.v`), it retains the verilog extension and is moved from the vhdl folder to the verilog folder.
Library name (VHDL only)	Name of the VHDL library, which must be compatible with VHDL simulators. For VHDL files, the dialog box is the same as that accessed by Project -> Set VHDL Library – see Set VHDL Library Command, on page 3-43
Last modified	Date the file was last modified
Save file	The format for the path type: choose either Relative to Project or with an Absolute Path.



File Path: D:\Designs\dac\src\eight_bit_uc.v
File type: verilog
Last Modified: 15:12:48 24-Apr-2002
Save file: [✓ Relative to Project] Absolute Path
Verilog Standard: Use Project Default

Figure 3-21: File Options dialog box, Verilog

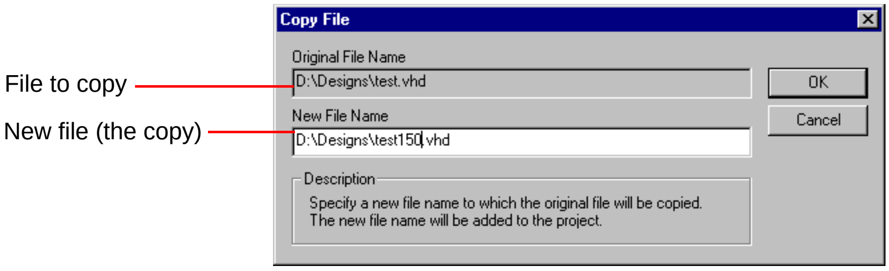


File: D:\Designs_A\tutorial\eight_bit_uc\vhdl\ALU.VHD
File type: vhdl
Library Name: work
Last Modified: 13:07:20 03-May-1998
Save file: (●) Relative to Project ( ) Absolute Path

Figure 3-22: File Options dialog box, VHDL

Copy File Command

With a file selected, select the Copy File popup menu command to copy the selected file and add it to the current project. This displays the Copy File dialog box where you specify the name of the new file.

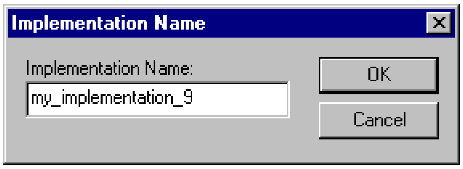


Original File Name
D:\Designs\test.vhd
New File Name
D:\Designs\test150.vhd
Description
Specify a new file name to which the original file will be copied.
The new file name will be added to the project.

Change Implementation Name/Copy Implementation Command

With an implementation selected, right-click and select the Change Implementation Name or Copy Implementation popup menu commands to display a dialog box where you specify the new name.

Change Implementation Name	The implementation name you specify is the new name for the implementation
Copy Implementation	The currently selected implementation is copied and saved to the project with the new implementation name you specify.

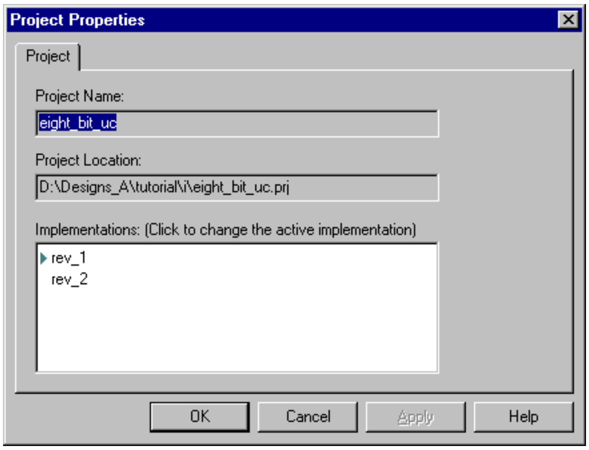


Implementation Name: my_implementation_9

Project Options Popup Menu Command

With a project or workspace selected, select the Project Options popup menu command to display the Project Properties dialog box and change the implementation of a project.

In the dialog box, select an implementation in the Implementations list, then click OK or Apply to make it the active implementation of the project.



Project Name: eight_bit_uc

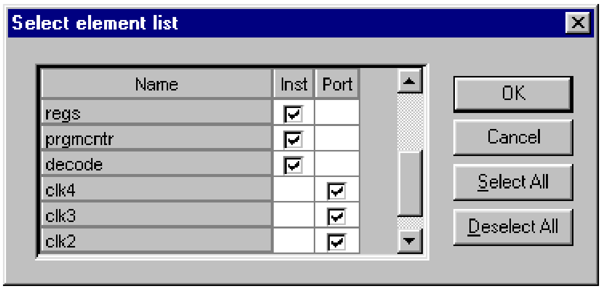
Project Location: D:\Designs_A\tutorial\i\eight_bit_uc.prj

Implementations: (Click to change the active implementation)
rev_1
rev_2

Text Editor Popup Menu

The popup menu in the Text Editor window contains the following commonly used text-editing commands from the Edit menu: Undo, Redo, Cut, Copy, Paste, and Toggle Bookmark. In addition, HDL Analyst specific commands appear when both an HDL Analyst view and it's corresponding HDL source file is open. For details of these commands, see Edit Menu Commands for the Text Editor, on page 3-12 and HDL Analyst Menu, on page 3-49.

The following table lists the commands that are unique to the popup menu:

Command	Description
Filter Analyst	Filters your design to show only the currently selected objects in the HDL text file. This is the same as HDL Analyst -> Filter Schematic.
Select in Analyst	Crossprobes from the Text Editor and selects the objects in the HDL Analyst view. To use this command, the Enhanced Text Crossprobing (option must be engaged.
Select from...	Displays the Select Element List dialog box showing the selected code elements from the text file. After you click OK, select Filter Analyst to update the schematic. 

Select element list

Name	Inst	Port
regs	✓	
prgmcntr	✓	
decode	✓	
clk4		✓
clk3		✓
clk2		✓

RTL and Technology Views Popup Menus

Some commands are only available from the popup menus in the RTL and Technology views, but most of the commands are duplicates of commands from the HDL Analyst, Edit, and View menus. The popup menus in the RTL and Technology views are nearly identical.

This section describes the following popup menu commands:

- Hierarchy Browser Popup Menu Commands, on page 3-97
- RTL View and Technology View Popup Menu Commands, on page 3-97

Hierarchy Browser Popup Menu Commands

The following commands become available when you right-click in the Hierarchy Browser of an RTL or Technology view. The Filter, Hide Instances, and Unhide Instances commands are the same as the corresponding commands in the HDL Analyst menu. The following commands are unique to this popup menu.

Command	Description
Collapse All	Collapses all trees in the Hierarchy Browser.
Expand All	Expands all trees in the Hierarchy Browser.
Reload	Refreshes the Hierarchy Browser. Use this if the Hierarchy Browser and schematic view do not match.

RTL View and Technology View Popup Menu Commands

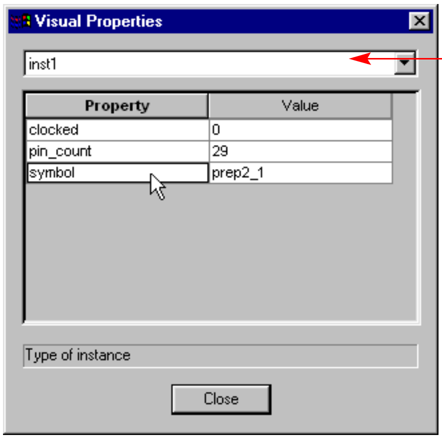
The commands on the popup menu are context-sensitive, and vary depending on the object selected, the kind of view, and where you click. In general, if you have a selected object and you right-click in the background, the menu includes global commands as well as selection-specific commands for the objects.

Most of the commands duplicate commands available on the HDL Analyst menu (see HDL Analyst Menu, on page 3-49). The following table lists the unique commands.

Command	Description
Properties	Displays the Visual Properties dialog box. See Properties Popup Menu Command, on page 3-98
SCOPE-> Edit Attributes (object <*name*>)	Opens a SCOPE window where you can enter attributes for the selected object. It displays the Select Constraint File dialog box (SCOPE -> Edit Attributes Command, on page 3-99), where you select the constraint file to edit. If no constraint file exists, you are prompted to create one. After you choose or create the constraint file, you can edit the attributes of the selected instance in the SCOPE Attributes dialog box. See SCOPE Attributes Dialog Box, on page 3-100.
Sheet *N* (where *N* is a sheet number)	Goes to the sheet numbered *N*, when you have a sheet connector selected. This is equivalent to picking the sheet in the Goto Sheet dialog box (View Sheets Command, on page 3-25).

Properties Popup Menu Command

The software displays property information about the selected object when you right-click on a net, instance, pin, or port in a HDL Analyst view. See Viewing Object Properties, on page 4-14 in the *Synplify User Guide* for more information about viewing object properties.



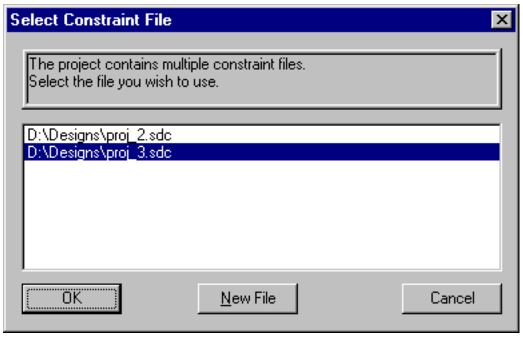
Property	Value
clocked	0
pin_count	29
symbol	prep2_1

Type of instance

Lists pins, if the selected object is an instance or net. Lists bits, if the selected object is a port.

SCOPE -> Edit Attributes Command

You use the Select Constraint File dialog box to choose or create a constraint file. You can open the constraint file and edit it.



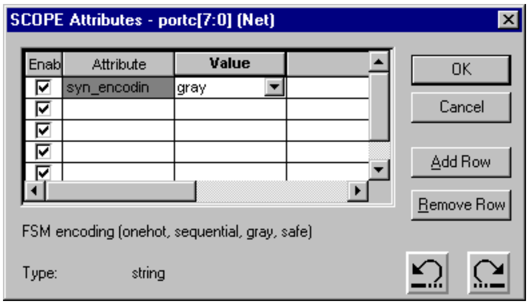
The project contains multiple constraint files.
Select the file you wish to use.

D:\Designs\proj_2.sdc
D:\Designs\proj_3.sdc

For more information about creating constraint files, see Setting Constraints in the SCOPE Window, on page 3-13 of the *Synplify Pro User Guide*.

SCOPE Attributes Dialog Box

You use the SCOPE Attributes dialog box to edit the attributes of an instance, port, or net selected in the RTL or Technology view.



Enab	Attribute	Value
✓	syn_encodin	gray
✓		
✓		
✓		
✓		

FSM encoding (onehot, sequential, gray, safe)

Type: string

Figure 3-23: SCOPE Attributes dialog box

Field/Option	Description
Enable	Enables the Attribute on the same line.
Attribute	Clicking the Attribute column displays a pulldown list of available attributes. You choose an attribute, then define its Value. The attribute type and description are indicated at the bottom of the dialog box.
Value	Clicking a cell in the Value column displays the default value for the attribute in the Attribute column. You can use the pulldown list to assign one of the available values.
Comment	You can enter a comment here.
Add Row	Adds an new, empty row above the current row in the table.
Remove Row	Removes the selected row from the table.
Undo	Undoes the last action.
Redo	Performs the action undone by Undo.

Chapter 4

Files

This chapter describes the input and output files used by the synthesis tool.

Input Files

The following table describes the input files used by the synthesis tool.

File	Extension	Description
Configuration/ Initialization	`.ini`	Governs the behavior of the synthesis tool. You normally do *not* need to edit this file; use the Schematic Options dialog box, instead, to customize behavior. See HDL Analyst Options Command, on page 3-73. On the Microsoft® Windows® operating system, the `.ini` file is in the `WINDOWS` (or `WINNT`) directory. On UNIX workstations, it is in the `windows` subdirectory of your home directory (`~/.synplicity`, where ~ is your home directory, which can be set with the environment variable `$HOME`).
Constraint	`.sdc`	Contains the timing constraints (clock parameters, I/O delays, and timing exceptions) in Tcl format. You can either create this file manually or generate it by entering constraints in the SCOPE window. For more information about constraint files, see Constraint Files, on page 7-29.
Project	`.prj`	Contains all the information required to complete a design. It is in Tcl format, and contains references to source files, compilation, mapping, and optimization switches, specifications for target technology and other runtime options.
Source files (VHDL)	`.vhd (`	Contains the design data in VHDL format. See VHDL, on page 4-3 and Chapter 10, *VHDL Language Support* for details.
Source files (Verilog)	`.v`	Contains the design data in VHDL format. For more information about the Verilog language, and the synthesis commands and attributes you can include, see Verilog, on page 4-4 and Chapter 9, *Verilog Language Support*.

HDL Source Files

The HDL source files for a project can be in either VHDL (.vhd) or Verilog (.v) format, but they must all be one or the other.

The Synplify synthesis tool contains built-in macro libraries for vendor macros like gates, counters, flip-flops, and I/Os. If you use the built-in macro libraries, you can easily instantiate vendor macros directly into the VHDL designs, and forward-annotate them to the output netlist. Refer to the appropriate vendor support documentation for more information.

VHDL

The Synplify synthesis tool supports a synthesizable subset of VHDL93 (IEEE 1076), and the following IEEE library packages:

- numeric_bit
- numeric_std
- std_logic_1164

The synthesis tool also supports the following industry standards in the IEEE libraries:

- std_logic_arith
- std_logic_signed
- std_logic_unsigned

The Synplify synthesis tool library contains an attributes package (*installation_dir*/lib/vhd/synattr.vhd) of built-in attributes and timing constraints that you can use with VHDL designs. The package includes declarations for timing constraints (including black-box timing constraints), vendor-specific attributes and synthesis attributes. To access these built-in attributes, add the following two lines to the beginning of each of the VHDL design units that uses them:

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

For more information about the VHDL language, and the synthesis commands and attributes you can include, see Chapter 10, *VHDL Language Support*. Here is additional reading material on VHDL:

- Ben Cohen, *VHDL Coding Styles and Methodologies,* Boston, MA: Kluwer Academic Publishers, 1995.
- *IEEE Standard VHDL Language Reference Manual (ANSI/IEEE Std 1076-1993),* New York, NY: The Institute of Electrical and Electronics Engineers, Inc, 1994.
- Sundar Rajan, *Essential VHDL: RTL Synthesis Done Right*, S & G Publishing, 1998.

Verilog

The Synplify synthesis tool supports a synthesizable subset of Verilog 2001 and Verilog 95 (IEEE 1364). For more information about the Verilog language, and the synthesis commands and attributes you can include, see Chapter 9, *Verilog Language Support*.

The Synplify synthesis tool contains built-in macro libraries for vendor macros like gates, counters, flip-flops, and I/Os. If you use the built-in macro libraries, you can instantiate vendor macros directly into Verilog designs and forward-annotate them to the output netlist. Refer to the User Guide for more information.

Here is some reading material on Verilog:

- Donald E. Thomas and Philip R. Moorby, *The Verilog Hardware Description Language, Second Edition,* Boston, MA: Kluwer Academic Publishers, 1995.
- Sternheim, Singh, Madhavan and Trivedi, *Digital Design and Synthesis with Verilog HDL,* San Jose, CA: Automata Publishing Company, 1993.
- *Verilog Hardware Description Language Reference Manual* (LRM) Version 2.0, Los Gatos, CA: Open Verilog International, 1993.

Output Files

The synthesis tool generates reports about the synthesis run and files that you can use for simulation or placement and routing..The following table describes the output files, categorizing them as either synthesis result and report files, or output files generated as input for other tools.

File	Description
Synthesis Results and Reports	
Synthesis log file (`.srr`)	Provides information on the synthesis run, as well as area and timing reports. See Log File, on page 4-7, for more information.
Hierarchical Area Report (`.areasrr`)	Reports area-specific information such as sequential and combinational ATOMS, RAMs, DSPs, and Black Boxes on each module in the design.
Vendor-specific results file	Results file that contains the synthesized netlist, written out in a format appropriate to the technology and the place-and-route you are using. Generally, the format is EDIF. Specify this file on the Implementation Results panel of the Options for Implementation dialog box (Implementation Results Panel, on page 3-32).
Compiler output file (`.srs`)	Output file after the compiler stage of the synthesis process. It contains an RTL-level representation of a design. This is the representation that appears graphically in an RTL view.
Mapping output files (`.srm`)	Output file after mapping. It contains the actual technology-specific mapped design. This is the representation that appears graphically in a Technology view.
Design component files (`.info`)	Design-dependent. Contains detailed information about design components like state machines or ROMs.
FSM information file (`.fse`)	Design-dependent. Contains information about state machine encodings.
Intermediate mapping files (`.srd`)	Used to save mapping information between synthesis runs. You do not need to use these files.

File	Description
Output Files Generated for Other Tools	
Constraints file for forward annotation	Contains synthesis constraints to be forward-annotated to the place-and-route tool. The constraints vary with the vendor and the technology. Refer to the vendor chapters for specific information about the constraints you can forward-annotate. Specify this file on the Implementation Results panel of the Options for Implementation dialog box (Implementation Results Panel, on page 3-32).
Mapped Verilog/VHDL netlist files (`.vm` or `.vhm`)	Optional post-synthesis netlist file in Verilog (`.vm`) or VHDL (`.vhm`) format. This is a structural netlist of the synthesized design, and differs from the original RTL you used as input for synthesis. Specify these files on the Implementation Results panel of the Options for Implementation dialog box (Implementation Results Panel, on page 3-32). Typically, you use this netlist for gate-level simulation, to verify your synthesis results. Some designers prefer to simulate before and after synthesis, and also after place-and-route. This approach helps them to isolate the stage of the design process where a problem occurred. The Verilog and VHDL output files are for functional simulation only. When you input stimulus into a simulator for functional simulation, use a cycle time for the stimulus of 1000 time ticks.

Log File

The log file is named *`project_name`*`.srr`, where *`project_name`* is the name of your project. The log file is written each time you compile or synthesize (compile and map) the design. When you compile a design without mapping it, the log file contains only compilation information, such as syntax errors and warnings.

The Synplify synthesis tool writes messages and all reports on synthesis, timing, and usage to the log file, located in the implementation directory. You can view the log file with the View Log button in the Project view (Buttons and Options, on page 2-36).

For further details about different parts of the log file, see the following:

For information about...	See...
Compiled files, messages (warnings, errors, and notes), user options set for synthesis, state machine extraction information, including a list of reachable states	Synthesis Run Information, on page 4-7
Buffers added to clocks in certain supported technologies.	Clock Buffering Report, on page 4-9
Buffers added to nets.	Net Buffering Report, on page 4-9
Timing results. This section of the log file begins with "`START TIMING REPORT`" section.	Timing Report, on page 7-47
Report of resources used by synthesis mapping.	Resource Usage Report, on page 4-10

Synthesis Run Information

The first section of the log file contains detailed information about the synthesis run, including the input files, the time taken to complete various stages, the options used for synthesis, and the output files that are generated after the run.. It is divided into the compiler phase and the mapper phase. Each phase includes any messages about problems detected during synthesis.

Compiler Phase

This phase starts with the compiler version and date, and includes the following:

- Project information: names of the source files, and the top-level module.
- Design information: HDL syntax and synthesis checks, black box instantiations, and inferred RAMs/ROMs. It also includes informational or warning messages about unused ports, removal of redundant logic, and latch inference. See Messages, on page 4-8 for details about the kinds of messages.

Mapper Phase

This phase begins with the mapper version and date, and reports the following:

- Project information: the names of the constraint files, target technology, and attributes set in the design.
- Design information: flattened instances, extraction of counters, FSM implementations, clock nets, buffered nets, replicated logic, flip-flop optimizations, and informational or warning messages. See Messages, on page 4-8 for details about the kinds of messages.

Messages

This section of the log file also includes any errors, warnings, notes, and informational messages detected during synthesis.

- Error messages begin with “`@E.`”
- Warning messages begin with “`@W.`”
- Notes begin with “`@N.`”
- Informational messages begin with “`@I.`”

Colors distinguish different types of messages:

Color	Message Type	Example
Blue	`Information (@I)` `Notes (@N)`	`@I::"D:\t\alu.v"` `@N:"D:\t\p.v":58:0:58:5|Trying to extract...`
Purple	`Warnings (@W)`	`@W:"D:\t\e.v":53:9:53:12|Creating black_b...`
Red	`Errors(@E)`	`@E:"D:\t\e.v":53:9:53:12|Reference to und...`

You can double-click a note, warning, or error to crossprobe to the corresponding HDL source code.

Clock Buffering Report

This section reports any clocks that were buffered. For example:

```
Clock Buffers:
Inserting Clock buffer for port clock0,TNM=clock0
```

Net Buffering Report

Net buffering reports are generated for most all of the supported FPGAs and CPLDs. This information is written in the log file, and includes the following information:

- The nets that were buffered or had their source replicated
- The number of segments created for that net
- The total number of buffers added during buffering
- The number of registers and look-up tables (or other cells) added during replication

Example: Net Buffering Report

```
Net buffering Report:
Badd_c[2] - loads: 24, segments 2, buffering source
Badd_c[1] - loads: 32, segments 2, buffering source
Badd_c[0] - loads: 48, segments 3, buffering source
Aadd_c[0] - loads: 32, segments 3, buffering source
Added 10 Buffers
Added 0 Registers via replication
Added 0 LUTs via replication
```

Timing Reports

A default timing report is written to the log file (*project_name*.srr) in the "START TIMING REPORT" section. See Timing Report, on page 7-47, for details.

Resource Usage Report

A resource usage report is written in the log file each time you compile or synthesize. The format of the resource usage report varies, depending on the architecture you are using. The report provides the following information:

- The total number of cells, and the number of combinational and sequential cells in the design
- The number of clock buffers and I/O cells
- Details of how many of each type of cell in the design

CHAPTER 5

Tcl Commands and Scripts

This chapter describes Tcl commands and scripts. These are the chapter topics:

Introduction to Tcl

Tcl (Tool Command Language) is a popular scripting language for controlling software applications. Synplicity has extended the Tcl command set with additional commands that you can use to run the Synplicity programs. These commands are not intended for use in controlling interactive debugging, but you can use them to run synthesis multiple times with alternate options to try different technologies, timing goals, or constraints on a design.

Tcl scripts are text files that have a "`tcl`" file extension (`.tcl`) and contain a set of Tcl commands designed to complete a task on set of tasks.

The Synplicity Tcl commands are described here. For information on the standard Tcl commands, syntax, language, and conventions, refer to the Tcl online help (Help -> TCL Help).

Tcl Conventions

Here is a list of conventions to respect when entering Tcl commands and/or creating Tcl scripts.

- Tcl is case sensitive.
- Comments begin with a hash mark or pound sign (#).
- Enclose all path names and filenames in double quotes (").
- Use a forward slash (/) as the separator between directory and path names (even on the Microsoft® Windows® operating system). For example:

  ```
  designs/big_design/test.v
  ```

Tcl Scripts and Batch Mode

For procedures for creating Tcl scripts and using batch mode, see Chapter 7 in the *Synplify User Guide*:

- Running Batch Mode on a Project File, on page 7-2
- Running Batch Mode with a Tcl Script, on page 7-3

- Generating a Job Script, on page 7-5
- Creating a Tcl Synthesis Script, on page 7-5
- Using Tcl Variables to Try Different Clock Frequencies, on page 7-7
- Running Bottom-up Synthesis with a Script, on page 7-9

Tcl Commands for Synthesis

You use the following Tcl commands to create and synthesize projects; add and open files; and control synthesis. The commands are listed in alphabetical order.

- add_file, on page 5-4
- constraint_file, on page 5-5
- get_env, on page 5-7
- get_option, on page 5-7
- project, on page 5-9
- project_data, on page 5-11
- project_file, on page 5-12
- set_option, on page 5-13
- Information on Vendor-specific Tcl Commands, on page 5-17

add_file

Adds one or more files to a project.

Syntax

add_file [**-verilog** | **-vhdl**] [**-lib** *libName*] [**-_include**] *fileName***.***ext* [*fileName***.***ext* [...]]

Files are added to the individual project directories according to their filename extensions as shown in the following table:

filename extension	**project directory**
`.v`	verilog
`.vhd`, `.vhdl`	vhdl
`.sdc`	constraint
`.tcl`	tcl script
`.srs`, `.srm`	RTL Netlist
all others	other

Multiple files can be added by separating the individual filenames with a space, and different file types (extensions) can be specified within the same command. Depending on the type of file to be added, the other command line arguments may or may not be available.

VHDL files have an associated library. The default library is work. The -lib option sets the VHDL library to *libName*.

The -_include argument indicates that the specified file is to be added to the project as an include file (include files are added to the include directory regardless of their extension). Include files are not passed to the compiler, but are assumed to be referenced from within the HDL source code. Adding an include file to a project, although not required, allows it to be accessed in the user interface where it can be viewed, edited, or cross-probed.

The -verilog and -vhdl arguments allow HDL files with other extensions to be added to the verilog and vhdl directories so that they can be compiled with the project. For example, the following command adds the file `alu.v.new` to the project's verilog directory:

```
% add_file -verilog /designs/megachip/alu.v.new
```

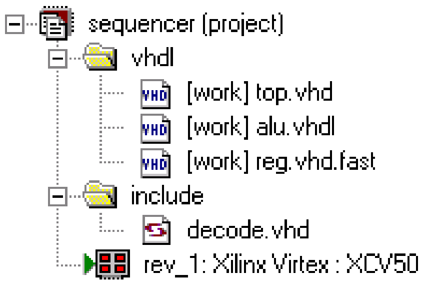
Without the -verilog argument, the file would be added to the other directory (.new is not a recognized Verilog extension), and the file would not be compiled with the existing Verilog files in the verilog directory.

Examples

Add a series of VHDL files to the vhdl directory and add an include file to the project:

```
% add_file /designs/sequencer/top.vhd
% add_file /designs/sequencer/alu.vhdl
% add_file -vhdl /designs/sequencer/reg.vhd.fast
% add_file -_include /designs/std/decode.vhd
```

The corresponding directory structure in the Project view is shown in the following figure:



```
sequencer (project)
    vhdl
        [work] top.vhd
        [work] alu.vhdl
        [work] reg.vhd.fast
    include
        decode.vhd
    rev_1: Xilinx Virtex : XCV50
```

constraint_file

Manipulates the constraint files used by the active implementation.

Syntax

constraint_file { **-enable** *constraintFileName* | **-disable** *constraintFileName* | **-list** | **-all** | **-clear** }

The following table describes the command options.

Table 5-1: constraint_file Tcl Command Options

Option	Description
-enable	Selects the specified constraint file to use for the active implementation.
-disable	Excludes the specified constraint file from being used for the active implementation
-list	Lists the constraint files used by the active implementation
-all	Selects (includes) all the project constraint files for the active implementation.
-clear	Clears (excludes) all the constraint files for the active implementation

Examples

List all constraint files added to a project, then disable one of these files for the next synthesis run.

```
% constraint_file -list
attributes.sdc clocks1.sdc clocks2.sdc eight_bit_uc.sdc

% constraint_file -disable eight_bit_uc.sdc
```

Disable all constraint files previously enabled for the project, then enable only one of them for the next synthesis run.

```
% constraint_file -clear

% constraint_file -enable clocks2.sdc
```

get_env

Reports the value of a predefined system variable.

Syntax

get_env *system_variable*

The following example shows you how to use the get_env command to see the value of the previously created MY_PROJECT environment variable. The MY_PROJECT variable contains the path to an HDL file directory, so get_env reports this path.

```
get_env MY_PROJECT
```

```
d:\project\hdl_files
```

In the project file, you can define a Tcl variable that contains the environment variable. In this example, my_project_dir contains the MY_PROJECT variable, which points to an HDL file directory.

```
set my_project_dir [get_env MY_PROJECT]
```

Then, use the $*system_variable* syntax to access the variable value. This is useful for specifying paths in your scripts, as in the following example which adds the file `myfile1.v` to the project.

```
add_file $my_project_dir/myfile1.v
```

get_option

Reports the settings of predefined project and device options. The options are the same as those for set_option. See set_option, on page 5-13 for details.

Syntax

get_option -*optionName*

hdl_param

Used to show or set HDL parameter overrides. For the GUI equivalent of this command, select Project->Implementation Options->Verilog/VHDL.

Syntax

hdl_param { **-add** [*paramName*] **|** **-list |** **-set** [*paramName*] [*paramValue*] **|**
-clear | **-override** }

The following table describes the command options.

Table 5-2: hdl_param Tcl Command Options

Option	Description
-add	Adds a parameter override to the project.
-list	Lists all the parameter overrides used in this project.
-set	Sets a parameter override and its value for the active implementation.
-clear	Clears all parameter overrides of the active implementation.
-overrides	Lists all the parameter override values used in this project.

project

Runs job flows to create, load, save, and close projects, and to change and examine project status.

Syntax

project { **-run** [-fg] [*mode*] [-tcl <*tclFile*>] |
-new [*projectPath*] | **-load** *projectPath* | **-close** | **-save** |
-result_file [*resultFilePath*] | **-log_file** [*logfileName*] |
-active [*projectName*] | **-dir** | **-file** | **-name** | **-list** | **-filelist** }

The following table describes the command options.

Table 5-3: project Tcl Command Options

Option	Description
-run [-fg] [*mode*] [-tcl <*tclFile*>]	Synthesizes the project, according to the specified options: • **-fg** – Synthesizes in the foreground. • **-tcl** – Synthesizes using the specified Tcl file. The *mode* is one of the following keywords: • **compile** – Compiles the active project, but does not map it. • **synthesis** – Default mode if no mode is specified. Compiles (if necessary) and synthesizes the currently active project. If followed by the -clean option (project -run synthesis -clean), resynthesizes the entire project, including the top level and *all compile points*, whether or not their constraints, implementation options or source code changed since the last synthesis. If not followed by -clean, only compile points that have been modified are resynthesized. • **synthesis_check** – Verifies that the design is functionally correct; errors are reported in the log file. • **syntax_check** – Verifies that the HDL is syntactically correct; errors are reported in the log file. • **timing** – Runs the Timing Analyst. This is equivalent to clicking the Generate button in the Timing Analyst dialog box using the default values. •
-new <*projectPath*>	Creates a new project in the current working directory. If *projectPath* is specified, creates the project in the specified project directory.

Table 5-3: project Tcl Command Options (Continued)

Option	Description
-load *<projectPath>*	Loads the project file specified by *projectPath*.
-close	Closes the currently active project.
-save	Saves the currently active project.
-log_file *<file>*	Reports the name of the project log file. If *file* is specified, changes the base name of the log file.
-active *<projectName>*	Shows the active project. If *projectName* is specified, makes the specified project the active project.
-result_file *<path>*	Names the synthesis result file. If *path* is specified, changes the name/location of the result file to the path specified.
-dir	Shows the project directory for the active project.
-file	Shows the active project filename.
-name	Returns the active project name.
-list	Returns a list of the loaded projects.
-insert	Adds selected projects to the workspace project.
-fileorder *<file1>* *<file2>* **...**	Reorder files by adding to the end of the project file list.
-movefile *<file1>* *<file2>*	Move *file1* to follow *file2* in HDL file list. If *file2* is not specified, move to top of the list.
-removefile *<file>*	Remove file from project.
-addfile *<file>*	Add file to project.

Examples

Load the project `top.prj` and compile the design without mapping it. Compiling makes it possible to create an `.sdc` file with the SCOPE spreadsheet and display an RTL schematic representation of the design.

```
% project -load top.prj
% project -run compile
```

Load a project and synthesize the design.

```
% project -load top.prj
%% project -run synthesis
```

In the example above, you can also use the command project -run, since the default is synthesis.

project_data

Used to show or set properties of a project.

Syntax

project_data { **-active** [*projectName*] **| -dir | -file** }

The following table describes the command options.

Table 5-4: project_file Tcl Command Options

Option	Description
-active	Set/show active project. With no argument, shows the active project. If *projectName* is specified, changes the active project to *projectName*.
-dir	Show directory of active project.
-file	Show the project file for the active project. The full path is included with the file name.

project_file

Used to manipulate and examine project files.

Syntax

project_file { **-lib** *fileName* [*libName*] **| -name** *fileName* [*newPath*] **|**
-time *fileName* [*format*] **| -date** *fileName* **|**
-type *fileName* **| -move** *fileName1* [*fileName2*] **|**
-remove *fileName* }

The following table describes the command options.

Table 5-5: project_file Tcl Command Options

Option	Description
-lib	Shows the project file library associated with *fileName*. If *libName* is specified, changes the project file library for the specified file to *libName*.
-name	Shows the project file path for the specified file. If *newPath* is specified, changes the location of the specified project file to the directory path specified by *newPath*.
-time	Shows the file time stamp. If a *format* is specified, changes the composition of the time stamp according to the combination of the following time formatting codes: **%H** (hour 00-23) **%M** (minute 00-59) **%S** (second 00-59) **%d** (day 01-31) **%b** (abbreviated month) **%Y** (year with century).
-date	Shows the file date.
-type	Shows the file type.
-move	Positions *fileName1* after *fileName2* in HDL file list. If *fileName2* is not specified, moves *fileName1* to the top of the list.
-remove	Removes the specified file from the project file list.

Examples

List the files added to a project. Remove a file.

```
% project -filelist
path_name1/cpu.v path_name1/cpu_cntrl.v path_name2/cpu_cntrl.vhd

% project_file -remove path_name2/cpu_cntrl.vhd
```

set_option

Sets the settings of predefined project and device options.

Syntax

set_option **-***optionName* *optionValue*

The following table lists the generic arguments for setting the device technology, part, and speed grade. These options are equivalent to the options available on the Device panel of the Options for implementation dialog box (see Device Panel, on page 3-30).

Table 5-6: set_option Tcl Command Options: Technology, Part, Speed Grade

Option Name	Description
-technology *keyword*	Sets the target technology for the implementation. *Keyword* is the vendor architecture. Refer to the appropriate vendor chapter or check the Device panel of the Options for implementation dialog box (see Device Panel, on page 3-30) for a list of supported families.
-part *part_name*	Specifies a part for the implementation. Check the Device panel of the Options for implementation dialog box (see Device Panel, on page 3-30) for available choices.
-speed_grade -*value*	Sets the speed grade for the implementation. Check the Device panel of the Options for implementation dialog box (see Device Panel, on page 3-30) for available choices.

Table 5-6: set_option Tcl Command Options: Technology, Part, Speed Grade

Option Name	Description
-package *value*	Sets the package for the implementation. This option is not available for certain vendor families, because it is set in the place-and-route software. Check the Device panel of the Options for implementation dialog box (see Device Panel, on page 3-30) for available choices.
-grade -*value*	Same as -speed_grade. Included for backwards compatibility.

The following table is an alphabetical list of other arguments to the set_option and get_option commands, with brief descriptions and GUI equivalents from the Options for implementation dialog box. Some options are technology-specific, and others have technology-specific defaults or limitations. The table in Information on Vendor-specific Tcl Commands, on page 5-17 summarizes where to go for vendor-specific details.

Table 5-7: set_option and get_option Tcl Command Options

Option	Description	Impl Options Equivalent
-areadelay *percent_value* **-area_delay_percent** *percent_value*	Sets the percentage of paths you want optimized. This option is available only in certain device technologies.	Percent of design to optimize for timing, Device Panel. See also Design Guidelines, on page 5-2 of the *Synplify User Guide*.
-autosm 1 | 0 **-symbolic_fsm_compiler 1 | 0**	Enables/disables the FSM compiler.	FSM Compiler check box, Options Panel.
-block 1 | 0 **-disable_io_insertion 1 | 0**	Enables/disables I/O insertion in some technologies.	Disable I/O Insertion check box, Device Panel.
-compiler_compatible 0 | 1	Disables pushing of tristates across process/block boundaries.	*Complement* of the Push Tristates Across Process/ Block Boundaries check box, VHDL Panel and Verilog Panel

Table 5-7: set_option and get_option Tcl Command Options (Continued)

Option	Description	Impl Options Equivalent
-default_enum_encoding default | onehot | gray | sequential	(VHDL only) Sets the default for enumerated types.	Default Enum Encoding, VHDL panel (see VHDL Panel and Verilog Panel).
-disable_io_insertion 1 | 0 **-block 1 | 0**	Enables/disables I/O insertion in some technologies.	Disable I/O Insertion, Device Panel.
-fanout_guide *value* **-maxfan** *value*	Sets the fanout limit guideline for the current project.	Fanout Guide, Device Panel.
-frequency *value*	Sets the global frequency.	Frequency, Constraints Panel.
-include_path *path*	(Verilog only) Defines the search path used by the ‘include commands in Verilog design files. Argument *path* is a string that is a semicolon-delimited list of directories.	Include Path Order, Verilog panel (see *VHDL Panel* and Verilog Panel, on page 3-37).
-maxfan *value* **-fanout_guide** *value*	Sets the fanout limit guideline for the current project.	Fanout Guide, Device Panel.
-maxfan_hard 1	For Actel designs, it specifies that the -maxfan value is a hard fanout limit that the Synplify synthesis tool must not exceed.	Hard Limit to Fanout, Device Panel.
-num_critical_paths *value*	Specifies the number of critical paths to report in the timing report.	Number of Critical Paths, Timing Report Panel.
-num_startend_points *value*	Specifies the number of start and end points to include when reporting paths with the worst slack in the timing report.	Number of Start/End Points, Timing Report Panel.

Table 5-7: set_option and get_option Tcl Command Options (Continued)

Option	Description	Impl Options Equivalent
-opcond *value*	Sets the operating condition for device performance in the areas of optimization, timing analysis, and timing reports. This option applies only to the Actel ProASIC (500K) and ProASIC Plus (PA) technologies. Values are Default, MIL-WC, IND-WC, COM-WC, and Automotive-WC.	Device Panel
-report_path *integer*	Sets the maximum number of critical paths in a forward-annotated SDF constraint file Actel 500K and PA designs.	Max Number of Critical Paths in SDF, Device Panel.
-resource_sharing 1 | 0	Enables/disables resource sharing.	Resource Sharing, Device Panel.
-result_file *filename*	Specifies the name of the results file.	Result File Name and Result Format, Implementation Results Panel.
-symbolic_fsm_compiler 1 | 0 **-autosm 1 | 0**	Enables/disables the FSM compiler.	FSM Compiler check box, Options Panel.
-synthesis_onoff_pragma 1 | 0	Determines whether code between synthesis on/off directives is ignored.	Synthesis on/off Implemented as Translate on/Off, VHDL Panel.
-top_module *name*	Specifies the top-level module.	Top-level Entity/Module, Device Panel.

Table 5-7: set_option and get_option Tcl Command Options (Continued)

Option	Description	Impl Options Equivalent
-vlog_std v2001 | v95	When enabled, the default Verilog standard for the project is Verilog 2001. When disabled, the default standard is Verilog 95. When not specified, the value is v95 for an existing project, v2001 for a new project.	Use Verilog 2001, VHDL Panel and Verilog Panel.
-write_apr_constraint 1 | 0	Writes vendor-specific constraint files.	Write Vendor Constraint File, Implementation Results Panel.
-write_verilog 1 | 0 **-write_vhdl 1 | 0**	Writes Verilog or VHDL mapped netlists.	Write Mapped Verilog/VHDL Netlist, Implementation Results Panel.

Information on Vendor-specific Tcl Commands

You can find vendor-specific Tcl commands in the appropriate vendor appendix.

Table 5-8: Vendor-specific Tcl Commands

Vendor/Family	Tcl Commands Described in...
Actel	Actel-specific Tcl Command Options, on page 11-9

Tcl Script Examples

This section provides several examples of Tcl scripts.

Using Several Target Technologies

```
# Run synthesis multiple times without exiting while trying different
# target technologies. View their implementations in the HDL Analyst tool.

# Open a new Project.
   project -new

# Set the design speed goal to 33.3 MHz.
   set_option -frequency 33.3

# Add a Verilog file to the source file list.
   add_file -verilog "D:/test/simpletest/prep2_2.v"

# Create a new Tcl variable, called $try_these, used to synthesize
# the design using different target technologies.

   set try_these {

         ISPGDX APEX20K Virtex2 # 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
   }
```

Different Clock Frequency Goals

```
# Run synthesis six times on the same design using different clock
# frequency goals. Check to see what the speed/area tradeoffs are for
# the different timing goals.

# Load an existing Project. This Project was created from an
# interactive session by saving the Project file, after adding all the
# necessary files and setting options in the Project -> Options for
# implementation dialog box.


   project -load "design.prj"

# Create a Tcl variable, called $try_these, that will be used to
# synthesize the design with different frequencies.
   set try_these {
      20.0
      24.0
      28.0
      32.0
      36.0
      40.0
   }

# Loop through each frequency, trying each one
   foreach frequency $try_these {

# Set the frequency from the try_these list
   set_option -frequency $frequency

# Since I want to keep all Log Files, save each one. Otherwise
# the default Log File name "<project_name>.srr" is used, which is
# overwritten on each run. Use the name "<$frequency>.srr" obtained from
the
# $try_these Tcl variable.
   project -log_file $frequency.srr

# Run synthesis.
   project -run

# Display the Log File for each synthesis run
   open_file -edit_file $frequency.srr
   }
```

Setting Options and Timing Constraints

```
# Set a number of options and use timing constraints on the design.

# Open a new Project
   project -new

# Set the target technology, part number, package, and speed grade options.
   set_option -technology VIRTEX2
   set_option -part XC2V40
   set_option -package CS144
   set_option -speed_grade -6

# Load the necessary VHDL files. Add the top-level design last.
   add_file -vhdl "statemach.vhd"
   add_file -vhdl "rotate.vhd"
   add_file -vhdl "memory.vhd"
   add_file -vhdl "top_level.vhd"

# Add a timing Constraint file and vendor-specific attributes.
   add_file -constraint "design.sdc"

# The top level file ("top_level.vhd") has two different designs, of
# which the last is the default entity. Try the first entity (design1)
# for this run. In VHDL, you could also specify the top level architecture
# using <entity>.<arch>
   set_option -top_module design1

# Turn on the Symbolic FSM Compiler to re-encode the state machine
# into one-hot.
   set_option -symbolic_fsm_compiler true

# Set the design frequency.
   set_option -frequency 30.0

# Save the existing Project to a file. The default synthesis Result File
# is named "<project_name>.<ext>". To name the synthesis Result File
# something other than "design.xnf", use project -result_file "<name>.xnf"
   project -save "design.prj"

# Synthesize the existing Project
   project -run

# Open an RTL View
   open_file -rtl_view

# Open a Technology View
   open_file -technology_view
```

```
# --------------------------------------------------
# This constraint file, "design.sdc," is read by "test3.tcl"
# with the add_file -constraint "design.sdc" command. Constraint files
# are for timing constraints and synthesis attributes.
# --------------------------------------------------
#  Timing Constraints:
# --------------------------------------------------
# The default design frequency goal is 30.0 MHz for four clocks. Except
# that clk_fast needs to run at 66.0 MHz. Override the 30.0 MHz default
# for clk_fast.
   define_clock {clk_fast} -freq 66.0

# The inputs are delayed by 4 ns
   define_input_delay -default 4.0

# except for the "sel" signal, which is delayed by 8 ns
   define_input_delay {sel} 8.0

# The outputs have a delay off-chip of 3.0 ns
   define_output_delay -default 3.0

# From a previous run it was noticed in Technology View that the critical
# paths are the flip-flop to flip-flop paths going to register "inst3.q[0]"
# (in the memory). Improve the paths going to inst3.q[0] by 3.0 ns.
   define_reg_input_delay {inst3.q[0]} -improve 3.0



# Set the maximum fanout to 10000.
   define_attribute {clk_slow} syn_maxfan 10000
```

Bottom-up Synthesis

```
# Bottom-up synthesis of a large design.

# The Source command reads in other Tcl scripts. Each of these scripts does
# a compile of one logic block and has its own constraint file.
   source "statemach.tcl"
   source "microproc.tcl"
   source "handshake.tcl"
   source "fifo.tcl"
   source "cherstrp.tcl"

# After synthesizing the individual logic blocks, create a Project for the
# top-level design.
   project -new
```

```
# Add the top level VHDL file.
   add_file -vhdl top_level.vhd

# Add the top level global constraint file.
   add_file -constraint top_level.sdc

# Set the top level options
   set_option -technology FLEX10K
   set_option -part EPF10K70
   set_option -speed_grade -3
   set_option -frequency 50.0
   set_option -symbolic_fsm_compiler true

# Set the output file information
   project -result_file top_level.edf
   project -log_file top_level.srr

# Save the Project to file
   project -save top_level.prj

# Run the Project
   project -run

# Open the top level RTL and Technology Views
   open_file -rtl_view
   open_file -technology_view

# --------------------------------------------------
# This file, "statemach.tcl," is read by "bottom_up.tcl," (the
# bottom up Tcl script) with the command "source statemach.tcl."
# The other .tcl scripts are similar.
# --------------------------------------------------
# Open a new Project for "statemach"
   project -new

# Add the VHDL file for this logic block.
   add_file -vhdl statemach.vhd

# Add the constraint file for this logic block.
   add_file -constraint statemach.sdc

# Set the other options for "statemach".
   set_option -technology FLEX10K
   set_option -part EPF10K70
   set_option -speed_grade -3
   set_option -frequency 50.0
   set_option -symbolic_fsm_compiler true
```

```
# Set the Project outputs
   project -result_file statemach.edf
   project -log_file statemach.srr

# Save this Project
   project -save statemach.prj

# Run this Project
   project  -run

# --------------------------------------------
# This file (statemach.sdc) is the constraint file read by
# "statemach.tcl," with the command add_file -constraint statemach.sdc.
# This constraint file is specific to this logic block " statemach "
# --------------------------------------------
# Timing Constraints:
# --------------------------------------------
   define_input_delay -default -100
   define_output_delay -default -100
   define_input_delay RESET -10
   define_reg_input_delay {q[8]} -improve 4.0
```

CHAPTER 6

HDL Analyst Operations

The HDL Analyst tool helps you examine your design and synthesis results, and analyze how you can improve design performance and area. For the Synplify synthesis tool, the HDL Analyst is a separate option and requires an additional license. Enter the HDL Analyst license in your Synplify tool license file. If you need assistance, ask your system administrator or contact Synplicity at this e-mail address:`support@synplicity.com`

This chapter describes the HDL Analyst tool and the operations you can perform with it.

For additional information, see the following:

- Descriptions of the HDL Analyst commands in Chapter 3, *User Interface Commands*:
- Chapter 4, *Result Analysis* in the *Synplify User Guide*

HDL Analyst Views and Commands

The HDL Analyst tool graphically displays information in two schematic views: the RTL and Technology views (see RTL View, on page 2-4 and Technology View, on page 2-6 for information). The graphic representation is useful for analyzing and debugging your design, because you can visualize where coding changes or timing constraints might reduce area or increase performance.

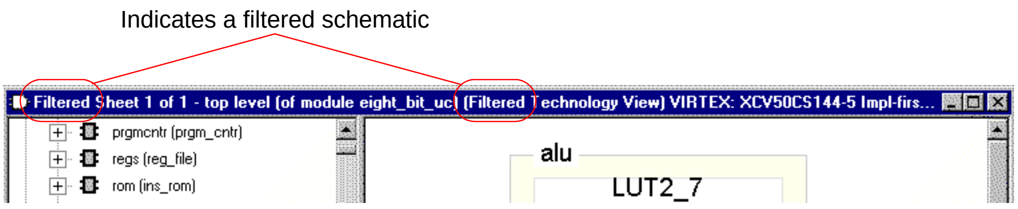
This section gives you information about the following:

- Filtered and Unfiltered Schematic Views, on page 6-2
- Accessing HDL Analyst Commands, on page 6-3

Filtered and Unfiltered Schematic Views

HDL Analyst views (RTL View, on page 2-4 and Technology View, on page 2-6) consist of schematics that let you analyze your design graphically. The schematics can be filtered or unfiltered. The distinction is important because the kind of view determines how objects are displayed for certain commands.

- Unfiltered schematics display all the objects in your design, at appropriate hierarchical levels.
- Filtered schematics show only a subset of the objects in your design, because the other objects have been filtered out by some operation. The Hierarchy Browser in the filtered view always list all the objects in the design, not just the filtered objects. Some commands, such as HDL Analyst -> Show Context, are only available in filtered schematics. Views with a filtered schematic have the word Filtered in the title bar.


Indicates a filtered schematic
Filtered Sheet 1 of 1 - top level (of module eight_bit_uc) (Filtered Technology View) VIRTEX: XCV50CS144-5 Impl-firs...
prgmcntr (prgm_cntr)
regs (reg_file)
rom (ins_rom)
alu
LUT2_7

Filtering commands affect only the displayed schematic, not the underlying design. For a detailed description of filtering, see Filtering and

Flattening Schematics, on page 6-28. For procedures on using filtering, see in the *Synplify User Guide*.

Accessing HDL Analyst Commands

You can access HDL Analyst commands in many ways, depending on the active view, the currently selected objects, and other design context factors. The software offers these alternatives to access the commands:

- HDL Analyst and View menus
- HDL Analyst popup menu
  To access it, right-click in an HDL Analyst view. The popup menu is context-sensitive, and includes commonly used commands from the HDL Analyst and View menus, as well as some additional commands.
- HDL Analyst toolbar icons
  The icons provide shortcuts to commonly used commands

For brevity, this document primarily refers to the menu method of accessing the commands and does not list alternative access methods.

See also:

- HDL Analyst Menu, on page 3-49
- View Menu, on page 3-20
- RTL and Technology Views Popup Menus, on page 3-97
- Analyst Toolbar, on page 2-24

Schematic Objects and Their Display

Schematic objects are the objects that you manipulate in an HDL Analyst schematic: instances, ports, and nets. Instances can be categorized in different ways, depending on the operation: hidden/unhidden, transparent/opaque, or primitive/hierarchical. The following topics describe schematic objects and the display of associated information in more detail:

- Object Information, on page 6-4
- Sheet Connectors, on page 6-5
- Primitive and Hierarchical Instances, on page 6-6
- Hidden Hierarchical Instances, on page 6-9
- Transparent and Opaque Display of Hierarchical Instances, on page 6-7
- Schematic Display, on page 6-9

For most objects, you select them to perform an operations. For some objects like sheet connectors, you do not select them but right-click on them and select from the popup menu commands.

Object Information

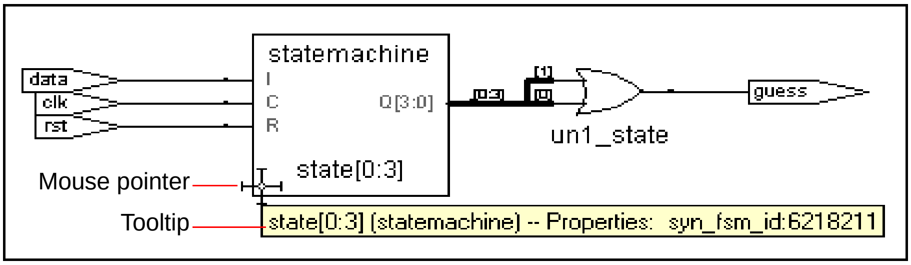
To obtain information about specific objects, you can view object properties with the Properties command from the right-click popup menu, or place the pointer over the object and view the object information displayed. With the latter method, information about the object displays in these two places until you move the pointer away:

- The status bar at the bottom of the synthesis window
  Displays the name of the instance, net, port, or sheet connector and other relevant information. Here is an example of the status bar information for a net:

  ```
  Net clock (local net clock) Fanout=4
  ```

  You can enable and disable the display of status bar information by toggling the command View -> Status Bar.

- In a tooltip at the mouse pointer
  Displays the name of the object and any attached attributes. The following figure shows tooltip information for a state machine:



state[0:3] (statemachine) -- Properties: syn_fsm_id:6218211

Figure 6-1: Tooltip Information for a State Machine

To disable tooltip display, select View -> Toolbars and disable the Show Tooltips option. Do this if you want to reduce clutter.

See also

- Pin and Pin Name Display for Opaque Objects, on page 6-10
- HDL Analyst Options Command, on page 3-73

Sheet Connectors

Whenever the HDL Analyst tool divides a schematic into multiple sheets, sheet connector symbols indicate how sheets are related. A sheet connector differs from a port symbol by having an empty diamond or a hexagon with sheet numbers at one end.

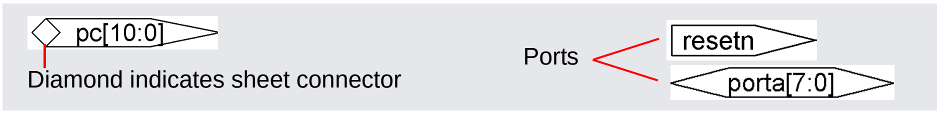


Figure 6-2: Sheet Connector Symbol Compared to Port Symbol

If you enable the Show Sheet Connector Index option in the (Options->HDL Analyst Options), the empty diamond becomes a hexagon with a list of the connected sheets. You go to a connecting sheet by right-clicking a sheet connector and choosing the sheet number from the popup menu. The menu has as many sheet numbers as there are sheets connected to the net at that point.

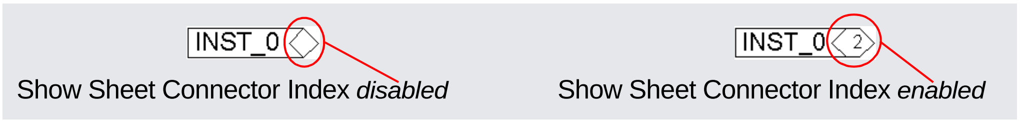

INST_0
Show Sheet Connector Index *disabled*
INST_0
2
Show Sheet Connector Index *enabled*

Figure 6-3: Effect of the Show Sheet Connector Index Option

See also

- Multiple-sheet Schematics, on page 6-18
- HDL Analyst Options Command, on page 3-73
- RTL and Technology Views Popup Menus, on page 3-97

Primitive and Hierarchical Instances

HDL Analyst instances are either primitive or hierarchical, and sorted into these categories in the Hierarchy Browser. Under Instances, the browser first lists hierarchical instances, and then lists primitive instances under Instances ->Primitives.

Primitive Instances

Although some primitive objects have hierarchy, the term is used here to distinguish these objects from *user-defined* hierarchies. Primitive instances include the following:

- Logic primitives, such as XOR gates
- Technology-specific primitives, such as LUTs
- Inferred ROMs, RAMs, and state machines
- Black boxes

In a schematic, logic gate primitives are represented with standard schematic symbols, and technology-specific primitives with various symbols (see Hierarchy Browser Symbols, on page 2-9). You can push into primitives like technology-specific primitives, inferred ROMs, and inferred state machines to view internal details. You cannot push into logic primitives.

Hierarchical Instances

Hierarchical instances are user-defined hierarchies; all other instances are considered primitive. Hierarchical instances correspond to Verilog modules and VHDL entities.

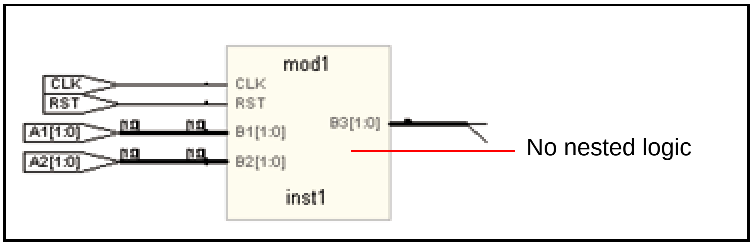
The Hierarchy Browser lists hierarchical instances under Instances, and uses this symbol: . In a schematic, the display of hierarchical instances depends on the combination of the following:

- Whether the instance is transparent or opaque. Transparent instances show their internal details nested inside them; opaque instances do not. You cannot directly control whether an object is transparent or opaque; the views are automatically generated by certain commands. See Transparent and Opaque Display of Hierarchical Instances, on page 6-7 for details.
- Whether the instance is hidden or not. This is user-controlled, and you can hide instances so that they are ignored by certain commands. See Hidden Hierarchical Instances, on page 6-9 for more information.

Transparent and Opaque Display of Hierarchical Instances

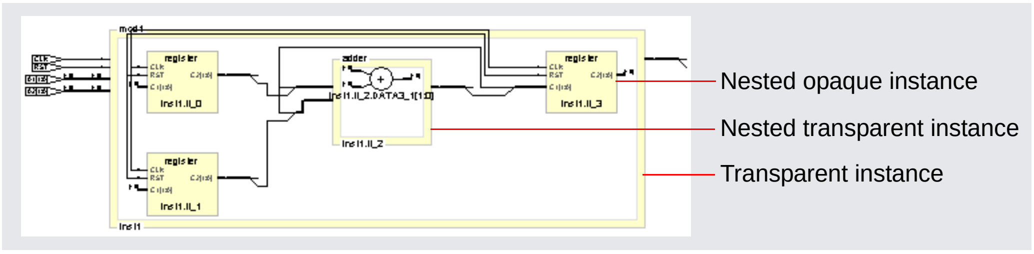
A hierarchical instance can be displayed transparently or opaquely. You cannot directly control the display; certain commands cause instances to be transparent. The distinction between transparent and opaque is important because some commands operate differently on transparent and opaque instances. For example, in a filtered schematic Flatten Current Schematic flattens only transparent hierarchical instances.

- Opaque instances are pale yellow boxes, and do not display their internal hierarchy. This is the default display.



- Transparent instances display some or all their lower-level hierarchy nested inside a hollow box with a pale yellow border. Transparent instances are only displayed in filtered schematics, and are a result of certain commands. See Looking Inside Hierarchical Instances, on page 6-26 for information about commands that generate transparent instances.

  A transparent instance can contain other opaque or transparent instances nested inside. The details inside a transparent instance are independent schematic objects and you can operate on them independently: select, push into, hide, and so on. Performing an operation on a transparent object does not automatically perform it on any of the objects nested inside it, and conversely.



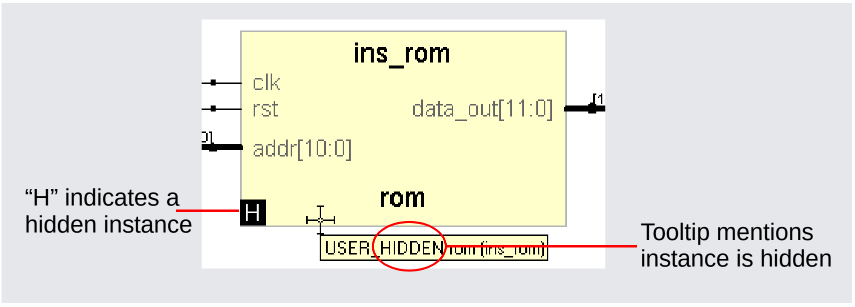
See also

- Looking Inside Hierarchical Instances, on page 6-26
- Multiple Sheets for Transparent Instance Details, on page 6-20
- Filtered and Unfiltered Schematic Views, on page 6-2

Hidden Hierarchical Instances

Certain commands do not operate on the lower-level hierarchy of hidden instances, so you can hide instances to focus the operation of a command and improve performance. You hide opaque or transparent hierarchical instances with the Hide Instances command (described in RTL and Technology Views Popup Menus, on page 3-97). Hiding and unhiding only affects the current HDL Analyst view, and does not affect the Hierarchy Browser. You can hide and unhide instances as needed. The hierarchical logic of a hidden instance is not removed from the design; it is only excluded from certain operations.

The schematics indicate hidden hierarchical instances with a small H in the lower left corner. When the mouse pointer is over a hidden instance, the status bar and the tooltip indicate that the instance is hidden.


ins_rom
clk
rst
data_out[11:0]
addr[10:0]
rom
H
USER_HIDDEN rom (ins_rom)
"H" indicates a hidden instance
Tooltip mentions instance is hidden

Schematic Display

The HDL Analyst Options dialog box controls general properties for all HDL Analyst views, and can determine the display of schematic object information. Setting a display option affects all objects of the given type in all views. Some schematic options only take effect in schematic windows opened after the setting change; others affect existing schematic windows as well.

The following are some commonly used settings that affect the display of schematic objects. See HDL Analyst Options Command, on page 3-73 for a complete list of display options.

Option	Controls the display of...
Show Cell Interior	Internal logic of technology-specific primitives
Compress Buses	Buses as bundles
Dissolve Levels	Hierarchical levels in a view flattened with HDL Analyst -> Dissolve Instances or Dissolve to Gates, by setting the number of levels to dissolve.
Instances Filtered Instances Instances added for expansion	Instances on a schematic by setting limits to the number of instances displayed
Instance Name Show Conn Name Show Symbol Name Show Port Name	Object labels
Show Pin Name HDL Analyst->Show All Hier Pins	Pin names. See Pin and Pin Name Display for Opaque Objects, on page 6-10 and Pin and Pin Name Display for Transparent Objects, on page 6-11 for details.

Pin and Pin Name Display for Opaque Objects

Although it always displays the pins, the software does not automatically display pin names for opaque hierarchical instances, technology-specific primitives, RAMS, ROMs, and state machines. To display pin names for these objects, enable Options-> HDL Analyst Options->Text->Show Pin Name. The following figures illustrate this display.

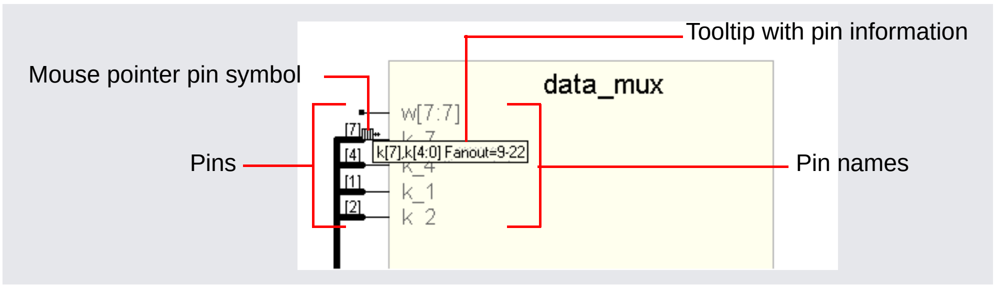


Figure 6-4: Pins and Pin Names of Opaque Hierarchical Instance

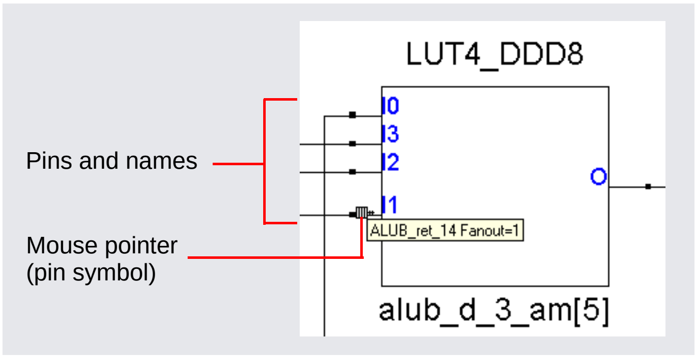


Figure 6-5: Pins of Technology-specific Primitive with Cell Contents Not Displayed

Pin and Pin Name Display for Transparent Objects

This section discusses pin name display for transparent hierarchical instances in filtered views and technology-specific primitives.

Transparent Hierarchical Instances

In a filtered schematic, some of the pins on a transparent hierarchical instance might not be displayed because of filtering. To display all the pins, select the instance and select HDL Analyst -> Show All Hier Pins.

To display pin names for the instance, enable Options-> HDL Analyst Options>Text ->Show Pin Name. The software temporarily displays the pin name when you move the cursor over a pin. To keep the pin name displayed even after you move the cursor away, select the pin. The name remains until you select something else.

Primitives

To display pin names for technology primitives in the Technology view, enable Options-> HDL Analyst Options->Text->Show Pin Name. The software displays the pin names until the option is disabled. If Show Pin Name is enabled when Options-> HDL Analyst Options->General->Show Cell Interior is also enabled, the primitive is treated like a transparent hierarchical instance, and primitive pin names are only displayed when the cursor moves over the pins. To keep a pin name displayed even after you move the cursor away, select the pin. The name remains until you select something else.

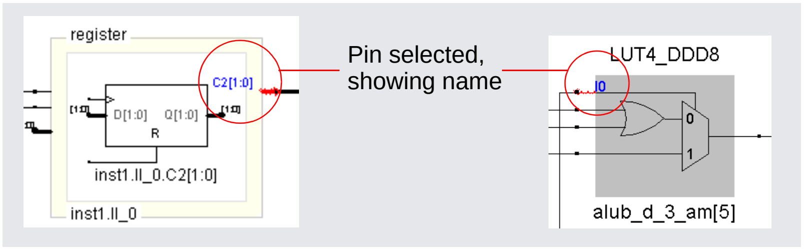


Figure 6-6: Pin Name Displayed Because Pin Is Selected

See also:

- HDL Analyst Options Command, on page 3-73
- Controlling the Amount of Logic on a Sheet, on page 6-18
- Annotated Timing Information, on page 6-34

Basic Operations on Schematic Objects

Basic operations on schematic objects include the following:

- Finding Schematic Objects, on page 6-13
- Selecting and Unselecting Schematic Objects, on page 6-14
- Crossprobing Objects, on page 6-15

For information about other operations on schematics and schematic objects, see the following:

- Filtering and Flattening Schematics, on page 6-28
- Timing Information and Critical Paths, on page 6-33
- Multiple-sheet Schematics, on page 6-18
- Exploring Design Hierarchy, on page 6-21

Finding Schematic Objects

You can use a Hierarchy Browser to browse and find schematic objects. This can be a quick way to locate an object by name if you are familiar with the design hierarchy.

You can also locate objects using the Edit -> Find command, described in Find Command (HDL Analyst), on page 3-15. For detailed procedures, see Finding Objects, on page 4-30 of the *Synplify User Guide*. The Find command displays the Object Query dialog box, which lists schematic objects by type (Instances, Symbols, Nets, or Ports) and lets you use wildcards to find objects by name. You can fine-tune your search by setting a range:

- Entire Design (all levels at once)
- Current Level & Below
- Current Level Only

All found objects are selected, whether or not they are displayed in the current schematic. Although you can search for hidden instances, you cannot find objects that are inside hidden instances at a lower level. Temporarily hiding an instance thus further refines the search range by excluding the internals of a a given instance. This can be very useful when working with transparent instances, because the lower-level details appear at the current level, and cannot be excluded by choosing Current Level Only.

Edit -> Find enhances filtering. Use Find to select by name and hierarchical level, and then filter the design to limit the display to the current selection. Unselected objects are removed. Because Find only adds to the current selection (it never deselects anything already selected), you can use successive searches to build up exactly the selection you need, before filtering.

Conversely, filtering helps your search:

- Filtering helps you to fine-tune the range of a search. You can search for objects just within a filtered schematic (by limiting the search range to the Current Level Only).
- Filtering adds to the expressive power of displaying search results. You can find objects on different sheets and filter them to see them all together at once. Filtering collapses the hierarchy visually, showing lower-level details nested inside transparent higher-level instances. The resulting display combines the advantage of a high-level, abstract view with detail-rich information from lower levels.

See Filtering and Flattening Schematics, on page 6-28 for further information, or the *Synplify User Guide* for detailed procedures.

Selecting and Unselecting Schematic Objects

Whenever an object is selected in one place it is selected and highlighted everywhere else in the synthesis tool, including all Hierarchy Browsers, all schematics, and the Text Editor. Many commands operate on the currently selected objects, whether or not those objects are visible.

The following briefly list selection methods; for a concise table of selection procedures, see Selecting Objects in the RTL/Technology Views, on page 4-17 in the *Synplify User Guide*.

Using the Mouse to Select a Range of Schematic Objects

In a Hierarchy Browser, you can select a *range* of schematic objects by clicking the name of an object at one end of the range, then holding the Shift key while clicking the name of an object at the other end of the range.To use the mouse for selecting and unselecting objects in a schematic, the cross-hairs symbol (┼) must appear as the mouse pointer. If this is not currently the case, right-click the schematic background.

Using Commands to Select Schematic Objects

You can select and deselect schematic objects using the commands in the HDL Analyst menu, or use Edit -> Find to find and select objects by name.

The HDL Analyst menu commands that affect selection include the following:

- Expansion commands like Expand, Expand to Register/Port, Expand Paths, and Expand Inwards select the objects that result from the expansion. This means that (except for Expand to Register/Port) you can perform successive expansions and expand the set of objects selected.
- The Select All Schematic and Select All Sheet commands select all instances or ports on the current schematic or sheet, respectively.
- The Select Net Driver and Select Net Instances commands select the appropriate objects according to the hierarchical level you have chosen.
- Unselect All unselects all objects in *all* HDL Analyst views.

See also

- Finding Schematic Objects, on page 6-13
- HDL Analyst Menu, on page 3-49

Crossprobing Objects

Crossprobing helps you diagnose where coding changes or timing constraints might reduce area or increase performance. When you crossprobe, you select an object in one place and it or its equivalent is automatically selected and highlighted in other places. For example, selecting text in the Text Editor automatically selects the corresponding logic in all HDL Analyst views. Whenever a net is selected, it is highlighted through all the hierarchical instances it traverses, at all schematic levels.

You can crossprobe between different views of the Synplify synthesis tool, and between these and any ToolNet-enabled application, such as the ModelSim® HDL simulator or the Visual Elite™ design tool.

Crossprobing Between Different Views

You can crossprobe objects (including logic inside hidden instances) between RTL views, Technology views, HDL source code files, and other text files. Some RTL and source code objects are optimized away during synthesis, so they cannot be crossprobed to certain views.

The following table summarizes crossprobing to and from HDL Analyst (RTL and Technology) views. For information about crossprobing procedures, see Crossprobing, on page 4-38 in the *Synplify User Guide*.

Table 6-1: Crossprobing Between Different Views

From . . .	To . . .	Do this . . .
Log file (Text Editor)	HDL source file (Text Editor)	Double-click a log file note, error, or warning. The corresponding HDL source code appears in the Text Editor.
Text Editor	Analyst view	The target view must be open. To crossprobe to the Technology view, the RTL view must also be open (but it can be minimized). Select the code in the Text Editor that corresponds to the object(s) you want to crossprobe. You can select a *column* of objects by pressing and holding the Alt key while dragging through the objects. On some platforms you need to use the key to which the Alt functionality is mapped, such as Meta. The object corresponding to the selected code is automatically selected in the target view, if an HDL source file is in the Text Editor. Otherwise, choose Select in Analyst from the Text Editor popup menu (available by right-clicking). (The Enhanced Text Crossprobing option must be enabled to crossprobe from text other than source code — see HDL Analyst Options Command, on page 3-73.)
Analyst view	Text Editor	Double-click an object. The source code corresponding to the object is automatically selected in the Text Editor, which is opened to show the selection. If you just select an object, without double-clicking it, the corresponding source code is still selected and displayed in the editor (provided it is open), but the editor window is not raised to the front.
Analyst view	another open view	Select an object in an HDL Analyst view. The object is automatically selected in all open views.
Tcl window	Text Editor	Double-click an error or warning message (available in the Tcl window errors or warnings panel, respectively). The corresponding source code is automatically selected in the Text Editor, which is opened to show the selection.

Crossprobing to the ModelSim HDL Simulator

The ModelSim® HDL simulator (version 5.2 or later), from Mentor Graphics Corporation, can be connected to the Synplify synthesis tool to help you debug your design by crossprobing waveforms.

For information on installing and running the ModelSim simulator, see the following:

- of the *Synplify User Guide*
- The *ModelSim User Guide.*

Using the Visual Elite Design Tool to Crossprobe and Run Synthesis

You can use the Visual Elite™ design tool to create and edit HDL designs. It is a product of Summit Design™; for more information see `http://www.sd.com/`.

After you have set up the Visual Elite software, you can launch synthesis from the Visual Elite design tool and crossprobe to the Visual Elite editor from HDL Analyst views. For information on setting up and using the Visual Elite software, see Working with Visual Elite, on page 7-12 of the *Synplify User Guide*.

Multiple-sheet Schematics

This section describes how to work with schematics that have more than a single sheet. It explains how to control the amount of logic on a sheet and how to navigate between sheets.

When there is too much logic to display on a single sheet, the HDL Analyst tool uses additional schematic sheets. Large designs can take several sheets. In a hierarchical schematic, each module consists of one or more sheets. Sheet connector symbols (Sheet Connectors, on page 6-5) mark logic connections from one sheet to the next.

For more information, see

- Controlling the Amount of Logic on a Sheet, on page 6-18
- Navigating Among Schematic Sheets, on page 6-18
- Multiple Sheets for Transparent Instance Details, on page 6-20

Controlling the Amount of Logic on a Sheet

You can control the amount of logic on a schematic sheet using the options in Options->HDL Analyst Options->Sheet Size. The Maximum Instances option sets the maximum number of instances on an unfiltered schematic sheet. The Maximum Filtered Instances option sets the maximum number of instances displayed at any given hierarchical level on a filtered schematic sheet.

See also:

- HDL Analyst Options Command, on page 3-73
- Setting Schematic View Preferences, on page 4-20 of the *Synplify User Guide*.

Navigating Among Schematic Sheets

This section describes how to navigate among the sheets in a given schematic. The window title bar lets you know where you are at any time.

Multisheet Orientation in the Title Bar

The window title bar of an RTL view or Technology view indicates the current context. For example, uc_alu (of module alu) in the title indicates that the current schematic level displays the instance uc_alu (which is of module alu). The objects shown are those comprising that instance.

The title bar also indicates, for the current schematic, the number of the displayed sheet, and the total number of sheets — for example, sheet 2 of 4. A schematic is initially opened to its first sheet.

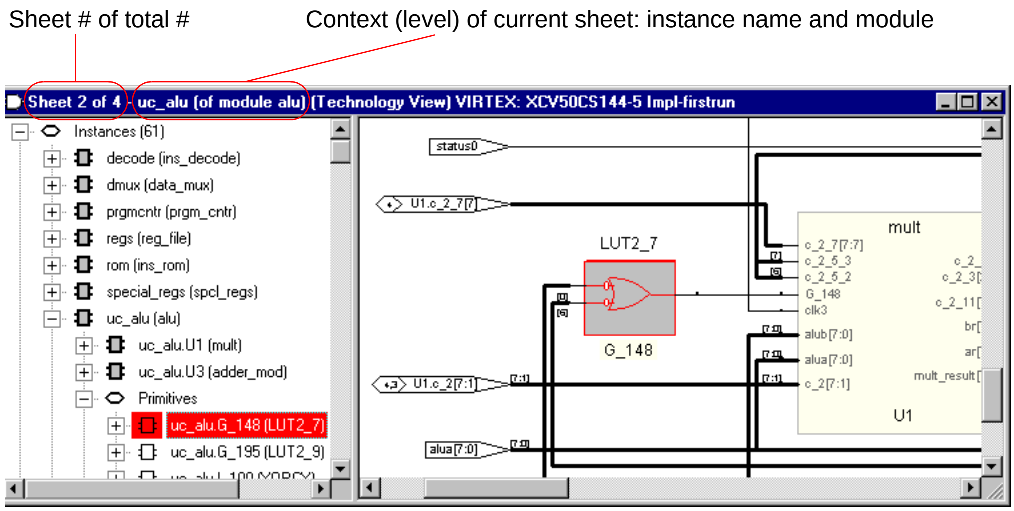


Figure 6-7: HDL Analyst View, Showing Title Bar

Navigating Among Sheets

You can navigate among different sheets of a schematic in these ways:

- Follow a sheet connector, by right-clicking it and choosing a connecting sheet from the popup menu
- Use the sheet navigation commands of the View menu: Next Sheet, Previous Sheet, and View Sheets, or their keyboard shortcut or icon equivalents
- Use the history navigation commands of the View menu (Back and Forward), or their keyboard shortcuts or icon equivalents to navigate to sheets stored in the display history

For details, see Working with Multisheet Schematics, on page 4-18 in the *Synplify User Guide*.

You can navigate among different design levels by pushing and popping the design hierarchy. Doing so adds to the display history of the View menu, so you can retrace your push/pop steps using View -> Back and Forward. After pushing down, you can either pop back up or use View -> Back.

See also:

- Filtering and Flattening Schematics, on page 6-28
- View Menu Commands for Project View, RTL and Technology Views, on page 3-21
- Pushing and Popping Hierarchical Levels, on page 6-21

Multiple Sheets for Transparent Instance Details

The details of a transparent instance in a filtered view are drawn in two ways:

- Generally, these interior details are spread out over multiple sheets at the same schematic level (module) as the instance that contains them.. You navigate these sheets as usual, using the methods described in Navigating Among Schematic Sheets, on page 6-18.
- If the number of nested contents exceeds the limit set with the Filtered Instances option (Options->HDL Analyst Options), the nested contents are drawn on separate sheets. The parent hierarchical instance is empty, with a notation (for example, Go to sheets 4-16) inside it, indicating which sheets contain its lower-level details. You access the sheets containing the lower-level details using the sheet navigation commands of the View menu, such as Next Sheet.

See also:

- Controlling the Amount of Logic on a Sheet, on page 6-18
- View Menu Commands for Project View, RTL and Technology Views, on page 3-21

Exploring Design Hierarchy

The hierarchy in your design can be explored in different ways. The following sections explain how to move between hierarchical levels:

- Pushing and Popping Hierarchical Levels, on page 6-21
- Navigating With a Hierarchy Browser, on page 6-24
- Looking Inside Hierarchical Instances, on page 6-26

Pushing and Popping Hierarchical Levels

You can navigate your design hierarchy by pushing down into a high-level schematic object or popping back up. Pushing down into an object takes you to a lower-level schematic that shows the internal logic of the object. Popping up from a lower level brings you back to the parent higher-level object.

Pushing and popping is best suited for traversing the hierarchy of a specific object. If you want a more general view of your design hierarchy, use the Hierarchy Browser instead. See Navigating With a Hierarchy Browser, on page 6-24 and Looking Inside Hierarchical Instances, on page 6-26 for other ways of viewing design hierarchy.

Pushable Schematic Objects

To push into an instance, it must have hierarchy. You can push into the object regardless of its position in the design hierarchy; for example, you can push into the object if it is shown nested inside a transparent instance. You can push down into the following kinds of schematic objects:

- Non-hidden hierarchical instances. To push into a hidden instance, unhide it first.
- Technology-specific primitives (not logic primitives)
- Inferred ROMs and state machines in RTL views. Inferred ROMs, RAMs, and state machines do not appear in Technology views, because they are resolved into technology-specific primitives.

When you push/pop, the HDL Analyst window displays the appropriate level of design hierarchy, except in the following cases:

- When you push into an inferred state machine in an RTL view, the `statemachine.info` text fileopens.
- When you push into an inferred ROM in an RTL view, the Text Editor window opens and displays the ROM data table (`rom.info` file).

You can use the following indicators to determine whether you can push into an object:

- The mouse pointer shape when Push/Pop mode is enabled. See How to Push and Pop Hierarchical Levels, on page 6-22 for details.
- A small H symbol (H) in the lower left corner indicates a hidden instance, and you cannot push into it.
- The Hierarchy Browser symbols indicates the type of instance and you can use that to determine whether you can push into an object. For example, hierarchical instance ( ), technology-specific primitive ( ), logic primitive such as XOR ( ), or other primitive instance ( ). The browser symbol does not indicate whether or not an instance is hidden.
- The *status bar* at the bottom of the main synthesis tool window reports information about the object under the pointer, including whether or not it is a hidden instance or a primitive.

How to Push and Pop Hierarchical Levels

You push/pop design levels with the HDL Analyst Push/Pop mode. To enable or disable this mode, toggle View -> Push/Pop Hierarchy, use the icon, or use the appropriate mouse strokes.

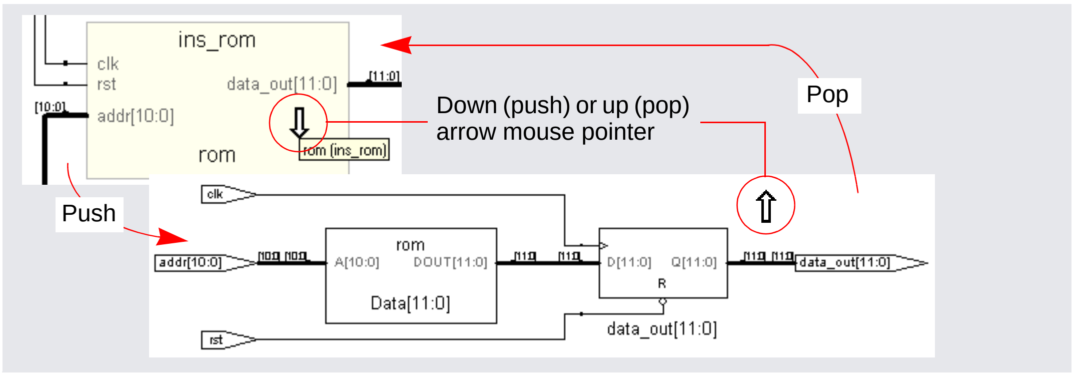


Figure 6-8: Pushing Down and Popping Up

Once Push/Pop mode is enabled, you push or pop as follows:

- To *pop*, place the pointer in an empty area of the schematic background, then click or use the appropriate mouse stroke. The background area inside a transparent instance acts just like the background area outside the instance.
- To *push* into an object, place the mouse pointer over the object and click or use the appropriate mouse stroke. To push into a transparent instance, place the pointer over its pale yellow border, not its hollow (white) interior. Pushing into an object nested inside a transparent hierarchical instance descends to a lower level than pushing into the enclosing transparent instance. In the following figure, pushing into transparent instance inst2 descends one level; pushing into nested instance inst2.II_3 descends two levels.

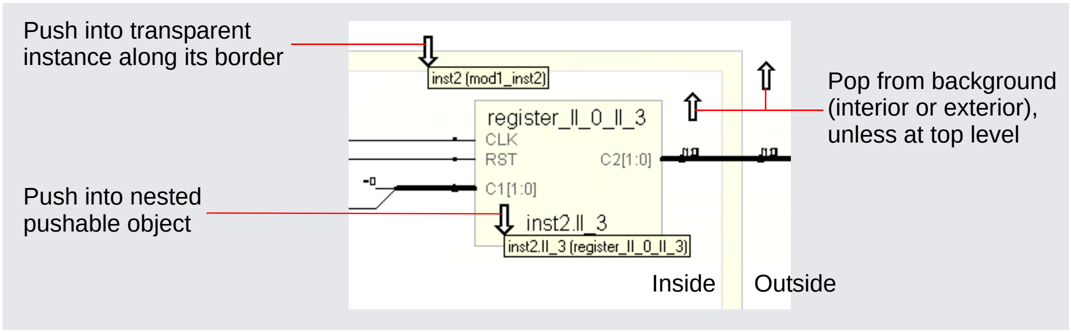

Push into transparent instance along its border
inst2 (mod1_inst2)
register_II_0_II_3
CLK
RST
C2[1:0]
C1[1:0]
inst2.II_3
inst2.II_3 (register_II_0_II_3)
Pop from background (interior or exterior), unless at top level
Push into nested pushable object
Inside
Outside

Figure 6-9: Push/Pop Areas of a Transparent Instance

The following arrow mouse pointers indicate status in Push/Pop mode. For other indicators, see Pushable Schematic Objects, on page 6-21.

A down arrow ⇩	Indicates that you can push (descend) into the object under the pointer and view its details at the next lower level.
An up arrow ⇧	Indicates that there is a hierarchical level above the current sheet.
A crossed-out double-headed arrow ⇳	Indicates that there is no accessible hierarchy above or below the current pointer position. If the pointer is over the schematic background it indicates that the current level is the top and you cannot pop higher. If the pointer is over an object, the object is an object you cannot push into: a non-hierarchical instance, a hidden hierarchical instance, or a black box.

See also:

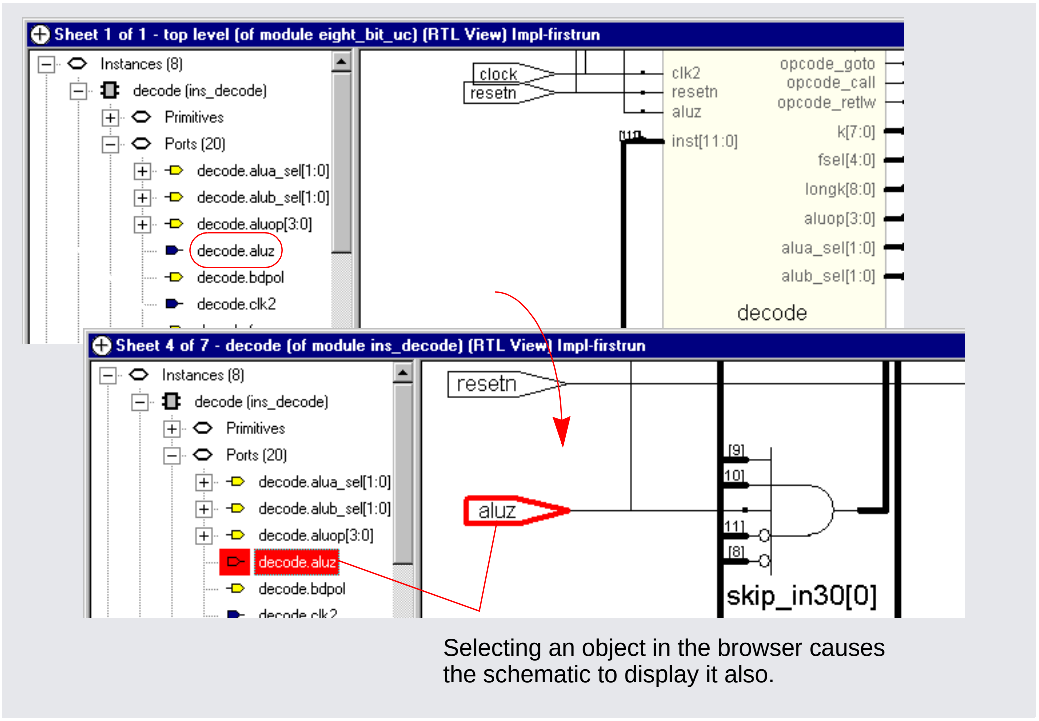
- Hidden Hierarchical Instances, on page 6-9
- Transparent and Opaque Display of Hierarchical Instances, on page 6-7
- Using Mouse Strokes, on page 2-15
- Navigating With a Hierarchy Browser, on page 6-24

Navigating With a Hierarchy Browser

Hierarchy Browsers are designed for locating objects by browsing your design. To move between design levels of a particular object, use Push/Pop mode (see Pushing and Popping Hierarchical Levels, on page 6-21 and Looking Inside Hierarchical Instances, on page 6-26 for other ways of viewing design hierarchy).

The browser in the RTL view displays the hierarchy specified in the RTL design description. The browser in the Technology view displays the hierarchy of your design after technology mapping.

Selecting an object in the browser displays it in the schematic, because the two are linked.Use the Hierarchy Browser to traverse your hierarchy and select ports, nets, components, and submodules. The browser categorizes the objects, and accompanies each with a symbol that indicates the object type.



Selecting an object in the browser causes the schematic to display it also.

Figure 6-10: Crossprobing Between a Schematic and Hierarchy Browser

Explore the browser hierarchy by expanding or collapsing the categories in the browser. You can also use the arrow keys (left, right, up, down) to move up and down the hierarchy and select objects. To select more than one object, press Ctrl and select the objects in the browser. To select a range of schematic objects, click an object at one end of the range, then hold the Shift key while clicking the name of an object at the other end of the range.

See also:

- Crossprobing Objects, on page 6-15
- Pushing and Popping Hierarchical Levels, on page 6-21
- Hierarchy Browser Popup Menu Commands, on page 3-97

Looking Inside Hierarchical Instances

An alternative method of viewing design hierarchy is to examine transparent hierarchical instances (see Navigating With a Hierarchy Browser, on page 6-24 and Navigating With a Hierarchy Browser, on page 6-24 for other ways of viewing design hierarchy). A transparent instance appears as a hollow box with a pale yellow border. Inside this border are transparent and opaque objects from lower design levels.

Transparent instances provide design context. They show the lower-level logic nested within the transparent instance at the current design level, while pushing shows the same logic a level down. The following figure compares the same lower-level logic viewed in a transparent instance and a push operation:

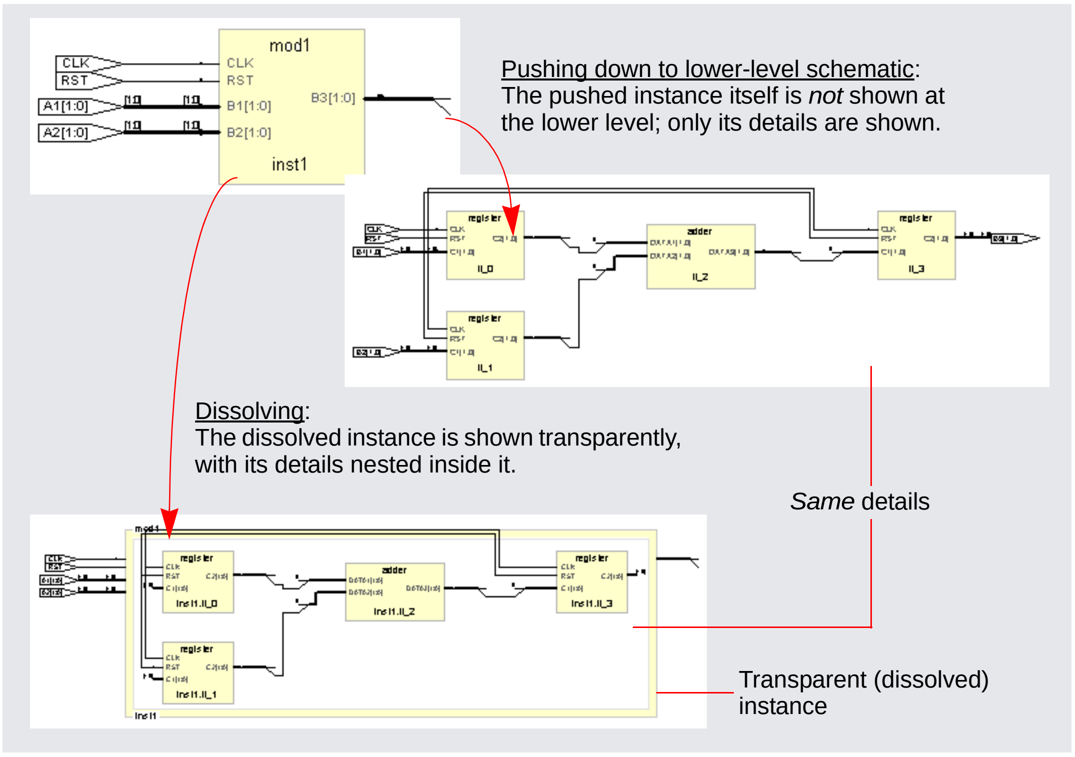

mod1
CLK
RST
A1[1:0]
A2[1:0]
B1[1:0]
B2[1:0]
B3[1:0]
inst1
Pushing down to lower-level schematic:
The pushed instance itself is *not* shown at the lower level; only its details are shown.
register
adder
IL_0
IL_1
IL_2
IL_3
Dissolving:
The dissolved instance is shown transparently, with its details nested inside it.
Same details
inst1.IL_0
inst1.IL_1
inst1.IL_2
inst1.IL_3
Transparent (dissolved) instance

Figure 6-11: A Transparent Instance with Nested Details

You cannot control the display of transparent instances directly. However, you can perform the following operations, which result in the display of transparent instances:

- Hierarchically expand an object (using the expansion commands in the HDL Analyst menu).
- Dissolve selected hierarchical instances in a *filtered* schematic (HDL Analyst -> Dissolve Instances).
- Filter a schematic, after selecting multiple objects at more than one level. See Commands That Result in Filtered Schematics, on page 6-28 for additional information.

These operations only make *non-hidden hierarchical* instances transparent. You cannot dissolve hidden or primitive instances (including technology-specific primitives). However, you can do the following:

- Unhide hidden instances, then dissolve them.
- Push down into technology-specific primitives to see their lower-level details, and you can show the interiors of all technology-specific primitives.

See also:

- Pushing and Popping Hierarchical Levels, on page 6-21
- Navigating With a Hierarchy Browser, on page 6-24
- HDL Analyst Command, on page 3-50
- Transparent and Opaque Display of Hierarchical Instances, on page 6-7
- Hidden Hierarchical Instances, on page 6-9

Filtering and Flattening Schematics

This section describes the HDL Analyst commands that result in filtered and flattened schematics. It describes

- Commands That Result in Filtered Schematics, on page 6-28
- Successive Filtering Operations, on page 6-29
- Returning to The Unfiltered Schematic, on page 6-29
- Commands That Flatten Schematics, on page 6-30
- Selective Flattening, on page 6-31
- Filtering Compared to Flattening, on page 6-32

Commands That Result in Filtered Schematics

A filtered schematic shows a subset of your design. Any command that *results in a filtered schematic* is a filtering command. Some commands, like the Expand commands, increase the amount of logic displayed, but they are still considered filtering commands because they result in a filtered view of the design. Other commands like Filter Schematic and Isolate Paths remove objects from the current display.

Filtering commands include the following:

- Filter Schematic, Isolate Paths — reduce the displayed logic.
- Dissolve Instances (in a filtered schematic) — makes selected instances transparent.
- Expand, Expand to Register/Port, Expand Paths, Expand Inwards, Select Net Driver, Select Net Instances — display logic connected to the current selection.
- Show Critical Path, Flattened Critical Path, Hierarchical Critical Path — show critical paths.

All the filtering commands, except those that display critical paths, operate on the currently selected schematic object(s). The critical path commands operate on your entire design, regardless of what is currently selected.

All the filtering commands except Isolate Paths are accessible from the HDL Analyst menubar menu; Isolate Paths is in the RTL view and Technology view popup menus (along with most of the other commands above).

For information about filtering procedures, see *Filtering Schematics, on page 4-48* in the *Synplify User Guide*.

See also:

- Filtered and Unfiltered Schematic Views, on page 6-2
- HDL Analyst Menu, on page 3-49 and RTL and Technology Views Popup Menus, on page 3-97

Successive Filtering Operations

Filtering operations are designed to be used in combination, successively. You can, for example, perform a sequence of operations like the following:

1. Filter Schematic — filter your design to examine a particular instance.
2. Expand — expand from one of the output pins of the instance to add its immediate successor cells to the display.
3. Select Net Driver — add the net driver of a net connected to one of the successors.
4. Isolate Paths — filter to isolate the net driver instance, together with those of its connecting paths that were already displayed.

Filtering operations add their resulting filtered schematics to the history of schematic displays, so you can use the View menu Forward and Back commands to switch between the filtered views. You can also combine filtering with the search operation. See Finding Schematic Objects, on page 6-13 for more information.

Returning to The Unfiltered Schematic

A filtered schematic often loses the design context, as it is removed from the display by filtering. After a series of multiple or complex filtering operations, you might want to view the context of a selected object. You can do this by

- Selecting a higher level object in the Hierarchy Browser; doing so always crossprobes to the corresponding object in the original schematic.
- Using Show Context to take you directly from a selected instance to the corresponding context in the original, unfiltered schematic.
- Using Goto Net Driver to go from a selected net to the corresponding context in the original, unfiltered schematic.

There is no Unfilter command. Use Show Context to see the unfiltered schematic containing a given instance. Use View -> Back to return to the previous, unfiltered display after filtering an unfiltered schematic. You can go back and forth between the original, unfiltered design and the filtered schematics, using the commands View -> Back and Forward.

See also:

- RTL and Technology Views Popup Menus, on page 3-97
- RTL and Technology Views Popup Menus, on page 3-97
- View Menu Commands for Project View, RTL and Technology Views, on page 3-21

Commands That Flatten Schematics

A flattened schematic contains no hierarchical objects. Any command that results in a flattened schematic is a flattening command. This includes the following.

Command	Unfiltered Schematic	Filtered Schematic
Dissolve Instances	Flattens selected instances	--
Flatten Current Schematic (Flatten Schematic)	Flattens at the current level and all lower levels. RTL view: flattens to generic logic level Technology view: flattens to technology-cell level	Flattens only non-hidden transparent hierarchical instances; opaque and hidden hierarchical instances are not flattened.
RTL -> Flattened View	Creates a new, unfiltered RTL schematic of the entire design, flattened to the level of generic logic cells.	
Technology -> Flattened View	Creates a new, unfiltered Technology schematic of the entire design, flattened to the level of technology cells.	

Command	Unfiltered Schematic	Filtered Schematic
Technology -> Flattened to Gates View	Creates a new, unfiltered Technology schematic of the entire design, flattened to the level of Boolean logic gates.	
Technology -> Flattened Critical Path	Creates a filtered, flattened Technology view schematic that shows only the instances with the worst slack times and their path.	
Unflatten Schematic	Undoes any flattening done by Dissolve Instances and Flatten Current Schematic at the current schematic level. Returns to the original schematic, as it was before flattening (and any filtering).	

All the commands are on the HDL Analyst menu except Unflatten Schematic, which is available in a schematic popup menu.

The most versatile commands, are Dissolve Instances and Flatten Current Schematic, which you can also use for selective flattening (Selective Flattening, on page 6-31).

See also:

- Filtering Compared to Flattening, on page 6-32
- Selective Flattening, on page 6-31

Selective Flattening

By default, flattening operations are not very selective. However, you can selectively flatten particular instances with these commands:

- Use Hide Instances to hide instances that you do *not* want to flatten, then flatten the others (flattening operations do not recognize hidden instances). After flattening, you can Unhide Instances that are hidden.
- Flatten selected hierarchical instances using one of these commands:
  - If the current schematic is unfiltered, use Dissolve Instances.
  - If the schematic is filtered, use Dissolve Instances, followed by Flatten Current Schematic. In a filtered schematic, Dissolve Instances makes the selected instances transparent and Flatten Current Schematic flattens only transparent instances.

The Dissolve Instances and Flatten Current Schematic (or Flatten Schematic) commands behave differently in filtered and unfiltered schematics:

Table 6-2: Effects of Selective Flattening Commands

Command	Unfiltered Schematic	Filtered Schematic
Dissolve Instances	Flattens selected instances	Provides virtual flattening: makes selected instances transparent, displaying their lower-level details.
Flatten Current Schematic Flatten Schematic	Flattens *everything* at the current level and below	Flattens only the non-hidden, *transparent* hierarchical instances: does not flatten opaque or hidden instances. See below for details of the process.

In a filtered schematic, flattening with Flatten Current Schematic is actually a two-step process:

1. The transparent instances of the schematic are flattened in the context of the entire design. The result of this step is the entire hierarchical design, with the transparent instances of the filtered schematic replaced by their internal logic.

2. The original filtering is then restored: the design is refiltered to show only the logic that was displayed before flattening.

Although the result displayed is that of Step 2, you can view the intermediate result of Step 1 with View -> Back. This is because the display history is erased before flattening (Step 1), and the result of Step 1 is added to the history as if you had viewed it.

See also:

- RTL and Technology Views Popup Menus, on page 3-97

Filtering Compared to Flattening

As a general rule, use filtering to examine your design, and flatten it only if you really need it. Here are some reasons to use filtering instead of flattening:

- Filtering before flattening is a more efficient use of computer time and memory. Creating a new view where everything is flattened can take considerable time and memory for a large design. You then filter anyway to remove the flattened logic you do not need.

- Filtering is selective. On the other hand, the default flattening operations are global: the entire design is flattened from the current level down. Similarly, the inverse operation (UnFlatten Schematic) unflattens everything on the current schematic level.
- Flattening operations eliminate the *history* for the current view: You can not use View -> Back after flattening. (You can, however, use UnFlatten Schematic to regenerate the unflattened schematic.)

See also:

- RTL and Technology Views Popup Menus, on page 3-97
- Selective Flattening, on page 6-31

Timing Information and Critical Paths

The HDL Analyst tool provides several ways of examining critical paths and timing information, to help you analyze problem areas. The different ways are described in the following sections.

- Annotated Timing Information, on page 6-34
- Timing Reports, on page 6-34
- Critical Paths and the Slack Margin Parameter, on page 6-35
- Examining Critical Path Schematics, on page 6-37

See also the following, for more information about timing and result analysis:

- Log File, on page 4-7
- The *Synplify User Guide*, Chapter 4, *Result Analysis*

Annotated Timing Information

To help you analyze timing, enable HDL Analyst -> Show Timing Information. This annotates all instances in a Technology view with their timing numbers. Two timing numbers are displayed above each instance:

Delay	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	The second number is the slack time of the worst path that goes through the instance. A negative value indicates that timing constraints could not be met. The command Show Critical Path uses the slack time, together with the slack margin, to determine which instances to display.

The Show Timing Information command also adds timing numbers to object status bar information. The commands that result in a critical path display (Show Critical Path, Hierarchical Critical Path, Flattened Critical Path) also enable Show Timing Information for that display.

See also:

- Critical Paths and the Slack Margin Parameter, on page 6-35

Timing Reports

When you synthesize a design, a default timing report is automatically written to the log file, which you can view using View -> View Log File. This report provides a clock summary, I/O timing summary, and detailed timing information for your design.

For certain device technologies, you can use the HDL Analyst->Timing Analyst command to generate a custom timing report. Use this command to specify start and end points of paths whose timing interests you, and set a limit the number of paths to analyze between these points.

By default, the sequential instances, input ports, and output ports that are currently selected in the Technology views of the design are the candidates for choosing start and end points. In addition, the start and end points of the previous Timing Analyst run become the default start and end points for the next run.

The custom timing report is stored in a text file named *`resultsfile`*`.ta`, where *`resultsfile`* is the name of the results file (see Implementation Results Panel, on page 3-32). In addition, a corresponding output netlist file is generated, named *`resultsfile`*`_ta.srm`. Both files are in the implementation results directory.

The Timing Analyst dialog box provides check boxes for viewing the text report (Open Report) in the Text Editor and the corresponding netlist (Open Schematic) in a Technology view. This Technology view of the timing path, labeled Timing View in the title bar, is special in two ways:

- The Timing View shows only the paths you specify in the Timing Analyst dialog box. It corresponds to a special design netlist that contains critical timing data.
- The Timing Analyst and Show Critical Path commands (and equivalent icons and shortcuts) are unavailable whenever the Timing View is active.

See also:

- Timing Report, on page 7-47
- Log File, on page 4-7

Critical Paths and the Slack Margin Parameter

The HDL Analyst tool can isolate critical paths in your design, so that you can analyze problem areas, add timing constraints where appropriate, and resynthesize for better results.

After you successfully run synthesis, you can display just the critical paths of your design using any of the following commands from the HDL Analyst menu:

- Hierarchical Critical Path
- Flattened Critical Path
- Show Critical Path

The first two commands create a new Technology view, hierarchical or flattened, respectively. The Show Critical Path command reuses the current Technology view. Neither the current selection nor the current sheet display have any effect on the result. The result is flat if the entire design was already flat; otherwise it is hierarchical. Use Show Critical Path if you want to maintain the existing display history.

All these commands filter your design to show only the instances (and their paths) with the worst slack times. They also enable HDL Analyst -> Show Timing Information, displaying timing information.

Negative slack times indicate that your design has not met its timing requirements. The worst (most negative) slack time indicates the amount by which delays in the critical path cause the timing of the design to fail. You can also obtain a *range* of worst slack times by setting the *slack margin* parameter to control the sensitivity of the critical-path display. Instances are displayed only if their slack times are within the slack margin of the (absolutely) worst slack time of the design.

The slack margin is the criterion for distinguishing worst slack times. The larger the margin, the more relaxed the measure of worst, so the greater the number of critical-path instances displayed. If the slack margin is zero (the default value), then only instances with the worst slack time of the design are shown. You use HDL Analyst -> Set Slack Margin to change the slack margin.

The critical-path commands do not calculate a single critical path. They filter out instances whose slack times are not too bad (as determined by the slack margin), then display the remaining, worst-slack instances, together with their connecting paths.

For example, if the worst slack time of your design is -10 ns and you set a slack margin of 4 ns, then the critical path commands display all instances with slack times between -6 ns and -10 ns.

See also:

- HDL Analyst Menu, on page 3-49
- HDL Analyst Command, on page 3-50
- Handling Negative Slack, on page 4-65 of the *Synplify User Guide*
- Annotated Timing Information, on page 6-34

Examining Critical Path Schematics

Use successive filtering operations to examine different aspects of the critical path. After filtering, use View -> Back to return to the previous point, then filter differently. For example, you could use the command Isolate Paths to examine the cone of logic from a particular pin, then use the Back command to return to the previous display, then use Isolate Paths on a different pin to examine a different logic cone, and so on. For step-by-step procedures, see in the *Synplify User Guide*.

Also, the Show Context and Goto Net Driver commands are particularly useful after you have done some filtering. They let you get back to the original, unfiltered design, putting selected objects in context.

See also:

- Returning to The Unfiltered Schematic, on page 6-29
- Filtering and Flattening Schematics, on page 6-28

CHAPTER 7

Timing Constraints

This chapter contains information about timing constraints.

Timing Constraints

Timing constraints define the timing targets for the design that the Synplify synthesis tool must meet to improve synthesis results. Timing constraints define the performance goals for a design.

The following sections describe methods for entering timing constraints and constraint priority.

Entering Timing Constraints

Timing constraints are passed to the synthesis environment in constraint files that have a .sdc extension (Synplicity design constraints). These files contain Tcl commands .

Currently, you can define timing constraintsusing one of the following methods.

- In a SCOPE spreadsheet, which automatically generates constraint files in Tcl format

  Use this method for specifying constraints wherever possible. You can use it for most constraints, except for source code directives. See SCOPE Constraints, on page 7-9 for more information about this tool.

- As Tcl commands in an `.sdc` file that you create manually in a text editor.

  It is easier to use the SCOPE spreadsheet to generate the constraint syntax, as described in SCOPE Constraint Types, on page 7-9. For information about Tcl constraint syntax for manually created constraint files, see Constraint Files, on page 7-29.

- As source code attributes or directives

You must enter black-box timing directives in the source code. Do not include any other timing constraints in the source code, as the source code becomes less portable, and you must recompile the design for the constraints to take effect. If there are multiple constraints on the same object, the software uses the guidelinbes described in Constraint Priority, on page 7-3 to determine which constraint takes precedence.

Constraint Priority

When there are multiple timing constraints on the same object, two factors determine which constraint takes priority:

- Where the constraint is applied: bit vs. bus
- The type of constraint applied

Bit vs. Bus Constraints

In the same way that a local constraint overrides a global constraint, a constraint on a bit takes priority over a bus constraint. This means that the bit constraint applies to that bit only, while all other bits on the bus are constrained by the bus constraint.

Order of Precedence for Constraints

The software constrains the design using the following order of priority when there is more than one constraint on the same object:

- False path constraints
  These constraints have the highest priority and take precedence over any other type of constraint.
- Path delay constraints
  These constraints are next in order of priority.
- Multicycle path constraints
  These constraints have a lower priority than path delay, because the path delay is a precise value while the multicycle constraint value is a multiple.

The SCOPE Window

This section contains a description of the SCOPE interface, information about initialization, and entering constraints. It describes the following:

- The SCOPE Spreadsheet, on page 7-4
- SCOPE Initialization Dialog Box, on page 7-5
- SCOPE Constraint Wizards, on page 7-6
- Insert Quick Mode, on page 7-8

For details about the SCOPE constraints, see SCOPE Constraints, on page 7-9.

The SCOPE Spreadsheet

The SCOPE (Synthesis Constraints Optimization Environment®) window presents a spreadsheet interface for entering and managing timing constraints and synthesis attributes. This is the easiest way to create constraint files.

To create a new constraint file, do one of the following in an open, compiled design. (If the design has not been compiled, you can still use the SCOPE spreadsheet, but it will not automatically initialize the clocks and I/O ports.)

– Choose File -> New -> Constraint file (SCOPE) from the Project view.

– Click the SCOPE icon in the toolbar.

From the SCOPE window, you can set the following:

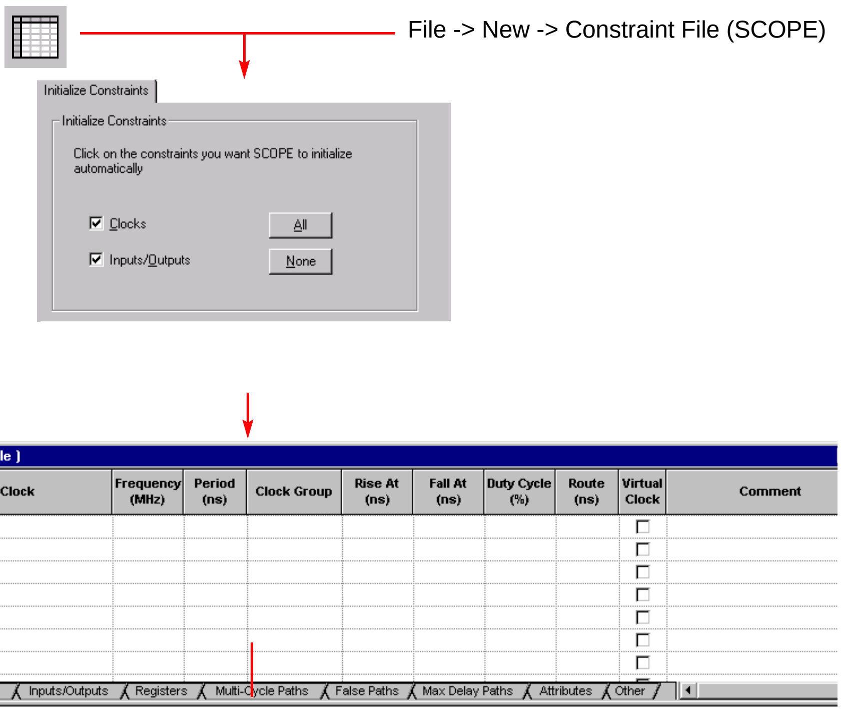
- Timing *constraints* for clocks, ports, and registers. For more information, see Entering and Editing Constraints in the SCOPE Window, on page 3-15 in the *Synplify User Guide*. You cannot set black-box constraints from the SCOPE window. The information you enter in the SCOPE window is stored in an `.sdc` constraint file that must be included in your project.
- Synthesis *attributes*. For more information, see Adding Attributes in the SCOPE Window, on page 3-38 in the *Synplify User Guide*.

You can also use the SCOPE window to edit constraints. Open an existing constraint file by double-clicking it in the Project view. Alternatively, you can use File -> Open, specifying the file type as Constraint Files (*.sdc). The alternative to editing the file in a SCOPE window is to use a text editor. Be sure to close the SCOPE spreadsheet before editing a constraint file with the text editor, so that saving with one tool does not overwrite modifications made with the other. See Using a Text Editor for Constraint Files, on page 3-33 in the *Synplify User Guide* for information about editing Tcl constraint files with a text editor.

SCOPE Initialization Dialog Box

Before the SCOPE spreadsheet opens, you see the Create a New Top-Level SCOPE File dialog box.

You use the Initialize Constraints panel of the dialog box to specify the constraints to be automatically initialized: Clocks and/or Inputs/Outputs. Clicking the buttons All and None provides a quick way to enable both or neither of the options. Unless you are an advanced user, it is a good idea to simply accept the default settings (click OK), automatically initializing All.



Initialize Constraints

Click on the constraints you want SCOPE to initialize automatically

☑ Clocks

☑ Inputs/Outputs

	Enabled	Clock	Frequency (MHz)	Period (ns)	Clock Group	Rise At (ns)	Fall At (ns)	Duty Cycle (%)	Route (ns)	Virtual Clock	Comment
1	☑									☐	
2	☑									☐	
3	☑									☐	
4	☑									☐	
5	☑									☐	
6	☑									☐	
7	☑									☐	

SCOPE Window

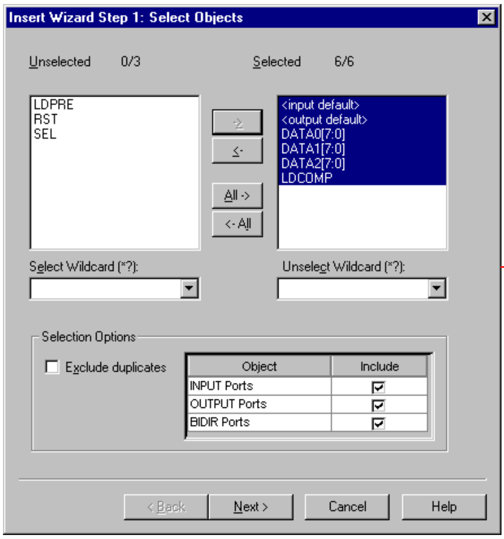
For information about using the SCOPE window to create and edit constraints, see Entering and Editing Constraints in the SCOPE Window, on page 3-15 in the *Synplify User Guide*. Once you have done this, save the constraints file. You can optionally add it to the project file when you save it.

SCOPE Constraint Wizards

The Clock, Inputs/Outputs, Registers, and Attributes panels of the SCOPE spreadsheet have wizards that you can use to set constraints. The Attributes panel wizard is described in SCOPE Attribute Wizard, on page 8-10.

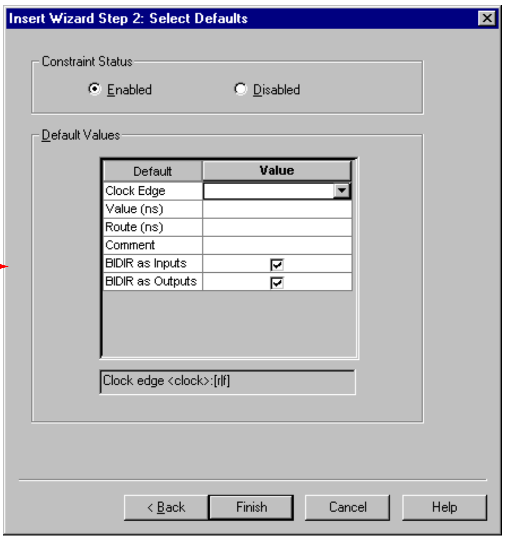
A wizard is most useful for assigning the same constraint to a number of objects at once; a typical use is setting default constraint values. An alternative way to assign the same constraint to a number of objects is provided by the insert quick mode (see Insert Quick Mode, on page 7-8). To assign a constraint to a single object, it is usually simpler to work directly in the SCOPE spreadsheet – see Entering and Editing Constraints in the SCOPE Window, on page 3-15 in the *Synplify User Guide* for details..

To start a wizard, right-click in a SCOPE panel, then choose Insert Wizard from the popup menu (or from the Edit menubar menu). For step-by-step instructions about using the wizards, see Entering Default Constraints, on page 3-18 in the *Synplify User Guide*. Each constraint wizard consists of two dialog boxes similar to these:



Object	Include
INPUT Ports	✓
OUTPUT Ports	✓
BIDIR Ports	✓

Step 1



Default	Value
Clock Edge	
Value (ns)	
Route (ns)	
Comment	
BIDIR as Inputs	✓
BIDIR as Outputs	✓

Step 2

In the first dialog box, choose the objects to which to assign a constraint, by moving them from the Unselected list to the Selected list. (If both lists are empty, then disable Exclude Duplicates.) Click the right arrow (->) to make the highlighted Unselected objects Selected. Click Next when the Selected list is complete, to display the second dialog box.

In the second wizard dialog box, you set the Value of the constraint (for *all* of the selected objects). For some constraint types you can set a separate Value for each SCOPE column (some columns, such as Comment, are optional; others, such as the Period of a clock, are required).

Enable or Disable the constraint assignment, then click Finish. The SCOPE spreadsheet then reflects your choices. The constraints you set are saved in the constraint file.

Insert Quick Mode

The insert quick mode is most useful for assigning the same constraint to a number of objects at once. An alternative way to assign the same constraint to a number of objects is provided by SCOPE Insert wizards (see SCOPE Constraint Wizards, on page 7-6). To assign a constraint to a single object, it is usually simpler to work directly in the SCOPE spreadsheet – see Entering and Editing Constraints in the SCOPE Window, on page 3-15 in the *Synplify User Guide*.

The insert quick mode can only be used with the Clocks or Inputs/Outputs panel, and it is only appropriate if you did *not* click Initialize in the Initialize New Constraint File dialog box when you started the SCOPE spreadsheet. To quickly generate default values for either of these panels, display the panel by clicking its tab, then follow these steps.

1. Choose Edit -> Insert Quick. The synthesis tool lists all of the clocks, or all of the ports and their types, respectively.
2. Enter values in the cells of the Value column. If you need the same value in a number of cells, select the column of cells, enter the value in the topmost cell, then press Ctrl-d. All the selected cells (in the same column) receive the same value.

SCOPE Constraints

This section contains a description of the SCOPE interface, and details about the different parts of the interface. It describes the following:

- SCOPE Constraint Types, on page 7-9
- Clocks Panel, on page 7-10
- Clock Constraint Options, on page 7-12
- Clock to Clock Panel, on page 7-16
- Inputs/Outputs Panel, on page 7-18
- Registers Panel, on page 7-20
- Multicycle Paths Panel, on page 7-21
- False Paths Panel, on page 7-24
- Max Delay Paths Panel, on page 7-27
- Other Panel, on page 7-28

For information about the SCOPE Attributes panel, see Specifying Attributes with the SCOPE Spreadsheet, on page 8-7.

SCOPE Constraint Types

You use the different SCOPE panels to enter specific constraint information. The following table shows the constraints you can set:

Constraint type	SCOPE panel
Clock	Clocks Panel, on page 7-10
Clock-to-clock delay	Clock to Clock Panel, on page 7-16
Input/output delays	Inputs/Outputs Panel, on page 7-18
Register	Registers Panel, on page 7-20

Constraint type	SCOPE panel
Multiple-cycle path exceptions	Multicycle Paths Panel, on page 7-21
False path exceptions	False Paths Panel, on page 7-24
Maximum delay limits	Max Delay Paths Panel, on page 7-27

You can also use the SCOPE spreadsheet to set attributes. See Specifying Attributes with the SCOPE Spreadsheet, on page 8-7 for details.

Clocks Panel

You use the Clocks panel of the SCOPE spreadsheet to define a signal as a clock. The equivalent Tcl constraint is define_clock; its syntax is described in define_clock, on page 7-34. For information about the Clocks panel wizard, see SCOPE Constraint Wizards, on page 7-6.

The Clocks panel includes the following fields:

SCOPE Cell	Description
Enabled	Turn this on to enable the constraint. Turn it off to disable an existing constraint.
Clock	(Required.) Specifies the name of the clock. The name can be either a top-level port in the design or the name of an internal instance used as a clock generator for the chip. In the case of virtual clocks, the field must contain a unique name not associated with any port or instance in the design. See the Virtual Clock field for more information.
Frequency (Mhz)	(Required.) Specifies the clock frequency in MHz. If you fill in this field and click in the Period field, the Period value is filled in automatically. For frequencies other than that implied by the clock pin, refer to syn_reference_clock, on page 8-50.
Period (ns)	(Required.) Specifies the clock period in nanoseconds. If you fill in this field and click in the Frequency field, the frequency is filled in automatically.

SCOPE Cell	Description
Clock Group	Assigns a clock to a clock group. Clocks in the same clock group are related. By default, the software assumes all clocks are related and assigns them to default_clkgroup. • The Synplify synthesis tool calculates the relationship between clocks in the same clock group and analyzes all paths between them. Paths between clocks in different groups are ignored (considered false paths). • By default, clocks are in the same clock group (default_clkgroup is the default name). See Clock Groups, on page 7-12 for more information.
Rise At (ns) Fall At (ns)	Specifies non-default rising and falling edges for clocks.By default, the tool assumes that the clock is a 50% duty cycle clock, with rising edge at 0 and falling edge at Period/2. See Rise and Fall Constraints, on page 7-14 for details. Setting rise/fall calculates the duty cycle automatically.
Duty Cycle (%)	Specifies the clock duty cycle as a percentage of the clock period. If you have a duty cycle that is not 50% (the default), specify the rising and falling edge values (Rise At/Fall At) instead of Duty Cycle, and the duty cycle value is calculated automatically. There is no corresponding Tcl command option.
Route (ns)	Improves the path delays of registers controlled by the clock. The value shrinks the effective period for synthesis, without affecting the clock period that is forward-annotated to the place-and-route tool. This is an advanced user option. Before you use this option, evaluate the path delays on individual registers in the optimization timing report and try to improve the delays only on the registers that need them (Registers panel). See Route Option, on page 7-15 for a detailed explanation of this option.
Virtual Clock	Designates the specified clock as a virtual clock. It lets you specify arrival and required times on top level ports that are enabled by clocks external to the chip (or block) that you are synthesizing. The clock name can be a unique name not associated with any port or instance in the synthesized design.
Comment	Lets you enter comments that are included in the constraints file.

For more information on using the Clocks panel, see Defining Clocks, on page 3-22 of the *Synplify User Guide*.

Clock Constraint Options

This section explains the following clock constraint options on the SCOPE Clocks panel (define_clocks):

- Clock Groups, on page 7-12
- Rise and Fall Constraints, on page 7-14
- Route Option, on page 7-15

Clock Groups

The timing engine uses clock groups to analyze and optimize the design. It assumes that clocks in the same clock group are synchronized with each other and treats them as related clocks. Typically, clocks in a clock group are derived from the same base clock. The timing analyzer automatically calculates the relationships between the related clocks in a clock group and analyzes all paths between them. For paths between clocks of the same group, the timing engine calculates the time available based on the period of the two clocks. If you want to override the calculation, set a clock delay on the Clock to Clock tab (define_clock_delay).

The waveforms in the following figure show how the Synplify synthesis tool determines the worst posedge-to-posedge timing between clocks CLK1 and CLK2. All paths that begin at CLK1 rising and end at CLK2 rising are constrained at 10 ns.

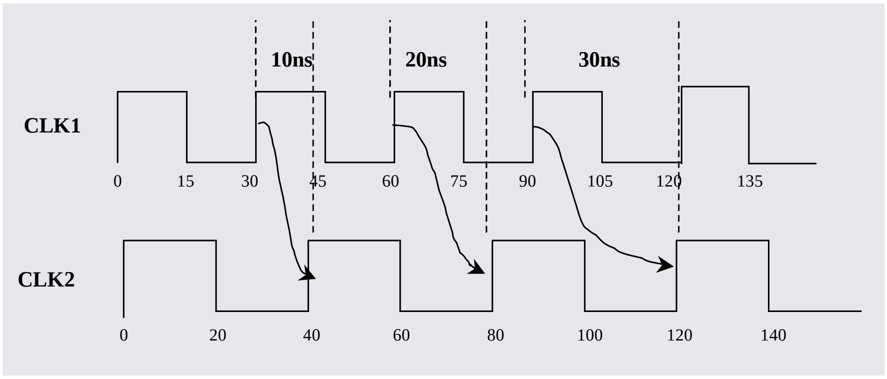


Figure 7-1: Worst case posedge-to-posedge timing

The following figure shows how the timing analyzer calculates worst case edge-to-edge timing between all possible transitions of the two related clocks. In this example, CLK1 has a period of 5 ns and CLK2 has a period of 10 ns.

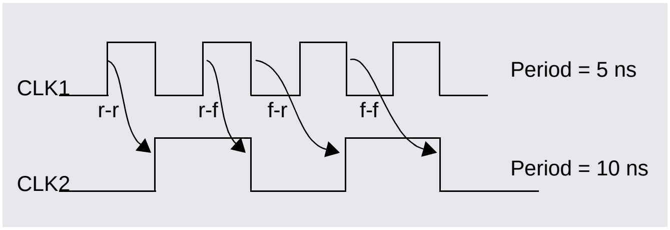


Figure 7-2: Worst case edge-to-edge timing

Conversely, clocks in different clock groups are considered unrelated or asynchronous. Paths between clocks from different groups are automatically marked as false paths and ignored during timing analysis and optimization.

By default, all clocks in a design are assigned to the same clock group, called default_clkgroup. For most accurate results, assign each clock explicitly to a clock group. If you have an unrelated clock, you must re-assign the unrelated clock to a new clock group using the Clock Group field in the SCOPE spreadsheet.

For example, if your design has three clocks (clk1, clk2, and clk3), by default they are all assigned to default_clkgroup. All paths between the three clocks are analyzed and optimized. If clk2 is unrelated to the other two clocks but is left in the same clock group, it skews the results of timing analysis and negatively affects optimization because the timing engine treats it as a related clock. Similarly, in cases where clock relationships are forward-annotated, it negatively affects place-and-route results.

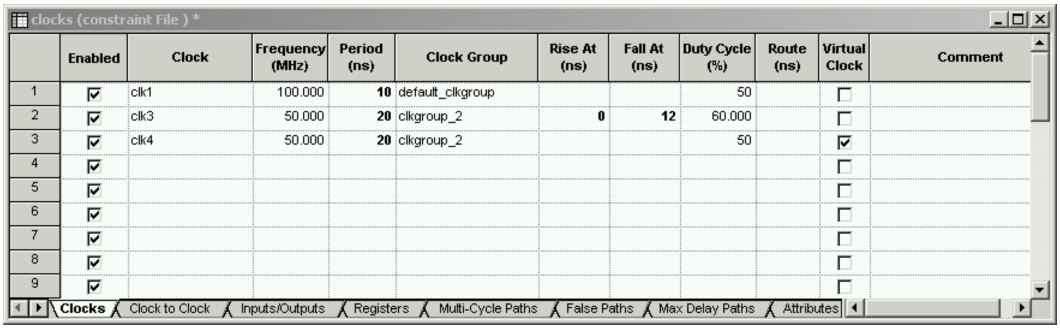
For more accurate results, specify a different clock group name for clk2, such as clkgrp2. The timing analyzer now only analyzes the relationship between clk1 and clk3. Paths between clk1 and clk2, and paths between clk3 and clk2, are considered false paths and are ignored.

Rise and Fall Constraints

The Synplify synthesis tool assumes an ideal clock network. By default, the constraints assume a 50% duty cycle clock with the rising edge at 0 and the falling edge at Period/2.

The synthesis tool computes relationships between the source clock and destination clock on a path by using the Rise At and Fall At numbers. When you enter or change these values, the Duty Cycle is calculated or recalculated automatically. The timing engine can calculate the rise/fall values from the Duty Cycle, but entering the rise/fall values directly is more accurate.

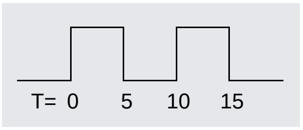
To understand how the relationships between source and destination clocks are computed using the Rise At and Fall At values, consider the following example.


clocks (constraint File) *

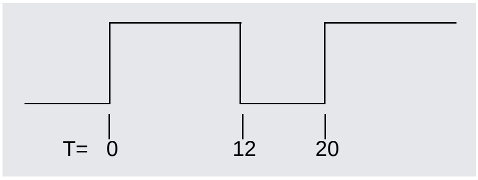
	Enabled	Clock	Frequency (MHz)	Period (ns)	Clock Group	Rise At (ns)	Fall At (ns)	Duty Cycle (%)	Route (ns)	Virtual Clock	Comment
1	✓	clk1	100.000	10	default_clkgroup			50		☐	
2	✓	clk3	50.000	20	clkgroup_2	0	12	60.000		☐	
3	✓	clk4	50.000	20	clkgroup_2			50		✓	
4	✓									☐	
5	✓									☐	
6	✓									☐	
7	✓									☐	
8	✓									☐	
9	✓									☐	

In this example:

- CLK1 is a clock with a period of 10 ns (100 MHz) in clock group default_clkgroup. Since none of the other fields are specified, this is a 50% duty cycle clock rising at 0 and falling at 5.



- CLK3 is a clock with a period of 20 ns (50 MHz) in clkgroup2. This means all paths between CLK3 and CLK1 are automatically treated as false paths. In addition, CLK3 has a Rise At value of 0 and a Fall At value of 12, which means it has a 60% duty cycle.



- CLK4 is a virtual clock. This means that there can be no port or instance named CLK4 in this design. However, there may be top-level ports on the chip which are clocked by CLK4 outside the chip. Input arrival times and output required times for such ports can be specified relative to CLK4.

Route Option

Use the Route (-route) option if you do not meet timing goals because the routing delay after placement and routing exceeds the delay predicted by the Synplify synthesis tool. The delay reported on an instance along a path from an input port is the amount of time by which the clock frequency goal is exceeded (positive slack), or not met (negative slack) because of the input delay. You can view slack times in the timing report section of the log file, or with the HDL Analyst tool.

Rerun synthesis with this option to include the actual route delay (from place-and-route results) so that the tool can meet the required clock frequency. Using the Route option is equivalent to putting a register delay (define_reg_input_delay) on all registers controlled by that clock.

Assume the frequency goal is 50 MHz (clock period goal of 20.0 ns). If the synthesis timing reports show the frequency goal is met, but the detailed timing report from placement and routing shows that the actual clock period is 21.8 ns because of paths through input_a, you can either tighten the constraints in the place-and-route tool, or you can rerun synthesis with a -route 1.8 option added to the define_input_delay constraint.

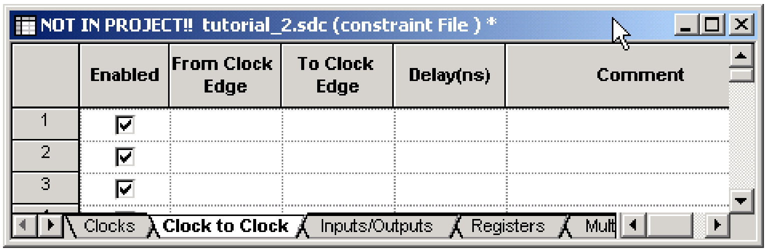
```
# In this example, input_a has a 10.0 ns input delay.
# Rerun synthesis with an added 1.8 ns constraint, to
# improve accuracy and match post-place-and-route results.

define_input_delay {input_a} 10.0 -route 1.8
```

Using this option adds 1.8 ns to the route calculations for paths to input_a and improves timing by 1.8 ns.

Clock to Clock Panel

Defines the time available from data signals on all paths between the specified clock edges. By default, the software automatically calculates clock delay based on the clock parameters defined in the Clocks pane of the SCOPE UI. If you define a clock-to-clock constraint, this delay value overrides any automatic calculations made by the software and is reflected in the values shown in the Clock Relationships section of the timing report.

For the equivalent Tcl syntax, see define_clock_delay, on page 7-36.



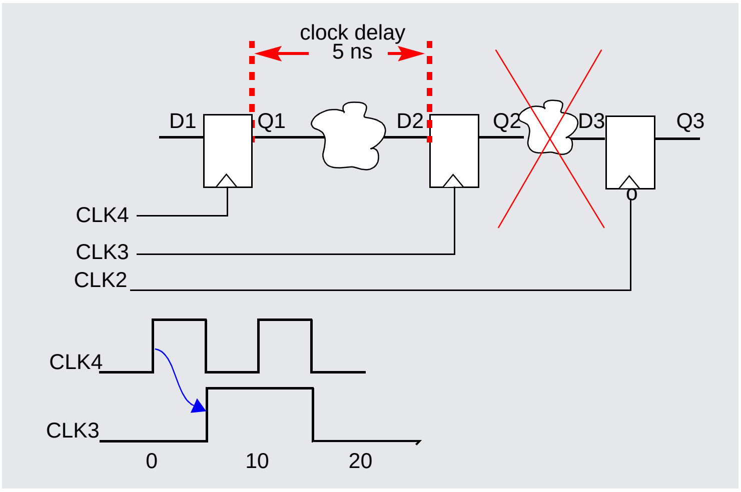
	Enabled	From Clock Edge	To Clock Edge	Delay(ns)	Comment
1	✓				
2	✓				
3	✓				

This table provides brief descriptions of the fields shown above.

Delay Parameter	**Description**
From Clock Edge	Clock edge that triggers the event of the starting point: *clock_name,* followed by a colon and the edge: (*r* for rising, *f* for falling).
To Clock Edge	Clock edge that triggers the event of the ending point: *clock_name*:r or f (rise or fall).
Delay (ns)	Path delay in nanoseconds. Alternatively, you can use the keyword false. When you use the false keyword, the timing engine disables the delay field and places an implicit false path constraint on the path. See Defining False Paths, on page 3-27 in the *Synplify User Guide* for details.

You can use the false keyword to specify false paths for all paths between two clock edges. Clocks that you have assigned to different clock groups are unrelated, which means the software treats any paths between them as implicit false paths. You can use the constraint to override the implicit false path for paths between clocks in different clock groups.

The following is an example of the constraints defined in the Clock to Clock panel above.



A 5 nanosecond delay constraint is placed on CLK4 rising to CLK3 rising and an implicit false path constraint is placed on CLK3 rising to CLK2 falling.

Inputs/Outputs Panel

This panel models the interface of the FPGA with the outside environment. You use it to specify delays outside the device. The default delay outside an FPGA is 0.0 ns.

You can specify multiple constraints on the same I/O port. See Multiple I/O Constraints, on page 7-20 for details.

For the equivalent Tcl constraints, see define_input_delay, on page 7-38 and define_output_delay, on page 7-40. For information about the Inputs/Outputs panel wizard, see SCOPE Constraint Wizards, on page 7-6.

The SCOPE Inputs/Outputs fields are shown in the following table:

Field	Description
Enabled	(Required.) Turn this on to enable the constraint, or off to disable a previous constraint.
Port	(Required.) Specifies the name of the port. If you have initialized a compiled design, you can select a port name from the pulldown list. The first two entries let you specify global input and output delays, which you can then override with additional constraints on individual ports.
Type	(Required.) Specifies whether the delay is an input or output delay.
Clock Edge	(Recommended.) The rising or falling edge that controls the event. The syntax for this field is the clock name, followed by a colon and the edge: *r* for rising, *f* for falling. For example, CLK1:r.

Field	Description
Value	(Required.) Specifies the delay value. You do not need this value if you supply a Route value.
Route	Improves the delay of the paths to and from the port. The value shrinks the effective synthesis constraint without affecting the constraint that is forward-annotated to the place-and-route tool. This is an advanced user option. See Route Option, on page 7-19 for details.
Comment	Lets you enter comments that are included in the constraints file.

Route Option

Use this option if you do not meet timing goals because the routing delay after placement and routing exceeds the delay predicted by the Synplify synthesis tool. The delay reported on an instance along a path from an input port is the amount of time by which the clock frequency goal is exceeded (positive slack), or not met (negative slack) because of the input delay. You can view slack times in the timing report section of the log file, or using the HDL Analyst tool.

Rerun synthesis using this option, which includes the actual route delay (from place-and-route results) so that the tool can meet the required clock frequency. Using the Route option is equivalent to putting a register delay (define_reg_input_delay) on all registers controlled by that clock.

Assume the frequency goal is 50 MHz (clock period goal of 20.0 ns). If the synthesis timing reports show the frequency goal is met, but the detailed timing report from placement and routing shows that the actual clock period is 21.8 ns because of paths through input_a, you can either tighten the constraints in the place-and-route tool, or you can rerun optimization after entering 1.8 in the SCOPE Route column or adding the following define_input_delay -route constraint to the constraint file:

```
# In this example, input_a has a 10.0 ns input delay.
# Rerun synthesis with an added 1.8 ns constraint, to
# improve accuracy and match post-place-and-route results.

define_input_delay {input_a} 10.0 -route 1.8
```

Using this option adds 1.8 ns to the route calculations for paths to input_a and improves timing by 1.8 ns.

Multiple I/O Constraints

You can specify multiple input and output delays (define_input_delay and define_output_delay) constraints for the same I/O port. This is useful for cases where a port is driven by or feeds multiple clocks. The priority of a constraint and its use in your design is determined by a few factors:

- The software applies the tightest constraint for a given clock edge, and ignores all others. All applicable constraints are reported in the timing report.
- You can apply I/O constraints on three levels, with the most specific overriding the more global:
  - Global (top-level netlist), for all inputs and outputs
  - Port-level, for the whole bus
  - Bit-level, for single bits
- Where there are multiple constraints from different levels, the more specific overrides the more global. 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. For example, take the following constraints:

  ```
  a[3:0]3clk1:r
  a[3:0]3clk2:r
  a[0]2 clk1:r
  ```

  In this case, port a[0] only gets 1 constraint of 2 ns. Bits 1, 2, and 3 get 2 constraints of 3 ns each.
- If at any given level (bit port, port, global), there is a constraint with a reference clock specified, then any constraint without a reference clock is ignored. In this example, the 1 ns constraint on bit 0 is ignored.

  ```
  a[0]2 clk1:r
  a[0]1
  ```

Registers Panel

This panel lets the advanced user add delays to paths feeding into/out of registers, in order to further constrain critical paths. You use this constraint to speed up the paths feeding a register. See define_reg_input_delay, on page 7-42, and define_reg_output_delay, on page 7-43 for the equivalent Tcl commands. For information about the Registers panel wizard, see SCOPE Constraint Wizards, on page 7-6.

The Registers SCOPE panel includes the following fields:

Field	Description
Enabled	(Required.) Turn this on to enable the constraint.
Register	(Required.) Specifies the name of the register. If you have initialized a compiled design, you can choose from the pulldown list.
Type	(Required.) Specifies whether the delay is an input or output delay.
Route	(Required.) Improves the speed of the paths to or from the register by the given number of nanoseconds. The value shrinks the effective period for the constrained registers without affecting the clock period that is forward-annotated to the place-and-route tool. For an explanation of this advanced user option, see Route Option, on page 7-19.
Comment	Lets you enter comments that are included in the constraints file.

Multicycle Paths Panel

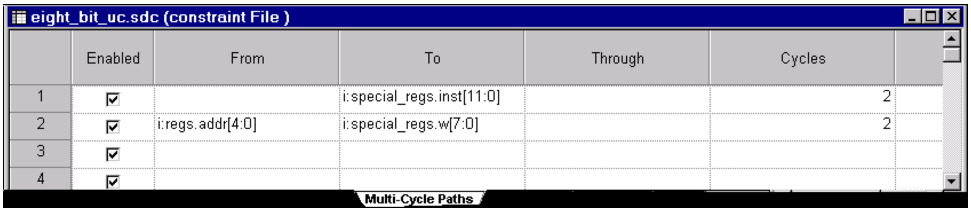
This panel lets you specify paths with multiple clock cycles. The following table defines the parameters for this constraint. For the equivalent Tcl constraints, see define_multicycle_path, on page 7-39.

The following table describes the fields. You can use any combination of from, to, and through points: from–through, from, from–through–to, and so on.

Field	Description
Enabled	(Required.) Turn this on to enable the constraint.
From	Specifies the start point for the multicycle path. From points can be registers, top-level input or bidirectional ports. For more information, see Defining From/To/Through for Timing Exceptions, on page 3-28 in the *Synplify User Guide*.

Field	Description
To	Specifies the end point for the multicycle path. To points can be registers, top-level output or bidirectional ports. SeeDefining From/To/Through for Timing Exceptions, on page 3-28 in the *Synplify User Guide*. for more information.
Through	Specifies the intermediate points for the timing exception. You can use any nets in the design as intermediate points. The intermediate points can be specified as a space-separated list, which is treated as an OR list. The exception is applied to all paths that cross any instance in the list. See Defining From/To/Through for Timing Exceptions, on page 3-28 in the *Synplify User Guide*.
Cycles	Number of cycles for the required time calculated for the path.
Comment	Lets you enter comments that are included in the constraints file.

The following shows an example of a SCOPE multicycle constraint:

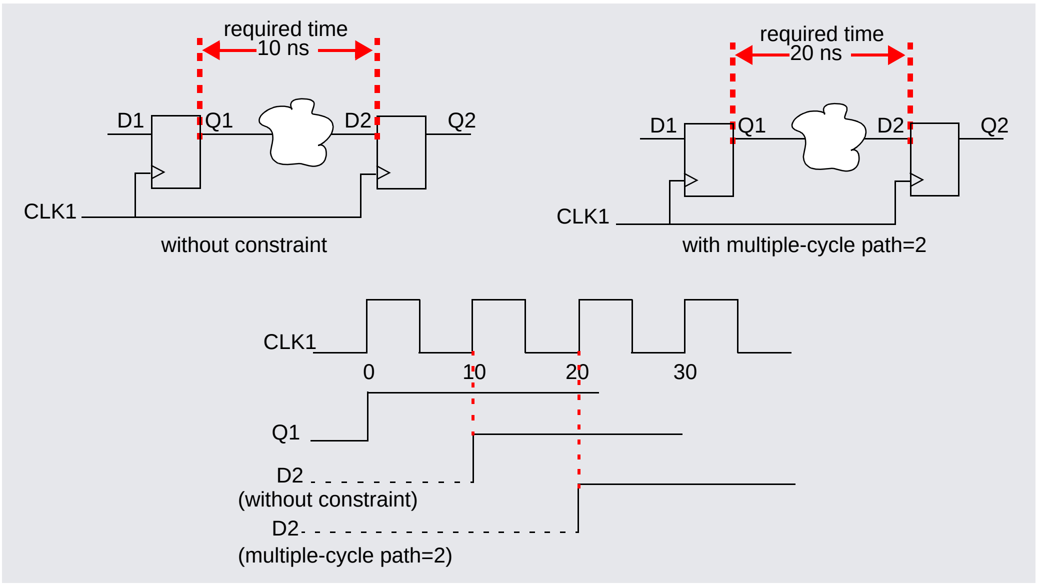


	Enabled	From	To	Through	Cycles
1	☑		i:special_regs.inst[11:0]		2
2	☑	i:regs.addr[4:0]	i:special_regs.w[7:0]		2
3	☑				
4	☑				

Multicycle Path Examples

Example 1

If you apply a multicycle path constraint from clk1 to clk2, the allowed time is #cycles x normal time between clk1 and clck2. In the following figure, CLK1 has a period of 10 ns. The data in this path has only one clock cycle before it must reach D2. To allow more time for the signal to complete this path, add a multiple-cycle constraint that specifies two clock cycles (10 x 2 or 20 ns) for the data to reach D2.



Example 2

The design has a multiplier that multiplies signal_a with signal_b and puts the result into signal_c. Assume that signal_a and signal_b are outputs of registers register_a and register_b, respectively. On clock cycle 1, a state machine enables an input enable signal to load signal_a into register_a and signal_b into register_b. At the beginning of clock cycle 2, the multiply begins. After two clock cycles, the state machine enables an output_enable signal on clock cycle 3 to load the result of the multiplication (signal_c) into an output register (register_c).

The design frequency goal is 50 MHz (20 ns) and the multiply function takes 35 ns, but it is given 2 clock cycles. After optimization, this 35 ns path is normally reported as a timing violation because it is more than the 20 ns clock-cycle timing goal. To avoid reporting the paths as timing violations, use the SCOPE window to set 2-cycle constraints (From column) on register_a and register_b, or include the following in the timing constraint file:

```
# Paths from register_a use 2 clock cycles
define_multicycle_path -from register_a 2

# Paths from register_b use 2 clock cycles
define_multicycle_path -from register_b 2
```

Alternatively, you can specify a 2-cycle SCOPE constraint (To column) on register_c, or add the following to the constraint file:

```
# Paths to register_c use 2 clock cycles
define_multicycle_path -to register_c 2
```

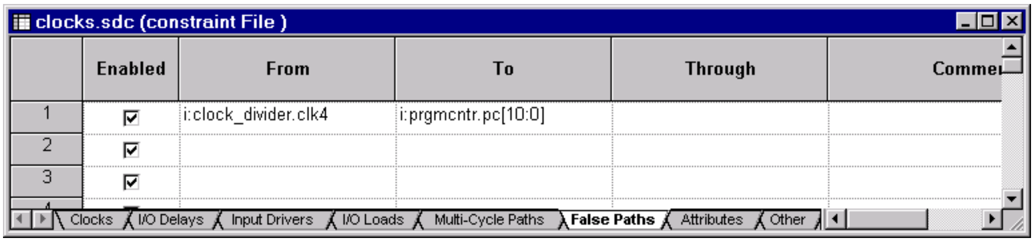
False Paths Panel

You use the False Paths constraint to specify clock paths that you want the Synplify synthesis tool to ignore during timing analysis and assign low (or no) priority during optimization. The equivalent Tcl constraint is described in define_false_path, on page 7-37.

The following table describes the fields. You can use any combination of from, to, and through points: from–through, from, from–through–to, and so on.

Field	Description
Enabled	(Required.) Turn this on to enable the constraint.
From	Specifies the start point for the timing exception. From points can be registers, top-level input or bidirectional ports, or black-box outputs. For more information, see Defining From/To/Through for Timing Exceptions, on page 3-28 in the *Synplify User Guide*.
To	Specifies the end point for the timing exception. To points can be registers, top-level output or bidirectional ports, or black-box inputs. For more information, see Defining From/To/Through for Timing Exceptions, on page 3-28 in the *Synplify User Guide*.
Through	Specifies the intermediate points (nets) for the timing exception. The nets can be specified as a space-separated list, which is treated as an OR list. The exception is applied to all paths that cross any instance in the list. See Defining From/To/Through for Timing Exceptions, on page 3-28 in the *Synplify User Guide*.
Comment	Lets you enter comments that are included in the constraints file.

The following is an example of a SCOPE entry:



clocks.sdc (constraint File)

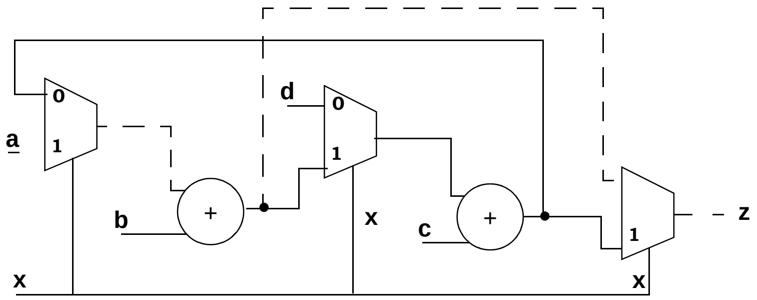
	Enabled	From	To	Through	Comme
1	✓	i:clock_divider.clk4	i:prgmcntr.pc[10:0]		
2	✓				
3	✓				

False Paths

A false path is a path that is not important for timing analysis. There are two types of false paths:

- Architectural false paths

  These are false paths that the designer is aware of, like an external reset signal that feeds internal registers but which is synchronized with the clock. The following example shows an architectural false path where the primary input x is always 1, but which is not optimized because the software does not optimize away primary inputs.



- Code-introduced false paths

  These are false paths that you identify after analyzing the schematic.

False Path Constraint Examples

In this case, the design frequency goal is 50 MHz (20ns) and the path from register_a to register_c is a false path with a large delay of 35 ns. After optimization, this 35 ns path is normally reported as a timing violation because it is more than the 20 ns clock-cycle timing goal.

To lower the priority of this path during optimization, define it as a false path. If all paths from register_a to any register or output pins are not timing-critical, then add a false path constraint to register_a in the SCOPE interface (From), or put the following line in the timing constraint file:

```
#Paths from register_a are ignored
define_false_path -from {register_a}
```

If all paths to register_c are not timing-critical, then add a false path constraint to register_c in the SCOPE interface (To), or include the following line in the timing constraint file:

```
#Paths to register_c are ignored
define_false_path -to {register_c}
```

If only the paths between register_a and register_c are not timing-critical, add a From/To constraint to the registers in the SCOPE interface (From and To), or include the following line in the timing constraint file:

```
#Paths to register_c are ignored
define_false_path -from {register_a} -to {register_c}
```

To lower the priority of this path during optimization, define it as a false path. If all paths from register_a to any register or output pins are not timing-critical, then add the following line to the timing constraint file:

```
#Paths from register_a are ignored
define_false_path -from {register_a}
```

If all paths to register_c are not timing-critical, then add the following line in the timing constraint file:

```
#Paths to register_c are ignored
define_false_path -to {register_c}
```

False Path Constraint Priority

False path constraints can be either*explicit* or *implicit*, and the priority of the constraint depends on the type of constraint it is.

- An explicit false path constraint is one that you apply to a path using the False Paths pane of the SCOPE UI, or the following Tcl syntax:

  define_false_path { **-from** *point* **|** **-to** *point* **|** **-through** *point* }

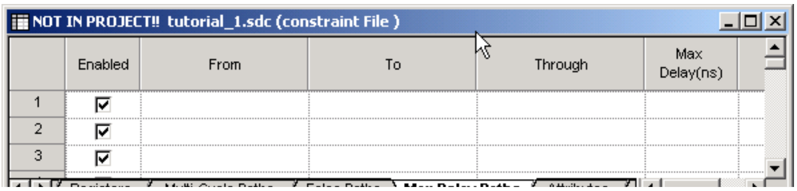
  This type of false path constraint has the highest priority of any of the types of constraints you can place on a path. Any path containing an explicit false path constraint is ignored by the software, even if you place a different type of constraint on the same path.

- Lower-priority false path constraints are those that the software automatically applies as a result of any of the following:
  - Clocks pane of SCOPE UI: you assign clocks to different groups.
  - Clock to Clock pane of SCOPE UI: you click on the Delay (ns) column and select the false option.
  - Project->Implementation Options->Constraints: you disable the Use clock period for unconstrained IO option.

  Implicit false path constraints are overridden by any subsequent constraints you place on a path. For example, if you assign two clocks to different clock groups, then place a maximum delay constraint on a path that goes through both clocks, the delay constraint has priority.

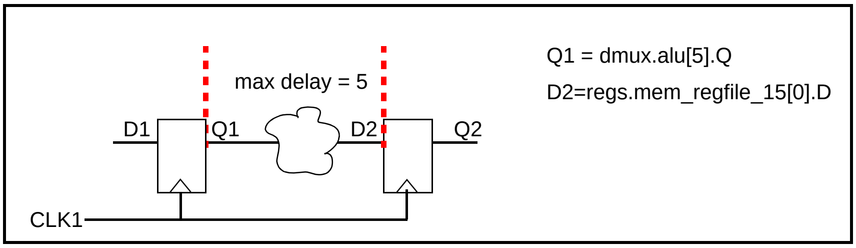
Max Delay Paths Panel

Specifies point-to-point delay constraints for paths. For the equivalent Tcl constraint, see define_path_delay, on page 7-41.



NOT IN PROJECT!! tutorial_1.sdc (constraint File)

	Enabled	From	To	Through	Max Delay(ns)
1	✓				
2	✓				
3	✓				



This table provides brief descriptions of the fields in the Max Delay Paths panel. You can use any combination of from, to, and through points: from–through, from, from–through–to, and so on.

SCOPE Cell	Description
From	Starting point of the path. From points can be registers, top-level input or bi-directional ports, or black box outputs. You can use this option alone, or combine it with To and Through. For more information, see Defining From/To/Through for Timing Exceptions, on page 3-28 in the *Synplify User Guide*.
To	Ending point of the path. To points can be registers, top-level output or bi-directional ports, or black box inputs.You can use this option alone, or combine it with To and Through. For more information, see Defining From/To/Through for Timing Exceptions, on page 3-28 in the *Synplify User Guide*.
Through	Defines one or more points of the path. You can use this option alone, or combine it with To and Through. See Defining From/To/Through for Timing Exceptions, on page 3-28 in the *Synplify User Guide* for information on defining through points.
Max Delay (ns)	Maximum delay value in nanoseconds.

Other Panel

The Other panel is intended for advanced users to enter newly-supported constraints. This panel contains the following fields

Field	Description
Enabled	(Required.) Turn this on to enable the constraint.
Command	(Required.) Specifies a command that you want to pass to the place-and-route tool.
Arguments	(Required.) Specifies arguments to the command.
Comment	Lets you enter a comment about the commands that you are passing to the place-and-route tool.

Constraint Files

Constraint files have an sdc file extension (.sdc). They can include timing constraints, general attributes, and vendor-specific attributes. You can manually create constraint files in a text editor using Tcl commands, but you typically use the SCOPE spreadsheet, instead, to generate the file automatically. If you edit a constraint file with a text editor, be sure to close the SCOPE spreadsheet first, so that saving with one tool does not overwrite modifications made with the other.

After you create a constraint file, include it in the project. You add constraint files to the files of a project by clicking Add in the Project view and selecting the files to add. You can change the List Files of Type field in the dialog box to constraint (.sdc) files, to see the available constraint files. Alternatively, you can read in a constraint file during a synthesis run from the project file with the add_file -constraint command.

When several constraint files are present in a project, make sure that the implementation options have the correct constraint file (or files) selected.

For more information, see the following:

- Object Naming Syntax, on page 7-29
- Tcl Syntax Guidelines for Constraint Files, on page 3-32 in the *Synplify User Guide*
- Using a Text Editor for Constraint Files, on page 3-33 in the *Synplify User Guide*
- Adding Attributes and Directives, on page 3-36 in the *Synplify User Guide*

Object Naming Syntax

This section describes the prefix syntax for identifying objects when different design objects share the same name. Objects like Verilog modules, VHDL design units, component instances, ports, and internal nets can have shared names. The prefix identifiers are used in constraint files to assign timing constraints and synthesis attributes to the correct object. The syntax varies slightly, depending on the language used to define the module or component. This section discusses the following object naming subjects:

- Verilog, on page 7-30
- VHDL, on page 7-31
- Naming Objects in the SCOPE Window, on page 7-32
- Object Naming Examples, on page 7-32

Verilog

This section provides syntax for object names in Verilog. No spaces are allowed in names. You can use the ***** and **?** characters as wildcards in names. These characters do not match the dot (**.**) used as a hierarchy separator. Examples of using wildcards in object names include the following: `mybus*`, `mybus[??]`, `mybus[*]`.

Verilog uses the following syntax for module names:

v: *cell*

v:	Identifies a view object
cell	The name of the Verilog module

Instance, ports, and net names have the following syntax:

cell typespec object_path

cell	The name of the Verilog module
typespec	A single letter followed by a colon. The letter can be one of the following, and is used to determine the kind of object when objects have the same names: **i:** identifies the name of an instance. **p:** identifies the name of an entire port (the port itself). **b:** identifies the name of a slice of a port (bit-slice of the port). **n:** identifies the name of an internal net. This designation is required for all internal nets.
object_path	An instance path with a dot (.) used as a hierarchy separator in the name. The object path ends with the name of the desired object.

See *Object Naming Examples, on page 7-32*.

VHDL

This section provides syntax for object names in VHDL. No spaces are allowed in names. You can use the ***** and **?** characters as wildcards in names. These characters do not match the dot (**.**) used as a hierarchy separator. Examples of using wildcards in object names include the following: `mybus*`, `mybus[??]`, `mybus[*]`.

VHDL has the following syntax for design unit names:

v:[*lib***.**]*cell* [**.***view*]

v:	Identifies a view object
lib	The name of a library that contains the design unit. The default is the library containing the top-level design unit.
cell	The name of the design unit entity.
view	The name of the design unit architecture. The use of *view* with VHDL designs is optional; it is required only when the design unit has more than one architecture.

Instance, port, and net names have the following syntax:

[[*lib***.**] *cell* [**.***view*]] | [*typespec*] *objpath*

lib	The name of a library that contains the design unit. The default is the library containing the top-level design unit.
cell	The name of the VHDL entity.
view	The name of the design unit architecture. The use of *view* with VHDL designs is optional; it is required only when the design unit has more than one architecture.
typespec	A single letter followed by a colon that is used (if needed) to resolve the ambiguity of two or more objects in the design that have the same name. *typespec* can have the following values: **i:** identifies the name of an instance. **p:** identifies the name of an entire port (the port itself). **b:** identifies the name of a slice of a port (bit-slice of the port). **n:** identifies the name of an internal net. This designation is required for all internal nets.
object_path	An instance path using dot (.) as a hierarchy separator. The object path ends with the name of the desired object.

See *Object Naming Examples, on page 7-32*.

Naming Objects in the SCOPE Window

If you choose an object from a SCOPE pulldown menu, it has the appropriate prefix appended automatically. If you drag and drop an object from an RTL view, for example, make sure to add the prefix appropriate to the language used for the module. See Verilog, on page 7-30 and VHDL, on page 7-31 for details.

Object Naming Examples

To identify all bits of instance statereg in module statemod:

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

To identify instances one level of hierarchy from the top with names that begin with state:

```
i:*.state*
```

To identify port mybus[19:0] instead of a driving register that also has the name mybus[19:0]:

```
p:mybus[19:0]
```

To identify top-level port mybus bit 1 instead of a driving register that also has the name mybus[1]:

```
b:mybus[1]
```

The following shows the constraint file (.sdc) entries that correspond to the previous source code examples:

```
define_attribute {statemod|i:statereg[*]} syn_encoding sequential
define_multicycle_path -to {*.*slowreg[*]} 2
```

Tcl Timing Constraints

This section discusses the following timing constraints, listed in alphabetical order:

- define_clock, on page 7-34
- define_clock_delay, on page 7-36
- define_false_path, on page 7-37
- define_input_delay, on page 7-38
- define_multicycle_path, on page 7-39
- define_reg_input_delay, on page 7-42
- define_reg_output_delay, on page 7-43

For black-box timing models, see Black-box Timing Models, on page 7-44.

The synthesis tool global frequency constraint is set in the Frequency (MHz) field of the Project view. This constraint can also be set in the project file using the set_option -frequency Tcl command.

define_clock

Defines a clock with a specific duty cycle and frequency or clock period goal. For the equivalent SCOPE spreadsheet interface and descriptions of the options, see Clocks Panel, on page 7-10 and Clock Constraint Options, on page 7-12, respectively.

You can have multiple clocks with different clock frequencies. Set the default frequency for all clocks with the set_option -frequency Tcl command in the project file. If you do not specify a global frequency, the timing analyzer uses a default. Use the define_clock timing constraint to override the default and specify unique clock frequency goals for specific clock signals. Additionally, you can use the define_clock timing constraint to set the clock frequency for a clock signal output of clock divider logic. The clock name is the output signal name for the register instance. This is the syntax:

define_clock [**-disable**] [**-virtual**] **-name {** *clock_name* **}**
[**-freq** *MHz* **|** **-period** *ns*] [**-clockgroup** *domain*]
[**-rise** *value* **-fall** *value*] [**-route** *ns*]
[**-comment** *text_string*]

-disable	Disables a previous clock definition.
-virtual	Specifies arrival and required times on top level ports that are enabled by clocks external to the chip (or block) that you are synthesizing. When specifying -name for the virtual clock, the field can contain a unique name not associated with any port or instance in the design.
-name	(Required). Specifies the clock name. The name can be a top-level port in the design or an internal instance used as a clock generator for the chip. In the case of virtual clocks, the field can contain a unique name not associated with any port or instance in the design.
-freq	Defines the frequency of the clock in MHz. You can specify either this or -period, but not both.
-period	Specifies the period of the clock in ns. Specify either this or -freq, but not both.
-clockgroup	Allows you to specify clock relationships. You put related (synchronized) clocks in the same clock group and unrelated clocks in different groups. • The synthesis tool calculates the relationship between clocks in the same clock group, and analyzes all paths between them. Paths between clocks in different groups are ignored (false paths). • By default, all clocks are in the same clock group (default_clkgroup is the default name). See Clock Groups, on page 7-12 for more information.
-rise/-fall	Specifies a non-default duty cycle. By default, the Synplify synthesis tool assumes that the clock is a 50% duty cycle clock, with the rising edge at 0 and the falling edge at period/2. If you have another duty clock cycle, specify the appropriate Rise At and Fall At values. For more information, see Rise and Fall Constraints, on page 7-14.

-route	An advanced user option that improves the path delays of all registers controlled by this clock. The value is the difference between the synthesis timing report path delays and the value in the place-and-route timing report. The -route constraint applies globally to the clock domain, and can over constrain registers where constraints are not needed. For details, see Route Option, on page 7-15. Before you use this option, evaluate the path delays on individual registers in the optimization timing report and try to improve the delays by applying the constraints define_reg_input_delay and define_reg_output_delay only on the registers that need them.

Here are some syntax examples:

```
define_clock -name {clock} -period 10.0 -clockgroup d1

define_clock -virtual -name {CLK5} -period 20.0 -clockgroup d2

define_clock -name {CLK4} -period 20.000 -clockgroup d2
   -rise 0.000 -fall 10.000
```

define_clock_delay

Defines delay between the clocks in the design. By default, the software automatically calculates clock delay based on the clock parameters you define through the define_clock command. However, if you use define_clock_delay, the specified delay value overrides any calculations made by the software. The results shown in the timing report, in the Clock Relationships section, are based on calculations made using this constraint.

Here is the syntax:

define_clock_delay [**-rise|fall**] **{***clock_name1***}** [**-rise|fall**] **{***clock_name2***}** *delay_value*

-rise|fall	**Specifies the clock edge**
clock_name	Specifies the clocks to constrain The clock must be already defined with define_clock.
-delay_value	Specifies the delay, in nanoseconds, between the two clocks. You can also specify a value false which defines the path as a false path.

define_false_path

Defines paths to ignore (remove) during timing analysis and give lower (or no) priority during optimization. The false paths are also passed on to supported place-and-route tools. For information about the equivalent SCOPE spreadsheet interface and the options, see False Paths Panel, on page 7-24.

This is the syntax:

define_false_path -from { *reg* | *input* **}** | **-to {** *reg* | *output* **}** |
-through { *net* **}** [**-comment** *text_string*]

-from	Specifies the start point for the false path. From points can be: registers, top-level input, or bidirectional ports. For more information, see Defining From/To/Through for Timing Exceptions, on page 3-28 in the *Synplify User Guide*. You can combine this option with -to or -through to get a specific path.
-to	Specifies the end point for the false path. To points can be: registers, top-level output or bidirectional ports. For more information, see Defining From/To/Through for Timing Exceptions, on page 3-28 in the *Synplify User Guide*. You can combine this option with -from or -through to get a specific path.
-through	Specifies the intermediate points (nets) for the timing exception. You can use any nets in the design as intermediate points. The nets can be specified as a space-separated list, which is treated as an OR list. The exception is applied to all paths that cross any instance in the list. See Defining From/To/Through for Timing Exceptions, on page 3-28 in the *Synplify User Guide*. You can combine this option with -to or -from to get a specific path. To keep the signal name intact throughout synthesis when you this option, set the syn_keep directive (Verilog or VHDL) on the signal.

See False Path Constraint Examples, on page 7-25, and False Path Constraint Priority, on page 7-26, for additional information.

define_input_delay

Specifies the external input delays on top-level ports in the design. It is the delay outside the chip before the signal arrives at the input pin. This constraint is used to model the interface of the inputs of the FPGA with the outside environment. The tool has no way of knowing what the input delay is unless you specify it in a timing constraint. For information about the equivalent SCOPE spreadsheet interface and multiple I/O constraints, see Inputs/Outputs Panel, on page 7-18.

Here is the syntax:

> **define_input_delay** [**-disable**] **{** *input_port_name* **}** | **-default** *ns* [**-route** *ns*]
> [**-ref** *clock_name*:*edge*] [**-comment** *text_string*]

-disable	Disables a previous delay specification on the named port.
input_port_name	The name of the input port.
-default	Sets a default input delay for all inputs. Use this option to set an input delay for all inputs. You can then set define_input_delay on individual inputs to override the default constraint. This example sets a default input delay of 3.0 ns: define_input_delay -default 3.0 This constraint overrides the default and sets the delay on input_a to 10.0 ns: define_input_delay {input_a} 10.0
-ref	(Recommended.) Clock name and edge that triggers the event. The value must include either the rising edge or falling edge. **r** - rising edge **f** - falling edge For example: define_input_delay {portb[7:0]} 10.00 -ref clock2:f
-route	An advanced option, which includes route delay when the Synplify synthesis tool tries to meet the clock frequency goal. Use the -route option on an input port when the place-and-route timing report shows that the timing goal is not met because of long paths through the input port. See Route Option, on page 7-19 for an example of its use.

Here are some examples of the define_input_delay constraint:

```
define_input_delay {porta[7:0]} 7.8 -ref clk1:r
define_input_delay -default 8.0
define_input_delay -disable {resetn}
```

define_multicycle_path

Specifies a path that is a timing exception because it uses multiple clock cycles. This constraint provides extra clock cycles to the designated paths for timing analysis and optimization. For information about the equivalent SCOPE spreadsheet interface, see Multicycle Paths Panel, on page 7-21.

Here is the syntax:

> **define_multicycle_path -from {** *reg* | *input* **}** | **-to {** *reg* | *output* **}** |
> **-through {** *net* **}** *clock_cycles* [**-comment** *text_string*]

-from	Specifies the start point for the multiple-cycle timing exception. From points can be registers, top-level input, or bidirectional ports. For more information, see *Defining From/To/Through for Timing Exceptions, on page 3-28* in the *Synplify User Guide*. You can combine this option with -to or -through to get a specific path.
-to	Specifies the end point for the timing exception. To points can be registers, top-level output, or bidirectional ports. For more information, see Defining From/To/Through for Timing Exceptions, on page 3-28 in the *Synplify User Guide*. You can combine this option with -from or -through to get a specific path.
-through	Specifies the intermediate points for the timing exception. You can use any nets in the design as intermediate points. The intermediate points can be specified as a space-separated list, which is treated as an OR list. The exception is applied to all paths that cross any instance in the list. See Defining From/To/Through for Timing Exceptions, on page 3-28 in the *Synplify User Guide*. You can combine this option with -to or -from to get a specific path. To keep the signal name intact throughout synthesis when you this option, set the syn_keep directive (Verilog or VHDL) on the signal.
clock_cycles	The number of clock cycles of the start clock to use for this path.

Here are some examples of the define_multicycle_path constraint syntax:

```
define_multicycle_path -from{i:regs.addr[4:0]}
   -to{i:special_regs.w[7:0]} 2
```

```
define_multicycle_path -to {i:special_regs.inst[11:0]} 2
```

```
define_multicycle_path -from {p:porta[7:0]}
   -through {n:prgmcntr.pc_sel44[0]} -to {p:portc[7:0]} 2
```

For additional examples of multicycle paths, see *Multicycle Path Examples, on page 7-22*.

define_output_delay

Specifies the delay of the logic outside the FPGA driven by the top-level outputs. This constraint is used to model the interface of the outputs of the FPGA with the outside environment. The default delay outside the FPGA is 0.0 ns. Output signals typically drive logic that exists outside the FPGA, but the tool has no way of knowing the delay for that logic unless you specify it using a timing constraint. For information about the equivalent SCOPE spreadsheet interface, see Inputs/Outputs Panel, on page 7-18.

Here is the syntax:

> **define_output_delay** [**-disable**] **{** *output_port_name* **}** |
> **-default** *ns* [**-route** *ns*] [**-ref** *clock_name*:*edge*] [**-comment** *text_string*]

-disable	Disables a previous delay specification on the named port.
output_port_name	The name of the output port.

-default	Sets a default input delay for all outputs. Use this option to set a delay for all outputs. You can then set define_output_delay on individual outputs to override the default constraint. This example sets a default output delay of 8.0 ns. The delay is outside the FPGA. define_output_delay -default 8.0 The following constraint overrides the default and sets the output delay on output_a to 10.0 ns. This means that output_a drives 10 ns of combinational logic before the relevant clock edge. define_output_delay {output_a} 10.0
-ref	Defines the clock name and edge that controls the event. Value must be one of the following: **r** - rising edge **f** - falling edge For example: define_output_delay {portb[7:0]} 10.00 -ref clock2:f
-route	An advanced option, which includes route delay when the Synplify synthesis tool tries to meet the clock frequency goal. See Route Option, on page 7-19 for an example of its use with an input register.

define_path_delay

Specifies point-to-point delay, in nanoseconds, for maximum and minimum delay constraints. You must specify both start and end points for the constraint.

If you specify both define_path_delay -max and define_multicycle_path constraints for the same path, the more restrictive of the two constraints is used by the software.

Here is the syntax:

define_path_delay [**-disable**] [**-comment** *string* **]**
-from *start_point* **-to** *end-point*
-max *value*

-disable	Disables the constraint.
-comment	Optional comment.
from *start_point*	Defines the starting point of the path, and can be one of the following: register, input port, or black box.

to *end_point*	Defines the ending point of the path, and can be one of the following: register, input port, or black box.
through	Specifies the intermediate points for the timing exception. You can use any nets in the design as intermediate points. The intermediate points can be specified as a space-separated list, which is treated as an OR list. The exception is applied to all paths that cross any instance in the list. See Defining From/To/Through for Timing Exceptions, on page 3-28 in the *Synplify User Guide*. You can combine this option with -to or -from to get a specific path. To keep the signal name intact throughout synthesis when you this option, set the syn_keep directive (Verilog or VHDL) on the signal.
max *delay_value*	Sets the maximum allowable delay for the specified path. This is an absolute value in nanoseconds. Shown as max analysis in the timing report.

Here is a syntax example:

```
define_path_delay -from {i:dmux.alua[5]}
-to {i:regs.mem_regfile_15[0]} -max 0.800
```

define_reg_input_delay

Speeds up paths feeding a register by a given number of nanoseconds. The Synplify synthesis tool attempts to meet the global clock frequency goals for a design as well as the individual clock frequency goals (set with define_clock). Use this constraint to speed up the paths feeding a register. For information about the equivalent SCOPE spreadsheet interface, see Registers Panel, on page 7-20.

This is the syntax:

define_reg_input_delay { *register_name* **}** [**-route** *ns*] [**-comment** *text_string*]

register_name	A single bit, an entire bus, or a slice of a bus.
-route	Advanced user option that you use to tighten constraints during resynthesis, when the place-and-route timing report shows the timing goal is not met because of long paths to the register. See Route Option, on page 7-19 for an example of its use.

define_reg_output_delay

Speeds up paths coming from a register by a given number of nanoseconds. The Synplify synthesis tool attempts to meet the global clock frequency goals for a design as well as the individual clock frequency goals (set with define_clock). Use this constraint to speed up the paths coming from a register. For information about the equivalent SCOPE spreadsheet interface, see Registers Panel, on page 7-20.

This is the syntax:

define_reg_output_delay { *register_name* **}** [**-route** *ns*] [**-comment** *text_string*]

register_name	A single bit, an entire bus, or a slice of a bus.
-route	Advanced user option that you use to tighten constraints during resynthesis, when the place-and-route timing report shows the timing goal is not met because of long paths from the register. For an example of its use (on an input register), see Route Option, on page 7-19.

Black-box Timing Models

This section describes black-box code directives.

Black-box Source Code Directives

Black-box timing models are specific to components that you have instantiated (or declared) as black boxes using syn_black_box. You specify the models with source code directives.In addition, you can specify black box pin and pad attributes (Black Box Pin Definitions, on page 7-46).

You must put the directives in the source code because the models are specific to individual instances. There are no corresponding Tcl directives available for use in a constraint file. The following are the attributes you use with syn_black_box to characterize black-box timing:

- syn_isclock – specifies a clock port on a black box. See syn_isclock, on page 8-72 for details.
- syn_tpd*n* – timing propagation for combinational delay through the black box. See syn_tpd<n>, on page 8-96 for details.
- syn_tsu*n* – timing setup delay required for input pins relative to the clock. See syn_tsu<n>, on page 8-100 for details.
- syn_tco*n* – timing clock to output delay through the black box, where *n* is an integer from 1 through 10, inclusive. See syn_tco<n>, on page 8-93 for details.

For syn_tpd, syn_tsu, and syn_tco, there are 10 instances available of each of these directives; for example: syn_tpd1, syn_tpd2, syn_tpd3, … syn_tpd10. If you need more than 10, you can declare the desired amount (start with an integer greater than 10), for example:

```
attribute syn_tpd11 : string;
attribute syn_tpd12 : string;
```

A *bundle* is a collection of buses and scalar signals. To assign values to bundles, use the following syntax. The values are in ns.

For syn_tpd: **"***bundle* **->** *bundle* **=** *value***"**
For syn_tsu: **"***bundle* **->** [**!**]*clock* **=** *value***"**
For syn_tco: **"**[**!**]*clock* **->** *bundle* **=** *value***"**

The optional exclamation mark (**!**) indicates a negative edge for a clock. The objects of a bundle must be separated by commas with no spaces between. A valid bundle is A,B,C which lists three signals.

VHDL Example

In addition to the syntax used in the code below, you can also use the following Verilog-style syntax to specify the black-box attributes:

```
attribute syn_tco1 of inputfifo_coregen : component is
   "rd_clk->dout[48:0]=3.0";
```

The following is an example of a VHDL black box:

```
-- A USE clause for the synthesis tool Attributes
-- package was included earlier to make the attribute
-- definitions visible here.
architecture top of top is
component rcf16x4z port (
   ado, ad1, ad2, ad3 : in std_logic;
   dio, 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_tco1 of rcf16x4z : component is
   "clk -> do0,do1 = 4.0";
attribute syn_tpd1 of rcf16x4z : component is
   "ado,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
   "ado,ad1,ad2,ad3 -> ck = 1.2";
attribute syn_tsu2 of rcf16x4z : component is
   "wren,wpe -> ck = 0.0";

-- Other code
```

Verilog Example

```
module ram32x4(z, d, addr, we, clk);
/* synthesis black_box
syn_tco1="clk->z[3:0]=4.0"
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
```

Black Box Pin Definitions

You define the pins on a black box with the following directives in the source code:

- black_box_pad_pin – indicates that a black box is an I/O pad for the rest of the design. See black_box_pad_pin, on page 8-56 for details.
- black_box_tri_pins – indicates tristates on black boxes. See black_box_tri_pins, on page 8-58 for further information.

Timing Report

A timing report is included as part of the log file (*project_name*.srr – see Log File, on page 4-7) in the results directory. You can find it by opening the log file (View -> Log File) after a synthesis run, and searching for “START TIMING REPORT”.

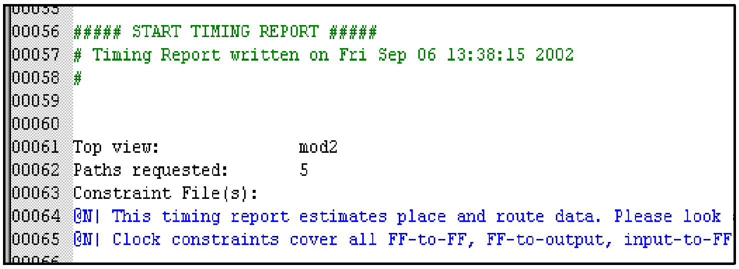
You can control the size of the timing report by choosing Project -> Implementation Options and clicking the tab of the Timing Report panel. This panel lets you specify the number of start/end points and the number of critical paths to report. See Timing Report Panel, on page 3-34, for more information.

The timing report has the following sections:

- Timing Report Header, on page 7-47
- Performance Summary, on page 7-48
- Clock Relationships, on page 7-49
- Interface Information, on page 7-49
- Detailed Clock Report, on page 7-50

Timing Report Header

The timing report header lists the date and time, the name of the top-level module, the number of paths requested for the timing report, and the constraint files used.



```
##### START TIMING REPORT #####
# Timing Report written on Fri Sep 06 13:38:15 2002
#


Top view:               mod2
Paths requested:        5
Constraint File(s):
@N| This timing report estimates place and route data. Please look
@N| Clock constraints cover all FF-to-FF, FF-to-output, input-to-FF
```

Figure 7-3: Timing Report header

Performance Summary

The Performance Summary section of the timing report reports estimated and requested frequencies for the clocks, with the clocks sorted by negative slack. The timing report has a different section for detailed clock information (see Detailed Clock Report, on page 7-50). The Performance Summary lists the following information for each clock in the design:

Performance Summary	Description
Starting Clock	The name of the clock. If the clock name is system, the clock is a collection of clocks with an undefined clock event. Rising and falling edge clocks are reported as one clock domain.
Requested Frequency	Target frequency.
Estimated Frequency	Estimated frequency after synthesis.
Requested Period	Target clock period.
Estimated Period	Estimated period after synthesis.
Slack	Difference between Requested Period and Estimated Period. The none category is a collection of clocks with an undefined clock event. This can include clocks that use the global frequency. Slack for a starting clock listed as none is the worst slack for all paths in the none category.
Clock Type	The type of clock: inferred, declared, derived or system. The system clock is the delay for the combinatorial path.

The Synplify synthesis tool does not report inferred clocks that have an unreasonable slack time.

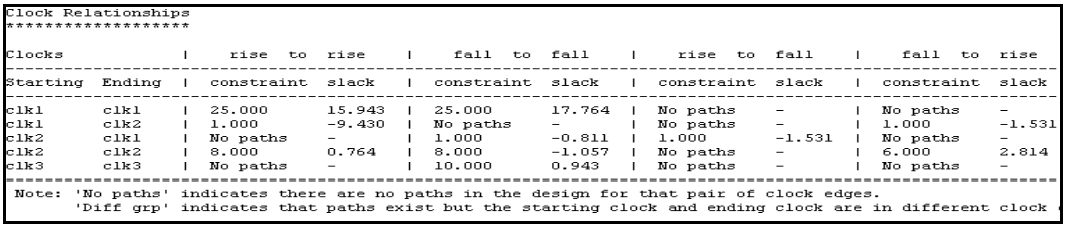
Also, a real clock might have a negative period. For example, suppose you have a clock going to a single flip-flop, which has a single path going to an output. If you specify an output delay of –1000 on this output, then the synthesis tool cannot calculate the clock frequency. It reports a negative period and no clock.

Clock Relationships

For each pair of clocks in the design, the Clock Relationships section of the timing report lists both the required time (constraint) and the worst slack time for each of the intervals rise to rise, fall to fall, rise to fall, and fall to rise.

This information is provided for the paths between related clocks (that is, clocks in the same clock group). If there is no path at all between two clocks, then that pair is not reported. If there is no path for a given pair of edges between two clocks, then an entry of No paths appears.

For information about how these relationships are calculated, see Clock Groups, on page 7-12. For tips on using clock groups, see Defining Other Clock Requirements, on page 3-25 in the *Synplify User Guide*.



```
Clock Relationships
*******************

Clocks              |    rise  to  rise    |    fall  to  fall    |    rise  to  fall    |    fall  to  rise
------------------------------------------------------------------------------------------------------------------
Starting  Ending    |  constraint  slack   |  constraint  slack   |  constraint  slack   |  constraint  slack
------------------------------------------------------------------------------------------------------------------
clk1      clk1      |  25.000      15.943  |  25.000      17.764  |  No paths    -       |  No paths    -
clk1      clk2      |  1.000       -9.430  |  No paths    -       |  No paths    -       |  1.000       -1.531
clk2      clk1      |  No paths    -       |  1.000       -0.811  |  1.000       -1.531  |  No paths    -
clk2      clk2      |  8.000       0.764   |  8.000       -1.057  |  No paths    -       |  6.000       2.814
clk3      clk3      |  No paths    -       |  10.000      0.943   |  No paths    -       |  No paths    -
==================================================================================================================
 Note: 'No paths' indicates there are no paths in the design for that pair of clock edges.
       'Diff grp' indicates that paths exist but the starting clock and ending clock are in different clock
```

Interface Information

The interface section of the timing report contains information on arrival times, required times, and slack for the top-level ports. It is divided into two subsections, one each for Input Ports and Output Ports. Bidirectional ports are listed under both. For each port, the interface report contains the following information.

Port parameter	**Description**
Port Name	Port name.
Starting Reference Clock	The reference clock.
User Constraint	The input/output delay. If a port has multiple delay records, the report contains the values for the record with the worst slack. The reference clock corresponds to the worst slack delay record.

Port parameter	**Description**
Arrival Time	Input ports: define_input_delay, or default value of 0. Output ports: path delay (including clock-to-out delay of source register). For purely combinational paths, the propagation delay is calculated from the driving input port.
Required Time	Input ports: clock period - (path delay + setup time of receiving register + define_reg_input_delay value). Output ports: clock period – define_output_delay. Default value of define_output_delay is 0.
Slack	Required Time – Arrival Time

Detailed Clock Report

Each clock reported in the performance summary also has a detailed clock report section in the timing report. The clock reports are listed in order of negative slack.

General Critical Path Information

This section contains general information about the most critical paths in the design.

Clock Information	**Description**
N most critical start points	Start points can be input ports or registers. If the start point is a register, you see the starting pin in the report. To change the number of start points reported, choose Project -> Implementation Options, and set the number on the Timing Report panel.
N most critical end points	End points can be output ports or registers. If the end point is a register, you see the ending pin in the report. To change the number of end points reported, select Project -> Implementation Options, and set the number on the Timing Report panel.

Clock Information	Description
N worst path information (see the next table for details)	Starting with the most critical path, the Worst Path Information sections contain details of the worst paths in the design. Paths from clock A to clock B are reported as critical paths in the section for clock A. You can change the number of critical paths on the Timing Report panel of the Options for implementation dialog box.

Worst Path Information

For each critical path, the timing report has a detailed description. It starts with a summary of the information and is followed by a detailed pin-by-pin report. The summary reports information like requested period, actual period, start and end points, and logic levels. Note that the requested period here is period -route, while the requested period in the Performance Summary (Performance Summary, on page 7-48) is just the clock period.

The detailed path report uses this format: Output pin – Net – Input pin – Output pin – Net – Input pin. The following table describes the critical path information reported:

Critical path information	Description
Instance/Net Name	Technology view names for the instances and nets in the critical path
Type	Type of cell
Pin Name	Name of the pin
Pin Dir	Pin direction
Delay	The delay value.
Arrival Time	Clock delay at the source + the propagation delay through the path
Fan Out	Number of fanouts for the point in the path

CHAPTER 8

Synthesis Attributes and Directives

This chapter shows you how to specify attributes and directives. It contains individual attribute and directive descriptions, syntax definitions, and examples.

These are the chapter topics:

Summary of Attributes and Directives

The following sections summarize the synthesis attributes and directives:

- Specifying Attributes and Directives, on page 8-2
- Attribute and Directive Summary, by Vendor, on page 8-3
- Attribute Summary (Alphabetical), on page 8-3
- Directive Summary (Alphabetical), on page 8-4

For detailed descriptions of individual attributes and directives, see Attributes, on page 8-15 and Directives, on page 8-55, respectively.

Specifying Attributes and Directives

Synthesis attributes and directives let you direct the way a design is analyzed, optimized, and mapped during synthesis. For example, you can specify how you want the Synplify synthesis tool to manage hierarchy during optimization or which objects to treat as black boxes.

By definition, *attributes* control mapping optimizations and *directives* control compiler optimizations. Because of this difference, directives must be entered in the HDL source code, but attributes can be entered either in an `.sdc` constraint file or in HDL source code. You typically use a constraint file to assign attributes.

If you specify attributes in source code, whenever you change their values you must recompile the design. Recompilation can be very time consuming for large designs. If you use a constraint file instead, you can easily change attribute values at any time, without recompiling. The preferred way to specify attributes is using the SCOPE spreadsheet (see The SCOPE Window, on page 7-4), which automatically generates and edits an `.sdc` constraint file.

For information on specifying attributes in constraint files, in particular using the SCOPE spreadsheet, see Specifying Attributes in a Constraint File, on page 8-7.

For information on specifying directives (and attributes) in the HDL source code, see Specifying Directives and Attributes in HDL, on page 8-11. Remember, though, that it is more flexible to specify attributes using the SCOPE spreadsheet.

Attribute and Directive Summary, by Vendor

For a list of vendor-specific attributes and directives, see the appropriate vendor documentation, as shown in the following table. For alphabetical summaries of attributes and directives, see *Attribute Summary (Alphabetical), on page 8-3* and *Directive Summary (Alphabetical), on page 8-4*, respectively.

Table 8-1: Vendor-specific Attributes and Directives

Vendor	Attributes and Directives
Actel	Actel Attribute and Directive Summary, on page 11-11

Attribute Summary (Alphabetical)

The following table summarizes the synthesis attributes. For details on individual attributes, see the alphabetical list in section Attributes, on page 8-15, . For a summary of the directives, see Directive Summary (Alphabetical), on page 8-4.

Table 8-2: Attributes, Alphabetical Summary

Attribute	Description
alsloc	Preserves relative placement in Actel designs.
alspin	Assigns scalar or bus ports to I/O pin numbers in Actel designs.
alspreserve	Specifies nets that must be preserved by the Actel place-and-route tool.
syn_direct_enable	Assigns clock enable nets to dedicated flip-flop enable pins. It can also be used as a compiler directive that marks flip-flops with clock enables for inference.
syn_encoding	Specifies the encoding style for state machines.
syn_global_buffers	Determines the number of global buffers available.
syn_hier	Determines hierarchical control across module or component boundaries.
syn_maxfan	Sets maximum fanout for individual nets or registers, overriding the default.

Table 8-2: Attributes, Alphabetical Summary (Continued)

Attribute	**Description**
syn_netlist_hierarchy	Controls hierarchy generation in EDIF and VQM output files
syn_noarrayports	Specifies ports as individual signals or bus arrays.
syn_noclockbuf	Disables automatic clock buffer insertion.
syn_preserve_sr_priority	Forces set/resets to observe priority (Actel).
syn_radhardlevel	Specifies radiation-resistant design technique (Actel).
syn_ramstyle	Determines how RAMs are implemented.
syn_reference_clock	Specifies a clock frequency other than that implied by the signal on the clock pin of the register.
syn_replicate	Controls replication.

Directive Summary (Alphabetical)

The following table summarizes the synthesis directives. For details on individual directives, see the alphabetical list in section Directives, on page 8-55, . For a summary of the attributes, see Attribute Summary (Alphabetical), on page 8-3.

Table 8-3: Directives, Alphabetical Summary

Directive	**Description**
black_box_pad_pin	Specifies that a pin on a black box is an I/O pad. It is applied to a component, architecture, or module, with a value that specifies the set of pins on the module or entity.
black_box_tri_pins	Specifies that a pin on a black box is a tristate pin. It is applied to a component, architecture, or module, with a value that specifies the set of pins on the module or entity.
full_case	Specifies that a Verilog case statement has covered all possible cases.
loop_limit	Specifies a loop iteration limit for for loops.

Table 8-3: Directives, Alphabetical Summary (Continued)

Directive	**Description**
parallel_case	Specifies a parallel multiplexed structure in a Verilog case statement, rather than a priority-encoded structure.
syn_black_box	Defines a black box for synthesis.
syn_direct_enable	When used as a compiler directive, this attribute marks the flip-flops with clock enables for the compiler to infer.
syn_enum_encoding	Specifies the encoding style for enumerated types (VHDL only).
syn_isclock	Specifies that a black-box input port is a clock, even if the name does not indicate it is one.
syn_keep	Prevents the internal signal from being removed during synthesis.
syn_macro	
syn_noprune	Controls the automatic removal of instances that have outputs that are not driven.
syn_preserve	Preserves registers that can be optimized due to redundancy or constraint propagation.
syn_sharing	Enables/disables resource sharing of operators inside a module.
syn_state_machine	Determines if the FSM Compiler extracts a structure as a state machine.
syn_tco<n>	Defines timing clock to output delay through a black box. The *n* indicates a value between 1 and 10.
syn_tpd<n>	Specifies timing propagation for combinational delay through a black box. The *n* indicates a value between 1 and 10.
.sdc File Syntax and Example	Specifies that a black-box pin is a tristate pin.

Table 8-3: Directives, Alphabetical Summary (Continued)

Directive	**Description**
syn_tsu<n>	Specifies the timing setup delay for input pins, relative to the clock. The *n* indicates a value between 1 and 10.
translate_off/translate_on	Specifies sections of code to exclude from synthesis, such as simulation-specific code.

Specifying Attributes in a Constraint File

Attributes direct the way a design is optimized during the mapping phase of synthesis. You can specify attributes in either HDL source code (see Specifying Directives and Attributes in HDL, on page 8-11) or a constraint file (.sdc). A constraint file can be either generated automatically by the SCOPE spreadsheet or typed in manually using a text editor.

If you specify attributes in source code, whenever you change them you must *recompile* the design. Recompilation can be very time consuming for large designs. If you use a constraint file, you can easily change attributes at any time, without recompiling.

Specifying Attributes with the SCOPE Spreadsheet

You typically specify attributes using the SCOPE spreadsheet, which automatically generates and updates a constraint file (.sdc) that specifies the attributes.

To use the SCOPE spreadsheet to enter attributes, start with a compiled design. Using a compiled design allows the SCOPE software to access the objects and values from the design. Then, click the SCOPE icon (▦). When the SCOPE spreadsheet opens, click the Attributes tab. Use the displayed Attributes panel to assign attributes, either directly in individual spreadsheet cells or through the insert wizard (see SCOPE Attribute Wizard, on page 8-10). The Attributes panel columns are as follows:

Table 8-4: SCOPE Attributes Panel

Column	Description
Enabled	(Required.) Turn this on to enable the constraint.
Object Type	Specifies the type of object to which the attribute is assigned. Choose from the pulldown list, to filter the available choices in the Object field.
Object	(Required.) Specifies the object to which the attribute is attached. This field is synchronized with the Attribute field, so selecting an object here filters the available choices in the Attribute field.

Table 8-4: SCOPE Attributes Panel (Continued)

Attribute	(Required.) Specifies the attribute name. You can choose from a pulldown list that includes all available attributes for the specified technology. This field is synchronized with the Object field. If you select an object first, the attribute list is filtered. If you select an attribute first, the Synplify synthesis tool filters the available choices in the Object field. You must select an attribute before entering a value.
Value	(Required.) Specifies the attribute value. You must specify the attribute first. Clicking in the column displays the default value; a drop-down arrow lists available values where appropriate.
Val Type	Specifies the kind of value for the attribute. For example, string or boolean.
Description	Contains a one-line description of the attribute.
Comment	Lets you enter comments about the attributes.

Enter the appropriate attributes and their values, either individually, by clicking in a cell and choosing from the pulldown menu, or in groups, using the Attributes panel insert wizard (see SCOPE Attribute Wizard, on page 8-10). To specify an object to which you want to assign an attribute, you may also drag-and-drop it from the RTL or Technology view into a cell in the Object column. After you have entered the attributes, save the constraint file and add it to your project. The constraint file, which has an “sdc” file extension (.sdc), may then be found in your project directory.

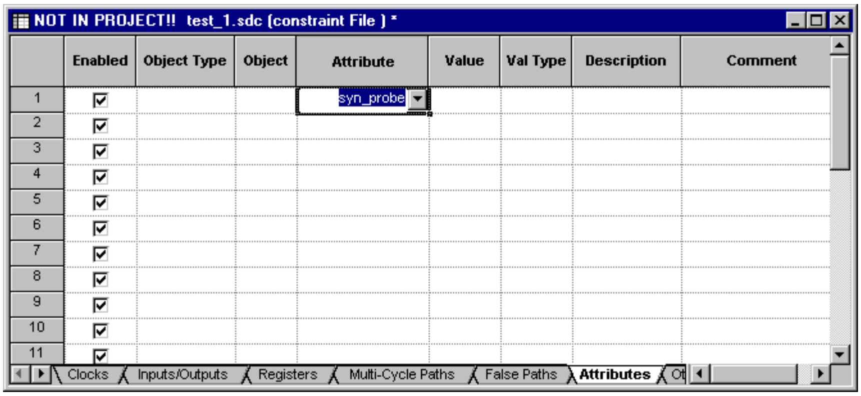
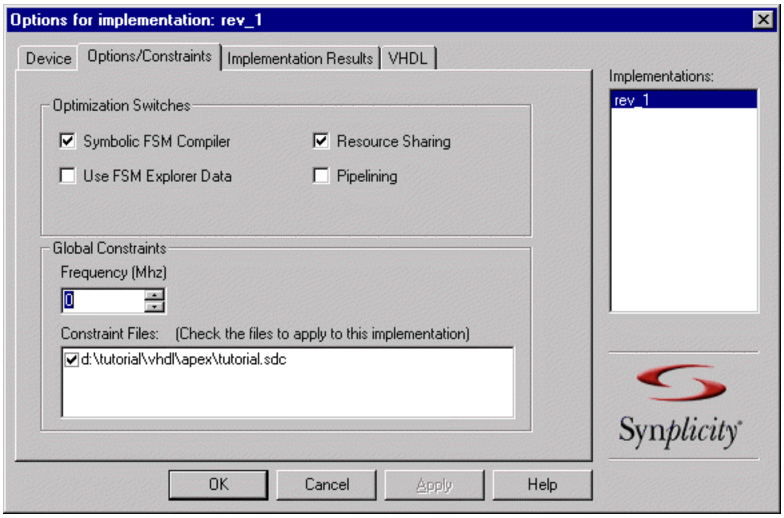


Figure 8-1: SCOPE Attributes panel

Add the constraint file to the source file list. Choose Project -> Implementation Options, and check that the file is selected on the Options/Constraints panel before you run synthesis. If you have more than one constraint file, select all those that apply to the implementation.


Options for implementation: rev_1

Device | Options/Constraints | Implementation Results | VHDL

Optimization Switches
- [✓] Symbolic FSM Compiler
- [✓] Resource Sharing
- [ ] Use FSM Explorer Data
- [ ] Pipelining

Global Constraints
Frequency (Mhz): 0

Constraint Files: (Check the files to apply to this implementation)
- [✓] d:\tutorial\vhdl\apex\tutorial.sdc

Implementations: rev_1

Figure 8-2: Specifying constraint files

.sdc File Syntax

When you use the SCOPE spreadsheet to create and modify the timing constraint files, the proper syntax is automatically generated. This is the syntax used to define attributes in the .sdc file.

define_attribute { *object* **}** *attribute_name* **{** *value* **}**

object	The design object, such as module, signal, input, instance, port, or wire name. The object naming syntax varies, depending on whether your source code is in Verilog or VHDL format. See Object Naming Syntax, on page 7-29 for details about the syntax conventions.
attribute_name	The name of the synthesis attribute. This must be an attribute, not a directive, as they are not supported in constraint files.
value	String, integer or boolean (0 or 1) value.

For more information on using the Attributes panel of the SCOPE spreadsheet to define attributes, see Adding Attributes in the SCOPE Window, on page 3-38 of the *Synplify User Guide*.

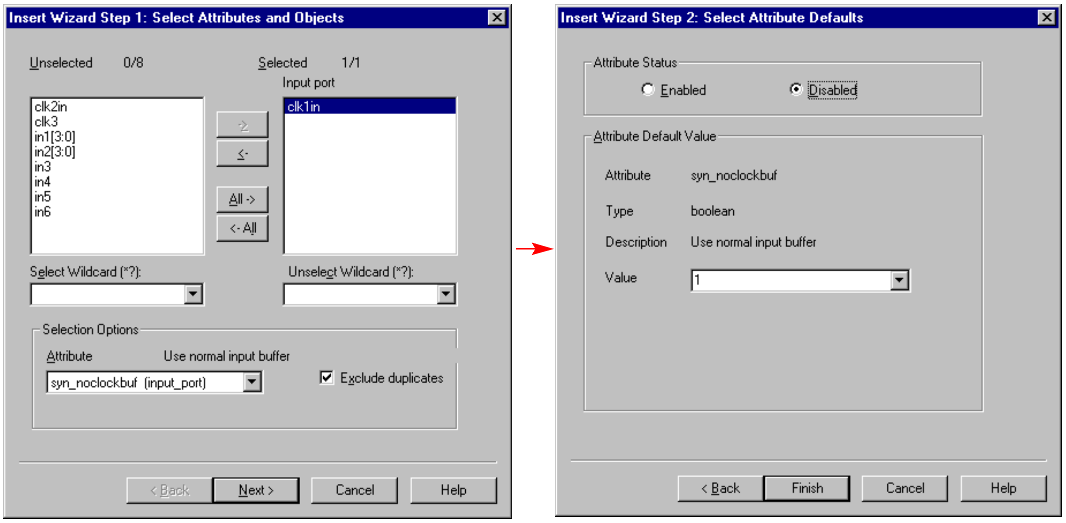
SCOPE Attribute Wizard

The SCOPE attribute wizard provides an easy interface for assigning attributes to design objects. It is most useful for assigning (or unassigning) the same attribute value to *several* objects at once. It provides a convenient way to enable, disable, and re-enable multiple identical attribute assignments. One common use is assigning default attribute values.

To assign an attribute using the attribute wizard, right-click in the SCOPE spreadsheet Attributes panel, then choose Insert Wizard from the popup menu (or from the Edit menubar menu).

The wizard consists of two dialog boxes. In the first, choose an attribute from the Selection Options pulldown list. The selected attribute determines the choices available in the Unselected list. (If you do not see a list, disable Exclude Duplicates.) To move Unselected objects to the Selected list, select them (with the mouse or with wildcards in the Select Wildcards (*?) field), then click the right arrow (->). Alternatively, select an object and double-click to move it to the Selected list.

Click Next to go to the second dialog box. Enable the attribute, set the value, and click Finish. The attribute is set in the SCOPE spreadsheet and saved in the constraint file.



Step 1 Step 2

Figure 8-3: Insert Wizard dialog boxes

In the second wizard dialog box, you set the Value of the attribute (for *all* of the selected objects). You can Enable or Disable the attribute assignment, then click Finish. The SCOPE spreadsheet reflects your choices.

Specifying Directives and Attributes in HDL

Directives control the design analysis during the compilation phase of synthesis. Because they are required at the time a design is compiled, directives can only be specified in the HDL source code.

Attributes are read after a design is compiled, during the mapping phase of synthesis. They can therefore be specified in either HDL source code or an .sdc constraint file. Specifying attributes in a constraint file provides the flexibility of changing them at any time without having to recompile the design. You typically define attributes in a constraint file. See Specifying Attributes in a Constraint File, on page 8-7 for more information.

The following sections describe how to specify directives and attributes in Verilog or VHDL source code.

Verilog Attribute and Directive Syntax

To define the directives (or attributes) that specify your synthesis preferences, attach them to the appropriate objects in the source code as regular Verilog or C-style comments. Each specification begins with the keyword synthesis. The directive or attribute value is either a string, placed within double quotes, or a Boolean integer (0 or 1). Directives, attributes, and their values are case sensitive and are usually in lower case.

Here is the syntax using a regular Verilog comment:

> ***//* synthesis** *directive* **|** *attribute* [**= "***value***"**]

This example shows how to use the syn_hier attribute:

```
// synthesis syn_hier = "firm"
```

This example shows the parallel_case directive:

```
// synthesis parallel_case
```

This directive forces a multiplexed structure in Verilog designs. It is implicitly true whenever you use it, which is why there is no associated value. Here is the syntax for the C-style comment:

> ***/** synthesis** *directive* **|** *attribute* [**= "***value***"**] ****/***

This example shows how you use the syn_hier attribute in a C-style comment:

```
/* synthesis syn_hier = "firm" */
```

To specify more than one directive or attribute for a given design object, place them within the same comment, separated by whitespace. Do *not* use commas (,) as separators. Here is an example where the syn_preserve and syn_implement directives are specified in a single comment:

```
module radhard_dffrs(q,d,c,s,r)
        /* synthesis syn_preserve=1 syn_state_machine=0 */;
```

If you specify a directive or attribute using a `/* . . . */` C-style comment, you must place the comment *before* the semicolon of the statement. For example:

```
module bl_box(out, in) /* synthesis syn_black_box */ ;
```

To make source code more readable, you can split long comment lines by inserting a backslash character (\) followed immediately by a newline character (carriage return). A line split this way is still read as a single line; the backslash causes the newline following it to be ignored. You can split a comment line this way any number of times. However:

- The first split cannot occur before the first attribute or directive specification.
- A given attribute or directive specification cannot be split before its equal sign (=).

For example, you cannot split the following comment line before the first equal sign or within the text “`xc_loc=`”. You can split it, for example, anywhere within the string value `"P20,P21,P22,P23,P24,P25,P26,P27"`.

```
/* synthesis syn_probe=1 xc_loc="P20,P21,P22,P23,P24,P25,P26,P27" */;
```

VHDL Attribute and Directive Syntax

All of the Synplify synthesis tool directives and attributes that you can use to specify your synthesis preferences are predefined in the attributes package in the synthesis tool library.

You can either use the attributes package or redeclare the types of directives and attributes each time you use them. You typically use the attributes package.

Using the attributes Package

Here is the syntax for using directives and attributes from the attributes package in your code:

library synplify;
use synplify.attributes.all;
-- *design_unit_declarations*
attribute *synplify_attribute* **of** *object* **:** *object_type* **is** *value* **;**

The attributes package can be found in the following location:

synplify_installation_dir/lib/vhd/synattr.vhd

The following is an example using syn_noclockbuf from the attributes package:

```
library synplify;
use synplify.attributes.all;
entity simpledff is
   port (q : out bit_vector(7 downto 0);
         d : in bit_vector(7 downto 0);
         clk : in bit);

// No explicit type declaration is necessary
attribute syn_noclockbuf of clk : signal is true;

-- Other code
```

Without Using the attributes Package

Here is the syntax for explicitly defining directives and attributes in your code, without referencing the attributes package:

-- *design_unit_declarations*
attribute *attribute_name* **:** *data_type* **;**
attribute *attribute_name* **of** *object* **:** *object_type* **is** *value* **;**

Here is an example using the syn_noclockbuf attribute:

```
entity simpledff is
   port (q : out bit_vector(7 downto 0);
         d : in bit_vector(7 downto 0);
         clk : in bit);

// Explicit type declaration
attribute syn_noclockbuf : boolean;
attribute syn_noclockbuf of clk : signal is true;

-- Other code
```

Attributes

The individual synthesis attributes are detailed in this section, in alphabetical order. Each attribute description includes the following:

- Technology support
- Detailed attribute description
- File syntax and examples for the following:
  - `.sdc` constraint file
  - Verilog source code
  - VHDL source code

For an alphabetical summary of the attributes, see Attribute Summary (Alphabetical), on page 8-3. For a summary of the attributes by vendor, refer to the appropriate vendor chapter or see Attribute and Directive Summary, by Vendor, on page 8-3.

For general information about specifying attributes, see Specifying Attributes in a Constraint File, on page 8-7.

alsloc

Attribute; Actel (except 500K and PA). Preserves relative placements of macros and IP blocks in the Actel Designer place-and-route tool. The alsloc attribute has no effect on synthesis, but is passed directly to Actel Designer.

.sdc File Syntax and Example

tests/feature_flow/docs/attributes/alsloc/sdc

define_attribute { *object* **} alsloc {** *location* **}**

In the attribute syntax, *object* is the name of a macro or IP block and *location* is the row-column address of the macro or IP block.

Following is an example of setting alsloc on a macro (u1).

```
define_attribute {u1} alsloc {R15C6}
```

Verilog Syntax and Example

object **/* synthesis alsloc = "***location***" */ ;**

Where *object* is a macro or IP block and *location* is the row-column string. For example:

tests/feature_flow/docs/attributes/alsloc/vlog

```
module test(in1, in2, in3, clk, q);
input in1, in2, in3, clk;
output q;

wire out1 /* synthesis syn_keep = 1 */, out2;

and2a u1 (.A (in1), .B (in2), .Y (out1))
            /* synthesis alsloc="R15C6" */;
assign out2 = out1 & in3;
df1 u2 (.D (out2), .CLK (clk), .Q (q))
         /* synthesis alsloc="R35C6" */;

endmodule
```

```
module and2a(A, B, Y);  // synthesis syn_black_box
input A, B;
output Y;

endmodule

module df1(D, CLK, Q); // synthesis syn_black_box
input D, CLK;
output Q;

endmodule
```

VHDL Syntax and Example

attribute alsloc of *object* **: label is "***location***" ;**

Where *object* is a macro or IP block and *location* is the row-column string.

tests/feature_flow/docs/attributes/alsloc/vhdl

```
library IEEE;
use IEEE.std_logic_1164.all;

entity test is
port(in1, in2, in3, clk : in std_logic;
                 q : out std_logic);
end test;

architecture rtl of test is
signal out1, out2 : std_logic;

component AND2A
port( A, B : in std_logic;
         Y : out std_logic);
end component;
component df1
port( D, CLK : in std_logic;
         Q : out std_logic);
end component;

attribute syn_keep : boolean;
attribute syn_keep of out1 : signal is true;
attribute alsloc: string;
attribute alsloc of U1: label is "R15C6";
attribute alsloc of U2: label is "R35C6";
attribute syn_black_box : boolean;
attribute syn_black_box of AND2A, df1 : component is true;
```

```
begin
U1: AND2A port map (A => in1, B => in2, Y => out1);
out2 <= in3 and out1;
U2: df1 port map (D => out2, CLK => clk, Q => q);
end rtl;
```

alspin

Attribute; Actel (except 500K and PA). The alspin attribute assigns the scalar or bus ports of the design to Actel I/O pin numbers (pad locations). Refer to the Actel databook for valid pin numbers. If you want to use alspin for bus ports or for slices of bus ports, you must also use the syn_noarrayports attribute. See *Specifying Locations for Actel Bus Ports, on page 6-3* of the *Synplify User Guide* for information on assigning pin numbers to buses and slices.

.sdc File Syntax and Example

define_attribute { *port_name* **} alspin {** *pin_number* **}**

In the attribute syntax, *port_name* is the name of the port and *pin_number* is the Actel I/O pin.

```
define_attribute {DATAOUT} alspin {48}
```

Verilog Syntax and Example

object **/* synthesis alspin = "***pin_number***" */ ;**

Where *object* is the port and *pin_number* is the Actel I/O pin. For example:

```
module comparator (datain, clk, dataout);
output dataout /* synthesis alspin="48" */;
input [7:0] datain;
input clk;

// Other code
```

VHDL Syntax and Example

attribute alspin of *object* **:** *object_type* **is "***pin_number***" ;**

Where *object* is the port, *object_type* is signal, and *pin_number* is the Actel I/O pin. For example:

```
entity comparator is
   port (datain : in bit_vector(7 downto 0);
      clk : in bit;
      dataout : out bit);
attribute alspin : string;
attribute alspin of dataout : signal is "48";

-- Other code
```

alspreserve

Attribute; Actel (except 500K and PA). Specifies a net that you do not want removed (optimized away) by the Actel Designer place-and-route tool. The alspreserve attribute has no effect on synthesis, but is passed directly to the Actel Designer place-and-route software. However, to prevent the net from being removed during the synthesis process, you must also use the syn_keep directive.

.sdc File Syntax and Example

define_attribute { n:*net_name* **} alspreserve { 1 }**

In the attribute syntax, *net_name* is the name of the net to preserve.

tests/feature_flow/docs/attributes/alspreserve/sdc

```
define_attribute {n:and_out3} alspreserve {1};
define_attribute {n:or_out1} alspreserve {1};
```

Verilog Syntax and Example

object **/* synthesis alspreserve = 1 */ ;**

Where *object* is the name of the net to preserve. For example:

tests/feature_flow/docs/attributes/alspreserve/vlog

```
module complex (in1, out1);
input [6:1] in1;oh
output out1;
wire out1;
wire or_oosut1 /* synthesis syn_keep=1 alspreserve=1 */;
wire and_out1;
wire and_out2;
wire and_out3 /* synthesis syn_keep=1 alspreserve=1 */;
assign and_out1 = in1[1] & in1[2];
assign and_out2 = in1[3] & in1[4];
assign and_out3 = in1[5] & in1[6];
assign or_out1 = and_out1 | and_out2;
assign out1 = or_out1 & and_out3;
endmodule
```

VHDL Syntax and Example

attribute alspreserve of *object* **: signal is true ;**

Where *object* is the name of the net to preserve.

For example:

tests/feature_flow/docs/attributes/alspreserve/vhdl

```
library ieee;
use ieee.std_logic_1164.all;
library synplify;
use synplify.attributes.all;
entity complex is
port (input : in std_logic_vector (6 downto 1);
output : out std_logic);
end complex;

architecture RTL of complex is
signal and_out1 : std_logic;
signal and_out2 : std_logic;
signal and_out3 : std_logic;
signal or_out1 : std_logic;
attribute syn_keep of and_out3 : signal is true;
attribute syn_keep of or_out1 : signal is true;
attribute alspreserve of and_out3 : signal is true;
attribute alspreserve of or_out1 : signal is true;

begin
and_out1 <= input(1) and input(2);
and_out2 <= input(3) and input(4);
and_out3 <= input(5) and input(6);
or_out1 <= and_out1 or and_out2;
output <= or_out1 and and_out3;
end;
```

syn_direct_enable

Attribute; The syn_direct_enable attribute controls the assignment of a clock enable net to the dedicated enable pin of a storage element (flip-flop). Using this attribute, you can direct the mapper to use a particular net as the only clock enable when the design has multiple clock-enable candidates.

You can also use this attribute as a compiler directive to infer flip-flops with clock enables. To do so, enter syn_direct_enable as a directive in source code, not the SCOPE spreadsheet.

The syn_direct_enable data type is Boolean. A value of 1 or true enables net assignment to the clock enable pin.

Verilog Syntax and Example

object **/* synthesis syn_direct_enable = 1 */ ;**

For example:

```
`timescale 1 ns/ 100 ps
module dff2(q1, d1, clk, e1, e2, e3);
input [4:0] d1;
input clk;
output [4:0] q1;
reg [4:0] q1;
input e1, e3;
input e2 /* synthesis syn_direct_enable = 1 */;

always @(posedge clk)
begin
   if (e1 & e2 & e3 ) begin
      q1 = d1;
   end
end
endmodule
```

VHDL Syntax and Examples

attribute syn_direct_enable of *object* **:** *object_type* **is true;**

The first example shows signals:

```
architecture dff2_arch of dff2 is
signal enable : std_logic;
attribute syn_direct_enable : boolean;
attribute syn_direct_enable of enable : signal is true;
```

The second example shows ports in an entity.

```
entity dff2 is
port (q1 : out std_logic_vector (4 downto 0);
      dl: in std_logic_vector (4 downto 0);
      clk , e1, e2 , e3 : in std_logic);
attribute syn_direct_enable : boolean;
attribute syn_direct_enable of e2 : signal is true;
end dff2;
```

syn_encoding

Attribute. Overrides the default FSM Compiler encoding for a state machine. This attribute takes effect only when FSM Compiler is enabled. See FSM Compiler, on page 2-13, for more information on FSM Compiler.

The default encoding style automatically assigns encoding based on the number of states in the state machine as follows:

- sequential for 0-4 states
- onehot for 5-24 states
- gray for >24 states

Use the syn_encoding attribute when you want to override these defaults. You can also use syn_encoding when you want to disable the FSM Compiler globally but there are a select number of state registers in your design that you want extracted. In this case, use this attribute with the syn_state_machine directive on just those specific registers.

syn_encoding can have the following values:

- default – synthesis selects encoding style based on the number of states. See default values above.
- onehot – Only two bits of the state register change (one goes to 0, one goes to 1) and only one of the state registers is hot (driven by 1) at a time. For example:

  ```
  0001, 0010, 0100, 1000
  ```

  This encoding style is usually fast and is well suited to FPGA architectures because it requires the least amount of state decode logic. FPGAs usually have a large number of flip-flops, so onehot encoding is effective in most designs.
- gray – More than one of the state registers can be hot. The Synplify synthesis tool *attempts* to have only one bit of the state registers change at a time, but it can allow more than one bit to change, depending upon certain conditions for optimization. For example:

  ```
  000, 001, 011, 010, 110
  ```

  Because gray is not a simple encoding (more than one bit can be set), the value must be decoded to determine the state. This encoding style can be faster than a onehot style if you have a large output decoder following a state machine.

- sequential – More than one bit of the state register can be hot. The synthesis tool makes no attempt at limiting the number of bits that can change at a time. For example:

  ```
  000, 001, 010, 011, 100
  ```

  This is one of the smallest encoding styles, so it is often used when area is a concern. Because more than one bit can be set (1), the value must be decoded to determine the state. This encoding style can be faster than a onehot style if you have a large output decoder following a state machine.

- safe – This implements the state machine in the default encoding and adds reset logic to force the state machine to a known state if it reaches an invalid state. This value can be used in combination with any of the other encoding styles described above. You specify safe before the encoding style. The safe value is only valid for a state register, in conjunction with an encoding style specification.

  For example, if the default encoding is onehot and the state machine reaches a state where all the bits are 0, which is an invalid state, the safe value ensures that the state machine is reset to a valid state.

  If recovery from an invalid state is a concern, it may be appropriate to use this encoding style, in conjunction with onehot, sequential or gray, in order to force the state machine to reset. When you specify safe, the state machine can be reset from an unknown state to its reset state.

The encoding style is implemented during the mapping phase. A message appears when the synthesis tool extracts a state machine, for example:

```
@N: "c:\design\..."|Trying to extract state machine for register
current_state
```

The log file reports the encoding styles used for the state machines in your design.

See also the following:

- For information on enabling state machine optimization for individual modules, see syn_state_machine, on page 8-89.
- For VHDL designs, see syn_encoding compared with syn_enum_encoding, on page 8-70 for comparative usage information.

Verilog Syntax and Examples

object **/* synthesis syn_encoding = "***value***" */ ;**

where *object* can be reg (register definition signals that hold the state values of state machines) and *value* can be: onehot, gray, sequential, safe, default.

In this example, syn_encoding overrides the default encoding style for current_state using the gray encoding style.

```
module prep3 (CLK, RST, IN, OUT);
input CLK, RST;
input [7:0] IN;
output [7:0] OUT;
reg [7:0] OUT;
reg [7:0] current_state /* synthesis syn_encoding="gray" */;

// Other code
```

Here is an example using safe,gray.

```
module prep3 (CLK, RST, IN, OUT);
input CLK, RST;
input [7:0] IN;
output [7:0] OUT;
reg [7:0] OUT;
reg [7:0] current_state /* synthesis syn_encoding="safe,gray" */;

// Other code
```

In this example, the encoding style for register OUT is gray. By specifying safe, if the state machine reaches an invalid state the Synplify synthesis tool will reset the values to a valid state.

VHDL Syntax and Example

attribute syn_encoding of *object* **:** *object_type* **is "***string***" ;**

where object is a signal that holds the state values of the state machines. Here is an example of using syn_encoding to define the gray encoding style for the signal s1.

```
library synplify;
use synplify.attributes.all;
package my_states is
type state is (Xstate, st0, st1, st2, st3, st4, st5, st6, st7,
               st8, st9, st10, st11, st12, st13, st14, st15);
signal s1 : state;
attribute syn_encoding of s1 : signal is "gray";
end my_states;
```

syn_global_buffers

Attribute; Actel ProAsic3E. Specifies the number of global buffers to be used in a design. The Synplify synthesis tool automatically adds global buffers for clock nets with high fanout; use this attribute to specify a maximum number of buffers and restrict the amount of global buffer resources used. Also, if there is a black box in the design that has global buffers, you can use syn_global_buffers to prevent the synthesis tool from inferring clock buffers or exceeding the number of global resources.

You specify the attribute globally on the top-level module or entity. The value you specify varies with the technology. For Actel ProAsic3E designs, it can be any integer between 6 and 18. If you specify an integer less than 6, the software infers 6 global buffers.

.sdc File Syntax and Example

define_global_attribute syn_global_buffers { *maximum* **}**

For example:

```
define_global_attribute syn_global_buffers {10}
```

Verilog Syntax and Example

object **/* synthesis syn_global_buffers = <***maximum***> */;**

For example:

```
module top (clk1, clk2, clk3, clk4, clk5, clk6, clk7,clk8,clk9,
      clk10, clk11, clk12, clk13, clk14, clk15, clk16, clk17, clk18,
      clk19, clk20, d1, d2, d3, d4, d5, d6, d7, d8, d9, d10, d11,
      d12, d13, d14, d15, d16, d17, d18, d19, d20, q1, q2, q3, q4, q5,
      q6, q7, q8, q9, q10, q11, q12, q13, q14, q15, q16, q17, q18,
      q19, q20, reset) /* synthesis syn_global_buffers = 10 */;

input clk1, clk2, clk3, clk4, clk5, clk6, clk7,clk8,clk9, clk10,
      clk11, clk12, clk13, clk14, clk15, clk16, clk17, clk18,
      clk19, clk20;
input d1, d2, d3, d4, d5, d6, d7, d8, d9, d10, d11, d12, d13, d14,
      d15, d16, d17, d18, d19, d20;
output q1, q2, q3, q4, q5, q6, q7, q8, q9, q10, q11, q12, q13, q14,
      q15, q16, q17, q18, q19, q20;
input reset;
```

```
reg q1, q2, q3, q4, q5, q6, q7, q8, q9, q10,
      q11, q12, q13, q14, q15, q16, q17, q18, q19, q20;

always @(posedge clk1 or posedge reset)
   if (reset)
      q1 <= 1'b0;
   else
      q1 <= d1;

always @(posedge clk2 or posedge reset)
   if (reset)
      q2 <= 1'b0;
   else
      q2 <= d2;

always @(posedge clk3 or posedge reset)
   if (reset)
      q3 <= 1'b0;
   else
      q3 <= d3;

always @(posedge clk4 or posedge reset)
   if (reset)
      q4 <= 1'b0;
   else
      q4 <= d4;

always @(posedge clk5 or posedge reset)
   if (reset)
      q5 <= 1'b0;
   else
      q5 <= d5;

always @(posedge clk6 or posedge reset)
   if (reset)
      q6 <= 1'b0;
   else
      q6 <= d6;

always @(posedge clk7 or posedge reset)
   if (reset)
      q7 <= 1'b0;
   else
      q7 <= d7;

always @(posedge clk8 or posedge reset)
   if (reset)
      q8 <= 1'b0;
   else
      q8 <= d8;
```

```
always @(posedge clk9 or posedge reset)
   if (reset)
      q9 <= 1'b0;
   else
      q9 <= d9;

always @(posedge clk10 or posedge reset)
   if (reset)
      q10 <= 1'b0;
   else
      q10 <= d10;

always @(posedge clk11 or posedge reset)
   if (reset)
      q11 <= 1'b0;
   else
      q11 <= d11;

always @(posedge clk12 or posedge reset)
   if (reset)
      q12 <= 1'b0;
   else
      q12 <= d12

always @(posedge clk13 or posedge reset)
   if (reset)
      q13 <= 1'b0;
   else
      q13 <= d13;

always @(posedge clk14 or posedge reset)
   if (reset)
      q14 <= 1'b0;
   else
      q14 <= d14;

always @(posedge clk15 or posedge reset)
   if (reset)
      q15 <= 1'b0;
   else
      q15 <= d15;

always @(posedge clk16 or posedge reset)
   if (reset)
      q16 <= 1'b0;
   else
      q16 <= d16;
```

```
always @(posedge clk17 or posedge reset)
   if (reset)
      q17 <= 1'b0;
   else
      q17 <= d17;

always @(posedge clk18 or posedge reset)
   if (reset)
      q18 <= 1'b0;
   else
      q18 <= d18;

always @(posedge clk19 or posedge reset)
   if (reset)
      q19 <= 1'b0;
   else|
      q19 <= d19;

always @(posedge clk20 or posedge reset)
   if (reset)
      q20 <= 1'b0;
   else
      q20 <= d20;

endmodule
```

VHDL Syntax and Example

attribute syn_global_buffers of *object* **:** *object_type* **is <***maximum***>;**

For example:

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

entity top is
   port(
      clk   : in std_logic_vector(19 downto 0);
      d     : in std_logic_vector(19 downto 0);
      q     : out std_logic_vector(19 downto 0);
      reset : in std_logic
       );
end top;

architecture behave of top is
attribute syn_global_buffers : integer;
attribute syn_global_buffers of behave : architecture is 10;
```

```
begin
process (clk, reset)

   begin
   for i in 0 to 19 loop
      if (reset = '1') then
         q(i) <= '0';
      elsif clk(i) = '1' and clk(i)' event then
         q(i) <= d(i);
      end if;
   end loop;

end process;

end behave;
```

syn_hier

Attribute. Allows you to control the amount of hierarchical transformation that occurs across boundaries on module or component instances during optimization.

During synthesis, the Synplify tool dissolves as much hierarchy as possible to allow efficient optimization of logic across hierarchical boundaries while maintaining optimal run times. The tool then rebuilds the hierarchy as close as possible to the original source, in an effort to preserve the topology of the design. However, you can use the syn_hier attribute to address specific needs you might have to maintain design hierarchy during optimization. This attribute allows you manual control over flattening for instances, modules or architectures in the design.

The following table shows the values you can use for syn_hier. For additional information about using this attribute, see Controlling Hierarchy Flattening, on page 5-10 and Preserving Hierarchy, on page 5-12 in the *Synplify User Guide*.

soft (default)	The synthesis tool determines the best optimization across hierarchical boundaries. This attribute affects only the design unit in which it is specified.
firm	Preserves the interface of the design unit and allows for cell packing across the boundary. This attribute affects only the design unit in which it is specified.
hard	(*Actel, except 500K and PA,* same as firm). Preserves the interface of the design unit with no exceptions. This attribute affects only the specified design units.
remove	Removes the level of hierarchy for the design unit in which it is specified. The hierarchy at lower levels is unaffected. This only affects synthesis optimization. The hierarchy is reconstructed in the netlist and in Technology view schematics.
macro	(*Actel except 500K and PA*. Preserves the interface and contents of the design with no exceptions.
flatten	Flattens the hierarchy of all levels below, but not the one in which it is specified. This only affects synthesis optimization. The hierarchy is reconstructed in the netlist and in Technology view schematics. To create a flattened netlist, set syn_netlist_hierarchy to false. You can also use flatten in combination with other syn_hier values, as described in Combinining flatten with Other Values, on page 8-35.

Combinining flatten with Other Values

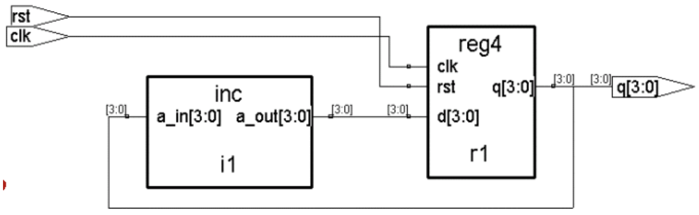
You can combine flatten with other syn_hier values as shown below:

flatten,soft	Same as flatten.
flatten,firm	Flattens all lower levels of the design but preserves the interface of the design unit in which it is specified. This option also allows optimization of cell packing across the boundary.
flatten,remove	Flattens all lower levels of the design, including the one on which it is specified.

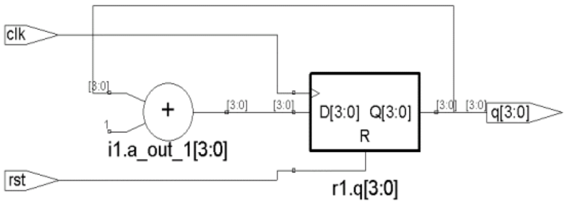
If you use flatten in combination with another option, the tool flattens as directed until encountering another syn_hier attribute at a lower level. The lower level syn_hier attribute then takes precedence over the higher level one.

Example of syn_hier

Here is an example of two versions of a design: one with syn_hier set on modules inc and reg4; the other shows what happens to those modules with the automatic flattening that occurs during synthesis.



With syn_hier="hard"



Automatic flattening, counter inferred

.sdc File Syntax and Example

define_attribute { *object* **} syn_hier {** *value* **}**

where *object* can be module declarations or architecture names. Check the attribute values to determine where to attach the attribute. The syntax is

```
define_attribute {v:fifo} syn_hier {hard}
```

Verilog Syntax and Examples

object **/* synthesis syn_hier = "***value***" */ ;**

where *object* can be a module declaration. Check the attribute values to determine where to attach the attribute. The Verilog syntax is

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

// Other code
```

This next example demonstrates the flatten,remove value, which flattens the current level of the hierarchy and all levels below it (unless another syn_hier attribute is found at a lower level).

```
module top1 (Q, CLK, RST, LD, CE, D)
               /* synthesis syn_hier = "flatten,remove" */;

// Other code
```

VHDL Syntax and Examples

attribute syn_hier of *object* **: architecture is "***value***" ;**

where *object* can be an architecture name. Check the attribute values to determine the level at which to attach the attribute.

```
architecture struct of cpu is

attribute syn_hier : string;
attribute syn_hier of struct: architecture is "firm";

-- Other code
```

This next example demonstrates the flatten,remove value for this attribute. This value flattens the current level of the hierarchy and all levels below it (unless another syn_hier attribute is found at a lower level).

```
architecture struct of cpu is

attribute syn_hier : string;
attribute syn_hier of struct: architecture is "flatten,remove";

-- Other code
```

syn_maxfan

Attribute; Overrides the default (global) fanout guide for an individual input port, net or register output. You set the default Fanout Guide for a design through the Device panel on the Options for Implementation dialog box or with the set_option -fanout_limit command or -fanout_guide (for Virtex) in the project file. Use the syn_maxfan attribute to specify a different (local) value for individual I/Os.

Generally, syn_maxfan and the default fanout guide are suggested guidelines only, but in certain cases they function as hard limits. When they are guidelines, the synthesis tool takes them into account but does not always respect them absolutely. If they impose constraints that interfere with optimization, they are not respected. The attribute value functions as a hard limit when it is attached to nets, ports, primitive instances, and registers in Actel ProAsic designs. See Setting Fanout Limits, on page 5-7 of the *Synplify User Guide,* for details.

You can apply the syn_maxfan attribute to a register, instance, port, or net. For Actel ProAsic designs, you can also apply it to a module or entity. If you apply it to a net, the synthesis tool creates a KEEPBUF component and attaches the attribute to it, because the net itself might be optimized away during synthesis. If you attach the attribute to a lower-level module or entity that is subsequently optimized during synthesis, the synthesis tool moves the syn_maxfan attribute up to the next higher level. If you do not want syn_maxfan moved up during optimization, set the syn_hier attribute for the entity or module to hard. This prevents the module or entity from being flattened when the design is optimized.

The syn_maxfan attribute is often used along with the syn_noclockbuf attribute on an input port that you do not want buffered. There are a limited number of clock buffers in a design, so if you want to save these special clock buffer resources for other clock inputs, put the syn_noclockbuf attribute on the clock signal. If timing for that clock signal is not critical, you can turn off buffering completely, to save area. To turn off buffering, set the maximum fanout to a very high number; for example, 1000.

Similarly, you use syn_maxfan with the syn_replicate attribute in certain technologies to control replication.

.sdc File Syntax and Example

define_attribute { *object* **} syn_maxfan {** *integer* **}**

The following example limits the fanout for the signal clk to 200:

```
define_attribute {clk} syn_maxfan {200}
```

Verilog Syntax and Example

object **/* synthesis syn_maxfan = "***value***" */ ;**

For example:

```
module test (registered_data_out, clock, data_in);
output [31:0] registered_data_out;
input clock;
input [31:0] data_in /* synthesis syn_maxfan=1000 */;
reg [31:0] registered_data_out /* synthesis syn_maxfan=1000 */;

// Other code
```

VHDL Syntax and Example

attribute syn_maxfan of *object* **:** *object_type* **is "***value***" ;**

For example:

```
entity test is
   port(clock : in bit;
      data_in : in bit_vector(31 downto 0);
      registered_data_out: out bit_vector(31 downto 0)
      )

attribute syn_maxfan : integer;
attribute syn_maxfan of data_in : signal is 1000;

-- Other code
```

syn_netlist_hierarchy

Attribute. A global attribute that controls the generation of hierarchy in the EDIF or VQM output (result file) when you assign it to the top-level module of your design. The default (true) is to allow hierarchy generation.

.sdc File Syntax and Example

define_global_attribute syn_netlist_hierarchy { 0 | 1 }

For example:

```
define_global_attribute syn_netlist_hierarchy {0}
```

Verilog Syntax and Example

object **/* synthesis syn_netlist_hierarchy = 0 | 1 */ ;**

where *object* can be a top-level module declaration. For example:

```
module top (clk, qout, a, b)
               /*synthesis syn_netlist_hierarchy=0 */;

// Other code
```

VHDL Syntax and Example

attribute syn_netlist_hierarchy of *object* **:** *object_type* **is true | false ;**

where *object* can be a top-level architecture name. The data type is Boolean. For example:

```
architecture top of top is

attribute syn_netlist_hierarchy : boolean;
attribute syn_netlist_hierarchy of top : architecture is false;

-- Other code
```

syn_noarrayports

Attribute. Specifies that the ports of a design unit be treated as individual signals (scalars), not as buses (arrays) in the EDIF file.

.sdc File Syntax and Example

define_global_attribute syn_noarrayports { 0 | 1 }

For example:

```
define_global_attribute syn_noarrayports {1}
```

Verilog Syntax and Example

object **/* synthesis syn_noarrayports = 0 | 1 */ ;**

Where *object* can be a module declarations. For example:

```
module adder8(cout, sum, a, b, cin)
               /* synthesis syn_noarrayports = 1 */;

// Other code
```

VHDL Syntax and Example

attribute syn_noarrayports of *object* **:** *object_type* **is true | false ;**

where *object* is an architecture name. Data type is Boolean.

In this example, the ports of adder8 are treated as scalars during synthesis.

```
architecture adder8 of adder8 is

attribute syn_noarrayports : boolean;
attribute syn_noarrayports of adder8 : architecture is true;

-- Other code
```

syn_noclockbuf

Attribute. The Synplify synthesis tool uses clock buffer resources, if they exist in the target module, and puts them on the highest fanout clock nets. You can turn off automatic clock buffer usage by using the syn_noclockbuf attribute. For example, you can put a clock buffer on a lower fanout clock that has a higher frequency and a tighter timing constraint.

You can turn off automatic clock buffering for entire modules or nets, or just for specific input ports. The value is Boolean with 1 or true turning off automatic clock buffering.

.sdc File Syntax and Example

define_attribute { *clock_port* **} syn_noclockbuf { 1 | 0 }**

define_global_attribute syn_noclockbuf { 1 | 0 }

For example:

```
define_attribute {clk} syn_noclockbuf {1}

define_global_attribute syn_noclockbuf {1}
```

Verilog Syntax and Examples

object **/* synthesis syn_noclockbuf = 1 | 0 */ ;**

Here is an example of turning off automatic clock buffering for an entire module.

```
module my_design(out, in, clk_in)
               /* synthesis syn_noclockbuf = 1 */;
```

This next example demonstrates turning off automatic clock buffering for a specific input port:

```
module my_design(out, in, clk_in);
output out;
input in;
input clk_in  /* synthesis syn_noclockbuf = 1 */;

// Other code
```

VHDL Syntax and Examples

attribute syn_noclockbuf of *object* **:** *object_type* **is true | false ;**

Where *object* can be an architecture or an input (clock) port.

This example turns off automatic clock buffering for an entire architecture using the syn_noclockbuf attribute from the Synplicity attributes package.

```
library ieee, synplify;
use ieee.std_logic_1164.all;
entity simpledff is
   port (q : out std_logic_vector(7 downto 0);
         d : in std_logic_vector(7 downto 0);
         clk : in std_logic);
end simpledff;

architecture behavior of simpledff is
-- Turn off automatic clock buffers for this architecture, by
-- setting the attribute on the architecture itself.

attribute syn_noclockbuf : boolean;
attribute syn_noclockbuf of behavior : architecture is true;

-- Coding for the behavior of this architecture
```

The following example demonstrates turning off automatic clock buffering for an individual signal:

```
library ieee, synplify;
use ieee.std_logic_1164.all;
entity simpledff is
   port (q : out std_logic_vector(7 downto 0);
         d : in std_logic_vector(7 downto 0);
         clk : in std_logic);

-- Turn off automatic clock buffering on clk
attribute syn_noclockbuf : boolean;
attribute syn_noclockbuf of clk : signal is true;
end simpledff;

architecture behavior of simpledff is

-- Coding for the behavior of this architecture
```

syn_preserve_sr_priority

Attribute; Actel ACT1 and 40MX. Globally implements hardware that forces set/reset flip-flops to honor the priority for the set or reset, as coded in the design. This attribute can increase the area of your design.

Sequential Actel components do not have a well defined behavior when both the set and reset signals are active at the same time, but you can have the Synplify synthesis tool automatically add hardware to force the priority that you have coded in the source code of your design.

.sdc File Syntax and Example

define_global_attribute syn_preserve_sr_priority { 1 }

For example:

```
define_global_attribute syn_preserve_sr_priority {1}
```

Most language models for a set/reset level-sensitive latch or D flip-flop define the behavior for this condition. Using this attribute sets the priority of set/reset as it is specified in your HDL source code. The following Verilog code implies that reset has priority over set if both are active:

```
if (reset)
   q=0;
else if (set)
   q=1;
else
   q=d;
end if;
```

Verilog Syntax and Example

object **/* synthesis syn_preserve_sr_priority = 1 */ ;**

The following example sets the syn_preserve_sr_priority attribute on the latch3 module. The value 1 indicates that the attribute is on.

```
module latch3(q,data,set,reset,clk)
               /* synthesis syn_preserve_sr_priority = 1 */;
output q;
input data, clk, set, reset ;
reg q;
```

```
always @(clk or data or set or reset)
begin
   if (reset)
      q = 0;
   else if (set)
      q = 1;
   else if (clk)
      q = data;
   end
endmodule
```

VHDL Syntax and Example

attribute syn_preserve_sr_priority of *object* **: architecture is true ;**

Where *object* is the entity.

Here is an example:

```
library ieee;
use ieee.std_logic_1164.all;
entity dff1 is
port (data, clk, reset, set : in std_logic;
      qrs: out std_logic);

end dff1;

architecture async_set_reset of dff1 is

--  Set the attribute to "true" for async_set_reset architecture
attribute syn_preserve_sr_priority : boolean;
attribute syn_preserve_sr_priority of async_set_reset :
              architecture is true;
begin
setreset: process (clk, reset, set)

begin
   if reset = '1' then
      qrs <= '0';
   elsif set = '1' then
      qrs <= '1';
   elsif rising_edge(clk)then
      qrs <= data;
   end if;
end process setreset;
end async_set_reset;
```

syn_radhardlevel

Attribute; Actel (RT, RH and RD radhard devices). Specifies the radiation-resistant design technique to use on an object. This attribute can be applied to a module/architecture or a register output signal (inferred register in VHDL), and is used in conjunction with the Actel macro files supplied with the software.

Some techniques are not available or appropriate for all Actel families. Contact Actel technical support for more information The design technique must be one that is valid for the project. You can apply syn_radhardlevel globally to the top-level module or architecture of your design and then selectively override it on different portions of the design. You can also control the design technique to apply on individual registers. See Working with Radhard Designs, on page 6-6 in the *User Guide* for details.

Values for syn_radhardlevel are as follows:

Value	Description
none	Standard design techniques are used.
cc	Combinational cells with feedback are used to implement storage rather than flip-flop or latch primitives.
tmr	Triple module redundancy or triple voting is used to implement registers. Each register is implemented by three flip-flops or latches that “vote” to determine the state of the register.
tmr_cc	Triple module redundancy is used where each voting register is composed of combinational cells with feedback rather than flip-flop or latch primitives.

.sdc File Syntax and Example

define_attribute { *object* } syn_radhardlevel { none | cc | tmr | tmr_cc }

where *object* is a register. For example:

```
define_attribute {dataout[3:0]} syn_radhardlevel {cc}
```

Verilog Syntax and Example

object **/* synthesis syn_radhardlevel ="none | cc | tmr | tmr_cc" */ ;**

Where *object* is a module or a register output signal. For example:

```
module top (clk, dataout, a, b);
input clk;
input a;
input b;
output dataout [3:0];
reg [3:0] dataout /* synthesis syn_radhardlevel="cc" */;

// Other code
```

VHDL Syntax and Example

attribute syn_radhardlevel of *object* **:** *object_type* **is "none | cc | tmr | tmr_cc" ;**

Where *object* is a module or a register output signal, *object_type* is architecture or signal. For example:

```
library synplify;
architecture top of top is
attribute syn_radhardlevel : string;
attribute syn_radhardlevel of top: architecture is "cc";

-- Other code
```

syn_ramstyle

Attribute. The syn_ramstyle attribute specifies the implementation to use for an inferred RAM. You apply syn_ramstyle globally, to a module, or to a RAM instance. To turn off RAM inference, set its value to registers.

The available values for syn_ramstyle vary with the technology being used. The following table lists all the valid syn_ramstyle values, some of which only apply to certain technologies.

Table 8-5: syn_ramstyle Values

registers	Specifies that an inferred RAM be mapped to registers (flip-flops and logic), not technology-specific RAM resources. If your RAM resources are limited, for whatever reason, you can map additional RAMs to registers instead of RAM resources using this attribute.
block_ram	Specifies that the inferred RAM be mapped to the appropriate vendor-specific memory. It uses the dedicated memory resources in the FPGA.
select_ram	Implements RAMs using the distributed RAM resources in the CLBs. For distributed RAMs, the write operation must be synchronous and the read operation asynchronous. For dual port RAMs, the synthesis tool adds glue logic to direct the RAM input to go to the output when the read address is the same as the write address.

.sdc File Syntax and Example

define_attribute { *signal_name* **[** *bit_range* **] } syn_ramstyle {** *string* **}**

If you edit a constraint file to apply syn_ramstyle, be sure to include the range of the signal with the signal name. For example:

```
define_attribute {mem[7:0]} syn_ramstyle {registers};
```

Verilog Syntax and Example

object **/* synthesis syn_ramstyle = "***string***" */ ;**

where *object* can be register definition (reg) signals. The data type is string.

Here is an example:

```
module ram4 (datain,dataout,clk);
output dataout[31:0];
input clk;
input datain[31:0];
reg [7:0] dataout[31:0] /* synthesis syn_ramstyle="M4K" */;

// Other code
```

VHDL Syntax and Example

attribute syn_ramstyle of *object* **:** *object_type* **is "***string***" ;**

where *object* can be a signal that defines a RAM or a label of a component instance. Data type is string.

Signal Defining the RAM

```
library ieee;
use ieee.std_logic_1164.all;
entity ram4 is
   port (d : in std_logic_vector(7 downto 0);
         addr : in  std_logic_vector(2 downto 0);
         we : in  std_logic;
         clk : in  std_logic;
         ram_out : out std_logic_vector(7 downto 0));
end ram4;

library synplify;
architecture rtl of ram4 is
type mem_type is array (127 downto 0) of std_logic_vector (7
   downto 0);
signal mem : mem_type;
-- mem is the signal that defines the RAM

attribute syn_ramstyle : string;
attribute syn_ramstyle of mem : signal is "block_ram";

-- Other code
```

syn_reference_clock

Attribute. The syn_reference_clock attribute lets you specify a clock frequency other than that implied by the signal on the clock pin of the register. For example, when flip-flops have an enable with a regular pattern, such as every second clock cycle, use syn_reference_clock to have timing analysis treat the flip-flops as if they were connected to a clock at half the frequency.

To use syn_reference_clock, define a new clock, then apply its name to the registers you want to change.

.sdc File Syntax and Examples

define_attribute { *register* **} syn_reference_clock {** *clk_name* **}**

For example:

```
define_attribute {myreg[31:0]} syn_reference_clock {sloClock}
```

You can also use syn_reference_clock to constrain multiple-cycle paths through the enable signal. The following example shows how you can apply the constraint to all registers with the enable signal en40:

```
define_attribute {find -reg -enable en40} syn_reference_clock
             {clk2}
```

Note: You apply syn_reference_clock only in an `.sdc` file; you cannot use it in source code.

syn_replicate

Attribute; Controls replication. The Synplify synthesis tool automatically replicates registers during the following optimization processes: fixing fanouts, packing I/Os, and improving quality of results.

You can use this attribute to disable replication either globally or on a per-register basis. When you disable replication globally, it disables I/O packing and quality-of-results optimizations. The synthesis tool uses only buffering to meet maximum fanout guidelines.

To disable I/O packing on specific registers , set the attribute to 0. Similarly, you can use it on a register between clock boundaries to prevent replication. For example, the synthesis tool replicates 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. By setting the attribute for the register to 0, you can disable replication.

.sdc File Syntax and Example

define_global_attribute syn_replicate = { 1 | 0 }

For example, to disable all replication in the design:

```
define_global_attribute syn_replicate {0}
```

Verilog Syntax and Example

object **/* synthesis syn_replicate = 1 | 0 */;**

For example:

```
module norep (Reset, Clk, Drive, OK, ADPad, IPad, ADOut);
input Reset, Clk, Drive, OK;
input [31:0] ADOut;
inout [31:0] ADPad;
output [31:0] IPad;

reg [31:0] IPad;

reg DriveA /* synthesis syn_replicate = 0 */;

assign ADPad = DriveA ? ADOut : 32'bz;
always @(posedge Clk or negedge Reset)
   if (!Reset)
      begin
         DriveA <= 0;
```

```
            IPad   <= 0;
         end
      else
         begin
            DriveA <= Drive & OK;
            IPad   <= ADPad;
         end
   endmodule
```

VHDL Syntax and Example

attribute syn_replicate of *object* **:** *object_type* **is true | false ;**

For example:

```
library IEEE;
use ieee.std_logic_1164.all;
entity norep is port (
            Reset : in std_logic;
            Clk : in std_logic;
            Drive : in std_logic;
            OK : in std_logic;
            ADPad : inout std_logic_vector (31 downto 0);
            IPad : out std_logic_vector (31 downto 0);
            ADOut : in std_logic_vector (31 downto 0));
end norep;

architecture archnorep of norep is
signal DriveA : std_logic;
attribute syn_replicate : boolean;
attribute syn_replicate of DriveA : signal is false;

begin
ADPad <= ADOut when DriveA='1' else (others => 'Z');
process (Clk, Reset)
   begin
   if Reset='0' then
      DriveA <= '0';
      IPad <= (others => '0');
   elsif rising_edge(clk) then
      DriveA <= Drive and OK;
      IPad <= ADPad;
   end if;
end process;
end archnorep;
```

syn_useenables

Attribute; Generates register instances with clock enable pins. By default, the Synplify synthesis tool tries to use the enable pin. You set syn_useenables to 0 to turn off clock-enable extraction.

.sdc File Syntax and Example

The constraint file syntax for the attribute is:

define attribute { *register* **|** *signal* **} syn_useenables { 0 | 1 }**

For example:

```
define_attribute {q[3:0]} syn_useenables {0}
```

Verilog Syntax and Example

object **/* synthesis syn_useenables = 0 | 1 */ ;**

where *object* can be a component register or signal. Data type is Boolean.

For example:

```
reg [3:0] q /* synthesis syn_useenables = 0 */;
always @(posedge clk)
if (enable)
   q <= d;
```

VHDL Syntax and Example

attribute syn_useenables of *object* **:** *object_type* **is true | false ;**

where *object* can be labels of component registers or signals.

For example:

```
signal q_int : std_logic_vector(3 downto 0);
attribute syn_useenables of q_int : signal is false;
...
begin
...
```

```
process(clk)
begin
   if (clk'event and clk = '1') then
      if (enable = '1') then
         q_int <= d;
      end if;
   end if;
end process;
```

Directives

The individual synthesis directives are described in this section, in alphabetical order. Each directive description includes the following:

- Technology support
- Directive definition
- File syntax and examples for Verilog and VHDL source code

For an alphabetical summary of the directives, see Directive Summary (Alphabetical), on page 8-4. For a summary of the directives by vendor, refer to the appropriate vendor chapter or see Attribute and Directive Summary, by Vendor, on page 8-3.

For a general information on specifying directives, see Specifying Directives and Attributes in HDL, on page 8-11.

black_box_pad_pin

Directive. Specifies pins on a user-defined black-box component, as I/O pads that are visible to the environment outside of the black box. If there is more than one port that is an I/O pad, list the ports inside double-quotes ("), separated by commas (,), and without enclosed spaces. See the Verilog and VHDL syntax and examples, below.

You can use this directive with any of the following black-box directives:

- black_box_tri_pins
- syn_black_box
- syn_isclock
- syn_tco<n>
- syn_tpd<n>
- syn_tsu<n>

To instantiate an I/O from your programmable logic vendor, you usually do not need to define a black box or this directive. The Synplify synthesis tool provides predefined black boxes for vendor I/Os. For more information, refer to your vendor section under FPGA and CPLD Support.

Verilog Syntax and Example

object **/* synthesis syn_black_box black_box_pad_pin = "***port_list***" */ ;**

where *port_list* is a spaceless, comma-separated list of the names of the ports on black boxes that are I/O pads.

For example:

```
module BBDLHS(D,E,GIN,GOUT,PAD,Q)
   /* synthesis syn_black_box black_box_pad_pin="GIN[2:0],Q" */;
```

VHDL Syntax and Example

attribute black_box_pad_pin of *object* **:** *object_type* **is "***port_list***" ;**

where *object* can be an architecture or component declaration of a black box. Data type is string; *port_list* is a spaceless, comma-separated list of the black-box port names that are I/O pads. For example:

```
library ieee;
use ieee.std_logic_1164.all;
package components is

component BBDLHS
   port(
      D: in std_logic;
      E: in std_logic;
      GIN : in std_logic_vector(2 downto 0);
      Q : out std_logic
      );
end component;

attribute syn_black_box : boolean;
attribute syn_black_box of BBDLHS : component is true;
attribute black_box_pad_pin : string;
attribute black_box_pad_pin of BBDLHS : component is "GIN(2:0),Q";
end components;
```

black_box_tri_pins

Directive. Specifies that an output port, on a component defined as a black box, is a tristate. This directive is used to eliminate multiple driver errors when the output of a black box has more than one driver. A multiple driver error is issued unless you use this directive to specify that the outputs are tristates. If there is more than one port that is a tristate, list the ports within double-quotes ("), separated by commas (,), and without enclosed spaces. See the Verilog and VHDL syntax and examples, below.

You can use this directive with any of the following black-box directives:

- black_box_pad_pin
- syn_black_box
- syn_isclock
- syn_tco<n>
- syn_tpd<n>
- syn_tsu<n>

Verilog Syntax and Examples

object **/* synthesis syn_black_box black_box_tri_pins = "***port_list***" */ ;**

where *port_list* is a spaceless, comma-separated list of multiple pins.

Here is an example with a single port name:

```
module BBDLHS(D,E,GIN,GOUT,PAD,Q)
         /* synthesis syn_black_box black_box_tri_pins="PAD" */;
```

Here is an example with a list of multiple pins:

```
module bb1(D,E,tri1,tri2,tri3,Q)
/* synthesis syn_black_box black_box_tri_pins="tri1,tri2,tri3" */;
```

For a bus, you specify the port name followed by all the bits on the bus:

```
module bb1(D,bus1,E,GIN,GOUT,Q)
   /* synthesis syn_black_box black_box_tri_pins="bus1[7:0]" */;
```

VHDL Syntax and Examples

attribute black_box_tri_pins of *object* **:** *object_type* **is "***port_list***" ;**

where *object* can be a component declaration or architecture. Data type is string, and *port_list* is a spaceless, comma-separated list of the tristate output port names. For example:

```
library ieee;
use ieee.std_logic_1164.all;
package components is

component BBDLHS
port(
   D: in std_logic;
   E: in std_logic;
   GIN : in std_logic;
   GOUT : in std_logic;
   PAD : inout std_logic;
   Q: out std_logic
   );
end component;

attribute syn_black_box : boolean;
attribute syn_black_box of BBDLHS : component is true;
attribute black_box_tri_pins : string;
attribute black_box_tri_pins of BBDLHS : component is "PAD";

end components;
```

Multiple pins on the same component can be specified as a list:

```
attribute black_box_tri_pins of bb1 : component is
              "tri,tri2,tri3";
```

To apply this directive to a port that is a bus, you specify all the bits on the bus:

```
attribute black_box_tri_pins of bb1 : component is "bus1[7:0]";
```

full_case

Directive. For Verilog designs only. When used with a case, casex, or casez statement, this directive indicates that all possible values have been given, and that no additional hardware is needed to preserve signal values.

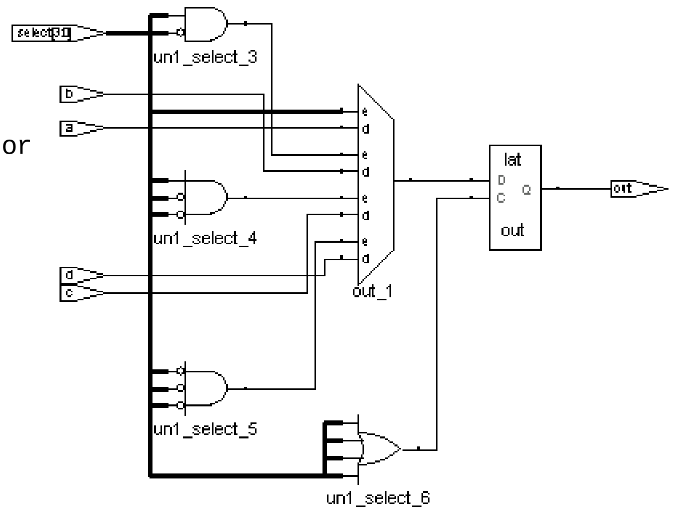
Verilog Syntax and Example

object **/* synthesis full_case */**

where *object* can be case, casex, or casez statement declarations.

The following casez statement creates a 4-input multiplexer with a pre-decoded select bus (a decoded select bus has exactly one bit enabled at a time):

```
module muxnew1 (out, a, b, c, d, select);
output out;
input a, b, c, d;
input [3:0] select;
reg out;

always @(select or a or b or
c or d)

begin
  casez (select)
    4'b???1: out = a;
    4'b??1?: out = b;
    4'b?1??: out = c;
    4'b1???: out = d;
  endcase
end
endmodule
```



This code does not specify what to do if the select bus has all zeros. If the select bus is being driven from outside the current module, the current module has no information about the legal values of select, and the synthesis tool must preserve the value of the output out when all bits of select are zero. Preserving the value of out requires the tool to add extraneous level-sensitive latches if out is not assigned elsewhere through every path of the always block. A warning message like the following is issued:

```
“Latch generated from always block for signal out, probably missing
assignment in branch of if or case.”
```

If you add the full_case directive, it instructs the synthesis tool not to preserve the value of out when all bits of select are zero.

```
module muxnew3 (out, a, b, c, d, select);
output out;
input a, b, c, d;
input [3:0] select;
reg out;

always @(select or a or b or c or d)

begin
   casez (select) /* synthesis full_case */
      4'b???1: out = a;
      4'b??1?: out = b;
      4'b?1??: out = c;
      4'b1???: out = d;
   endcase
end
endmodule
```

If the select bus is decoded in the same module as the case statement, the synthesis tool automatically determines that all possible values are specified, so the full_case directive is unnecessary.

Assigned Default and full_case

As an alternative to full_case, you can assign a default in the case statement. The default is assigned a value of 'bx (a 'bx in an assignment is treated as a “don't care”). The software assigns the default at each pass through the casez statement in which the select bus does not match one of the explicitly given values; this ensures that the value of out is not preserved and no extraneous level-sensitive latches are generated.

The following code shows a default assignment in Verilog:

```
module muxnew2 (out, a, b, c, d, select);
output out;
input a, b, c, d;
input [3:0] select;
reg out;
always @(select or a or b or c or d)

begin
```

```
      casez (select)
         4'b???1: out = a;
         4'b??1?: out = b;
         4'b?1??: out = c;
         4'b1???: out = d;
         default: out = 'bx;
      endcase
   end
   endmodule
```

Both techniques help keep the code concise because you do not need to declare all the conditions of the statement.The following table compares them:

Default Assignment	full_case
Stays within Verilog to get the desired hardware	Have to use a synthesis directive to get the desired hardware
Helps simulation debugging, because you can easily find that the invalid select is assigned a 'bx	Could cause mismatches between pre- and post-synthesis simulation because the simulator does not use full_case

loop_limit

Directive. Specifies a loop iteration limit for for loops in the design when the loop index is not a constant but a variable. The compiler uses the default iteration limit of 1999 when the exit or terminating condition does not compute a constant value, or to avoid infinite loops. The default limit ensures the effective use of runtime and memory resources.

If your design requires a variable loop index or if the number of loops is greater than the default limit, use the loop_limit directive to specify a new limit for the compiler. If you do not, you get a compiler error. You must hard code the limit at the beginning of the loop statement. The limit cannot be an expression. The higher the value you set, the longer the runtime.

Verilog Syntax and Example

beginning of loop statement **/* synthesis loop_limit** *<integer>* ***/**

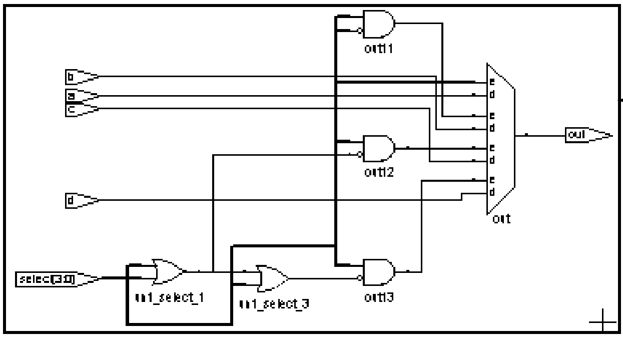
The following is an example where the loop limit is set to 2000:

```
module test(din,dout,clk);
input[1999 : 0] din;
input clk;
output[1999 : 0] dout;
reg[1999 : 0] dout;
integer i;

always @(posedge clk)
begin
   /* synthesis loop_limit 2000 */
   for(i=0;i>=1999;i=i+1) /
   begin
   dout[i] <= din[i];
   end
end

endmodule
```

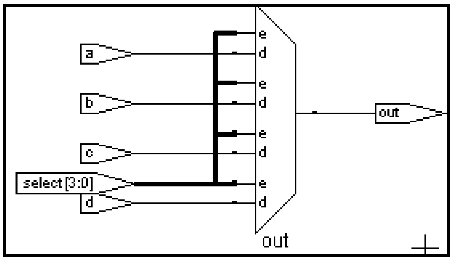
parallel_case

Directive. For Verilog designs only. Forces a parallel-multiplexed structure rather than a priority-encoded structure. This is useful because case statements are defined to work in priority order, executing (only) the first statement with a tag that matches the select value.

If the select bus is driven from outside the current module, the current module has no information about the legal values of select, and the software must create a chain of disabling logic so that a match on a statement tag disables all following statements. However, if you know the legal values of select, you can eliminate extra prioirty-encoding logic with the parallel_case directive. In the following example, the only legal values of select are 4'b1000, 4'b0100, 4'b0010, and 4'b0001, and only one of the tags can be matched at a time. Specify the parallel_case directive so that tag-matching logic can be parallel and independent, instead of chained.



Extra logic for priority encoding (without parallel_case)



Extra logic eliminated with parallel_case

Verilog Syntax and Example

You specify the directive as a comment immediately following the select value of the case statement.

object **/* synthesis parallel_case */**

where *object* is a case, casex or casez statement declaration.

```
module muxnew4 (out, a, b, c, d, select);
output out;
input a, b, c, d;
input [3:0] select;
reg out;

always @(select or a or b or c or d)

begin
   casez (select) /* synthesis parallel_case */
      4'b???1: out = a;
      4'b??1?: out = b;
      4'b?1??: out = c;
      4'b1???: out = d;
      default: out = 'bx;
   endcase
end
endmodule
```

If the select bus is decoded within the same module as the case statement, the parallelism of the tag matching is determined automatically, and the parallel_case directive is unnecessary.

syn_black_box

Directive. Specifies that a module or component is a black box with only its interface defined for synthesis. The contents of a black box cannot be optimized during synthesis. A module can be a black box whether or not it is empty. This directive has an implicit Boolean value of 1 or true. Common uses of syn_black_box include the following:

- Vendor primitives and macros (including I/Os).
- User-designed macros whose functionality is defined in a schematic editor, IP, or another input source in which the place-and-route tool merges design netlists from different sources.

To instantiate vendor I/Os and other vendor macros, you usually do not need to define a black box since the Synplify synthesis tool provides pre-defined black boxes for the vendor macros.

You can use this directive with any of the following black-box attributes:

- black_box_pad_pin
- black_box_tri_pins
- syn_isclock
- syn_tco<n>
- syn_tpd<n>
- syn_tsu<n>

For more information on black boxes, see Instantiating Black Boxes in Verilog, on page 9-49, and Instantiating Black Boxes in VHDL, on page 10-88.

Verilog Syntax and Example

object **/* synthesis syn_black_box */ ;**

where object is a module declaration. For example:

```
module bl_box(out,data,clk) /* synthesis syn_black_box */;

// Other code
```

VHDL Syntax and Example

attribute syn_black_box of *object* **:** *object_type* **is true ;**

where *object* is a component declaration, label of an instantiated component to define as a black box, architecture, or component. Data type is Boolean.

```
library synplify;
architecture top of top is

component ram4
   port (myclk : in bit;
         opcode : in bit_vector(2 downto 0);
         a, b : in bit_vector(7 downto 0);
         rambus : out bit_vector(7 downto 0)
         );
end component;

attribute syn_black_box : boolean;
attribute syn_black_box of ram4: component is true;

-- Other code
```

syn_enum_encoding

Directive. For VHDL designs. Defines how enumerated data types are implemented. The type of implementation affects the performance and device utilization.

If FSM Compiler is enabled, do *not* use this directive to specify the encoding styles of extracted state machines; use the syn_encoding attribute instead. However, if you have enumerated data types and you turn off the FSM Compiler so that no state machines are extracted, the syn_enum_encoding style is implemented in the final circuit. See syn_encoding compared with syn_enum_encoding, on page 8-70 for more information. For step-by-step details about setting codng styles with this attribute see Defining State Machines in VHDL, on page 5-14 of the *Synplify User Guide*.

Values for syn_enum_encoding are as follows:

- default – Automatically assigns an encoding style based on the number of states:

sequential	0-4 enumerated types
onehot	5-24 enumerated types
gray	>24 enumerated types

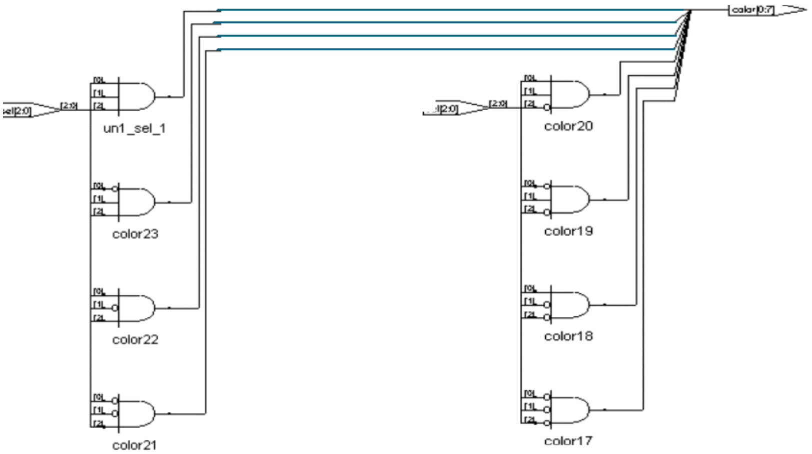
- sequential – More than one bit of the state register can change at a time, but because more than one bit can be hot, the value must be decoded to determine the state. For example: 000, 001, 010, 011, 100
- onehot – Only two bits of the state register change (one goes to 0; one goes to 1) and only one of the state registers is hot (driven by a 1) at a time. For example: 0000, 0001, 0010, 0100, 1000
- gray – Only one bit of the state register changes at a time, but because more than one bit can be hot, the value must be decoded to determine the state. For example: 000, 001, 011, 010, 110
- <*string*> – This can be any value you define. For example: 001, 010, 101. See Example of syn_enum_encoding for User-Defined Encoding, on page 8-71.

A message appears in the log file when you use the syn_enum_encoding directive; for example:

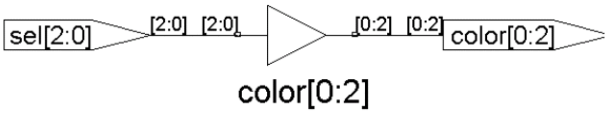
```
@N:"c:\design\..":17:11:17:12|Using onehot encoding for type
mytype (red="10000000")
```

Effect of Encoding Styles

The following figure provides an example of two versions of a design: one with the default encoding style, the other with the syn_enum_encoding directive overriding the default enumerated data types that define a set of eight colors.



syn_enum_encoding = “default”
Based on 8 states, onehot assigned



syn_enum_encoding = “sequential”

In this example, using the default value for syn_enum_encoding, onehot is assigned because there are 8 states in this design. The onehot style implements the output color as 8 bits wide and creates decode logic to convert the input sel to the output. Using sequential for syn_enum_encoding, the logic is reduced to a buffer. The size of output color is 3 bits.

See the following section for the source code used to generate the schematics above.

VHDL Syntax and Examples

attribute syn_enum_encoding of *object* **:** *object_type* **is "***value***" ;**

Where *object* is an enumerated type and *value* is one of the following: default, sequential, onehot or gray.

Here is the code used to generate the second schematic in the previous figure. (The first schematic will be generated instead, if ”sequential” is replaced by ”onehot” as the syn_enum_encoding value.)

```
package testpkg is
type mytype is (red, yellow, blue, green, white,
                violet, indigo, orange);
attribute syn_enum_encoding : string;
attribute syn_enum_encoding of mytype : type is "sequential";
end package testpkg;

library IEEE;
use IEEE.std_logic_1164.all;
use work.testpkg.all;
entity decoder is
port(
      sel : in std_logic_vector(2 downto 0);
      color : out mytype
   );
end decoder;
architecture rtl of decoder is
begin
   process(sel)
   begin
      case sel is
       when "000" => color <= red;
       when "001" => color <= yellow;
       when "010" => color <= blue;
       when "011" => color <= green;
       when "100" => color <= white;
       when "101" => color <= violet;
       when "110" => color <= indigo;
       when others => color <= orange;
      end case;
   end process;
end rtl;
```

syn_encoding compared with syn_enum_encoding

To implement a state machine with a particular encoding style when the FSM Compiler is enabled, use the syn_encoding attribute. The syn_encoding attribute affects how the technology mapper implements state machines in the final netlist. The syn_enum_encoding directive only affects how the compiler interprets the associated enumerated data types. Therefore, the encoding defined by syn_enum_encoding is *not propagated* to the implementation of the state machine.

However, when FSM Compiler is disabled, the value of syn_enum_encoding is implemented in the final circuit.

Example of syn_enum_encoding for User-Defined Encoding

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

entity shift_enum is
   port(clk, rst : bit;
    O : out std_logic_vector(2 downto 0));
end shift_enum;

architecture behave of shift_enum is
   type state_type is (S0, S1, S2);
   attribute syn_enum_encoding: string;
   attribute syn_enum_encoding of state_type : type is "001 010
101";
   signal machine : state_type;
begin
   process (clk, rst)
   begin
    if rst = ’1’ then
      machine <= S0;
    elsif clk = ’1’ and clk’event then
      case machine is
      when S0 => machine <= S1;
      when S1 => machine <= S2;
      when S2 => machine <= S0;
      end case;
    end if;
   end process;

   with machine select
    O <= "001" when S0,
    "010" when S1,
    "101" when S2;
end behave;
```

syn_isclock

Directive. Specifies an input port on a black box as a clock.

Use the syn_isclock directive to specify that an input port on a black box is a clock, even though its name does not correspond to one of the recognized names. Using this directive connects it to a clock buffer if appropriate. The data type is Boolean. You can use this directive with any of the following black-box attributes:

- black_box_pad_pin
- black_box_tri_pins
- syn_black_box
- syn_tco<n>
- syn_tpd<n>
- syn_tsu<n>

Verilog Syntax and Examples

object **/* synthesis syn_isclock = 1 */ ;**

where *object* is an input port on a black box.

```
module ram4 (myclk,out,opcode,a,b) /* synthesis syn_black_box */;
output [7:0] out;
input myclk /* synthesis syn_isclock = 1 */;
input [2:0] opcode;
input [7:0] a, b;

//Other code
```

VHDL Syntax and Examples

attribute syn_isclock of *object***:** *object_type* **is true ;**

where *object* is a black-box input port.

Example:

```
library synplify;

entity ram4 is
   port (myclk : in bit;
         opcode : in bit_vector(2 downto 0);
         a, b : in bit_vector(7 downto 0);
         rambus : out bit_vector(7 downto 0)
         );
attribute syn_isclock : boolean;
attribute syn_isclock of myclk: signal is true;

-- Other code
```

syn_keep

Directive. Keeps the specified net intact during optimization. This directive preserves a net throughout synthesis. When you use this directive, the compiler places a temporary keep buffer primitive on the net as a placeholder throughout synthesis. You can view this buffer in the schematic views (see Effects of Using syn_keep, on page 8-74 for an example). However the buffer is not part of the final netlist, so no extra logic is generated.

As a result of optimization, the compiler might remove some nets, although it maintains ports, registers, and instantiated components. To preserve a net for simulation results or to obtain a different synthesis implementation, use syn_keep on the net.

There are other situations that might require you to use syn_keep. For example, use this directive to prevent duplicate cells from being merged during optimization.

You can also use syn_keep as a placeholder to apply the -through option of the define_multicycle_path or define_false_path timing constraint allowing you to specify a unique path as a multiple-cycle or false path. Apply the constraint to the keep buffer.

Do not apply syn_keep to a reg or signal that will become a sequential object.

Effects of Using syn_keep

The following figure shows the technology view for two versions of a design. One version shows syn_keep set on two registers, out1 and out2, to prevent sharing. The other version is without syn_keep.

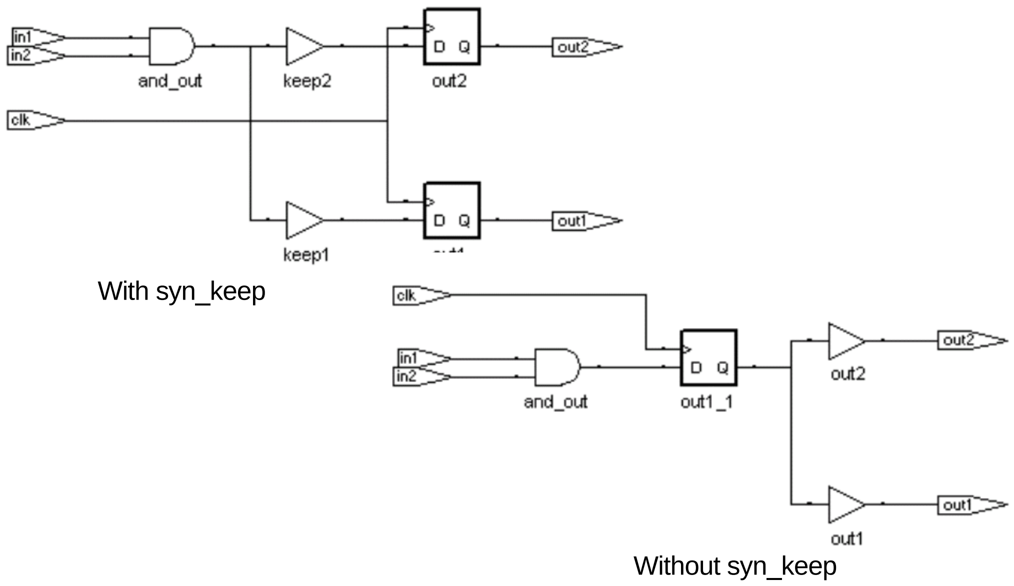


Figure 8-4: syn_keep directive used to prevent sharing

syn_keep was applied at the input of the registers to obtain registered outputs for out1 and out2. Without syn_keep, out1 and out2 optimize to one register. See the following HDL syntax and example sections for the source code used to generate the schematics above.

Usage Compared: syn_keep, syn_preserve, syn_noprune

The following comparsion will help you understand the use of the syn_keep, syn_preserve, and syn_noprune directives:

- syn_keep ensures that 1) a wire is kept during synthesis and 2) that no optimizations cross the wire. This directive is usually used to break unwanted optimizations and to ensure manually created replications. It works only on nets and combinational logic.
- syn_preserve ensures that registers are not optimized away.
- syn_noprune ensures that a black box is not optimized away when its outputs are unused (i.e., when its outputs do not drive any logic).

Verilog Syntax and Example

object **/* synthesis syn_keep = 1 */ ;**

where *object* is a wire or reg declaration. Make sure that there is a space between the object name and the beginning of the comment slash (/).

Here is the source code used to produce the results shown in Effects of Using syn_keep, on page 8-74

```
module example2(out1, out2, clk, in1, in2);
output out1, out2;
input clk;
input in1, in2;
wire and_out;
wire keep1 /* synthesis syn_keep=1 */;
wire keep2 /* synthesis syn_keep=1 */;
reg out1, out2;

assign and_out=in1&in2;
assign keep1=and_out;
assign keep2=and_out;

always @(posedge clk)begin;
   out1<=keep1;
   out2<=keep2;
end
endmodule
```

VHDL Syntax and Example

attribute syn_keep of *object* **:** *object_type* **is true ;**

where *object* is a single or multiple-bit signal.

Here is the source code used to produce the schematics shown in Effects of Using syn_keep, on page 8-74.

```
entity example2 is
   port  (
         in1, in2 : in bit;
         clk : in bit;
         out1, out2 : out bit
         );
end example2;

architecture rt1 of example2 is
attribute syn_keep : boolean;
signal and_out, keep1, keep2: bit;
attribute syn_keep of keep1, keep2 : signal is true;

begin
   and_out <= in1 and in2;
   keep1 <= and_out;
   keep2 <= and_out;

   process(clk)
   begin
      if (clk'event and clk = '1') then
         out1 <= keep1;
         out2 <= keep2;
      end if;
   end process;
end rt1;
```

syn_macro

Directive; QuickLogic only. Prevents instantiated macros from being merged or otherwise optimized away.

Verilog Syntax and Example

object **/* synthesis syn_macro = 1 | 0 */ ;**

For example:

```
multipler_module
   decoder_macro1 (out, in, clk_in) /* synthesis syn_macro=1 */,
   decoder_macro2 (out, in, clk_in) /* synthesis syn_macro=1 */,
   decoder_macro3 (out, in, clk_in) /* synthesis syn_macro=1 */;
```

VHDL Syntax and Example

attribute syn_macro of *object* **:** *object_type* **is true ;**

For example:

```
architecture top of top is

component decoder_macro
   port (clk : in bit;
         opcode : in bit_vector(2 downto 0);
         a : in bit_vector(7 downto 0);
         data0 : out bit_vector(7 downto 0)
         );
end component;

-- The components u1, u2, u3, and u4 are instantiations
-- of the decoder_macro which are specified later the body of
-- this architecture.
attribute syn_macro : boolean;
attribute syn_macro of u1 : label is true;
attribute syn_macro of u2 : label is true;
attribute syn_macro of u3 : label is true;
attribute syn_macro of u4 : label is true;

-- Other code
```

syn_noprune

Directive. Prevents instance optimization for black-box modules (including technology-specific primitives) with unused output ports. During optimization, if a module does not drive any logic, it is removed by the Synplify synthesis tool. If you want to keep the black-box instance of the module in the design, use the syn_noprune directive on the instance or module along with syn_hier set to hard .

Effects of using syn_noprune

The following figure shows the technology view for two versions of a design: one version using syn_noprune on black-box instance U1, one version without syn_noprune.

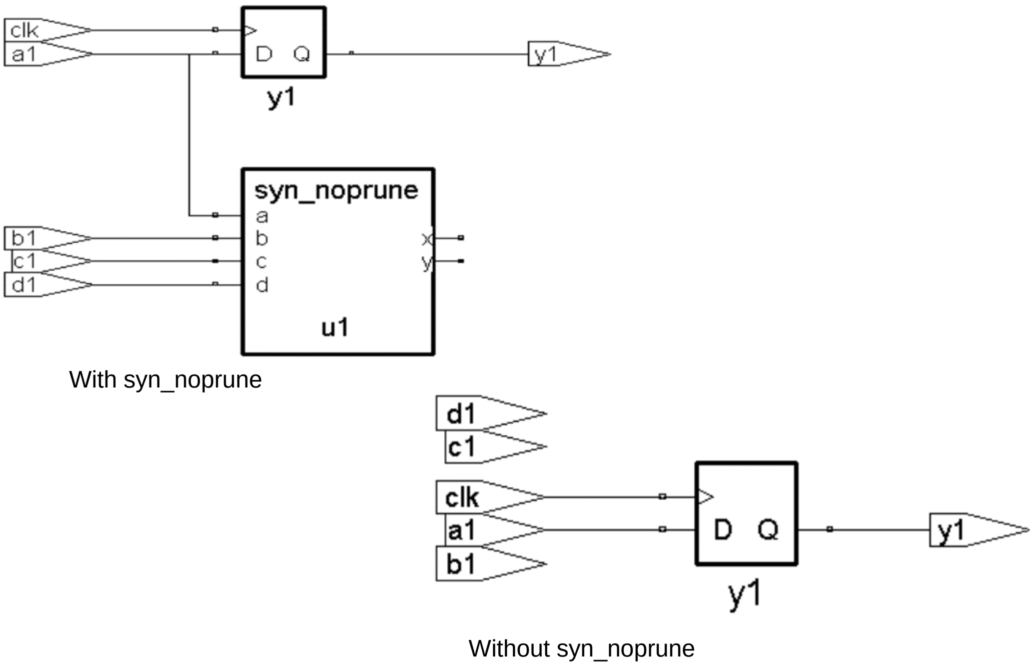


Figure 8-5: syn_noprune directive used to prevent instance optimization

With syn_noprune, module U1 remains in the design. Without syn_noprune the module is optimized away. See the following HDL syntax and example sections for the source code used to generate the schematics above.

Usage Compared: syn_keep, syn_preserve, syn_noprune

The following comparison will help you understand the use of the syn_keep, syn_preserve, and syn_noprune directives:

- syn_keep ensures that 1) a wire is kept during synthesis and 2) that no optimizations cross the wire. This directive is usually used to break unwanted optimizations and to ensure manually created replications. It works only on nets and combinational logic.
- syn_preserve ensures that registers are not optimized away.
- syn_noprune ensures that a black box is not optimized away when its outputs are unused (i.e., when its outputs do not drive any logic).

Verilog Syntax and Examples

object **/* synthesis syn_noprune = 1 */ ;**

where object can be a module declaration or an instance. The data type is Boolean.

The following example shows the source code used for the schematics in Figure 8-5.

```
module top(a1,b1,c1,d1,y1,clk);
output y1;
input a1,b1,c1,d1;
input clk;
wire x2,y2;
reg y1;
syn_noprune u1(a1,b1,c1,d1,x2,y2) /* synthesis syn_noprune=1 */;

always @(posedge clk)
   y1<= a1;

endmodule

module syn_noprune (a,b,c,d,x,y)/* synthesis syn_hier="hard" */;
output x,y;
input a,b,c,d;
endmodule
```

In this example, syn_noprune can be applied in two places, on the module declaration of syn_noprune or in the top-level instantiation. The most common place to use syn_noprune is in the declaration of the module. By placing it here, all instances of the module are protected.

```
module syn_noprune (a,b,c,d,x,y); /* synthesis syn_noprune=1 */;

// Other code
```

Here is an example of using syn_noprune on black-box instances. If your design uses multiple instances with a single module declaration, the synthesis comment must be placed before the comma (,) following the port list for each of the instances.

```
my_design my_design1(out,in,clk_in) /* synthesis syn_noprune=1 */;
my_design my_design2(out,in,clk_in) /* synthesis syn_noprune=1 */;
```

In this example, only the instance my_design2 will be removed if the output port is not mapped.

```
my_design
   my_design1 (out, in, clk_in) /* synthesis syn_noprune=1 */,
   my_design2 (out, in, clk_in),
   my_design3 (out, in, clk_in) /* synthesis syn_noprune=1 */;
```

VHDL Syntax and Example

attribute syn_noprune of *object* **:** *object_type* **is true ;**

where the data type is boolean, and *object* can be an architecture, a component, or a label of an instantiated component. See Architectures, on page 8-82, Component Declaration, on page 8-83, Component Instance, on page 8-83, for details of the objects.

The following example shows the source code used for the schematics in Figure 8-5, on page 8-79.

```
library ieee;
use ieee.std_logic_1164.all;
entity noprune is
   port (
         a, b, c,d  : in std_logic;
         x,y   : out std_logic
         );
end noprune;
```

```
architecture behave of noprune is
attribute syn_hier : string;
attribute syn_hier of behave : architecture is "hard" ;
begin
   x <= a and b;
   y <= c and d;
end behave;

library ieee;
use ieee.std_logic_1164.all;
entity top is
   port (
         a1, b1 : in std_logic;
         c1,d1,clk  : in std_logic;
         y1   :out std_logic
         );
end ;

architecture behave of top is
component noprune
port (
      a, b, c, d : in std_logic;
      x,y    : out std_logic
      );
end component;

signal x2,y2 : std_logic;
attribute syn_noprune : boolean;
attribute syn_noprune of u1 : label is true;
begin
   u1: noprune port map(a1, b1, c1, d1, x2, y2);
   process begin
   wait until (clk = '1') and clk'event ;
      y1 <= a1;
   end process;
end;
```

Architectures

The syn_noprune attribute is normally associated with the names of architectures. Once it is associated, any component instantiation of the architecture (design unit) is protected from being deleted.

```
library synplify;
architecture mydesign of rtl is

attribute syn_noprune : boolean;
attribute syn_noprune of mydesign : architecture is true;

-- Other code
```

Component Declaration

Here is an example:

```
architecture top_arch of top is
component gsr
   port (gsr : in std_logic);
end component;

attribute syn_noprune : boolean;
attribute syn_noprune of gsr: component is true;
```

See Instantiating Black Boxes in VHDL, on page 10-88, for more information.

Component Instance

The syn_noprune attribute works the same on component instances as with a component declaration.

```
architecture top_arch of top is
component gsr
   port (gsr : in bit);
end component;

attribute syn_noprune : boolean;
attribute syn_noprune of u1_gsr: label is true;
```

syn_preserve

Directive. Prevents sequential optimization such as constant propagation, inverter push-through and FSM extraction.

Use syn_preserve to keep registers for simulation purposes or to preserve the logic of registers driven by a constant 1 or 0. To preserve the associated flip-flop and prevent optimization of the signal, you can set syn_preserve on individual registers or on the module/architecture so that the directive is applied to all registers in the module. For example, assume that the input of a flip-flop is always driven to the same value, such as logic 1. The Synplify synthesis tool ties that signal to VCC and removes the flip-flop. Using syn_preserve on the registered signal prevents the removal of the flip-flop.

Another use for this attribute is to preserve a particular state machine. When you enable the symbolic FSM compiler for your entire design and state-machine optimizations are performed, you can use syn_preserve to retain a particular state machine during optimization.

Effects of using syn_preserve

The following figure shows an example where reg1 and out2 are preserved during optimization using syn_preserve.

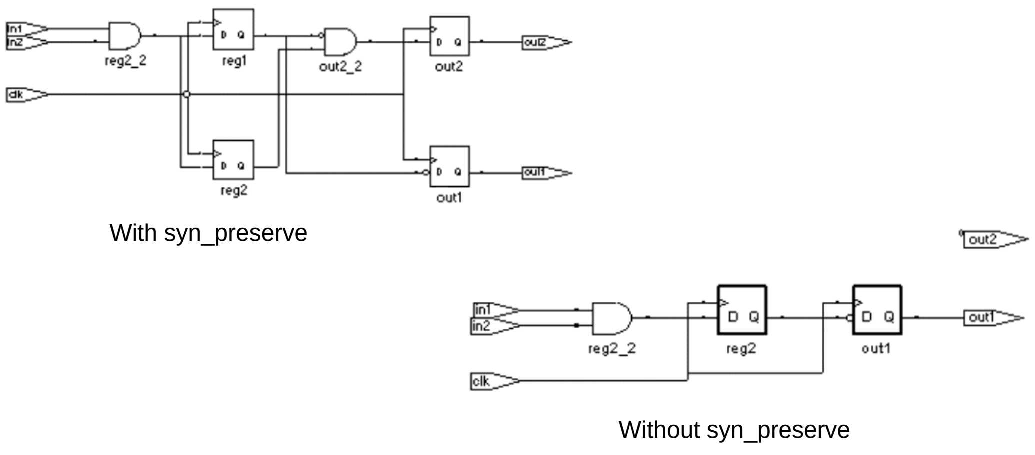


Figure 8-6: How syn_preserve retains registers during optimization

Without syn_preserve, reg1 and reg2 are shared because they are driven by the same source; out2 obtains the result of the AND of reg2 and NOT reg1. This is equivalent to the AND of reg1 and NOT reg1, which is a 0. As this is a constant, register out2 is also removed and output out2 is always 0.

When registers are removed during synthesis, a warning message appears in the log file. For example,

```
@W:...Register bit out2 is always 0, optimizing ...
```

See the HDL syntax and example sections below for the source code used to generate the schematics above.

Usage Compared: syn_keep, syn_preserve, syn_noprune

The following comparsion will help you understand the use of the syn_keep, syn_preserve, and syn_noprune directives:

- syn_keep ensures that 1) a wire is kept during synthesis and 2) that no optimizations cross the wire. This directive is usually used to break unwanted optimizations and to ensure manually created replications. It works only on nets and combinational logic.
- syn_preserve ensures that registers are not optimized away.
- syn_noprune ensures that a black box is not optimized away when its outputs are unused (i.e., when its outputs do not drive any logic).

Verilog Syntax and Examples

object **/* synthesis syn_preserve = 1 */ ;**

where object is a register definition signal or a module.

In the following example, syn_preserve is used on a registered signal so that the flip-flop is not removed and optimization does not occur across the register. This is useful when you are not finished with the design but want to synthesize to find the area utilization.

```
reg foo /* synthesis syn_preserve = 1 */;
```

Following is an example of using syn_preserve for a state register.

```
reg [3:0] curstate /* synthesis syn_preserve = 1 */;
```

The following example shows the source code used for the schematics in Figure 8-6, on page 8-84.

```
module syn_preserve (out1,out2,clk,in1,in2)
              /* synthesis syn_preserve=1 */;

output out1, out2;
input clk;
input in1, in2;

reg out1;
reg out2;
reg reg1;
reg reg2;

always@ (posedge clk)begin
   reg1 <= in1 &in2;
   reg2 <= in1&in2;
   out1 <= !reg1;
   out2 <= !reg1 & reg2;
end
endmodule
```

VHDL Syntax and Examples

attribute syn_preserve of *object* **:** *object_type* **is true ;**

where *object* is an output port or an internal signal that holds the value of a state register or architecture.

```
library ieee, synplify;
use ieee.std_logic_1164.all;
entity simpledff is
   port (q : out std_logic_vector(7 downto 0);
         d : in std_logic_vector(7 downto 0);
         clk : in std_logic);

-- Turn on flip-flop preservation for the q output
attribute syn_preserve : boolean;
attribute syn_preserve of q : signal is true;
end simpledff;

architecture behavior of simpledff is
begin
   process(clk)
   begin
      if rising_edge(clk) then
      -- Notice the continual assignment of "11111111" to q.
            q <= (others => '1');
         end if;
      end process;
   end behavior;
```

In this example, syn_preserve is used on the signal curstate that is later used in a state machine to hold the value of the state register.

```
architecture behavior of mux is
begin
signal curstate : state_type;
attribute syn_preserve of curstate : signal is true;

-- Other code
```

The following example shows the source code for the schematics in Figure 8-6, on page 8-84.

```
library ieee;
use ieee.std_logic_1164.all;
entity mod_preserve is
   port (
         out1     :  out  std_logic;
         out2     :  out std_logic;
         in1,in2,clk :  in std_logic
         );
end mod_preserve;

architecture behave of mod_preserve is
attribute syn_preserve : boolean;
attribute syn_preserve of behave: architecture is true;
   signal reg1 : std_logic;
   signal reg2 : std_logic;

begin process begin
   wait until clk'event and clk = '1';
      reg1 <= in1 and in2;
      reg2 <= in1 and in2;
      out1 <= not ( reg1);
      out2 <= (not (reg1) and reg2) ;
end process;
end behave;
```

syn_sharing

Directive. Enables/disables the resource sharing of operators inside a module during synthesis. Values for syn_sharing are on or off. See Sharing Resources, on page 5-5 in the *Synplify User Guide* for details about resource sharing.

You can set this directive globally for the entire design by setting it on the top-level module/architecture, or on individual modules. A lower-level directive overrides the global setting. By default, the attribute is enabled globally (value is on). If the Resource Sharing check box (Project->Implementation Options->Options or option in the Project view) is disabled, you can still enable resource sharing with the syn_sharing directive.

Verilog Syntax and Example

object **/* synthesis syn_sharing = on | off */ ;**

where *object* can be module definitions.

```
module my_design(out,in,clk_in) /* synthesis syn_sharing=off */;
// Other code
```

VHDL Syntax and Example

attribute syn_sharing of *object* **:** *object_type* **is " true | false " ;**

where *object* can be architecture names.

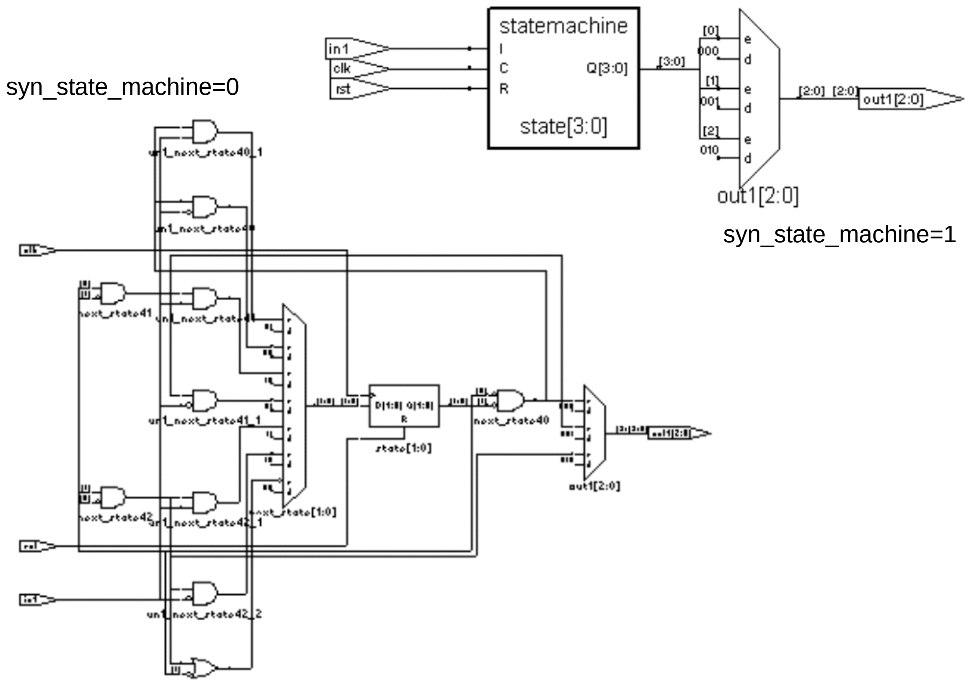
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity alu is
   port ( a, b : in std_logic_vector (7 downto 0);
         opcode: in std_logic_vector (1 downto 0);
         clk: in std_logic;
         result: out std_logic_vector (7 downto 0) );
end alu;

architecture behave of alu is
--  Turn on resource sharing for the architecture.
attribute syn_sharing of behave : architecture is "on";

begin
-- Behavioral source code for the design goes here.
end behave;
```

syn_state_machine

Directive. Enables/disables state-machine optimization on individual state registers in the design. If you disable FSM Compiler, so state-machines in your design *are not* automatically extracted, but you want to extract some of them, you can use this directive on just those individual state-registers. Conversely, if FSM Compiler is enabled, but there are state machines in your design that you do not want extracted, you can use syn_state_machine to override extraction on just those individual state registers.

Also, when FSM Compiler is enabled, all state machines are usually detected during synthesis. However, on occasion there are cases in which certain state machines are not detected. You can use this directive to declare those undetected registers as state machines.

The following figure shows an example of two implementations of a state machine: one with syn_state_machine enabled, the other with the directive disabled.



See the following HDL syntax and example sections for the source code used to generate the schematics above. See also:

- syn_encoding, on page 8-25 for information on overriding default encoding styles for state machines.
- For VHDL designs, syn_encoding compared with syn_enum_encoding, on page 8-70 for usage information about these two directives.

Verilog Syntax and Examples

object **/* synthesis syn_state_machine = 0 | 1 */ ;**

where *object* is a state register. Data type is Boolean: 0 does not extract an FSM, 1 extracts an FSM.

Following is an example of syn_state_machine applied to register OUT.

```
module prep3 (CLK, RST, IN, OUT);
input CLK, RST;
input [7:0] IN;
output [7:0] OUT;
reg [7:0] OUT;
reg [7:0] current_state /* synthesis syn_state_machine=1 */;

// Other code
```

Here is the source code used for the example in the previous figure.

```
module FSM1 (clk, in1, rst, out1);
input        clk, rst, in1;
output [2:0] out1;

`define s0 3'b000
`define s1 3'b001
`define s2 3'b010
`define s3 3'bxxx

reg [2:0] out1;
reg [2:0] state /* synthesis syn_state_machine = 1 */;
reg [2:0] next_state;

always @(posedge clk or posedge rst)
   if (rst) state <= `s0;
   else     state <= next_state;
```

```
// Combined Next State and Output Logic
always @(state or in1)
   case (state)
      `s0  : begin
      out1 <= 3'b000;
      if (in1) next_state <= `s1;
      else     next_state <= `s0;
      end
      `s1  : begin
         out1 <= 3'b001;
         if (in1) next_state <= `s2;
         else     next_state <= `s1;
      end
      `s2  : begin
         out1 <= 3'b010;
         if (in1) next_state <= `s3;
         else     next_state <= `s2;
      end
      default : begin
         out1 <= 3'bxxx;
         next_state <= `s0;
      end
   endcase
endmodule
```

VHDL Syntax and Examples

attribute syn_state_machine of *object* **:** *object_type* **is true|false ;**

where *object* is a signal that holds the value of the state machine. For example:

```
attribute syn_state_machine of current_state: signal is true;
```

Following is the source code used for the example in the previous figure.

```
library ieee;
use ieee.std_logic_1164.all;
entity FSM1 is
   port(
         clk,rst,in1 : in std_logic;
         out1 : out std_logic_vector (2 downto 0)
      );
end FSM1;
```

```
architecture behave of FSM1 is
type state_values is ( s0, s1, s2,s3 );
signal state, next_state: state_values;
attribute syn_state_machine : boolean;
attribute syn_state_machine of state : signal is false;

begin
   process (clk, rst)
   begin
      if rst = '1' then
               state <= s0;
      elsif rising_edge(clk) then
         state <= next_state;
      end if;
   end process;

   process (state, in1) begin
      case state is
         when s0 =>
            out1 <= "000";
            if in1 = '1' then next_state <= s1;
               else next_state <= s0;
            end if;
         when s1 =>
            out1 <= "001";
            if in1 = '1' then next_state <= s2;
               else next_state <= s1;
            end if;
         when s2 =>
            out1 <= "010";
            if in1 = '1' then next_state <= s3;
               else next_state <= s2;
            end if;
         when others =>
            out1 <= "XXX"; next_state <= s0;
      end case;
   end process;
end behave;
```

syn_tco<*n*>

Directive. Supplies the clock to output timing-delay through a black box.

You can use syn_tco<*n*> with any of the following black-box directives:

- black_box_pad_pin
- black_box_tri_pins
- syn_black_box
- syn_isclock
- syn_tpd<n>
- syn_tsu<n>

Note: The syn_tco<*n*> directive can be entered as an attribute using the Attribute panel of the SCOPE editor. The information in the object, attribute, and value fields must be manually entered. See .sdc File Syntax and Example, on page 8-95.

Verilog Syntax and Example

object **/* syn_tco***n* **= "**[**!**]*clock* **->** *bundle* **=** *value***" */ ;**

A *bundle* is a collection of buses and scalar signals. To assign values to bundles, use the following syntax. The values are in ns.

"[**!**]*clock* **->** *bundle* **=** *value***"**

The optional exclamation mark (**!**) indicates a negative edge for a clock. The objects of a bundle must be separated by commas with no spaces between. A valid bundle is A,B,C which lists three signals.

An example defining syn_tco<*n*> with other black-box constraints follows:

```
module ram32x4(z,d,addr,we,clk);
/* synthesis syn_black_box syn_tco1="clk->z[3:0]=4.0"
               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
```

VHDL Syntax and Examples

attribute syn_tco*n* **of** *object* **:** *object_type* **is "**[**!**]*clock* **->** *bundle* **=** *value***" ;**

A *bundle* is a collection of buses and scalar signals. To assign values to bundles, use the following syntax. The values are in ns.

"[**!**]*clock* **->** *bundle* **=** *value***"**

The optional exclamation mark (**!**) indicates a negative edge for a clock. The objects of a bundle must be separated by commas with no spaces between. A valid bundle is A,B,C which lists three signals.

In VHDL, there are 10 predefined instances of each of these directives in the synplify library, for example: syn_tco1, syn_tco2, syn_tco3, ... syn_tco10. If you are entering the timing directives in the source code and you require more than 10 different timing delay values for any one of the directives, declare the additional directives with an integer greater than 10. For example:

```
attribute syn_tco11 : string;
attribute syn_tco11 of bitreg : component is
   "clk -> do0,do1 = 2.0";
attribute syn_tco12 : string;
attribute syn_tco12 of bitreg : component is
   "clk -> do2,do3 = 1.8";
```

Following is an example of assigning syn_tco<*n*> along with some of the other black-box constraints:

```
-- A USE clause for the Synplify Attributes package
-- was included earlier to make the timing constraint
-- definitions visible here.
architecture top of top is
component rcf16x4z port (
   ad0, ad1, ad2, ad3 : in std_logic;
   di0, di1, di2, di3 : in std_logic;
   clk, wren, wpe : in std_logic;
   tri : in std_logic;
   do0, do1, do2, do3 : out std_logic);
end component;
```

```
attribute syn_tco1 of rcf16x4z : component is
                                  "clk -> do0,do1 = 4.0";
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 -> clk = 1.2";
attribute syn_tsu2 of rcf16x4z : component is
                                  "wren,wpe -> clk = 0.0";
-- Other code
```

.sdc File Syntax and Example

The constraint file syntax for the directive is:

define_attribute {v:*blackbox_module***} syn_tco***n* **{** [**!**]*clock* **->** *bundle* **=** *value***}**

For details about the syntax, see the following table:

v:	Constraint file syntax that indicates that the directive is attached to the view.
blackbox_module	The symbol name of the black-box.
n	A numerical suffix that lets you specify different clock to output timing delays for multiple signals/bundles.
!	The optional exclamation mark indicates that the clock is active on its falling (negative) edge.
clock	The name of the clock signal.
bundle	A collection of buses and scalar signals. The objects of a bundle must be separated by commas with no intervening spaces. A valid bundle is A,B,C, which lists three signals.
value	Clock to output delay value in ns.

Constraint file example:

```
define_attribute {v:RCV_CORE} syn_tco1 {CLK-> R_DATA_OUT[63:0]=20}
define_attribute {v:RCV_CORE) syn_tco2 {CLK-> DATA_VALID=30}
```

syn_tpd<*n*>

Directive. Supplies information on timing propagation for combinational delay through a black box.

You can use syn_tpd<*n*> with any of the following black-box directives:

- black_box_pad_pin
- black_box_tri_pins
- syn_black_box
- syn_isclock
- syn_tco<n>
- syn_tsu<n>

Note: The syn_tpd<*n*> directive can be entered as an attribute using the Attribute panel of the SCOPE editor. The information in the object, attribute, and value fields must be manually entered. See .sdc File Syntax and Example, on page 8-98.

Verilog Syntax and Example

object **/* syn_tpd***n* **= "***bundle* **->** *bundle* **=** *value***" */ ;**

A *bundle* is a collection of buses and scalar signals. To assign values to bundles, use the following syntax. The values are in ns.

"*bundle* **->** *bundle* **=** *value***"**

The objects of a bundle must be separated by commas with no intervening spaces. A valid bundle is A,B,C which lists three signals.

Following is an example of defining syn_tpd<*n*> along with some of the other black-box timing constraints:

```
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
```

VHDL Syntax and Examples

attribute syn_tpd*n* **of** *object* **:** *object_type* **is "***bundle* **->** *bundle* **=** *value***" ;**

In VHDL, there are 10 predefined instances of each of these directives in the synplify library, for example: syn_tpd1, syn_tpd2, syn_tpd3, … syn_tpd10. If you are entering the timing directives in the source code and you require more than 10 different timing delay values for any one of the directives, declare the additional directives with an integer greater than 10. For example:

```
attribute syn_tpd11 : string;
attribute syn_tpd11 of bitreg : component is
   "di0,di1 -> do0,do1 = 2.0";
attribute syn_tpd12 : string;
attribute syn_tpd12 of bitreg : component is
   "di2,di3 -> do2,do3 = 1.8";
```

A *bundle* is a collection of buses and scalar signals. To assign values to bundles, use the following syntax. The values are in ns.

"*bundle* **->** *bundle* **=** *value***"**

The objects of a bundle must be separated by commas with no spaces between. A valid bundle is A,B,C which lists three signals.

Following is an example of assigning syn_tpd<*n*> along with some of the of the black-box constraints:

```
-- A USE clause for the Synplify Attributes package
-- was included earlier to make the timing constraint
-- definitions visible here.
architecture top of top is
component rcf16x4z port (
   ad0, ad1, ad2, ad3 : in std_logic;
   di0, di1, di2, di3 : in std_logic;
   clk, 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 -> clk = 1.2";
attribute syn_tsu2 of rcf16x4z : component is
               "wren,wpe -> clk = 0.0";
-- Other code
```

.sdc File Syntax and Example

The constraint file syntax for the directive is:

define_attribute {v:*blackbox_module***} syn_tpd***n* **{** *bundle* **->** *bundle* **=** *value***}**

For details about the syntax, see the following table:

v:	Constraint file syntax that indicates that the directive is attached to the view.
blackbox_module	The symbol name of the black-box.
n	A numerical suffix that lets you specify different input to output timing delays for multiple signals/bundles.
bundle	A collection of buses and scalar signals. The objects of a bundle must be separated by commas with no intervening spaces. A valid bundle is A,B,C, which lists three signals.
value	Input to output delay value in ns.

Constraint file example:

```
define_attribute {v:MEM} syn_tpd1 {MEM_RD->DATA_OUT[63:0]=20}
```

syn_tristate

Directive. Specifies that an output port, on a module defined as a black box, is a tristate. Use this directive to eliminate multiple driver errors if the output of a black box has more than one driver. A multiple driver error is issued unless you use this directive to specify that the outputs are tristate.

Verilog Syntax and Examples

object **/* synthesis syn_tristate = 1 */ ;**

where *object* can be black-box output ports. For example:

```
module BUFE(O, I, E); /* synthesis syn_black_box */
   output O /* synthesis syn_tristate = 1 */;

// Other code
```

syn_tsu<*n*>

Directive. Supplies information on timing setup delay required for input pins (relative to the clock) in a black box.

You can use syn_tsu<*n*> with any of the following black-box directives:

- black_box_pad_pin
- black_box_tri_pins
- syn_black_box
- syn_isclock
- syn_tco<n>
- syn_tpd<n>

Note: The syn_tsu<*n*> directive can be entered as an attribute using the Attribute panel of the SCOPE editor. The information in the object, attribute, and value fields must be manually entered. See .sdc File Syntax and Example, on page 8-102.

Verilog Syntax and Example

object **/* syn_tsu***n* **= "***bundle* **->** [**!**]*clock* **=** *value***" */ ;**

A *bundle* is a collection of buses and scalar signals. To assign values to bundles, use the following syntax. The values are in ns.

"*bundle* **->** [**!**]*clock* **=** *value***"**

The optional exclamation mark (**!**) indicates a negative edge for a clock. The objects of a bundle must be separated by commas with no spaces between. A valid bundle is A,B,C which lists three signals.

Following is an example of defining syn_tsu<*n*> along with some of the other black-box constraints:

```
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
```

VHDL Syntax and Examples

attribute syn_tsu*n* **of** *object* **:** *object_type* **is "***bundle* **->** [**!**]*clock* **=** *value***" ;**

In VHDL, there are 10 predefined instances of each of these directives in the synplify library, for example: syn_tsu1, syn_tsu2, syn_tsu3, … syn_tsu10. If you are entering the timing directives in the source code and you require more than 10 different timing delay values for any one of the directives, declare the additional directives with an integer greater than 10. For example:

```
attribute syn_tsu11 : string;
attribute syn_tsu11 of bitreg : component is
   "di0,di1 -> clk = 2.0";
attribute syn_tsu12 : string;
attribute syn_tsu12 of bitreg : component is
   "di2,di3 -> clk = 1.8";
```

A *bundle* is a collection of buses and scalar signals. To assign values to bundles, use the following syntax. The values are in ns.

"*bundle* **->** [**!**]*clock* **=** *value***"**

The optional exclamation mark (**!**) indicates a negative edge for a clock. The objects of a bundle must be separated by commas with no spaces between. A valid bundle is A,B,C which lists three signals.

In addition to the syntax used in the code below, you can also use the following Verilog-style syntax to specify this attribute:

```
attribute syn_tsu1 of inputfifo_coregen : component is
   "rd_clk->dout[48:0]=3.0";
```

Following is an example of assigning syn_tsu<*n*> along with some of the other black-box constraints:

```
-- A USE clause for the Synplify Attributes package
-- was included earlier to make the timing constraint
-- definitions visible here.
architecture top of top is
component rcf16x4z port (
   ad0, ad1, ad2, ad3 : in std_logic;
```

```
        di0, di1, di2, di3 : in std_logic;
        clk, wren, wpe : in std_logic;
        tri : in std_logic;
        do0, do1, do2, do3 : out std_logic);
    end component;

    attribute syn_tco1 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 -> clk = 1.2";
    attribute syn_tsu2 of rcf16x4z : component is
                  "wren,wpe -> clk = 0.0";
    -- Other code
```

.sdc File Syntax and Example

The constraint file syntax for the directive is:

define_attribute {v:*blackbox_module***} syn_tsu***n* **{** *bundle* **->** [**!**]*clock* **=** *value***}**

For details about the syntax, see the following table:

v:	Constraint file syntax that indicates that the directive is attached to the view.
blackbox_module	The symbol name of the black-box.
n	A numerical suffix that lets you specify different clock to output timing delays for multiple signals/bundles.
bundle	A collection of buses and scalar signals. The objects of a bundle must be separated by commas with no intervening spaces. A valid bundle is A,B,C, which lists three signals.
!	The optional exclamation mark indicates that the clock is active on its falling (negative) edge.
clock	The name of the clock signal.
value	Input to clock setup delay value in ns.

Constraint file example:

```
define_attribute {v:RTRV_MOD} syn_tsu4 {RTRV_DATA[63:0]->!CLK=20}
```

translate_off/translate_on

Directive. For compatibility with synthesis tools from other vendors than Synplicity, you can use translate_off and translate_on to synthesize designs originally written for use with other synthesis tools, without needing to modify source code. The Synplify synthesis tool ignores all source code between these two directives during synthesis.

Another use of these directives is to prevent the synthesis of stimulus source code that only has meaning for logic simulation. You can use translate_off/translate_on to skip over simulation-specific lines of code that are not synthesizable.

When you use translate_off in a module, the Synplify synthesis tool halts the synthesis of all source code that follows until translate_on is encountered. Every translate_off must have a corresponding translate_on. These directives cannot be nested, therefore, the translate_off directive can only be followed by a translate_on directive.

Verilog Syntax and Example

For Verilog designs, you can use the synthesis macro with the Verilog ‘ifdef directive instead of the translate on/off directives. See synthesis Macro, on page 9-53 for information.

The Verilog syntax for these directives is as follows:

/* synthesis translate_off */

/* synthesis translate_on */

For example:

```
module adder8(cout, sum, a, b, cin);

   // Place code here that you WANT synthesized

/* synthesis translate_off */

   // Place code here that you DO NOT want synthesized.
```

```
/* synthesis translate_on */

   // Place code here that you WANT synthesized.

endmodule
```

VHDL Syntax and Example

For VHDL designs, you can alternatively use the synthesis off/on directives. Select Project->Implementation Options->VHDL and enable the Synthesis On/Off Implemented as Translate On/Off option. This directs the compiler to treat the synthesis off/on directives like translate off/on and ignore any code between these directives.

The following is the VHDL syntax for translate off/on:

synthesis translate_off

synthesis translate_on

For example:

```
architecture behave of ram4 is
begin

-- synthesis translate_off
stimulus: process (clk, a, b)

-- Source code you DO NOT want synthesized

end process;
-- synthesis translate_on

-- Other source code you WANT synthesized
```

CHAPTER 9

Verilog Language Support

This chapter discusses Verilog support in the Synplify synthesis tool, including the following topics:

Language Constructs

This section describes the different Verilog language constructs that are supported by the synthesis tool.

Supported Verilog Language Constructs

The synthesis tool supports the following Verilog constructs:

- Net types: wire, tri, supply1, supply0; register types: reg, integer, time (64 bit reg); arrays of reg
- Continuous assignments
- Gate primitive and module instantiations
- always blocks, user tasks, user functions
- inputs, outputs, and inouts to a module
- All operators (-, -, *, /, %, <, >, <=, >=, ==, !=, ===, !==, &&, ||, !, ~, &, ~&, |, ~|, ^~, ~^, ^, <<, >>, ?:, { }, {{ }})

  Operators / and % are supported for compile-time constants and constant powers of two.
- Procedural statements: assign, if-else-if, case, casex, casez, for, repeat, while, forever, begin, end, fork, join
- Procedural assignments: blocking assignments =**,** non-blocking assignments <=

  Operator <= cannot be mixed with = for the same register. Use parameter override: # and defparam (down one level of hierarchy only).
- Compiler directives: `define, `ifdef, `else, `endif, `include, `undef
- Miscellaneous:
  - Parameter ranges
  - Local declarations to begin-end block
  - Variable indexing of bit vectors on the left and right sides of assignments

Unsupported Verilog Language Constructs

The synthesis tool does *not* support the following Verilog constructs. If found, it generates an error message and halts.

- Net types: trireg, wor, trior, wand, triand, tri0, tri1, and charge strength; register type: real
- Built-in unidirectional and bidirectional switches, and pull-up, pull-down
- Procedural statements: deassign, wait
- Named events and event triggers
- UDPs and specify blocks
- force, release, and hierarchical net names (for simulation only)

Ignored Verilog Language Constructs

The synthesis tool ignores the following Verilog constructs and continues the synthesis run.

- delay, delay control, and drive strength
- scalared, vectored
- initial block
- Compiler directives (except for `define, `ifdef, `else, `endif, `include, and `undef, which are supported)
- Calls to system tasks and system functions (they are only for simulation)

Verilog 2001 Support

You can choose the Verilog standard to use for a project or given files within a project: Verilog '95 or Verilog 2001. See File Options Command, on page 3-93 and Setting Verilog and VHDL Options, on page 3-9 of the *Synplify User Guide*. The synthesis tool supports the following Verilog 2001 features:

Table 9-1: Supported Verilog 2001 Features

Feature	**Description**
Combined data and port types	You can combine module data and port type declarations, for conciseness.
Comma-separated sensitivity list	Commas are allowed as separators in sensitivity lists (as in other Verilog lists).
Wildcards in sensitivity list	You can use @* or @(*) to include all signals in a procedural block, eliminating mismatches between RTL and post-synthesis simulation.
Signed arithmetic expressions	Data types net and reg, module ports, integers of different bases and signals can all be signed. Signed signals can be assigned and compared. Signed operations can be performed for vectors of any length.
Inline parameter value assignment by name	You can assign values to parameters by name, inline.
$signed and $unsigned built-in functions	You can use the built-in Verilog 2001 functions to convert types between signed and unsigned.
Power (exponent) operator (**)	Implemented as a left shift. Only base 2 is supported.
Generate	You use it to create multiple instances of an object in a module. You can use generate with loops and conditional statements.
Constant function	You can use constant functions to support the building of complex values at elaboration time.
Localparam	A constant which cannot be redefined or modified.
Multidimensional Arrays	You use it to group elements of the declared element type into multi-dimensional objects.

Example: Combined Data, Port Types (ANSI C-style Modules)

Verilog ‘95

```
module adder_16 (sum, cout, cin, a, b);
output [15:0] sum;
output cout;
input [15:0] a, b;
input cin;

reg [15:0] sum;
reg cout;
wire [15:0] a, b;
wire cin;
```

Verilog 2001

```
module adder_16(output reg [15:0] sum,
                output reg cout,
                input wire cin,
                input wire [15:0] a, b);
```

Example: Comma-separated Sensitivity List

Verilog ‘95

```
always @(a or b or cin)
   sum = a - b - cin;

always @(posedge clock or negedge reset)
   if (!reset)
      q <= 0;
   else
      q <= d;
```

Verilog 2001

```
always @(a, b or cin)
   sum = a - b - cin;

always @(posedge clock, negedge reset)
   if (!reset)
      q <= 0;
   else
      q <= d;
```

Example: Wildcard (*) in Sensitivity List

Verilog ‘95

```
always @(a or b or cin)
   sum = a - b - cin;
```

Verilog 2001

```
// One style
always @(*)
   sum = a - b - cin;

// Another style
always @*
   sum = a - b - cin;
```

Examples: Signed Signals (Verilog 2001)

Declaration

```
module adder   (output reg signed [31:0] sum,
                wire signed input [31:0] a, b;
```

Assignment

```
wire signed [3:0] a = 4’sb1001;
```

Comparison

```
wire signed [1:0] sel;
parameter   p0 = 2’sb00, p1 = 2’sb01,
            p2 = 2’sb10, p3 = 2’sb11;
case sel
   p0: ...
   p1: ...
   p2: ...
   p3: ...
endcase
```

Examples: Inline Parameter Assignment by Name (Verilog 2001)

```
module top( /* port list of top-level signals */ );
   dff #(.param1(10), .param2(5)) inst_dff(q, d, clk);
endmodule
```

where:

```
module dff #(parameter param1=1, param2=2) (q, d, clk);
   input d, clk;
   output q;
...
endmodule
```

Examples: $signed and $unsigned Built-in Functions (Verilog 2001)

```
c = $signed (s); /* Assign signed valued of s to c. */
d = $unsigned (s); /* Assign unsigned valued of s to d. */
```

Examples: Generate Statement (Verilog 2001)

```
// for loop
generate
begin:G1
   genvar i;
   for (i=0; i<=7; i=i-1)
   begin :inst
      adder8 add (sum [8*i-7 : 8*i], c0[i-1],
      a[8*i-7 : 8*i], b[8*i-7 : 8*i], c0[i]);
   end
endgenerate

// if-else
generate
   if (adder_width < 8)
      ripple_carry # (adder_width) u1 (a, b, sum);
   else
      carry_look_ahead # (adder_width) u1 (a, b, sum);
endgenerate
```

```
// case
parameter WIDTH=1;
generate
   case (WIDTH)
      1: adder1 x1 (c0, sum, a, b, ci);
      2: adder2 x1 (c0, sum, a, b, ci);
      default: adder # width (c0, sum, a, b, ci);
   endcase
endgenerate
```

Multidimensional Array (Verilog 2001)

Use arrays to group elements into multidimensional objects, called vector types. When working with multidimensional arrays, the association is always from right-to-left while the declarations are left-to-right.

Arrays are declared by specifying the element address ranges after the declared identifiers. Use a constant expression, when specifying the indices for the array. The constant expression value can be a positive integer, negative integer, or zero.

Example: Multidimensional Array (Verilog 2001)

A 2 dimensional wire object. "my_wire" is an eight-bit-wide vector with indices from 5 to 0.

```
wire [7:0]  my_wire [5:0];
```

A 3 dimensional wire object. "my_wire" is a eight-bit-wide vector with indices from 5 to 0 whose indices are from 3 down to 0.

```
wire [7:0]  my_wire [5:0] [3:0];
```

A 3 dimensional wire object. "my_wire" is a eight-bit-wide vector (-4 to 3) with indices from -3 to 1 whose indices are from 3 down to 0.

```
wire [-4:3]  my_wire [-3:1] [3:0];
```

All the above examples equally apply for register types.

```
reg [3:0] mem[7:0];  // A regular memory of 8 words with 4
bits/word.

reg [3:0] mem[7:0][3:0]; // A memory of memories.
```

You can access bits in memory words or wire types. The following example refers to bit 1 of the second word (word does not imply storage here) of my_wire:

```
wire[3:0] my_wire[3:0];
assign y = my_wire[2][1]; // refers to bit.
```

There currently exists a restriction in Verilog which prohibits bit access into memory words. Verilog 2001 removes all such restrictions. This applies equally to wires types also.

Consider :

```
wire[3:0] my_wire[3:0];

assign y = my_wire[2][1]; // refers to bit 1 of 2nd word (word does
not imply storage here) of my_wire.
```

This example shows some syntax restrictions:

```
reg [3:0] arrayb [7:0][0:255];
arrayb[1] = 0; // Illegal Syntax - Attempt to write to elements
[1][0]..[1][255]
arrayb[1][12:31] = 0; // Illegal Syntax - Attempt to write to
elements [1][12]..[1][31]
arrayb[1][0] = 0; // Okay. Assigns 32'b0 to the word referenced by
indices [1][0]
Arrayb[22][8] = 0; // Semantic Error, There is no word 8 in 2nd
dimension.
```

Constant Function

Used to build complex values at elaboration time. The following limitations apply to using constant functions:

- No hierarchical references
- Function calls inside constant function can only be constant functions
- System tasks inside constant functions are ignored
- System functions inside constant functions are illegal
- Parameter references inside a constant function should be already visible
- Identifiers need arguments and parameters to be local to the constant function

- Constant functions are illegal within the scope of a generate statement

Example: Constant Function (Verilog 2001)

```
module ram
   // Verilog 2001 ANSI parameter declaration syntax
   #( parameter depth= 129,
      parameter width=16)

   // Verilog 2001 ANSI port declaration syntax
   (  input clk, we,
      // Calculate addr width using Verilog 2001 constant function
      input [clogb2(depth)-1:0] addr,
      input [width-1:0] di,
      output reg [width-1:0] do
      );

function integer clogb2;
input [31:0] value;
   for (clogb2=0; value>0; clogb2=clogb2+1)
   value = value>>1;
endfunction

reg [width-1:0] mem[depth-1:0];

always @(posedge clk) begin
   if (we)
   begin
      mem[addr]<= di;
      do<= di;
   end
   else
      do<= mem[addr];
   end

endmodule
```

Localparam

A new datatype that is identical to a Verilog parameter as to sizing and sign, but which cannot be directly modified. To modify the parameter, you modify the assigned constant expression, which can contain a parameter.

You will get compiler errors if you modify a localparam with defparam or through name-based instantiations. Use caution when modifying parameters with positional-based instantiation. This is because the association is based on a one-to-one mapping. If the compiler encounters a localparam in the positional parameter list while expanding an instantiation, it ignores the localparam and associates the new value to the next available parameter.

Example: Localparams (Verilog 2001)

A localparam follows all the rules identical to a parament with regards to sizing and sign.

Example:

```
parameter ONE = 1
localparam TWO=2*ONE
localparam [3:0] THREE=TWO+1;
localparam signed [31:0] FOUR=2*TWO;
```

Verilog Synthesis Guidelines

This section provides guidelines for synthesis using Verilog.

General Synthesis Guidelines

Some general guidelines are presented here to help you synthesize your Verilog design. See Verilog Module Template, on page 9-16 for additional information.

- Top-level module – The synthesis tool picks the last module compiled that is not referenced in another module as the top-level module. Module selection can be overridden from the Verilog panel of the Options for Implementation dialog box.
- Simulate your design before synthesis to expose logic errors. Logic errors that you do not catch are passed through the synthesis tool, and the synthesized results will contain the same logic errors.
- Simulate your design after placement and routing – Have the place-and-route tool generate a post placement and routing (timing-accurate) simulation netlist, and do a final simulation before programming your devices.

- Avoid asynchronous state machines – To use the synthesis tool for asynchronous state machines, make a netlist of technology primitives from your target library.
- Level-sensitive latches – For modeling level-sensitive latches, use continuous assignment statements.

Constant Function Syntax Restrictions

For Verilog 2001, the syntax for constant functions is identical to the existing function definitions in Verilog. Restrictions on constraint functions are as follows:

- No hierarchal references are allowed
- Any function calls inside constant functions must be constant functions
- System tasks inside constant functions are ignored
- System functions inside constant functions are illegal
- Any parameter references inside a constant function should be visible
- All identifiers, except arguments and parameters, should be local to the constant function
- Constant function sare illegal inside the scope of a generate statement

Multi-dimensional Array Syntax Restrictions

For Verilog 2001, the following examples show multi-dimensional array syntax restrictions.

```
reg [3:0] arrayb [7:0][0:255];

arrayb[1] = 0;
// Illegal Syntax - Attempt to write to elements [1][0]..[1][255]

arrayb[1][12:31] = 0;
// Illegal Syntax - Attempt to write to elements [1][12]..[1][31]

arrayb[1][0] = 0;
// Okay. Assigns 32’b0 to the word referenced by indices [1][0]

Arrayb[22][8] = 0;
// Semantic Error, There is no word 8 in 2nd dimension.
```

It is important to remember that when using multi-dimension arrays, the association is always from right-to-left while the declarations are left-to-right.

Example 1

```
module test (input a,b, output z, input clk,  in1, in2);

reg tmp [0:1][1:0];


always @(posedge clk)
begin
   tmp[1][0] <= a ^ b;
   tmp[1][1] <= a & b;
   tmp[0][0] <= a | b;
   tmp[0][1] <= a &~ b;
end

assign z = tmp[in1][in2];

endmodule
```

Example 2

```
module bb(input [2:0] in, output [2:0] out)
/* synthesis syn_black_box = 1 */;
endmodule


module top(input [2:0] in, input [2:1] d1, output [2:0] out);

wire [2:0] w1[2:1];
wire [2:0] w2[2:1];

generate
begin : ABCD
   genvar i;
   for(i=1; i < 3; i = i+1)
   begin : CDEF
      assign w1[i] = in;
      bb my_bb(w1[i], w2[i]);
   end
end
```

```
endgenerate

assign out = w2[d1];

endmodule
```

Signed Multipliers in Verilog

This section applies only to those using Verilog compilers earlier than version 2001.

The software contains an updated signed multiplier module generator. A signed multiplier is used in VHDL whenever you multiply signed numbers. Because earlier versions of Verilog compilers do not support signed data types, an example is provided on how to write a signed multiplier in your Verilog design:

```
module smul4(a, b, clk, result);
input [3:0]a;
input [3:0]b;
input clk;
output [7:0]result;

reg [3:0]inputa;
reg [3:0]inputb;
reg [7:0]out, result;

always @(inputa or inputb)
begin
out = {4{inputa[3]},inputa} * {4{inputb[3]},inputb};
end

always @(posedge clk)
begin
inputa = a;
inputb = b;
result = out;
end

endmodule
```

Verilog Language Guidelines: always Blocks

An always block can have more than one event control argument, provided they are all edge-triggered events or all signals; these two kinds of arguments cannot be mixed in the same always block.

Examples

```
// OK: Both arguments are edge-triggered events
always @(posedge clk or posedge rst)

// OK: Both arguments are signals
always @(A or B)

// No good: One edge-triggered event, one signal
always @(posedge clk or rst)
```

An always block represents either sequential logic or combinational logic. The one exception is that you can have an always block that specifies level-sensitive latches and combinational logic. Avoid this style, however, because it is error prone and can lead to unwanted level-sensitive latches.

An event expression with posedge/negedge keywords implies edge-triggered sequential logic; and without posedge/negedge keywords implies combinational logic, a level-sensitive latch, or both.

Each sequential always block is triggered from exactly one clock (and optional sets and resets).

You must declare every signal assigned a value inside an always block as a reg or integer. An integer is a 32-bit quantity by default, and is used with the Verilog operators to do two's complement arithmetic.

Syntax:

integer [*msb*:*lsb*] *identifier* **;**

Avoid combinational loops in always blocks. Make sure all signals assigned in a combinational always block are explicitly assigned values every time the always block executes, otherwise the synthesis tool needs to insert level-sensitive latches in the design to hold the last value for the paths that do not assign values. This is a common source of errors, so the tool issues a warning message that latches are being inserted into your design. You will get an error message if you have combinational loops in your design that are not recognized as level-sensitive latches by the synthesis tool (for example if you have an asynchronous state machine).

It is illegal to have a given bit of the same reg or integer variable assigned in more than one always block.

Assigning a 'bx to a signal is interpreted as a “don't care” (there is no 'bx value in hardware); the synthesis tool then creates the hardware with the most efficient design.

Verilog Module Template

Hardware designs can include combinational logic, sequential logic, state machines, and memory. These elements are described in the Verilog module. You also can create hardware by directly instantiating built-in gates into your design (in addition to instantiating your own modules).

Within a Verilog module you can describe hardware with one or more continuous assignments, always blocks, module instantiations, and gate instantiations. The order of these statements within the module is irrelevant, and all execute concurrently. The following is the Verilog module template:

```
module <top_module_name>(<port list>);

/* Port declarations. followed by wire,
   reg, integer, task and function declarations */

/* Describe hardware with one or more continuous assignments,
   always blocks, module instantiations and gate instantiations */

// Continuous assignment
wire <result_signal_name>;
assign <result_signal_name> = <expression>;

// always block
always @(<event expression>)

begin
   // Procedural assignments
   // if statements
   // case, casex, and casez statements
   // while, repeat and for loops
   // user task and user function calls
end

// Module instantiation
<module_name> <instance_name> (<port list>);
```

```
// Instantiation of built-in gate primitive
gate_type_keyword  (<port list>);

endmodule
```

The statements between the begin and end statements in an always block execute sequentially from top to bottom. If you have a fork-join statement in an always block, the statements within the fork-join execute concurrently.

You can add comments in Verilog by preceding your comment text with // (two forward slashes). Any text from the slashes to the end of the line is treated as a comment, and is ignored by the synthesis tool. To create a block comment, start the comment with /* (forward slash followed by asterisk) and end the comment with */ (asterisk followed by forward slash). A block commentcan span any number of lines but cannot be nested inside another block comment.

Scalable Modules

This section describes creating and using scalable Verilog modules.

Creating a Scalable Module

You can create a Verilog module that is scalable, so that it can be stretched or shrunk to handle a user-specified number of bits in the port list buses.

Declare parameters with default parameter values. The parameters can be used to represent bus sizes inside a module.

Syntax

parameter *parameter_name* **=** *value* **;**

You can define more than one parameter per declaration by using comma-separated *parameter_name* **=** *value* pairs.

Example

```
parameter size = 1;
parameter word_size = 16, byte_size = 8;
```

Using Scalable Modules

To use scalable modules, instantiate the scalable module and then override the default parameter value with the defparam keyword. Give the instance name of the module you are overriding, the parameter name, and the new value.

Syntax

defparam *instance_name***.***parameter_name* **=** *new_value* **;**

Example

```
big_register my_register (q, data, clk, rst);
defparam my_register.size = 64;
```

Combine the instantiation and the override in one statement. Use a # (hash mark) immediately after the module name in the instantiation, and give the new parameter value. To override more than one parameter value, use a comma-separated list of new values.

Syntax

module_name **# (***new_values_list***)** *instance_name* **(***port_list***) ;**

Example

```
big_register #(64) my_register (q, data, clk, rst);
```

Creating a Scalable Adder

```
module adder(cout, sum, a, b, cin);

/* Declare a parameter, and give a default value */
parameter size = 1;
output cout;
```

```
/* Notice that sum, a, and b use the value of the size parameter */
output [size-1:0] sum;
input [size-1:0] a, b;
input cin;
assign {cout, sum} = a - b - cin;
endmodule
```

Scaling by Overriding a Parameter Value with defparam

```
module adder8(cout, sum, a, b, cin);
output cout;
output [7:0] sum;
input [7:0] a, b;
input cin;
adder my_adder (cout, sum, a, b, cin);

// Creates my_adder as an eight bit adder
defparam my_adder.size = 8;
endmodule
```

Scaling by Overriding the Parameter Value with

```
module adder16(cout, sum, a, b, cin);
output cout;

/* You can define a parameter at this level of hierarchy, and
   pass that value down to the lower level instance. In this
   example, a parameter called my_size is declared. You
   can declare a parameter with the same name as the lower
   level name (size) because this level of hierarchy has a
   different name range than the lower level and there is no
   conflict -- but there is no correspondence between the two
   names either, so you still have to explicitly pass the
   parameter value down through the hierarchy. */

parameter my_size = 16;    // I want a sixteen bit adder
output [my_size-1:0] sum;
input [my_size-1:0] a, b;
input cin;

/* my_size overrides size inside instance my_adder of adder */
// Creates my_adder as a sixteen bit adder
adder #(my_size) my_adder (cout, sum, a, b, cin);
endmodule
```

Built-in Gate Primitives

You can create hardware by directly instantiating built-in gates into your design (in addition to instantiating your own modules.) The built-in Verilog gates are called primitives.

Syntax

gate_type_keyword [*instance_name*] **(***port_list***) ;**

The gate type keywords for simple and tristate gates are listed in the following tables. The *instance_name* is a unique instance name, and is optional. The signal names in the *port_list* can be given in any order with the restriction that all outputs must come before the inputs. For tristate gates, outputs come first, then inputs, and then enable.

The following tables list the available keywords.

Table 9-2: Simple Gates

Keyword	**Definition**
buf	buffer
not	inverter
and	and gate
nand	nand gate
or	or gate
nor	nor gate
xor	exclusive or gate
xnor	exclusive nor gate

Table 9-3: Tristate Gates

Keyword	Definition
bufif1	tristate buffer with logic one enable
bufif0	tristate buffer with logic zero enable
notif1	tristate inverter with logic one enable
notif0	tristate inverter with logic zero enable

Combinational Logic

Combinational logic is hardware with output values based on some function of the current input values. There is no clock, and no saved states. Most hardware is a mixture of combinational and sequential logic.

You create combinational logic with an always block and/or continuous assignments.

Combinational Logic Examples

The following combinational logic synthesis examples are included in the `*synplicity_install_dir*/examples/verilog/combinat` directory:

- Adders
- ALU
- Bus Sorter
- 3-to-8 Decoder
- 8-to-3 Priority Encoders
- Comparator
- Multiplexers (concurrent signal assignments, case statements, or if-then-else statements can be used to create multiplexers; the tool automati-

cally creates parallel multiplexers when the conditions in the branches are mutually exclusive)

- Parity Generator
- Tristate Drivers

always Blocks for Combinational Logic

Use the Verilog always blocks to model combinational logic as shown in the following template.

```
always @(event_expression)
begin
   // Procedural assignment statements,
   // if, case, casex, and casez statements
   // while, repeat, and for loops
   // task and function calls
end
```

When modeling combinational logic with always blocks, keep the following in mind:

- The always block must have exactly one event control (@(*event_ expression*)) in it, located immediately after the always keyword.
- List all signals feeding into the combinational logic in the event expression. This includes all signals that affect signals that are assigned inside the always block. List all signals on the right side of an assignment inside an always block. The tool assumes that the sensitivity list is complete, and generates the desired hardware. However, it will issue a warning message if any signals on the right side of an assignment inside an always block are not listed, because your pre- and post-synthesis simulation results might not match.
- You must explicitly declare as reg or integer all signals you assign in the always block.

Note: Make sure all signals assigned in a combinational always block are explicitly assigned values each time the always block executes. Otherwise, the synthesis tool must insert level-sensitive latches in your design to hold the last value for the paths that do not assign values. This will occur, for instance, if there are combinational loops in your design. This often represents a coding error. The synthesis tool issues a warning message that latches are being inserted into your design because of combinational loops. You will get an error message if you have combinational loops in your design that are not recognized as level-sensitive latches by the synthesis tool.

Event Expression

Every always block must have one event control (@(*event_expression*)), that specifies the signal transitions that trigger the always block to execute. This is analogous to specifying the inputs to logic on a schematic by drawing wires to gate inputs. If there is more than one signal, separate the names with the or keyword.

Syntax

always @ **(***signal1* **or** *signal2* ...**)**

Example

```
/* The first line of an always block for a multiplexer that
   triggers when 'a', 'b' or 'sel' changes */
always @(a or b or sel)
```

Locate the event control immediately after the always keyword. Do not use the posedge or negedge keywords in the event expression; they imply edge-sensitive sequential logic.

Example: Multiplexer

See also Example: Comma-separated Sensitivity List, on page 9-5.

```
module mux (out, a, b, sel);
output out;
input a, b, sel;
reg out;

always @(a or b or sel)
begin
   if (sel)
      out = a;
   else
      out = b;
end
endmodule
```

Continuous Assignments for Combinational Logic

Use continuous assignments to model combinational logic. To create a continuous assignment:

1. Declare the assigned signal as a wire using the syntax:

   wire [*msb*:*lsb*] *result_signal* ;

2. Specify your assignment with the assign keyword, and give the expression (value) to assign.

   assign *result_signal* = *expression* ;

   or ...

   Combine the wire declaration and assignment into one statement:

   wire [*msb*:*lsb*] *result_signal* = *expression* ;

Each time a signal on the right side of the equal sign (=) changes value, the expression re-evaluates, and the result is assigned to the signal on the left side of the equal sign. You can use any of the built-in operators to create the expression.

The bus range [msb : lsb] is only necessary if your signal is a bus (more than one bit wide).

All outputs and inouts to modules default to wires; therefore the wire declaration is redundant for outputs and inouts and “assign *result_signal* = *expression*;” is sufficient.

Example: Bit-wise AND

```
module bitand (out, a, b);
output [3:0] out;
input [3:0] a, b;
/* This wire declaration is not required because "out" is an
   output in the port list */
wire [3:0] out;

assign out = a & b;

endmodule
```

Example: 8-bit Adder

```
module adder_8 (cout, sum, a, b, cin);
output cout;
output [7:0] sum;
input cin;
input [7:0] a, b;

assign {cout, sum} = a - b - cin;

endmodule
```

Signed Multipliers

A signed multiplier is inferred whenever you multiply signed numbers in Verilog 2001 or VHDL. However, Verilog 95 does not support signed data types. If your Verilog code does not use the Verilog 2001 standard, you can implement a signed multiplier in the following way:

```
module smul4(a, b, clk, result);
input [3:0]a;
input [3:0]b;
input clk;
output [7:0]result;

reg [3:0]inputa;
reg [3:0]inputb;
reg [7:0]out, result;

always @(inputa or inputb)
begin
out = {{4{inputa[3]}},inputa} * {{4{inputb[3]}},inputb};
end

always @(posedge clk)
begin
inputa = a;
inputb = b;
result = out;
end
endmodule
```

Sequential Logic

Sequential logic is hardware that has an internal state or memory. The state elements are either flip-flops that update on the active edge of a clock signal or level-sensitive latches that update during the active level of a clock signal.

Because of the internal state, the output values might depend not only on the current input values, but also on input values at previous times. A state machine is sequential logic where the updated state values depend on the previous state values. There are standard ways of modeling state machines in Verilog. Most hardware is a mixture of combinational and sequential logic.

You create sequential logic with always blocks and/or continuous assignments.

Sequential Logic Examples

The following sequential logic synthesis examples are included in the `*synplicity_install_dir*/examples/verilog/sequentl` directory:

- Flip-flops and level-sensitive latches
- Counters (up, down, and up/down)
- Register file
- Shift registers
- State machines

For additional information on synthesizing flip-flops and latches, see these topics:

- Flip-flops Using always Blocks, on page 9-28
- Level-sensitive Latches, on page 9-29
- Sets and Resets, on page 9-32

Flip-flops Using always Blocks

To create flip-flops/registers, assign values to the signals in an always block, and specify the active clock edge in the event expression.

always Block Template

```
always @(event_expression)
begin
   // Procedural statements
end
```

The always block must have one event control (@(*event_expression*)) immediately after the always keyword that specifies the clock signal transitions that trigger the always block to execute.

Syntax

always @ **(***edge_keyword clock_name***)**

where *edge_keyword* is posedge (for positive-edge triggered) or negedge (for negative-edge triggered).

Example

```
always @(posedge clk)
```

Assignments to Signals in always Blocks

- You must explicitly declare as reg or integer any signal you assign inside an always block.
- Any signal assigned within an edge-triggered always block will be implemented as a register; for instance, signal q in the following example.

Example

```
module dff_or (q, a, b, clk);
output q;
input a, b, clk;
reg q; // Declared as reg, since assigned in always block

always @(posedge clk)
begin
   q <= a | b;
end
endmodule
```

In this example, the result of a|b connects to the data input of a flip-flop, and the q signal connects to the q output of the flip-flop.

Level-sensitive Latches

The preferred method of modeling level-sensitive latches in Verilog is to use continuous assignment statements.

Example

```
module latchor1 (q, a, b, clk);
output q;
input a, b, clk;

assign q = clk ? (a | b) : q;
endmodule
```

Whenever clk, a, or b change, the expression on the right side re-evaluates. If your clk becomes true (active, logic 1), a|b is assigned to the q output. When the clk changes and becomes false (deactivated), q is assigned to q (holds the last value of q). If a or b changes and clk is already active, the new value a|b is assigned to q.

Although it is simpler to specify level-sensitive latches using continuous assignment statements, you can create level-sensitive latches from always blocks. Use an always block and follow these guidelines for event expression and assignments.

always Block Template

```
always@(event_expression)
begin   // Procedural statements
end
```

Whenever the assignment to a signal is incompletely defined, the event expression specifies the clock signal and the signals that feed into the data input of the level-sensitive latch.

Syntax

always @ **(***clock_name* **or** *signal1* **or** *signal2* ... **)**

Example

```
always @(clk or data)
begin
   if (clk)
      q <= data;
end
```

The always block must have exactly one event control (@(*event_expression*)) in it, and must be located immediately after the always keyword.

Assignments to Signals in always Blocks

You must explicitly declare as reg or integer any signal you assign inside an always block.

Any incompletely-defined signal that is assigned within a level-triggered always block will be implemented as a latch.

Whenever level-sensitive latches are generated from an always block, the tool issues a warning message, so that you can verify if a given level-sensitive latch is really what you intended. (If you model a level-sensitive latch using continuous assignment then no warning message is issued.)

Example: Creating Level-sensitive Latches You Want

```
module latchor2 (q, a, b, clk);
output q;
input a, b, clk;
reg q;

always @(clk or a or b)
begin
   if (clk)
      q <= a | b;
end
endmodule
```

If clk, a, or b change, and clk is a logic 1, then set q equal to a|b.

What to do when clk is a logic zero is not specified (there is no else in the if statement), so when clk is a logic 0, the last value assigned is maintained (there is an implicit q=q). The synthesis tool correctly recognizes this as a level-sensitive latch, and creates a level-sensitive latch in your design. The tool issues a warning message when you compile this module (after examination, you may choose to ignore this message).

Example: Creating Unwanted Level-sensitive Latches

```
module mux4to1 (out, a, b, c, d, sel);
output out;
input a, b, c, d;
input [1:0] sel;
reg out;

always @(sel or a or b or c or d)
begin
   case (sel)
      2'd0: out = a;
      2'd1: out = b;
      2'd3: out = d;
   endcase
end
endmodule
```

In the above example, the sel case value 2'd2 was intentionally omitted. Accordingly, out is not updated when the select line has the value 2'd2, and a level-sensitive latch must be added to hold the last value of out under this condition. The tool issues a warning message when you compile this module, and there can be mismatches between RTL simulation and post-synthesis simulation. You can avoid generating level-sensitive latches by adding the missing case in the case statement; using a “default” case in the case statement; or using the Verilog full_case directive.

Sets and Resets

A set signal is an input to a flip-flop that, when activated, sets the state of the flip-flop to a logic one. Asynchronous sets take place independent of the clock, whereas synchronous sets only occur on an active clock edge.

A reset signal is an input to a flip-flop that, when activated, sets the state of the flip-flop to a logic zero. Asynchronous resets take place independent of the clock, whereas synchronous resets take place only at an active clock edge.

Asynchronous Sets and Resets

Asynchronous sets and resets are independent of the clock. When active, they set flip-flop outputs to one or zero (respectively), without requiring an active clock edge. Therefore, list them in the event control of the always block, so that they trigger the always block to execute, and so that you can take the appropriate action when they become active.

Event Control Syntax

always @ **(** *edge_keyword clock_signal* **or** *edge_keyword reset_signal* **or** *edge_keyword set_signal* **)**

Edge_keyword is posedge for active-high set or reset (or positive-edge triggered clock) or negedge for active-low set or reset (or negative-edge triggered clock)

You can list the signals in any order.

Example: Event Control

```
// Asynchronous, active-high set (rising-edge clock)
always @(posedge clk or posedge set)

// Asynchronous, active-low reset (rising-edge clock)
always @(posedge clk or negedge reset)

// Asynchronous, active-low set and active-high reset
// (rising-edge clock)
always @(posedge clk or negedge set or posedge reset)
```

Example: always Block Template with Asynchronous, Active-high reset, set

```
always @(posedge clk or posedge set or posedge reset)
begin
   if (reset) begin

      /* Set the outputs to zero */

   end else if (set) begin

      /* Set the outputs to one */

   end else begin

      /* Clocked logic */
   end
end
```

Example: flip-flop with Asynchronous, Active-high reset and set

```
module dff1 (q, qb, d, clk, set, reset);
input d, clk, set, reset;
output q, qb;
// Declare q and qb as reg because assigned inside always
reg q, qb;

always @(posedge clk or posedge set or posedge reset)
begin
   if (reset) begin
      q <= 0;
      qb <= 1;
   end else if (set) begin
      q <= 1;
      qb <= 0;
   end else begin
```

```
            q <= d;
            qb <= ~d;
         end
      end
      endmodule
```

For simple, single variable flip-flops, the following template can be used.

```
always @(posedge clk or posedge set or posedge reset)

q = reset ? 1'b0 : set ? 1'b1 : d;
```

Synchronous Sets and Resets

Synchronous sets and resets set flip-flop outputs to logic 1 or 0 (respectively) on an active clock edge.

Do not list the set and reset signal names in the event expression of an always block so they do not trigger the always block to execute upon changing. Instead, trigger the always block on the active clock edge, and check the reset and set inside the always block first.

RTL View Primitives

The Verilog compiler can detect and extract the following flip-flops with synchronous sets and resets and display them in the RTL schematic view:

- sdffr—flip-flop with synchronous reset
- sdffs—flip-flop with synchronous set
- sdffrs—flip-flop with both synchronous set and reset
- sdffpat—vectored flip-flop with synchronous set/reset pattern
- sdffre—enabled flip-flop with synchronous reset
- sdffse—enabled flip-flop with synchronous set
- sdffpate—enabled, vectored flip-flop with synchronous set/reset pattern

You can check the name (type) of any primitive by placing the mouse pointer over it in the RTL view: a tooltip displays the name.

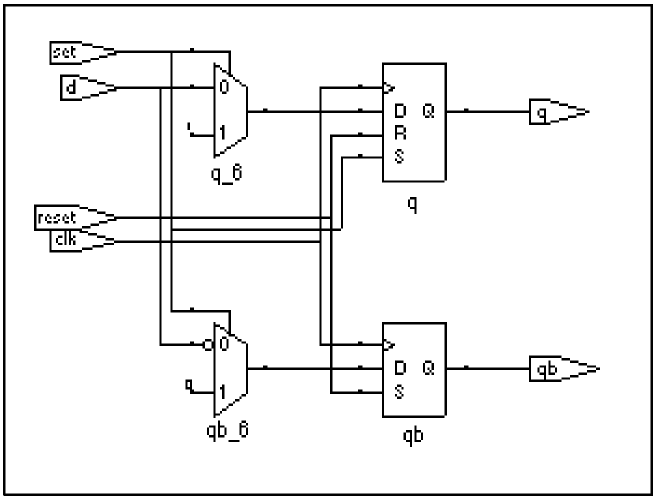


Figure 9-1: Flip-flops with synchronous sets and resets

Event Control Syntax

always @ **(***edge_keyword clock_name* **)**

In the syntax line, *edge_keyword* is posedge for a positive-edge triggered clock or negedge for a negative-edge triggered clock.

Example: Event Control

```
// Positive edge triggered
always @(posedge clk)

// Negative edge triggered
always @(negedge clk)
```

Example: always Block Template with Synchronous, Active-high reset, set

```
always @(posedge clk)
begin
   if (reset) begin
      /* Set the outputs to zero */
   end else if (set) begin
```

```
         /* Set the outputs to one */
      end else begin
         /* Clocked logic */
      end
   end
```

Example: D Flip-flop with Synchronous, Active-high set, reset

```
module dff2 (q, qb, d, clk, set, reset);
input d, clk, set, reset;
output q, qb;
reg q, qb;

always @(posedge clk)
begin
   if (reset) begin
      q <= 0;
      qb <= 1;
   end else if (set) begin
      q <= 1;
      qb <= 0;
   end else begin
      q <= d;
      qb <= ~d;
   end
end
endmodule
```

Verilog State Machines

This section describes Verilog state machines: guidelines for using them, defining state values, and dealing with asynchrony.

State Machine Guidelines

A finite state machine (FSM) is hardware that advances from state to state at a clock edge.

The synthesis tool works best with synchronous state machines. You typically write a fully synchronous design and avoid asynchronous paths such as paths through the asynchronous reset of a register. See Asynchronous State Machines, on page 9-41, for information about asynchronous state machines.

- The state machine must have a synchronous or asynchronous reset, to be inferred. State machines must have an asynchronous or synchronous reset to set the hardware to a valid state after power-up, and to reset your hardware during operation (asynchronous resets are available freely in most FPGA architectures).
- You can define state machines using multiple event controls in an always block only if the event control expressions are identical (for example, @(posedge clk)). These state machines are known as implicit state machines. However it is better to use the explicit style described here and shown in Example: FSM Coding Style, on page 9-38.
- Separate the sequential from the combinational always block statements. Besides making it easier to read, it makes what is being registered very obvious. It also gives better control over the type of register element used.
- Represent states with defined labels or enumerated types.
- Use a case statement in an always block to check the current state at the clock edge, advance to the next state, then set the output values. You can use if statements in an always block, but stay with case statements, for consistency.
- Always use a default assignment as the last assignment in your case statement and set the state variable to 'bx. See Example: default Assignment, on page 9-40.

- Set encoding style with the syn_encoding directive. This attribute overrides the default encoding assigned during compilation. The default encoding is determined by the number of states. If you have a syn_encoding attribute, its value is used during the mapping stage to determine encoding style.

  ```
  attribute syn_encoding : string;
  attribute syn_encoding of <typename> : type is "sequential";
  ```

  See Chapter 8, *Synthesis Attributes and Directives*, for details about the syntax and values.

  One-hot implementations are not always the best choice for state machines, even in FPGAs and CPLDs. For example, one-hot state machines might result in larger implementations, which can cause fitting problems. An example in an FPGA where one-hot implementation can be detrimental is a state machine that drives a large decoder, generating many output signals. In a 16-state state machine, for instance, the output decoder logic might reference sixteen signals in a one-hot implementation, but only four signals in a sequential representation.

Example: FSM Coding Style

```
module FSM1 (clk, rst, enable, data_in, data_out, state0, state1,
   state2);
input clk, rst, enable;
input [2:0] data_in;
output data_out, state0, state1, state2;

/* Defined state labels; this style preferred over ‘defines*/
parameter deflt=2'bxx;
parameter idle=2'b00;
parameter read=2'b01;
parameter write=2'b10;

reg data_out, state0, state1, state2;
reg [1:0] state, next_state;

/* always block with sequential logic*/
always @(posedge clk or negedge rst)
   if (!rst) state <= idle;
   else state <= next_state;
```

```
/* always block with combinational logic*/
always @(state or enable or data_in) begin
   /* Default values for FSM outputs*/
   state0 <= 1'b0;
   state1 <= 1'b0;
   state2 <= 1'b0;
   data_out <= 1'b0;

   case (state)
      idle : if (enable) begin
            state0 <= 1'b1;
            data_out <= data_in[0];
            next_state <= read;
         end
         else begin
            next_state <= idle;
         end
      read : if (enable) begin
            state1 <= 1'b1;
            data_out <= data_in[1];
            next_state <= write;
         end
         else begin
            next_state <= read;
         end
      write : if (enable) begin
            state2 <= 1'b1;
            data_out <= data_in[2];
            next_state <= idle;
         end
         else begin
            next_state <= write;
         end
      /* Default assignment for simulation */
      default : next_state <= deflt;
   endcase
end
endmodule
```

Example: default Assignment

```
default: state = 'bx;
```

Assigning 'bx to the state variable (a “don't care” for synthesis) tells the tool that you have specified all the used states in your case statement. Any remaining states are not used, and the synthesis tool can remove unnecessary decoding and gates associated with the unused states. You do not have to add any special, non-Verilog directives.

If you set the state to a used state for the default case (for example, default state = state1), the tool generates the same logic as if you assign 'bx, but there will be pre- and post-synthesis simulation mismatches until you reset the state machine. These mismatches occur because all inputs are unknown at start up on the simulator. You therefore go immediately into the default case, which sets the state variable to state1. When you power up the hardware, it can be in a used state, such as state2, and then advance to a state other than state1. Post-synthesis simulation behaves more like hardware with respect to initialization.

State Values

In Verilog, you must give explicit state values for states. You do this using parameters or `defines, typically using parameters: the state names are shown in the FSM Viewer only if you use parameters.

Example 1: Using Parameters for State Values

```
parameter state1 = 2'h1, state2 = 2'h2;
...
current_state = state2;  // Setting current state to 2'h2
```

Example 2: Using `define for State Values

```
`define state1     2'h1
`define state2     2'h2
...
current_state = `state2;  // Setting current state to 2'h2
```

Asynchronous State Machines

Avoid defining asynchronous state machines in Verilog. An asynchronous state machine has states, but no clearly defined clock, and has combinational loops.

Do not use tools to design asynchronous state machines; the synthesis tool might remove your hazard-suppressing logic when it performs logic optimization, causing your asynchronous state machines to work incorrectly.

The synthesis tool displays a “Found combinational loop” warning message for an asynchronous state machine when it detects combinational loops in continuous assignment statements, always blocks, and built-in gate-primitive logic.

To create asynchronous state machines, do one of the following:

- To use Verilog, make a netlist of technology primitives from your target library. Any instantiated technology primitives are left in the netlist, and not removed during optimization.
- Use a schematic editor (and not Verilog) for the asynchronous state machine part of your design.

The following asynchronous state machine examples generate warning messages.

Example 1: Asynchronous FSM with Continuous Assignment

```
module async1 (out, g, d);
output out;
input g, d;
assign out = g & d | !g & out | d & out;
endmodule
```

Example 2: Asynchronous FSM with an always Block

```
module async2 (out, g, d);
output out;
input g, d;
reg out;

always @(g or d or out)
begin
   out = g & d | !g & out | d & out;
end
endmodule
```

RAM Inference

The tool can infer synchronous and synchronously resettable RAMs from your Verilog source code and, where appropriate, generate technology-specific single or dual-port RAMs. No special input such as attributes or directives in your source code is needed.

The RTL-level RAM inferred by the compiler always has an asynchronous READ. During synthesis, the mappers convert this to the appropriate technology-specific RAM primitives. Depending on the technology used, the synthesized design can include RAM primitives with either synchronous or asynchronous READs. See Synchronous READ RAMs, on page 9-45, for information on coding your Verilog description to ensure that technology-specific RAM primitives with synchronous READs are used.

The following table lists the supported technology-specific RAMs that the synthesis tool can generate.

Example

```
module ram_test(q, a, d, we, clk);
output reg [7:0] q;
input  [7:0] d;
input  [7:0] a;
input  clk, we;
reg [7:0] mem [255:0] ;
always @(posedge clk) begin
   if(we)
      mem[a] <= d;
   end

always @ (posedge clk)
q = mem[a];
```

```
endmodule
```

RAMs with Special Write Enables

The synthesis tool can infer RAMs when the Write Enable is tied to Vcc or when the Write Enable is nested within IF statements.

always-enabled Write Enable

The RAM extraction code supports the inference of RAMs with their Write Enable tied permanently to Vcc rather than implementing the logic in flip-flops.

Example

```
module xyz (q, d , clk, addr);
input clk;
input [7:0] addr;
input [3:0] d;
output [3:0] q;
reg [255:0] mem [3:0];
wire we;

assign we = 1'b1;
always @(posedge clk)
begin
   if(we)
   begin
      mem[addr] = d;
   end
end

assign q = mem[addr];
endmodule
```

Nested Write Enable

The RAM extraction code can infer RAMs when the Write Enable is more complex as found in nested if statements. The compilers extract common terms from the feedback MUX Enables to derive the common Write Enable signal.

Example

```
module xyz (q, d , we1, we2, clk, addr);
input clk, we1, we2;
input [7:0] addr;
input [3:0] d;
output [3:0] q;
reg [255:0] mem [3:0];

always @(posedge clk)
begin
   if(we1)
   begin
      if(we2)
      begin
         mem[addr] = d;
      end
   end
end

assign q = mem[addr];
endmodule
```

Limited RAM Resources

If your RAM resources are limited, you can designate additional instances of inferred RAMs as flip-flops and logic using the syn_ramstyle attribute. This attribute takes the string argument of registers, which you place on the RAM instance.

Additional Components Generated

After inferring a RAM for some technologies, you might notice that the tool has generated a few additional components adjacent to the RAM. This glue logic ensures accuracy in your post place-and-route simulation results.

Synchronous READ RAMs

All RAM primitives that the tool generates for inferred RAMs have asynchronous READs.

The following examples shows how the READ address or the data output of the RAM can be registered to ensure generation of a synchronous READ RAM.

Example: READ Address Registered

```
module ram_test(q, a, d, we, clk);
output [7:0] q;
input  [7:0] d;
input  [6:0] a;
input  clk, we;

reg [6:0] read_add;
reg [7:0] mem [127:0];

always @(posedge clk) begin
   if(we)
      mem[a] <= d;
      read_add <= a;
   end

assign q = mem[read_add];
endmodule
```

Example: Data Output Registered

```
module ram_test(q, a, d, we, clk);
output [7:0] q;
input  [7:0] d;
input  [6:0] a;
input  clk, we;

reg [7:0] mem [127:0];

always @(posedge clk) begin
   q <= mem [a];
   if(we)
      mem[a] <= d;
   end
endmodule
```

ROM Inference

As part of BEST (Behavioral Extraction Synthesis Technology) feature, the synthesis tool infers ROMs (read-only memories) from your HDL source code, and generates block components for them in the RTL view.

The data contents of the ROMs are stored in a text file named `rom.info`. To quickly view `rom.info` in read-only mode, synthesize your HDL source code, open an RTL view, then push down into the ROM component.

Generally, the Synplify synthesis tool infers ROMs from HDL source code that uses `case` statements, or equivalent `if` statements, to make 16 or more signal assignments using constant values (words). The constants must all be the same width.

Another requirement for ROM inference is that values must be specified for at least half of the address space. For example, if the ROM has 5 address bits, then the address space is 32 and at least 16 of the different addresses must be specified.

Example

```
module rom(z,a);
   output [3:0] z;
   input [4:0] a;
   reg [3:0] z;

   always @(a) begin
      case (a)
      5'b00000 :z = 4'b0001;
      5'b00001 :z = 4'b0010;
      5'b00010 :z = 4'b0110;
      5'b00011 :z = 4'b1010;
      5'b00100 :z = 4'b1000;
      5'b00101 :z = 4'b1001;
      5'b00110 :z = 4'b0000;
      5'b00111 :z = 4'b1110;
      5'b01000 :z = 4'b1111;
      5'b01001 :z = 4'b1110;
      5'b01010 :z = 4'b0001;
      5'b01011 :z = 4'b1000;
      5'b01100 :z = 4'b1110;
      5'b01101 :z = 4'b0011;
      5'b01110 :z = 4'b1111;
      5'b01111 :z = 4'b1100;
      5'b10000 :z = 4'b1000;
      5'b10001 :z = 4'b0000;
      5'b10010 :z = 4'b0011;
      default  :z = 4'b0111;
      endcase
end
endmodule
```

ROM Table Data (rom.info File)

```
Note: This data is for viewing only

ROM work.rom4(behave)-z_1[3:0]
address width: 5
data width: 4
inputs:
0: a[0]
1: a[1]
2: a[2]
3: a[3]
4: a[4]
outputs:
0: z_1[0]
1: z_1[1]
2: z_1[2]
3: z_1[3]

data:
00000 -> 0001
00001 -> 0010
00010 -> 0110
00011 -> 1010
00100 -> 1000
00101 -> 1001
00110 -> 0000
00111 -> 1110
01000 -> 1111
01001 -> 1110
01010 -> 0001
01011 -> 1000
01100 -> 1110
01101 -> 0011
01110 -> 0010
01111 -> 0010
10000 -> 0010
10001 -> 0010
10010 -> 0010
default -> 0111
```

Instantiating Black Boxes in Verilog

Black boxes are modules with just the interface specified; internal information is ignored by the software. Black boxes can be used to directly instantiate:

- Technology-vendor primitives and macros (including I/Os).
- User-designed macros whose functionality was defined in a schematic editor, or another input source. (When the place-and-route tool can merge design netlists from different sources.)

Black boxes are specified with the syn_black_box directive. If the macro is an I/O, use black_box_pad_pin=1 on the external pad pin. See below for details.

For most of the technology-vendor architectures, macro libraries are provided (in `*installation_dir*/lib/*technology*/*family*.v`) that predefine the black boxes for their primitives and macros (including I/Os).

Verilog simulators require a functional description of the internals of a black box. To ensure that the functional description is ignored and treated as a black box, use the translate_off and translate_on directives. See translate_off/translate_on, on page 8-103 for information on the translate_off and translate_on directives.

If the black box has tristate outputs, you must define these outputs with a black_box_tri_pins directive (see black_box_tri_pins, on page 8-58).

The input, output, and delay through a black box is specified with special black box timing directives (see Black-box Timing Models, on page 7-44).

For information on how to instantiate black boxes and technology-vendor I/Os, see *Defining Black Boxes for Synthesis, on page 5-22* of the *Synplify User Guide*.

PREP Verilog Benchmarks

PREP (Programmable Electronics Performance) Corporation distributes benchmark results that show how FPGA vendors compare with each other in terms of device performance and area. The following PREP benchmarks are included in the *`synplicity_install_dir`*`/examples/verilog/prep` directory:

- PREP VHDL Benchmark 1. Data Path
- PREP VHDL Benchmark 2. Timer/Counter
- PREP VHDL Benchmark 3. Small State Machine
- PREP VHDL Benchmark 4. Large State Machine
- PREP VHDL Benchmark 5. Arithmetic Circuit
- PREP VHDL Benchmark 6. 16-Bit Accumulator
- PREP VHDL Benchmark 7. 16-Bit Counter
- PREP VHDL Benchmark 8. 16-Bit Pre-scaled Counter
- PREP VHDL Benchmark 9. Memory Map

The source code for the benchmarks can be used for design examples for synthesis or for doing your own FPGA vendor comparisons.

PREP Corp. has disbanded, but you can still obtain information from their Web site: www.prep.org.

Hierarchy: Structural Verilog

This section describes the creation and use of hierarchical Verilog designs.

Using Hierarchical Verilog Designs

The software accepts and processes hierarchical Verilog designs. You create hierarchy by instantiating a module or a built-in gate primitive within another module.

The signals connect across the hierarchical boundaries through the port list, and can either be listed by position (the same order that you declare them in the lower-level module), or by name (where you specify the name of the lower-level signals to connect to).

Connecting by name minimizes errors, and can be especially advantageous when the instantiated module has many ports.

Creating a Hierarchical Verilog Design

To create a hierarchical design:

1. Create modules.
2. Instantiate the modules within other modules. (When you instantiate modules inside of others, the ones that you have instantiated are sometimes called “lower-level modules” to distinguish them from the “top-level” module that is not inside of another module.)
3. Connect signals in the port list together across the hierarchy either “by position” or “by name” (see the examples, below).

Example: Creating Modules (Interfaces Shown)

```
module mux(out, a, b, sel); // mux
output [7:0] out;
input [7:0] a, b;
input sel;

// <mux functionality>

endmodule

module reg8(q, data, clk, rst); // Eight-bit register
output [7:0] q;
input [7:0] data;
input clk, rst;
// <Eight-bit register functionality>
endmodule

module rotate(q, data, clk, r_l, rst); // Rotates bits or loads
output [7:0] q;
input [7:0] data;
input clk, r_l, rst;
// When r_l is high, it rotates; if low, it loads data
// < Rotate functionality>
endmodule
```

Example: Top-level Module with Ports Connected by Position

```
module top1(q, a, b, sel, r_l, clk, rst);
output [7:0] q;
input [7:0] a, b;
input sel, r_l, clk, rst;
wire [7:0] mux_out, reg_out;

// The order of the listed signals here will match
// the order of the signals in the mux module declaration.
mux mux_1 (mux_out, a, b, sel);
reg8 reg8_1 (reg_out, mux_out, clk, rst);
rotate rotate_1 (q, reg_out, clk, r_l, rst);

endmodule
```

Example: Top-level Module with Ports Connected by Name

```
module top2(q, a, b, sel, r_l, clk, rst);
output [7:0] q;
input [7:0] a, b;
input sel, r_l, clk, rst;
wire [7:0] mux_out, reg_out;

/* The syntax to connect a signal "by name" is:
.<lower_level_signal_name>(<local_signal_name>)
*/
mux mux_1 (.out(mux_out), .a(a), .b(b), .sel(sel));

/* Ports connected "by name" can be in any order */
reg8 reg8_1 (.clk(clk), .data(mux_out), .q(reg_out), .rst(rst));
rotate rotate_1 (.q(q), .data(reg_out), .clk(clk), .r_l(r_l),
                 .rst(rst));
endmodule
```

synthesis Macro

Use this text macro along with the Verilog `ifdef compiler directive to conditionally exclude part of your Verilog code from being synthesized. The most common use of the synthesis macro is to avoid synthesizing stimulus that only has meaning for logic simulation.

The synthesis macro is defined so that the statement `ifdef synthesis is true. The statements in the `ifdef branch are compiled; the stimulus statements in the `else branch are ignored.

Note: Because Verilog simulators do *not* recognize a synthesis macro, the compiler for your simulator will use the stimulus in the `else branch.

In the following example, an AND gate is used for synthesis because the tool recognizes the synthesis macro to be defined (as true); the assign c = a & b branch is taken. During simulation, an OR gate is used instead, because the simulator does not recognize the synthesis macro to be defined; the assign c = a | b branch is taken.

Note: A macro in Verilog has a nonzero value only if it is defined.

```
module top (a,b,c);
   input a,b;
   output c;
`ifdef synthesis
   assign c = a & b;
`else
   assign c = a | b;
`endif
endmodule
```

text Macro

The directive ‘define creates a macro for text substitution. The compiler substitutes the text of the macro for the string *macro_name*. A text macro is defined using arguments that can be customized for each individual use.

The syntax for a text macro definition is as follows.

Test_macro_definition ::= **define** *text_macro_name macro_text*

text_macro_name ::= text_macro_identifier[(list_of_formal_arguments)]

list_of_formal_arguments ::= formal_argument_identifier {, formal_argument_identifier}

When formal arguments are used to define a text macro, the scope of the formal argument extends to the end of the macro text. You can use a formal argument in the same manner as an identifier.

A text macro with one or more arguments is expanded by replacing each formal argument with the actual argument expression.

Example 1

```
`define MIN(p1, p2) (p1)<(p2)?(p1):(p2)

module example1(i1, i2, o);
input i1, i2;
output o;
reg o;

always @(i1, i2) begin
o = `MIN(i1, i2);
end
endmodule
```

Example 2

```
`define SQR_OF_MAX(a1, a2) (`MAX(a1, a2))*(`MAX(a1, a2))
`define MAX(p1, p2) (p1)<(p2)?(p1):(p2)

module example2(i1, i2, o);
input i1, i2;
output o;
reg o;

always @(i1, i2) begin
o = `SQR_OF_MAX(i1, i2);
end
endmodule
```

Example 3 Include File ppm_top_ports_def.inc

```
//ppm_top_ports_def.inc

// Single source definition for module ports and signals
// of PPM TOP.
// Input
`DEF_DOT `DEF_IN([7:0]) in_test1 `DEF_PORT(in_test1) `DEF_END
`DEF_DOT `DEF_IN([7:0]) in_test2 `DEF_PORT(in_test2) `DEF_END

// In/Out
// `DEF_DOT `DEF_INOUT([7:0]) io_bus1 `DEF_PORT(io_bus1) `DEF_END

// Output
`DEF_DOT `DEF_OUT([7:0]) out_test2 `DEF_PORT(out_test2)
// No DEF_END here...
```

```
`undef DEF_IN
`undef DEF_INOUT
`undef DEF_OUT
`undef DEF_END
`undef DEF_DOT
`undef DEF_PORT
```

Verilog File top.v

```
// top.v

`define INC_TYPE  1
module ppm_top(
  `ifdef INC_TYPE
// Inc file Port def...
   `define DEF_IN(arg1) /* arg1 */
   `define DEF_INOUT(arg1) /* arg1 */
   `define DEF_OUT(arg1) /* arg1 */
   `define DEF_END ,
   `define DEF_DOT /* nothing */
   `define DEF_PORT(arg1) /* arg1 */

`include "ppm_top_ports_def.inc"
   `else
   // Non-Inc file Port def, above defines should expand to
   // what is below...
      /* nothing */ /* [7:0] */ in_test1 /* in_test1 */ ,
      /* nothing */ /* [7:0] */ in_test2 /* in_test2 */ ,

   // In/Out
   //`DEF_DOT `DEF_INOUT([7:0]) io_bus1 `DEF_PORT(io_bus1)
`DEF_END

   // Output
      /* nothing */ /* [7:0] */ out_test2 /* out_test2 */
// No DEF_END here...
  `endif
);

   `ifdef INC_TYPE
   // Inc file Signal type def...
   `define DEF_IN(arg1) input arg1
   `define DEF_INOUT(arg1) inout arg1
   `define DEF_OUT(arg1) output arg1
   `define DEF_END ;
   `define DEF_DOT /* nothing */
   `define DEF_PORT(arg1) /* arg1 */

`include "ppm_top_ports_def.inc"
   `else
   // Non-Inc file Signal type def, defines should expand to
   // what is below...
      /* nothing */ input [7:0] in_test1 /* in_test1 */ ;
      /* nothing */ input [7:0] in_test2 /* in_test2 */ ;

// In/Out
   //`DEF_DOT `DEF_INOUT([7:0]) io_bus1 `DEF_PORT(io_bus1)`DEF_END

// Output
   /* nothing */ output [7:0] out_test2 /* out_test2) */
// No DEF_END here...
   `endif

  ; /* Because of the 'No DEF_END here...' in line of the include
file. */

   assign out_test2 = (in_test1 & in_test2);

endmodule
```

Synthesis Attributes and Directives

Verilog synthesis attributes and directives allow you to associate information with your design to control the way it is analyzed, compiled, and mapped.

- *Attributes* direct the way your design is optimized and mapped during synthesis. Although you can place synthesis attributes directly in your source code, specify them in a constraint file instead, with the Attributes panel of the SCOPE spreadsheet. That way, changes can be made (to the constraint file), without requiring you to recompile.
- *Directives* control the way your design is analyzed prior to mapping. They must therefore be included directly in your source code; they cannot be specified in a constraint file.

Because Verilog does not have predefined attributes or directives for synthesis, you attach attributes and directives to source code items as *comments*.

In addition to the general attributes and directives described in Chapter 8, *Synthesis Attributes and Directives*, the tool also supports vendor-specific directives and attributes that apply to the specific programmable logic vendor you are targeting for your design.

CHAPTER 10

VHDL Language Support

This chapter discusses how you can use the VHDL language to create HDL source code for the synthesis tool:

Language Constructs

This section generally describes how the synthesis tool relates to different VHDL language constructs.

Supported VHDL Language Constructs

The following is a compact list of language constructs that are supported.

- Entity, architecture, and package design units
- Function and procedure subprograms
- All IEEE library packages, including:
  - std_logic_1164
  - std_logic_unsigned
  - std_logic_signed
  - std_logic_arith
  - numeric_std and numeric_bit
  - standard library package (std)
- In, out, inout, buffer, linkage ports
- Signals, constants, and variables
- Aliases
- Integer, physical and enumeration data types; subtypes of these
- Arrays of scalars and records
- Record data types
- All operators (-, -, *, /, **, mod, rem, abs, not, =, /=, <, <=, >, >=, and, or, nand, nor, xor, xnor, sll, srl, sla, sra, rol, ror, &)

Operators	Supported when...
/, mod, and rem	Arguments are compiler constraints or when the right argument is a power of 2.
**	Arguments are compiler constants. The left operand must always be a power of 2 or be zero. When the left operand is 2, the right operand can be a variable.
sla	The shift distance is a compiler constant.
rol and ror	The shift distance is a compiler constant. The data size is a constant power of 2 and the shift size is such that the maximum shift value does not overflow the data size.

- Sequential statements: signal and variable assignment, wait, if, case, loop, for, while, return, null, function, and procedure call
- Concurrent statements: signal assignment, process, block, generate (for and if), component instantiation, function, and procedure call
- Component declarations and four methods of component instantiations
- Configuration specification and declaration
- Generics; attributes; overloading
- Next and exit looping control constructs
- Predefined attributes: t'base, t'left, t'right, t'high, t'low, t'succ, t'pred, t'val, t'pos, t'leftof, t'rightof, integer'image, a'left, a'right, a'high, a'low, a'range, a'reverse_range, a'length, a'ascending, s'stable, s'event
- Unconstrained ports in entities
- Global signals declared in packages

Unsupported VHDL Language Constructs

If any of these constructs are found, an error message is reported and the synthesis run is cancelled.

- Access, and file types
- Register and bus kind signals
- Guarded blocks
- Expanded (hierarchical) names
- User-defined resolution functions. The synthesis tool only supports the resolution functions for std_logic and std_logic_vector.
- Slices with range indices that do not evaluate to constants

Ignored VHDL Language Constructs

The synthesis tool ignores the following constructs in your design. If found, the tool parses and ignores them and continues with the synthesis run.

- disconnect
- assert and report

VHDL Language Constructs

This section describes the synthesis language support that the synthesis tool provides for each VHDL construct. The language information is taken from the most recent VHDL Language Reference Manual (Revision ANSI/IEEE Std 1076-1993). The section names match those from the LRM, for easy reference.

Data Types

Predefined Enumeration Types

Enumeration types have a fixed set of unique values. The two predefined data types – bit and Boolean, as well as the std_logic type defined in the std_logic_1164 package are the types that represent hardware values. You can declare signals and variables (and constants) that are vectors (arrays) of these types by using the types bit_vector, and std_logic_vector. You typically use std_logic and std_logic_vector, because they are highly flexible for synthesis and simulation.

Table 10-1: Type std_logic

Values	Treated by the synthesis tool as...
'U' (uninitialized)	don't care
'X' (forcing unknown)	don't care
'0' (forcing logic 0)	logic 0
'1' (forcing logic 1)	logic 1
'Z' (high impedance)	high impedance
'W' (weak unknown)	don't care
'L' (weak logic 0)	logic 0
'H' (weak logic 1)	logic 1
'-' (don't care)	don't care

Table 10-2: Type bit

Values	Treated by the synthesis tool a...
'0'	logic 0
'1'	logic 1

Table 10-3: Type boolean

Values	Treated by the synthesis tool as...
false	logic 0
true	logic 1

User-defined Enumeration Types

You can create your own enumerated types. This is common for state machines because it allows you to work with named values rather than individual bits or bit vectors.

Syntax

type *type_name* **is (***value_list***) ;**

Examples

```
type states is ( state0, state1, state2, state3);
type traffic_light_state is ( red, yellow, green);
```

Integers

An integer is a predefined VHDL type that has 32 bits. When you declare an object as an integer, restrict the range of values to those you are using. This results in a minimum number of bits for implementation and on ports.

Data Types for Signed and Unsigned Arithmetic

For signed arithmetic, you have the following choices:

- Use the std_logic_vector data type defined in the std_logic_1164 package, and the package std_logic_signed.
- Use the signed data type, and signed arithmetic defined in the package std_logic_arith.
- Use an integer subrange (for example: signal mysig: integer range -8 to 7;). If the range includes negative numbers, the Synplify synthesis tool uses a twos-complement bit vector of minimum width to represent it (four bits in this example). Using integers limits you to a 32-bit range of values,

and is typically only used to represent small buses. Integers are most commonly used for indexes.

- Use the signed data type from the numeric_std or numeric_bit packages.

For unsigned arithmetic, you have the following choices:

- Use the std_logic_vector data type defined in the std_logic_1164 package and the package std_logic_unsigned.
- Use the unsigned data type and unsigned arithmetic defined in the package std_logic_arith.
- Use an integer subrange (for example: signal mysig: integer range 0 to 15;). If the integers are restricted to positive values, the Synplify synthesis tool uses an unsigned bit vector of minimum width to represent it (four bits in this example). Using integers limits you to a 32-bit range of values, and is typically only used to represent small buses (integers are most commonly used for indexes).
- Use the unsigned data type from the numeric_std or numeric_bit packages.

User-Defined Types on Primary Ports

The VHDL simulation netlist does not preserve user-defined types on the primary inputs and outputs of a design. The synthesis tool now preserves these user-defined types, such as record and multidimensional array, by creating a cross-reference file that maps the user types to the bit-blasted I/Os. This eliminates the need for the user to create a wrapper and ensures that the post-synthesis netlist can be used in an RTL test bench or harness.

Declaring and Assigning Objects in VHDL

VHDL objects (object classes) include signals (and ports), variables, and constants. The synthesis tool does not support the file object class.

Naming Objects

VHDL is case insensitive. A VHDL name (identifier) must start with a letter and can be followed by any number of letters, numbers, or underscores ('_'). Underscores cannot be the first or last character in a name and cannot be used twice in a row. No special characters such as '$', '?', '*', '-', or '!', can be used as part of a VHDL identifier.

Syntax

object_class object_name **:** *data_type* [**:=** *initial_value*] **;**

- object_class is signal, variable, or constant.
- object_name is the name (the identifier) of the object.
- data_type can be any predefined data type (such as bit or std_logic_vector) or user-defined data type.

Assignment Operators

<= Signal assignment operator.

:= Variable assignment and initial value operator.

Signals and Ports

In VHDL, the port list of the entity lists the I/O signals for the design. Ports of mode in can be read from, but not assigned (written) to. Ports of mode out can be assigned to, but not read from. Ports of mode inout are bidirectional and can be read from and assigned to. Ports of mode buffer are like inout ports but can have only one internal driver on them.

Internal signals are declared in the architecture declarative area and can be read from or assigned to anywhere within the architecture.

Examples

```
signal my_sig1 : std_logic;    -- Holds a single std_logic bit
begin                          -- An architecture statement area
my_sig1 <= '1';                -- Assign a constant value '1'

-- My_sig2 is a 4-bit integer
signal my_sig2 : integer range 0 to 15;
begin                         -- An architecture statement area
my_sig2 <= 12;

-- My_sig_vec1 holds 8 bits of std_logic, indexed from 0 to 7
signal my_sig_vec1 : std_logic_vector (0 to 7) ;
begin                         -- An architecture statement area

-- Simple signal assignment with a literal value
my_sig_vec1 <= "01001000";
```

```
-- 16 bits of std_logic, indexed from 15 down to 0
signal my_sig_vec2 : std_logic_vector (15 downto 0) ;
begin                       -- An architecture statement area

-- Simple signal assignment with a vector value
my_sig_vec2 <= "0111110010000101";

-- Assigning with a hex value FFFF
my_sig_vec2 <= X"FFFF";

-- Use package Std_Logic_Signed
signal my_sig_vec3 : signed (3 downto 0);
begin                       -- An architecture statement area

-- Assigning a signed value, negative 7
my_sig_vec3 <= "1111";

-- Use package Std_Logic_Unsigned
signal my_sig_vec4 : unsigned (3 downto 0);
begin                       -- An architecture statement area

-- Assigning an unsigned value of 15
my_sig_vec4 <= "1111";

-- Declare an enumerated type, a signal of that type, and
-- then make an valid assignment to the signal
type states is ( state0, state1, state2, state3);
signal current_state : states;
begin                       -- An architecture statement area
current_state <= state2;
```

Variables

VHDL variables are declared within a process or subprogram and then used internally. Generally, variables are not visible outside the process or subprogram they are declared unless passed as a parameter to another subprogram.

Example

```
process (clk)   -- What follows is the process declaration area
   variable my_var1 : std_logic := '0'; -- Initial value '0'

begin         -- What follows is the process statement area
   my_var1 := '1';
end process;
```

Example

```
process (clk, reset)
-- Set the initial value of the variable to hex FF
   variable my_var2 : std_logic_vector (1 to 8) := X"FF";
begin
-- my_var2 is assigned the octal value 44
   my_var2 := O"44";
end process;
```

VHDL Constants

VHDL constants are declared in any declarative region and can be used within that region. The value of a constant cannot be changed.

Example

```
package my_constants is
   constant num_bits : integer := 8;

-- Other package declarations

end my_constants;
```

Libraries and Packages

When you want to synthesize a design in VHDL, include the HDL files in the source files list of your synthesis tool project. Often your VHDL design will have more than one source file. List all the source files in the order you want them compiled; the files at the top of the list are compiled first.

Compiling Design Units into Libraries

All design units in VHDL, including your entities and packages are compiled into libraries. A library is a special directory of entities, architectures and/or packages. You compile source files into libraries by adding them to the source file list. In VHDL, the library you are compiling has the default name work. All entities and packages in your source files are automatically compiled into work. You can keep source files anywhere on your disk, even though you add them to libraries. You can have as many libraries as are needed.

1. To add a file to a library, select the file in the Project view.

   The library structure is maintained in the Project view. The name of the library where a file belongs appears on the same line as the filename, and directly in front of it.

2. Choose Project -> Set Library from the menu bar, then type the library name. You can add any number of files to a library.

3. If you want to use a design unit that you compiled into a library (one that is no longer in the default work library), you must use a library clause in the VHDL source code to reference it.

   For example, if you add a source file for the entity ram16x8 to library my_rams, and instantiate the 16x8 RAM in the design called top_level, you must add library my_rams; just before defining top_level.

Predefined Packages

The synthesis tool supports the two standard libraries ,`std` and `ieee,` that contain packages containing commonly used definitions of data types, functions, and procedures. These libraries and their packages are built in to the synthesis tool, so you do not compile them. The packages are described in the following table.

Table 10-4: Predefined VHDL Libraries and Packages

Library	**Package**	**Description**
std	standard	Defines the basic VHDL types including bit and bit_vector
ieee	std_logic_1164	Defines the 9-value std_logic and std_logic_vector types
ieee	numeric_bit	Defines numeric types and arithmetic functions. The base type is BIT.
ieee	numeric_std	Defines arithmetic operations on types defined in std_logic_1164

Table 10-4: Predefined VHDL Libraries and Packages (Continued)

Library	Package	Description
ieee	std_logic_arith	Defines the signed and unsigned types, and arithmetic operations for the signed and unsigned types
ieee	std_logic_signed	Defines signed arithmetic for std_logic and std_logic_vector
ieee	std_logic_unsigned	Defines unsigned arithmetic for std_logic and std_logic_vector

The Synplify synthesis tool also has vendor-specific built-in macro libraries for vendor macros like gates, counters, flip-flops, and I/Os. If you use the built-in macro libraries, you can easily instantiate vendor macros directly into the VHDL designs, and forward-annotate them to the output netlist. Refer to the appropriate vendor support chaptertopic for more information.

Additionally, the Synplify synthesis tool library contains an attributes package of built-in attributes and timing constraints (`installation_dir/lib/vhd/synattr.vhd`) that you can use with VHDL designs. The package includes declarations for timing constraints (including black-box timing constraints), vendor-specific attributes and synthesis attributes. To access these built-in attributes, add the following two lines to the beginning of each of the VHDL design units that uses them:

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

Accessing Packages

To gain access to a package include a library clause in your VHDL source code to specify the library the package is contained in, and a use clause to specify the name of the package. The library and use clauses must be included immediately before the design unit (entity or architecture) that uses the package definitions.

Syntax

library *library_name* **;**
use *library_name*.*package_name*.**all ;**

To access the data types, functions and procedures declared in std_logic_1164, std_logic_arith, std_logic_signed, or std_logic_unsigned, you need a library ieee clause and a use clause for each of the packages you want to use.

Example

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

-- Other code
```

Library and Package Rules

To access the standard package, no library or use clause is required. The standard package is predefined (built-in) in VHDL, and contains the basic data types of bit, bit_vector, Boolean, integer, real, character, string, and others along with the operators and functions that work on them.

If you create your own package, and compile it into the work library to access its definitions, you still need a use clause before the entity using them, but not a library clause (because work is the default library.)

To access packages other than those in work and std, you must provide a library and use clause for each package as shown in the following example of creating a resource library.

```
-- Compile this in library mylib
library ieee;
use ieee.std_logic_1164.all;

package my_constants is
constant max: std_logic_vector(3 downto 0):="1111";
   .
   .
   .
end package;

-- Compile this in library work
library ieee, mylib;
use ieee.std_logic_1164.all;
use mylib.my_constants.all;

entity compare is
port(a: in std_logic_vector(3 downto 0);
    z: out std_logic);
end compare;
```

```
architecture rtl of compare is
begin
   z <= '1' when (a = max) else '0';
end rtl;
```

The rising_edge and falling_edge functions are defined in the std_logic_1164 package. If you use these functions, your clock signal must be declared as type std_logic.

Instantiating Components in a Design

No library or use clause is required to instantiate components (entities and their associated architectures) compiled in the default work library. The files containing the components must be listed in the source files list before the file(s) that instantiates them.

To instantiate components from the built-in technology-vendor macro libraries, you must include the appropriate use and library clauses in your source code. Refer to the section for your vendor for more information.

To create a separate resource library to hold your components, put all the entities and architectures in one source file, and assign that source file the library components in the synthesis tool Project view. To access the components from your source code, put the clause library components; before the designs that instantiate them. There is no need for a use clause. The project file (`.prj`) must include both the files that create the package components and the source file that accesses them.

Operators

The synthesis tool supports creating expressions using all predefined VHDL operators:

Table 10-5: VHDL Arithmetic Operators

Operator	Description
-	addition
-	subtraction
*	multiplication
/	division (supported for compile-time constants and when divisor is a power of 2)
**	exponentiation (supported for compile-time constants and when left operand is 2; right operand can be a variable)
mod	modulus (supported for compile-time constants and when right operand is a power of 2)
rem	remainder (supported for compile-time constants and when right operand is a power of 2)

Table 10-6: VHDL Relational Operators

Operator	Description
=	equal (if either operand has a bit with an 'X' or 'Z' value, the result is 'X')
/=	not equal (if either operand has a bit with an 'X' or 'Z' value, the result is 'X')
<	less than (if, because of unknown bits in the operands, the relation is ambiguous, then the result is the unknown value 'X')
<=	less than or equal to (if, because of unknown bits in the operands, the relation is ambiguous, then the result is the unknown value 'X')
>	greater than (if, because of unknown bits in the operands, the relation is ambiguous, then the result is the unknown value 'X')
>=	greater than or equal to (if, because of unknown bits in the operands, the relation is ambiguous, then the result is the unknown value 'X')

Table 10-7: VHDL Logical Operators

Operator	Description
and	and
or	or
nand	nand
nor	nor
xor	xor
xnor	xnor
not	not (takes only one operand)

Table 10-8: VHDL Shift Operators

Operator	Description
sll	shift left logical – logically shifted left by R index positions
srl	shift right logical – logically shifted right by R index positions
sla	shift left arithmetic – arithmetically shifted left by R index positions
sra	shift right arithmetic – arithmetically shifted right by R index positions
rol	rotate left logical – rotated left by R index positions
ror	rotate right logical – rotated right by R index positions

Note: The software supports rotation by variable amount when the data size is a constant power of 2 or when the shift size is such that the maximum shift value does not overflow the data size. In these cases, the software doubles up the data and uses a normal left shift or right shift operator.

Table 10-9: Other VHDL Operators

Operator	Description
-	identity
-	negation
&	concatenation

Note: Initially, X's are treated as "don't-cares", but they are eventually converted to 0's or 1's in a way that minimizes hardware.

VHDL Process

The VHDL keyword process introduces a block of logic that is triggered to execute when one or more signals change value. Use processes to model combinational and sequential logic.

process Template to Model Combinational Logic

```
<optional_label> : process (<sensitivity_list>)

-- Declare local variables, data types,
-- and other local declarations here

begin
   --  Sequential statements go here, including:
   --    signal and variable assignments
   --    if and case statements
   --    while and for loops
   --    function and procedure calls
end process  <optional_label>;
```

Sensitivity List

The sensitivity list specifies the signal transitions that trigger the process to execute. This is analogous to specifying the inputs to logic on a schematic by drawing wires to gate inputs. If there is more than one signal, separate the names with commas.

A warning is issued when a signal is not in the sensitivity list but is used in the process, or when the signal is in the sensitivity list but not used by the process.

Syntax

process (*signal1***,** *signal2***, ...) ;**

A process can have only one sensitivity list, located immediately after the keyword process, or one or more wait statements. If there are one or more wait statements, one of these wait statements must be either the first or last statement in the process.

List all signals feeding into the combinational logic (all signals that affect signals assigned inside the process) in the sensitivity list. If you forget to list all signals, the Synplify synthesis tool generates the desired hardware, and reports a warning message that you are not triggering the process every time the hardware is changing its outputs, and therefore your pre- and post-synthesis simulation results might not match.

Any signals assigned in the process must either be outputs specified in the port list of the entity or declared as signals in the architecture declarative area.

Any variables assigned in the process are local and must be declared in the process declarative area.

Note: Make sure all signals assigned in a combinational process are explicitly assigned values each time the process executes,. Otherwise, the synthesis tool must insert level-sensitive latches in your design, in order to hold the last value for the paths that don't assign values (if, for example, you have combinational loops in your design). This usually represents coding error, so the synthesis tool issues a warning message that level-sensitive latches are being inserted into the design because of combinational loops. You will get an error message if you have combinational loops in your design that are not recognized as level-sensitive latches.

Common Sequential Statements

This section describes the if-then-else and case statements.

if-then-else Statement

Syntax

if *condition1* **then**
    sequential_statement(s)
[**elsif** *condition2* **then**
    sequential_statement(s)]
[**else**
    sequential_statement(s)]
end if ;

The else and elsif clauses are optional.

Example

```
library ieee;
use ieee.std_logic_1164.all;
entity mux is
   port (output_signal : out std_logic;
            a, b, sel : in std_logic );
end mux;

architecture if_mux of mux is
begin
   process (sel, a, b)
   begin
      if sel = '1' then
         output_signal <= a;
      elsif sel = '0' then
         output_signal <= b;
      else
         output_signal <= 'X';
      end if;
   end process ;
end if_mux;
```

case Statement

Syntax

case *expression* **is**
  when *choice1* **=>** *sequential_statement(s)*
  when *choice2* **=>** *sequential _statement(s)*

-- *Other case choices*

  when *choiceN* **=>** *sequential_statement(s)*
end case ;

Note: VHDL requires that the expression match one of the given choices. To create a default, have the final choice be when others => sequential_statement(s) or null. (Null means not to do anything.)

Example

```
library ieee;
use ieee.std_logic_1164.all;
entity mux is
   port (output_signal : out std_logic;
      a, b, sel : in std_logic );
end mux;

architecture case_mux of mux is
begin
   process (sel, a, b)
   begin
      case sel is
         when '1' =>
            output_signal <= a;
         when '0' =>
            output_signal <= b;
         when others =>
            output_signal <= 'X';
      end case;
   end process;
end case_mux;
```

Note: To test the condition of matching a bit vector, such as `"0-11"`, that contains one or more don't-care bits, do *not* use the equality relational operator (=). Instead, use the std_match function (in the ieee.numeric_std package), which succeeds (evaluates to true) whenever all of the explicit constant bits (0 or 1) of the vector are matched, regardless of the values of the bits in the don't-care (-) positions. For example, use the condition `std_match(a, "0-11")` to test for a vector with the first bit unset (0) and the third and fourth bits set (1).

Concurrent Signal Assignments

There are three types of concurrent signal assignments in VHDL.

- Simple
- Selected (with-select-when)
- Conditional (when-else)

Use the concurrent signal assignment to model combinational logic. Put the concurrent signal assignment in the architecture body. You can any number of statements to describe your hardware implementation. Because all statements are concurrently active, the order you place them in the architecture body is not significant.

Re-evaluation of Signal Assignments

Every time any signal on the right side of the assignment operator (<=) changes value (including signals used in the expressions, values, choices, or conditions), the assignment statement is re-evaluated, and the result is assigned to the signal on the left side of the assignment operator. You can use any of the predefined operators to create the assigned value.

Simple Signal Assignments

Syntax

signal **<=** *expression* **;**

Example

```
architecture simple_example of simple is
begin
   c <= a nand b ;
end simple_example;
```

Selected Signal Assignments

Syntax

with *expression* **select**
signal **<=** *value1* **when** *choice1* **,**
    value2 **when** *choice2* **,**
    .
    .
    .
    valueN **when** *choiceN* **;**

Example

```
library ieee;
use ieee.std_logic_1164.all;
entity mux is
   port (output_signal : out std_logic;
      a, b, sel : in std_logic );
end mux;

architecture with_select_when of mux is
begin
   with sel select
   output_signal <= a when '1',
               b when '0',
               'X' when others;
end with_select_when;
```

Conditional Signal Assignments

Syntax

signal **<=** *value1* **when** *condition1* **else**
    value2 **when** *condition2* **else**
    valueN-1 **when** *conditionN-1* **else**
    valueN **;**

Example

```
library ieee;
use ieee.std_logic_1164.all;
entity mux is
   port (output_signal: out std_logic;
         a, b, sel: in std_logic);
end mux;

architecture when_else_mux of mux is
begin
output_signal <= a when sel = '1' else
                 b when sel = '0' else
               'X';
end when_else_mux;
```

Note: To test the condition of matching a bit vector, such as `"0-11"`, that contains one or more don't-care bits, do *not* use the equality relational operator (=). Instead, use the std_match function (in the ieee.numeric_std package), which succeeds (evaluates to true) whenever all of the explicit constant bits (0 or 1) of the vector are matched, regardless of the values of the bits in the don't-care (-) positions. For example, use the condition `std_match(a, "0-11")` to test for a vector with the first bit unset (0) and the third and fourth bits set (1).

Resource Sharing

When you have mutually exclusive operators in a case statement, the synthesis tool shares resources for the operators in the case statements. For example, automatic sharing of operator resources includes adders, subtractors, incrementors, decrementors, and multipliers.

Combinational Logic

Combinational logic is hardware with output values based on some function of the current input values. There is no clock and no saved states. Most hardware is a mixture of combinational and sequential logic.

Create combinational logic with concurrent signal assignments and/or processes.

Sequential Logic

Sequential logic is hardware that has an internal state or memory. The state elements are either flip-flops that update on the active edge of a clock signal, or level-sensitive latches, that update during the active level of a clock signal.

Because of the internal state, the output values might depend not only on the current input values, but also on input values at previous times. State machines are made of sequential logic where the updated state values depend on the previous state values. There are standard ways of modeling state machines in VHDL. Most hardware is a mixture of combinational and sequential logic.

Create sequential logic with processes and/or concurrent signal assignments.

Component Instantiation in VHDL

A structural description of a design is made up of component instantiations that describe the subsystems of the design and their signal interconnects. The synthesis tool supports four major methods of component instantiation:

- Simple component instantiation (described below)
- Selected component instantiation
- Direct entity instantiation
- Configurations described in Configuration Specification, on page 10-62

Simple Component Instantiation

In this method, a component is first declared either in the declaration region of the architecture, or in a package of (typically) component declarations, and then instantiated in the statement region of the architecture. By default, the synthesis process binds a named entity (and architecture) in the working library to all component instances that specify a component declaration with the same name.

Syntax

label **:** [**component**] *declaration_name*
  [**generic map (***actual_generic1***,** *actual_generic2***, ...)**]
  [**port map (** *port1***,** *port2***, ...)**] **;**

The use of the reserved word component is optional in component instantiations.

Example: VHDL 1987

```
architecture struct of hier_add is
component add
   generic (size : natural);
   port (a : in bit_vector(3 downto 0);
   b : in bit_vector(3 downto 0);
   result : out bit_vector(3 downto 0)
   );
end component;

begin
-- Simple component instantiation
add1: add
   generic map(size => 4)
   port map(a => ain,
      b => bin,
      result => q);

-- Other code
```

Example: VHDL 1993

```
architecture struct of hier_add is
component add
   generic (size : natural);
   port (a : in bit_vector(3 downto 0);
      b : in bit_vector(3 downto 0);
      result : out bit_vector(3 downto 0)
      );
end component;

begin
-- Simple component instantiation
add1: component add -- Component keyword new in 1993
   generic map(size => 4)
   port map(a => ain,
      b => bin,
      result => q);

-- Other code
```

Note: If no entity is found in the working library named the same as the declared component, all instances of the declared component are mapped to a black box and the error message “Unbound component mapped to black box” is issued.

VHDL Selected Name Support

Selected Name Support (SNS) is provided in VHDL for constants, operators, and functions in library packages. SNS eliminates ambiguity in a design referencing elements with the same names, but that have unique functionality when the design uses the elements with the same name defined in multiple packages. By specifying the library, package, and specific element (constant, operator, or function), SNS designates the specific constant, operator, or function used. This section discusses all facets of SNS. Complete VHDL examples are included to assist you in understanding how to use SNS effectively.

Constants

SNS lets you designate the constant to use from multiple library packages. To incorporate a constant into a design, specify the library, package, and constant. Using this feature eliminates ambiguity when multiple library packages have identical names for constants and are used in an entity-architecture pair.

The following example has two library packages available to the design CONSTANTS. Each library package has a constant defined by the name C1 and each has a different value. SNS is used to specify the constant (see WORK.PACKAGE.C1 in the constants example below).

```
-- CONSTANTS PACKAGE1
library IEEE;
use IEEE.std_logic_1164.all;
package PACKAGE1 is
   constant Cl: std_logic_vector := "10001010";
end PACKAGE1;

-- CONSTANTS PACKAGE2
library IEEE;
use IEEE.std_logic_1164.all;
package PACKAGE2 is
   constant C1: std_logic_vector := "10110110";
end PACKAGE2;

-- CONSTANTS EXAMPLE
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity CONSTANTS is
   generic (num_bits : integer := 8) ;
      port (
            a,b: in std_logic_vector (num_bits -1 downto 0);
            out1, out2: out std_logic_vector (num_bits -1 downto 0)
          );
end CONSTANTS;

architecture RTL of CONSTANTS is
begin
   out1 <= a - work.PACKAGE1.Cl; --Example of specifying SNS
   out2 <= b - work.PACKAGE2.C1; --Example of specifying SNS
end RTL;
```

In the above design, outputs out1 and out2 use two C1 constants from two different packages. Although each output uses a constant named C1, the constants are not equivalent. For out1, the constant C1 is from PACKAGE1. For out2, the constant C1 is from PACKAGE2. For example:

`out1 <= a - work.PACKAGE1.Cl;` is equivalent to `out1 <= a - "10001010";`

whereas:

`out2 <= b - work.PACKAGE2.Cl;` is equivalent to `out2 <= b - "10110110";`

The constants have different values in different packages. SNS specifies the package and eliminates ambiguity within the design.

Functions and Operators

Functions and operators in VHDL library packages customarily have overlapping naming conventions. For example, the add operator in the IEEE standard library exists in both the std_logic_signed and std_logic_unsigned packages. Depending upon the add operator used, different values result. Defining only one of the IEEE library packages is a straightforward solution to eliminate ambiguity, but applying this solution is not always possible. A design requiring both std_logic_signed and std_logic_unsigned package elements must use SNS to eliminate ambiguity.

Functions

In the following example, multiple IEEE packages are declared in a 256x8 RAM design. Both std_logic_signed and std_logic_unsigned packages are included. In the RAM definition, the signal address_in is converted from type std_logic_vector to type integer using the CONV_INTEGER function, but which CONV_INTEGER function will be called? SNS determines the function to use. The RAM definition clearly declares the std_logic_unsigned package as the source for the CONV_INTEGER function.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;
use IEEE.numeric_std.all;

entity FUNCTIONS is
   port( address : in std_logic_vector(7 downto 0);
         data_in : in std_logic_vector(7 downto 0);
         data_out : out std_logic_vector(7 downto 0);
         we : in std_logic;
         clk : in std_logic
       );

end FUNCTIONS;

architecture RTL of FUNCTIONS is

type mem_type is array (255 downto 0) of std_logic_vector (7 downto
0);
signal mem: mem_type;
signal address_in: std_logic_vector(7 downto 0);
```

```
begin
data_out <= mem(IEEE.std_logic_unsigned.CONV_INTEGER(address_in));
process (clk)
   begin
   if rising_edge(clk) then
      if (we = '1') then
         mem(IEEE.std_logic_unsigned.CONV_INTEGER(address_in))
         <= data_in;
      end if;
      address_in <= address;
   end if;
end process;
end RTL;
```

Operators

In this example, comparator functions from the IEEE std_logic_signed and std_logic_unsigned library packages are used. Depending upon the comparator called, a signed or an unsigned comparison results. In the assigned outputs below, the op1 and op2 functions show the valid SNS syntax for operator implementation.

```
library IEEE;
use IEEE.std_logic_1164.std_logic_vector;
use IEEE.std_logic_signed.">";
use IEEE.std_logic_unsigned.">";

entity OPERATORS is
   port (
         in1 :std_logic_vector(1 to 4);
         in2 :std_logic_vector(1 to 4);
         in3 :std_logic_vector(1 to 4);
         in4 :std_logic_vector(1 to 4);
         op1,op2 :out boolean
        );
end OPERATORS;

architecture RTL of OPERATORS is
begin
   process(in1,in2,in3,in4)
   begin

      --Example of specifying SNS
      op1 <= IEEE.std_logic_signed.">"(in1,in2);

      --Example of specifying SNS
      op2 <= IEEE.std_logic_unsigned.">"(in3,in4);
   end process;
end RTL;
```

User-defined Function Support

SNS is not limited to predefined standard IEEE packages and packages supported by the synthesis tool; SNS also supports user-defined packages. You can create library packages that access constants, operators, and functions in the same manner as the packages supported by IEEE or the synthesis tool.

The following example incorporates two user-defined packages in the design. Each package includes a function named func. In PACKAGE1, func is an XOR gate, whereas in PACKAGE2, func is an AND gate. Depending on the package called, func results in either an XOR or an AND gate. The function call uses SNS to distinguish the function that is called.

```
-- USER DEFINED PACKAGE1
library IEEE;
use IEEE.std_logic_1164.all;
package PACKAGE1 is
   function func(a,b: in std_logic) return std_logic;
end PACKAGE1;

package body PACKAGE1 is
   function func(a,b: in std_logic) return std_logic is
begin
   return(a xor b);
end func;
end PACKAGE1;

-- USER DEFINED PACKAGE2
library IEEE;
use IEEE.std_logic_1164.all;

package PACKAGE2 is
   function func(a,b: in std_logic) return std_logic;
end PACKAGE2;

package body PACKAGE2 is
   function func(a,b: in std_logic) return std_logic is
begin
   return(a and b);
end func;
end PACKAGE2;

-- USER DEFINED FUNCTION EXAMPLE
library IEEE;
use IEEE.std_logic_1164.all;
```

```
entity USER_DEFINED_FUNCTION is
   port (
         in0: in  std_logic;
         in1: in  std_logic;
         out0: out std_logic;
         out1: out std_logic
       );
end USER_DEFINED_FUNCTION;

architecture RTL of USER_DEFINED_FUNCTION is
begin
   out0 <= work.PACKAGE1.func(in0, in1);
   out1 <= work.PACKAGE2.func(in0, in1);
end RTL;
```

Demand Loading

In the previous section, the user-defined function example successfully uses SNS to determine the func function to implement. However, neither PACKAGE1 nor PACKAGE2 was declared as a use package clause (for example, work.PACKAGE1.all;). How could func have been executed without a use package declaration? A feature of SNS is demand loading: this loads the necessary package without explicit use declarations. Demand loading lets you create designs using SNS without use package declarations, which supports all necessary constants, operators, and functions.

VHDL Synthesis Guidelines

This section provides guidelines for synthesis using VHDL.

General Synthesis Guidelines

Some general guidelines are presented here to help you synthesize your VHDL design.

- Top-level entity and architecture. The synthesis tool chooses the top-level entity and architecture – the last architecture for the last entity in the last file compiled. Entity selection can be overridden from the VHDL panel of the Options for Implementation dialog box. Files are compiled in the order they appear – from top to bottom in the Project view source files list.
- Simulate your design before synthesis because it exposes logic errors. Logic errors that are not caught are passed through the synthesis tool, and the synthesized results will contain the same logic errors.
- Simulate your design after placement and routing. Have the place-and-route tool generate a post placement and routing (timing-accurate) simulation netlist, and do a final simulation before programming your devices.
- Avoid asynchronous state machines. To use the synthesis tool for asynchronous state machines, make a netlist of technology primitives from your target library.
- For modeling level-sensitive latches, it is simplest to use concurrent signal assignments.

VHDL Language Guidelines

This section discusses VHDL language guidelines.

Processes

- A process must have either a sensitivity list or one wait statement.

- Each sequential process can be triggered from exactly one clock and only one edge of clock (and optional sets and resets).
- Avoid combinational loops in processes. Make sure all signals assigned in a combinational process are explicitly assigned values every time the process executes; otherwise, the synthesis tool needs to insert level-sensitive latches in your design to hold the last value for the paths that do not assign values. This might represent a mistake on your part, so the synthesis tool issues a warning message that level-sensitive latches are being inserted into your design. You will get an warning message if you have combinational loops in your design that are not recognized as level-sensitive latches (for example, if you have an asynchronous state machine).

Assignments

- Assigning an 'X' or '-' to a signal is interpreted as a “don't care”, so the Synplify synthesis tool creates the hardware that is the most efficient design.

Data Types

- Integers are 32-bit quantities. If you declare a port as an integer data type, specify a range (for example, my_input: in integer range 0 to 7). Otherwise, your synthesis result file will contain a 32-bit port.
- Enumeration types are represented as a vector of bits. The encoding can be sequential, gray, or one hot. You can manually choose the encoding for ports with an enumeration type.

Model Template

You can place any number of concurrent statements (signal assignments, processes, component instantiations, and generate statements) in your architecture body as shown in the following example. The order of these statements within the architecture is not significant, as all can execute concurrently.

- The statements between the begin and the end in a process execute sequentially, in the order you type them from top to bottom.

- You can add comments in VHDL by proceeding your comment text with two dashes "--". Any text from the dashes to the end of the line is treated as a comment, and ignored by the Synplify synthesis tool.

```
-- List libraries/packages that contain definitions you use
library <library_name> ;
use <library_name>.<package_name>.all ;

-- The entity describes the interface for your design.
entity <entity_name> is
   generic ( <define_interface_constants_here> ) ;
      port ( <port_list_information_goes_here> ) ;
end <entity_name> ;

-- The architecture describes the functionality (implementation)
-- of your design
architecture <architecture_name> of <entity_name> is

-- Architecture declaration region.
-- Declare internal signals, data types, and subprograms here

-- If you will create hierarchy by instantiating a
-- component (which is just another architecture), then
-- declare its interface here with a component declaration;
component <entity_name_instantiated_below>
   port (<port_list_information_as_defined_in_the_entity>) ;
end component ;

begin         -- Architecture body, describes functionality

-- Use concurrent statements here to describe the functionality
-- of your design. The most common concurrent statements are the
-- concurrent signal assignment, process, and component
-- instantiation.

-- Concurrent signal assignment (simple form):
<result_signal_name> <= <expression> ;

-- Process:
process <sensitivity list>)
-- Declare local variables, data types,
-- and other local declarations here
begin
--  Sequential statements go here, including:
   --    signal and variable assignments
   --    if and case statements
   --    while and for loops
   --    function and procedure calls
end process;
```

```
-- Component instantiation
<instance_name> : <entity_name>
   generic map (<override values here >)
   port map (<port list>) ;
end <architecture_name> ;
```

Constraint Files for VHDL Designs

In previous versions of the software, all object names output by the compiler were converted to lower case. This means that any constraints files created by dragging from the RTL view or through the SCOPE UI contained object names using only lower case. Case is preserved on design object names. If you use mixed-case names in your VHDL source, for constraints to be applied correctly, you must manually update any older constraint files or re-create constraints in the SCOPE editor.

Creating Flip-flops and Registers Using VHDL Processes

It is easy to create flip-flops and registers using a process in your VHDL design.

process Template

```
process (<sensitivity list>)
begin
   <sequential statement(s)>
end;
```

To create a flip-flop:

1. List your clock signal in the sensitivity list. Recall that if the value of any signal listed in the sensitivity list changes, the process is triggered, and executes. For example,

   ```
   process (clk)
   ```

2. Check for rising_edge or falling_edge as the first statement inside the process. For example,

```
process (clk)
begin
if rising_edge(clk) then
<sequential statement(s)>
```

or

```
process (clk)
begin
if falling_edge(clk) then
<sequential statement(s)>
```

Alternatively, you could use an if clk'event and clk = '1' then statement to test for a rising edge (or if clk'event and clk = '0' then for a falling edge). Although these statements work, for clarity and consistency, use the rising_edge and falling_edge functions from the VHDL 1993 standard.

3. Set your flip-flop output to a value, with no delay, if the clock edge occurred. For example, q <= d ;.

Complete Example

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

entity dff_or is
   port (a, b, clk: in std_logic;
         q: out std_logic);
end dff_or;

architecture sensitivity_list of dff_or is
begin
process (clk)  -- Clock name is in sensitivity list
begin
   if rising_edge(clk) then
      q <= a or b;
   end if;
end process;
end sensitivity_list ;
```

In this example, if clk has an event on it, the process is triggered and starts executing. The first statement (the if statement) then checks to see if a rising edge has occurred for clk. If the if statement evaluates to true, there was a rising edge on clk and the q output is set to the value of a or b. If the clk changes from 1 to 0, the process is triggered and the if statement executes, but it evaluates to false and the q output is not updated. This is the functionality of a flip-flop, and synthesis correctly recognizes it as such and connects the result of the `a` or `b` expression to the data input of a D-type flip-flop and the `q` signal to the `q` output of the flip-flop.

Note: The signals you set inside the process will drive the data inputs of D-type flip-flops.

Clock Edges

There are many ways to correctly represent clock edges within a process including those shown here.

The typical rising clock edge representation is:

```
rising_edge(clk)
```

Other supported rising clock edge representations are:

```
clk = '1' and clk'event
clk'last_value = '0' and clk'event
clk'event and clk /= '0'
```

The typical falling clock edge representation is:

```
falling_edge(clk)
```

Other supported falling clock edge representations are:

```
clk = '0' and clk'event
clk'last_value = '1' and clk'event
clk'event and clk /= '1'
```

Incorrect or Unsupported Representations for Clock Edges

Rising clock edge:

```
clk = '1'
clk and clk'event -- Because clk is not a Boolean
```

Falling clock edge:

```
clk = '0'
not clk and clk'event -- Because clk is not a Boolean
```

Defining an Event Outside a Process

The 'event attribute can be used outside of a process block. For example, the process block

```
process (clk,d)
begin
   if (clk='1' and clk'event) then
      q <= d;
   end if;
end process;
```

can be replaced by including the following line outside of the process statement:

```
q <= d when (clk='1' and clk'event);
```

Using a WAIT Statement Inside a Process

The synthesis tool supports a wait statement inside a process to create flip-flops, instead of using a sensitivity list.

Example

```
library ieee;
use ieee.std_logic_1164.all;
entity dff_or is
   port (a, b, clk: in std_logic;
         q: out std_logic);
end dff_or;

architecture wait_statement of dff_or is
begin
process       -- Notice the absence of a sensitivity list.
begin
-- The process waits here until the condition becomes true
   wait until rising_edge(clk);
   q <= a or b;
end process;

end wait_statement;
```

Rules for Using wait Statements Inside a Process

- It is illegal in VHDL to have a process with a wait statement and a sensitivity list.
- The wait statement must either be the first or the last statement of the process.

Clock Edge Representation in wait Statements

The typical rising clock edge representation is:

```
wait until rising_edge(clk);
```

Other supported rising clock edge representations are:

```
wait until clk = '1' and clk'event
wait until clk'last_value = '0' and clk'event
wait until clk'event and clk /= '0'
```

The typical falling clock edge representation is:

```
wait until falling_edge(clk)
```

Other supported falling clock edge representations are:

```
wait until clk = '0' and clk'event
wait until clk'last_value = '1' and clk'event
wait until clk'event and clk /= '1'
```

Level-sensitive Latches Using Concurrent Signal Assignments

To model level-sensitive latches in VHDL, use a concurrent signal assignment statement with the conditional signal assignment form (also known as when-else).

Syntax

signal **<=** *value1* **when** *condition1* **else**
value2 **when** *condition2* **else**
valueN-1 **when** *conditionN-1* **else**
valueN **;**

Example

In VHDL, you are not allowed to read the value of ports of mode out inside of an architecture that it was declared for. Ports of mode buffer can be read from and written to, but must have no more than one driver for the port in the architecture. In the following port statement example, q is defined as mode buffer.

```
library ieee;
use ieee.std_logic_1164.all;
entity latchor1 is
   port (a, b, clk : in std_logic;
-- q has mode buffer so it can be read inside architecture
         q: buffer std_logic );
end latchor1;

architecture behave of latchor1 is
begin
   q  <=  a or b when clk = '1' else q;
end behave;
```

Whenever clk, a, or b changes, the expression on the right side re-evaluates. If clk becomes true (active, logic 1), the value of a or b is assigned to the q output. When the clk changes and becomes false (deactivated), q is assigned to q (holds the last value of q). If a or b changes, and clk is already active, the new value of a or b is assigned to q.

Level-sensitive Latches Using VHDL Processes

Although it is simpler to specify level-sensitive latches using concurrent signal assignment statements, you can create level-sensitive latches with VHDL processes. Follow the guidelines given here for the sensitivity list and assignments.

process Template

```
process (<sensitivity list>)
begin
   <sequential statement(s)>
end process;
```

Sensitivity List

The sensitivity list specifies the clock signal, and the signals that feed into the data input of the level-sensitive latch. The sensitivity list must be located immediately after the process keyword.

Syntax

process (*clock_name***,** *signal1***,** *signal2***, ...)**

Example

```
process (clk, data)
```

process Template for a Level-sensitive Latch

```
process (<clock, data_signals ... > ...)
begin
   if (<clock> = <active_value>)
   <signals> <= <expression involving data signals> ;
   end if;
end process ;
```

All data signals assigned in this manner become logic into data inputs of level-sensitive latches.

Whenever level-sensitive latches are generated from a process, the synthesis tool issues a warning message so that you can verify if level-sensitive latches are really what you intended. Often a thorough simulation of your architecture will reveal mistakes in coding style that can cause the creation of level-sensitive latches during synthesis.

Example: Creating Level-sensitive Latches that You Want

```
library ieee;
use ieee.std_logic_1164.all;
entity latchor2 is
   port (a, b, clk : in std_logic ;
         q: out std_logic );
end latchor2;
```

```
architecture behave of latchor2 is
begin
process (clk, a, b)
begin
   if clk = '1' then
      q <= a or b;
   end if;
end process ;
end behave;
```

If there is an event (change in value) on either clk, a or b, and clk is a logic 1, set q to a or b.

What to do when clk is a logic 0 is not specified (there is no else), so when clk is a logic zero, the last value assigned is maintained (there is an implicit q=q). The Synplify synthesis tool correctly recognizes this as a level-sensitive latch, and creates a level-sensitive latch in your design. It will issue a warning message when you compile this architecture, but after examination, this warning message can safely be ignored.

Example: Creating Unwanted Level-sensitive Latches

This design demonstrates the level-sensitive latch warning caused by a missed assignment in the when two => case. The message generated is:

```
"Latch generated from process for signal odd, probably caused by a
missing assignment in an if or case statement".
```

This information will help you find a functional error even before simulation.

```
library ieee;
use ieee.std_logic_1164.all;
entity mistake is
   port (inp: in std_logic_vector (1 downto 0);
         outp: out std_logic_vector (3 downto 0);
         even, odd: out std_logic);
end mistake;

architecture behave of mistake is
   constant zero: std_logic_vector (1 downto 0):= "00";
   constant one: std_logic_vector (1 downto 0):= "01";
   constant two: std_logic_vector (1 downto 0):= "10";
   constant three: std_logic_vector (1 downto 0):= "11";
begin
process (inp)
begin
   case inp is
```

```
         when zero =>
            outp <= "0001";
            even <= '1';
            odd <= '0';
         when one =>
            outp <= "0010";
            even <= '0';
            odd <= '1';
         when two =>
            outp <= "0100";
            even <= '1';
-- Notice that assignment to odd is mistakenly commented out next.
            -- odd <= '0';
         when three =>
            outp <= "1000";
            even <= '0';
            odd <= '1';
   end case;
end process;
end behave;
```

Sets and Resets

This section describes VHDL sets and resets, both asynchronous and synchronous.

A set signal is an input to a flip-flop that, when activated, sets the state of the flip-flop to a logic one.

A reset signal is an input to a flip-flop that, when activated, sets the state of the flip-flop to a logic zero.

Asynchronous Sets and Resets

By definition, asynchronous sets and resets are independent of the clock and do not require an active clock edge. Therefore, you must include the set and reset signals in the sensitivity list of your process so they trigger the process to execute.

Sensitivity List

The sensitivity list is a list of signals (including ports) that, when there is an event (change in value), triggers the process to execute.

Syntax

process (*clk_name***,** *set_signal_name***,** *reset_signal_name* **)**

The signals are listed in any order, separated by commas.

Example: process Template with Asynchronous, Active-high reset, set

```
process (clk, reset, set)
begin
   if reset = '1' then
-- Reset the outputs to zero.
   elsif set = '1' then
-- Set the outputs to one.
   elsif rising_edge(clk) then -- Rising clock edge clock
-- Clocked logic goes here.
   end if;
end process;
```

Example: D Flip-flop with Asynchronous, Active-high reset, set

```
library ieee;
use ieee.std_logic_1164.all;
entity dff1 is
   port (data, clk, reset, set : in std_logic;
         qrs: out std_logic);
end dff1;

architecture async_set_reset of dff1 is
begin
setreset: process(clk, reset, set)
   begin
      if reset = '1' then
         qrs <= '0';
      elsif set = '1' then
         qrs <= '1';
      elsif rising_edge(clk) then
         qrs <= data;
      end if;
   end process setreset;
end async_set_reset;
```

Synchronous Sets and Resets

Synchronous sets and resets set flip-flop outputs to logic '1' or '0' respectively on an active clock edge.

Do not list the set and reset signal names in the sensitivity list of a process so they will not trigger the process to execute upon changing. Instead, trigger the process when the clock signal changes, and check the reset and set as the first statements.

RTL View Primitives

The VHDL compiler can detect and extract the following flip-flops with synchronous sets and resets and display them in the RTL schematic view:

- sdffr—flip-flop with synchronous reset
- sdffs—flip-flop with synchronous set
- sdffrs—flip-flop with both synchronous set and reset
- sdffpat—vectored flip-flop with synchronous set/reset pattern

- sdffre—enabled flip-flop with synchronous reset
- sdffse—enabled flip-flop with synchronous set
- sdffpate—enabled, vectored flip-flop with synchronous set/reset pattern

You can check the name (type) of any primitive by placing the mouse pointer over it in the RTL view: a tooltip displays the name.

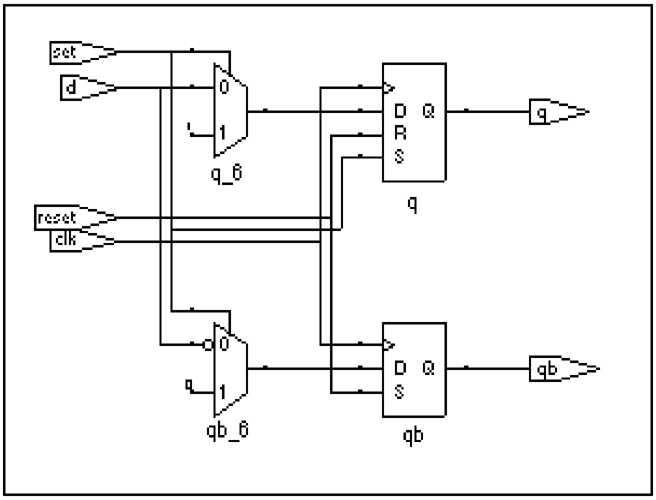


Figure 10-1: Flip-flops with synchronous sets and resets

Sensitivity List

The sensitivity list is a list of signals (including ports) that, when there is an event (change in value), triggers the process to execute.

Syntax

process (*clk_signal_name* **)**

Example: process Template with Synchronous, Active-high reset, set

```
process(clk)
begin
   if rising_edge(clk) then
      if reset = '1' then
         -- Set the outputs to '0'.
      elsif set = '1' then
```

```
            -- Set the outputs to '1'.
         else
            -- Clocked logic goes here.
         end if ;
      end if ;
   end process;
```

Example: D Flip-flop with Synchronous, Active-high reset, set

```
library ieee;
use ieee.std_logic_1164.all;
entity dff2 is
   port (data, clk, reset, set : in std_logic;
         qrs: out std_logic);
end dff2;

architecture sync_set_reset of dff2 is
begin
setreset: process (clk)
begin
   if rising_edge(clk) then
      if reset = '1' then
         qrs <= '0';
      elsif set = '1' then
         qrs <= '1';
      else
         qrs <= data;
      end if;
   end if;
end process setreset;

end sync_set_reset;
```

VHDL State Machines

This section describes VHDL state machines: guidelines for using them, defining state values with enumerated types, and dealing with asynchrony.

State Machine Guidelines

A finite state machine (FSM) is hardware that advances from state to state at a clock edge.

The synthesis tool works best with synchronous state machines. You typically write a fully synchronous design, avoiding asynchronous paths such as paths through the asynchronous reset of a register. See Asynchronous State Machines in VHDL, on page 10-56 for information about asynchronous state machines.

The following are guidelines for coding FSMs:

- The state machine must have a synchronous or asynchronous reset, to be inferred. State machines must have an asynchronous or synchronous reset to set the hardware to a valid state after power-up, and to reset your hardware during operation (asynchronous resets are available freely in most FPGA architectures).
- The synthesis tool does not infer implicit state machines that are created using multiple wait statements in a process.
- Separate the sequential process statements from the combinational ones. Besides making it easier to read, it makes what is being registered very obvious. It also gives better control over the type of register element used.
- Represent states with defined labels or enumerated types.
- Use a case statement in a process to check the current state at the clock edge, advance to the next state, and set the output values. You can also use if-then-else statements.
- Assign default values to outputs derived from the FSM before the case statement. This helps prevent the generation of unwanted latches and makes it easier to read because there is less clutter from rarely used signals.

- If you do not have case statements for all possible combinations of the selector, use a when others assignment as the last assignment in your case statement and set the state vector to some valid state. If your state vector is not an enumerated type, set the value to X. Assign the state to X in the default clause of the case statement, to avoid mismatches between pre- and post-synthesis simulations. See Example: Default Assignment, on page 10-53.
- Override the default encoding style with the syn_encoding attribute. The default encoding is determined by the number of states. However, if you have a syn_encoding attribute, its value is used during the mapping stage to determine encoding style.

  ```
  attribute syn_encoding : string;
  attribute syn_encoding of <typename> : type is “sequential”;
  ```

  See Chapter 8, *Synthesis Attributes and Directives,* for details about the syntax and values.

  One-hot implementations are not always the best choice for state machines, even in FPGAs and CPLDs. For example, one-hot state machines might result in higher speeds in CPLDs, but could cause fitting problems because of the larger number of global signals. An example in an FPGA with ineffective one-hot implementation is a state machine that drives a large decoder, generating many output signals. In a 16-state state machine, for example, the output decoder logic might reference sixteen signals in a one-hot implementation, but only four signals in an encoded representation.

  In general, do not use the directive syn_enum_encoding to set the encoding style. Use syn_encoding instead. The value of syn_enum_encoding is used by the compiler to interpret the enumerated data types but is ignored by the mapper when the state machine is actually implemented. The directive syn_enum_encoding affects the final circuit only when you have turned off the FSM Compiler. Therefore, if you are not using FSM Compiler or the syn_state_machine attribute, which use syn_encoding, you can use syn_enum_encoding to set the encoding style. See Chapter 8, *Synthesis Attributes and Directives,* for details about the syntax and values.
- Implement user-defined enumeration encoding, beyond the one-hot, gray, and sequential styles. Use the directive syn_enum_encoding to set the state encoding. See Example: FSM User-Defined Encoding, on page 10-53.

Example: FSM Coding Style

```
architecture behave of test is
   type state_value is (deflt, idle, read, write);
   signal state, next_state: state_value;
begin

-- Figure out the next state
   process (clk, rst)
   begin
      if rst = '0' then
         state <= idle;
      elsif rising_edge(clk) then
         state <= next_state;
      end if;
   end process;
   process (state, enable, data_in)

   begin
      data_out <= '0';
      -- Catch missing assignments to next_state
      next_state <= idle;
      state0 <= '0';
      state1 <= '0';
      state2 <= '0';
      case state is
         when idle =>
            if enable = '1' then
               state0 <= '1' ;data_out <= data_in(0);
               next_state <= read;
            else next_state <= idle;
            end if;
         when read =>
            if enable = '1' then
               state1 <= '1'; data_out <= data_in(1);
               next_state <= write;
            else next_state <= read;
            end if;
         when deflt =>
            if enable = '1' then
               state2 <= '1' ;data_out <= data_in(2);
               next_state <= idle;
            else next_state <= write;
            end if;
         when others => next_state <= deflt;
      end case;
   end process;
end behave;
```

Example: FSM User-Defined Encoding

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

entity shift_enum is
   port(clk, rst : bit;
    O : out std_logic_vector(2 downto 0));
end shift_enum;

architecture behave of shift_enum is
   type state_type is (S0, S1, S2);
   attribute syn_enum_encoding: string;
   attribute syn_enum_encoding of state_type : type is "001 010 
101";
   signal machine : state_type;
begin
   process (clk, rst)
   begin
    if rst = '1' then
       machine <= S0;
    elsif clk = '1' and clk'event then
       case machine is
       when S0 => machine <= S1;
       when S1 => machine <= S2;
       when S2 => machine <= S0;
       end case;
    end if;
   end process;

   with machine select
    O <= "001" when S0,
    "010" when S1,
    "101" when S2;
end behave;
```

Example: Default Assignment

The second others keyword in the following example pads (covers) all the bits. In this way, you need not remember the exact number of X's needed for the state variable or output signal.

```
 when others =>
    state := (others => 'X') ;
```

Assigning X to the state variable (a “don’t care” for synthesis) tells the synthesis tool that you have specified all the used states in your case statement, and any unnecessary decoding and gates related to other cases can therefore be removed. You do not have to add any special, non-VHDL directives.

If you set the state to a used state for the when others case (for example: when others => state <= delft), the synthesis tool generates the same logic as if you assign X, but there will be pre- and post-synthesis simulation mismatches until you reset the state machine. These mismatches occur because all inputs are unknown at start up on the simulator. You therefore go immediately into the when others case, which sets the state variable to state1. When you power up the hardware, it can be in a used state, such as state2, and then advance to a state other than state1. Post-synthesis simulation behaves more like hardware with respect to initialization.

Using Enumerated Types for State Values

Generally, you represent states in VHDL with a user-defined enumerated type.

Syntax

type *type_name* **is (** *state1_name*, *state2_name*, ... , *stateN_name* **) ;**

Example

```
type states is (st1, st2, st3, st4, st5, st6, st7, st8);
begin
-- The statement region of a process or subprogram.
next_state := st2 ;
--  Setting the next state to st2
```

Simulation Tips When Using Enumerated Types

You want initialization in simulation to mimic the behavior of hardware when it powers up. Therefore, do not initialize your state machine to a known state during simulation, because the hardware will not be in a known state when it powers up.

Creating an Extra Initialization State

If you use an enumerated type for your state vector, create an extra initialization state in your type definition (for example, stateX), and place it first in the list, as shown in the example below.

```
type state is (stateX, state1, state2, state3, state4);
```

In VHDL, the default initial value for an enumerated type is the leftmost value in the type definition (in this example, stateX). When you begin the simulation, you will be in this initial (simulation only) state.

Detecting Reset Problems

In your state machine case statement, create an entry for staying in stateX when you get in stateX. For example:

```
when stateX => next_state := stateX;
```

Look for your design entering stateX. This means that your design is not resetting properly.

Note: The synthesis tool does not create hardware to represent this initialization state (stateX). It is removed during optimization.

Detecting Forgotten Assignment to the Next State

Assign your next state value to stateX right before your state machine case statement.

Example

```
next_state := stateX;
case (current_state) is
...
   when state3 =>
      if (foo = '1') then
         next_state := state2;
      end if;
...
end case;
```

Asynchronous State Machines in VHDL

Avoid defining asynchronous state machines in VHDL. An asynchronous state machine has states, but no clearly defined clock, and has combinational loops. However, if you must use asynchronous state machines, you can do one of the following.

- Create a netlist of the technology primitives from the target library for your technology vendor. Any instantiated primitives that are left in the netlist are not removed during optimization.
- Use a schematic editor for the asynchronous state machine part of your design.

Do not use the synthesis tool to design asynchronous state machines; the tool might remove your hazard-suppressing logic when it performs logic optimization, causing your asynchronous state machine to work incorrectly.

The synthesis tool displays a “found combinational loop” warning message for an asynchronous FSM when it detects combinational loops in continuous assignment statements, processes and built-in gate-primitive logic.

Asynchronous State Machines that Generate Error Messages

In this example, both async1 and async2 will generate combinational loop errors, because of the recursive definition for output.

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

entity async is
-- output is a buffer mode so that it can be read
   port (output : buffer std_logic ;
         g, d : in std_logic ) ;
end async ;

-- Asynchronous FSM from concurrent assignment statement
architecture async1 of async is
begin
   -- Combinational loop error, due to recursive output definition.
   output <= ((((g and d) or (not g)) and output) or d) and
      output);
end async1;

-- Asynchronous FSM created within a process
architecture async2 of async is
begin
process(g, d, output)
begin
-- Combinational loop error, due to recursive output definition.
   output <= ((((g and d) or (not g)) and output) or d) and
      output);
end process;
end async2;
```

Hierarchical Designs in VHDL

This section describes the creation and use of hierarchical VHDL designs.

Creating a Hierarchical Design in VHDL

Creating hierarchy is similar to creating a schematic. You place available parts from a library onto a schematic sheet and connect them.

To create a hierarchical design in VHDL, you instantiate one design unit inside of another. In VHDL, the design units you instantiate are called components. Before you can instantiate a component, you must declare it (step 2, below).

The basic steps for creating a hierarchical VHDL design are:

1. Write the design units (entities and architectures) for the parts you wish to instantiate.
2. Declare the components (entity interfaces) you will instantiate.
3. Instantiate the components, and connect (map) the signals (including top-level ports) to the formal ports of the components to wire them up.

Step 1 – Write Entities and Architectures

Write entities and architectures for the design units to instantiate.

```
library ieee;
use ieee.std_logic_1164.all;
entity muxhier is
   port (outvec: out std_logic_vector (7 downto 0);
         a_vec, b_vec: in std_logic_vector (7 downto 0);
         sel: in std_logic);
end muxhier;

architecture mux_design of muxhier is
begin
-- <mux functionality>
end mux_design;
```

```
library ieee;
use ieee.std_logic_1164.all;
entity reg8 is
   port (q: buffer std_logic_vector (7 downto 0);
         data: in std_logic_vector (7 downto 0);
         clk, rst: in std_logic);
end reg8;

architecture reg8_design of reg8 is -- Eight bit register
begin
-- <Eight bit register functionality>
end reg8_design;

library ieee;
use ieee.std_logic_1164.all;
entity rotate is
   port (q: buffer std_logic_vector (7 downto 0);
         data: in std_logic_vector (7 downto 0);
         clk, rst, r_l: in std_logic);
end rotate;

architecture rotate_design of rotate is
begin
-- Rotates bits or loads
-- When r_l is high, it rotates; if low, it loads data
-- <Rotatation functionality>
end rotate_design;
```

Step 2 – Declare the Components

Components are declared in the declarative region of the architecture with a component declaration statement.

The component declaration syntax is:

component *entity_name*
  port (*port_list* **) ;**
end component ;

The entity_name and port_list of the component must match exactly that of the entity you will be instantiating.

Example

```
architecture structural of top_level_design is
-- Component declarations are placed here in the
-- declarative region of the architecture.
```

```
component muxhier  -- Component declaration for mux
   port (outvec: out std_logic_vector (7 downto 0);
         a_vec, b_vec: in std_logic_vector (7 downto 0);
         sel: in std_logic);
end component;

component reg8  -- Component declaration for reg8
   port (q: out std_logic_vector (7 downto 0);
         data: in std_logic_vector (7 downto 0);
         clk, rst: in std_logic);
end component;

component rotate -- Component declaration for rotate
   port (q: buffer std_logic_vector (7 downto 0);
         data: in std_logic_vector (7 downto 0);
         clk, rst, r_l: in std_logic);
end component;
begin
-- The structural description goes here.
end structural;
```

Step 3 – Instantiate the Components

Use the following syntax to instantiate your components:

unique_instance_name **:** *component_name*
[**generic map (***override_generic_values* **)**]
port map (*port_connections* **) ;**

You can connect signals either with positional mapping (the same order declared in the entity) or with named mapping (where you specify the names of the lower-level signals to connect). Connecting by name minimizes errors, and especially advantageous when the component has many ports. To use configuration specification and declaration, refer to Configuration Specification and Declaration, on page 10-62.

Example

```
library ieee;
use ieee.std_logic_1164.all;
entity top_level is
   port (q: buffer std_logic_vector (7 downto 0);
         a, b: in std_logic_vector (7 downto 0);
         sel, r_l, clk, rst: in std_logic);
end top_level;
```

```
architecture structural of top_level is
-- The component declarations shown in Step 2 go here.
-- Declare the internal signals here
signal mux_out, reg_out: std_logic_vector (7 downto 0);

begin
-- The structural description goes here.
-- Instantiate a mux, name it inst1, and wire it up.
-- Map (connect) the ports of the mux using positional association.
inst1:  muxhier port map (mux_out, a, b, sel);

-- Instantiate a rotate, name it inst2, and map its ports.
inst2: rotate port map (q, reg_out, clk, r_l, rst);

-- Instantiate a reg8, name it inst3, and wire it up.
-- reg8 is connected with named association.
-- The port connections can be given in any order.
-- Notice that the actual (local) signal names are on
-- the right of the '=>' mapping operators, and the
-- formal signal names from the component
-- declaration are on the left.
inst3: reg8 port map (
   clk => clk,
   data => mux_out,
   q => reg_out,
   rst => rst);

end structural;
```

Configuration Specification and Declaration

A configuration declaration or specification can be used to define binding information of component instantiations to design entities (entity-architecture pairs) in a hierarchical design. After the structure of one level of a design has been fully described using components and component instantiations, a designer must describe the hierarchical implementation of each component.

A configuration declaration or specification can also be used to define binding information of design entities (entity-architecture pairs) that are compiled in different libraries.

This section discusses usage models of the configuration declaration statement supported by the synthesis tool. The following topics are covered:

- Configuration Specification
- Configuration Declaration

Component declarations and component specifications are not required for a component instantiation where the component name is the same as the entity name. In this case, the entity and its last architecture denote the default binding. In direct-entity instantiations, the binding information is available as the entity is specified, and the architecture is optionally specified. Configuration declaration and/or configuration specification are required when the component name does not match the entity name. If configurations are not used in this case, VHDL simulators give error messages, and the synthesis tool creates a black box and continues synthesis.

Configuration Specification

A configuration specification associates binding information with component labels that represent instances of a given component declaration. A configuration specification is used to bind a component instance to a design entity, and to specify the mapping between the local generics and ports of the component instance and the formal generics and ports of the entity. Optionally, a configuration specification can bind an entity to one of its architectures. The synthesis tool supports a subset of configuration specification commonly used in RTL synthesis; this section discusses that support.

The following Backus-Naur Form (BNF) grammar is supported (VHDL-93 LRM pp.73-79):

configuration_specification ::=

for component_specification binding_indication **;**

component_specification ::=

instantiation_list : component_name

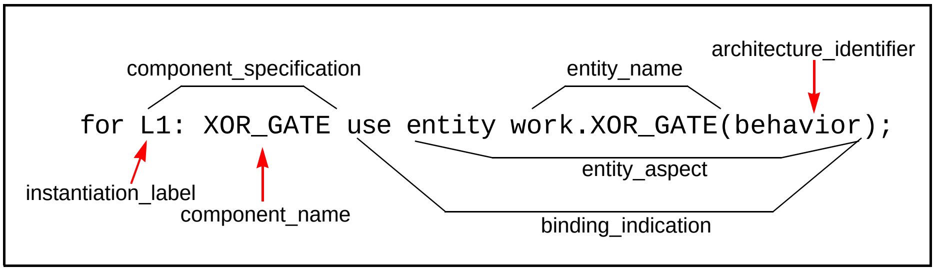
instantiation_list ::=

instantiation_label {, instantiation_label } | **others** | **all**

binding_indication ::= [**use** entity_aspect]

entity_aspect ::=

entity entity_name [(architecture_identifier)] |
configuration configuration_name



```
for others: AND_GATE use entity work.AND_GATE(structure);
for all: XOR_GATE use entity work.XOR_GATE;
```

Example: Configuration Specification

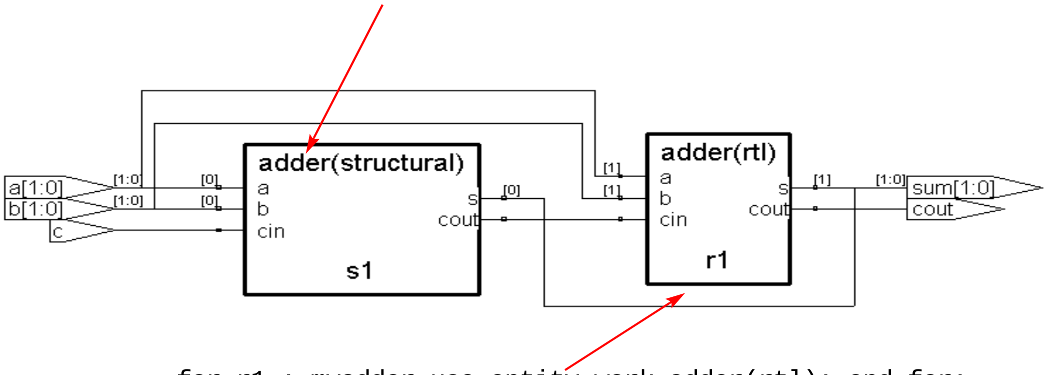
In the following example, two architectures (RTL and structural) are defined for an adder. There are two instantiations of an adder in design top. A configuration statement defines the adder architecture to use for each instantiation.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity adder is
   port ( a : in std_logic;
          b : in std_logic;
          cin : in std_logic;
          s : out std_logic;
          cout : out std_logic);
end adder;

library IEEE;
use IEEE.std_logic_unsigned.all;
architecture rtl of adder is
signal tmp : std_logic_vector(1 downto 0);
begin
   tmp <= ('0' & a) - b - cin;
   s <= tmp(0);
   cout <= tmp(1);
end rtl;

architecture structural of adder is
begin
   s <= a xor b xor cin;
   cout <= ((not a) and b and cin) or ( a and (not b) and cin)
             or (a and b and (not cin)) or ( a and b and cin);
end structural;

library IEEE;
use IEEE.std_logic_1164.all;
entity top is
   port ( a : in std_logic_vector(1 downto 0);
          b : in std_logic_vector(1 downto 0);
          c : in std_logic;
          cout : out std_logic;
          sum : out std_logic_vector(1 downto 0));
end top;
```

```
architecture top_a of top is
component myadder
   port ( a : in std_logic;
          b : in std_logic;
          cin : in std_logic;
          s : out std_logic;
          cout : out std_logic);
end component;

signal carry : std_logic;
for s1 : myadder use entity work.adder(structural);
for r1 : myadder use entity work.adder(rtl);
begin
   s1 : myadder port map ( a(0), b(0), c, sum(0), carry);
   r1 : myadder port map ( a(1), b(1), carry, sum(1), cout);
end top_a;
```

Results

```
for s1 : myadder use entity work.adder(structural); end for;
```



```
for r1 : myadder use entity work.adder(rtl); end for;
```

Unsupported Constructs for Configuration Specification

The following are the configuration specification constructs that are *not* supported by the synthesis tool. Appropriate messages are issued in the log file when these constructs are used.

1. The VHDL-93 LRM defines the binding indication in the configuration statement as:

    binding_indication ::=

        [**use** entity_aspect]

        [generic_map_aspect]

        [port_map_aspect]

    The synthesis tool supports the entity_aspect of a binding indication. It does not yet support the generic_map_aspect and port_map_aspect. If used, a warning is issued, and the tool ignores the generic and port maps.

2. The VHDL-93 LRM defines entity_aspect in the binding indication as:

    entity_aspect ::=

        entity entity_name [(architecture_identifier)] **|**
        configuration configuration_name **| open**

    The synthesis tool supports entity_name and configuration_name in the entity_aspect of a binding indication. The tool does not yet support the open construct.

Configuration Declaration

Configuration declaration specifies binding information of component instantiations to design entities (entity-architecture pairs) in a hierarchical design. Configuration declaration can bind component instantiations in an architecture, in either a block statement, a for…generate statement or an if…generate statement. It is also possible to bind different entity-architecture pairs to different indices of a for…generate statement.

The synthesis tool supports a subset of configuration declaration commonly used in RTL synthesis. The following Backus-Naur Form (BNF) grammar is supported (VHDL-93 LRM pp.11-17):

configuration_declaration ::=

configuration identifier **of** entity_name **is**

block_configuration

```
        end [ configuration ] [configuration_simple_name ] ;

block_configuration ::=

        for block_specification

            { configuration_item }

        end for ;

block_specification ::=

        achitecture_name |  block_statement_label |
        generate_statement_label [ ( index_specification ) ]

index_specification ::=

        discrete_range |  static_expression

configuration_item ::=

        block_configuration |  component_configuration

component_configuration ::=

        for component_specification
            [ binding_indication ; ]
            [ block_configuration ]
        end for ;
```

The BNF grammar for component_specification and binding_indication is described in Configuration Specification, on page 10-62.

Example 1: Configuration Declaration

The following example shows a configuration declaration describing the binding in a 3-level hierarchy, for…generate statement labeled label1, within block statement blk1 in architecture arch_gen3. Each architecture implementation of an instance of my_and1 is determined in the configuration declaration and depends on the index value of the instance in the for…generate statement.

```
entity and1 is
   port(a,b: in bit ; o: out bit);
end;

architecture and_arch1 of and1 is
begin
   o <= a and b;
end;

architecture and_arch2 of and1 is
begin
   o <= a and b;
end;

architecture and_arch3 of and1 is
begin
   o <= a and b;
end;

library WORK; use WORK.all;
entity gen3_config is
   port(a,b: in bit_vector(0 to 7);
       res: out bit_vector(0 to 7));
end;

library WORK; use WORK.all;
architecture arch_gen3 of gen3_config is
   component my_and1 port(a,b: in bit ; o: out bit); end component;
begin
   label1: for i in 0 to 7 generate
      blk1: block
      begin
         a1: my_and1 port map(a(i),b(i),res(i));
      end block;
   end generate;
end;
```

```
library work;
configuration config_gen3_config of gen3_config is
   for arch_gen3  -- ARCHITECTURE block_configuration "for
block_specification"
      for label1 (0 to 3) --GENERATE block_config "for
block_specification"
         for blk1  -- BLOCK block_configuration "for
block_specification"
         -- {configuration_item}
            for a1 : my_and1 -- component_configuration
            -- Component_specification "for idList : compName"
               use entity work.and1(and_arch1);  --
binding_indication
            end for;  -- a1 component_configuration
         end for;  -- blk1 BLOCK block_configuration
      end for;  -- label1 GENERATE block_configuration
      for label1 (4) -- GENERATE block_configuration "for
block_specification"
         for blk1
            for a1 : my_and1
               use entity work.and1(and_arch3);
            end for;
         end for;
      end for;

      for label1 (5 to 7)
         for blk1
            for a1 : my_and1
               use entity work.and1(and_arch2);
            end for;
         end for;
      end for;
   end for;  -- ARCHITECTURE block_configuration
end config_gen3_config;
```

Example 2: Configuration Declaration

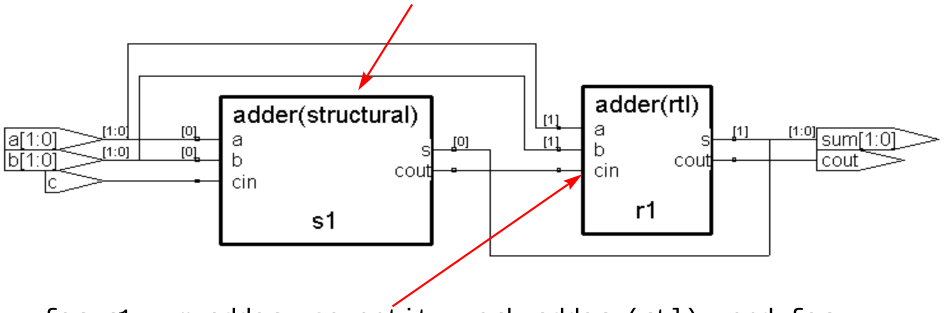
In the following example, two architectures (RTL and structural) are defined for an adder. There are two instantiations of an adder in design top. A configuration declaration defines the adder architecture to use for each instantiation. This example is similar to the configuration specification example.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity adder is
   port ( a : in std_logic;
          b : in std_logic;
          cin : in std_logic;
          s : out std_logic;
          cout : out std_logic);
end adder;

library IEEE;
use IEEE.std_logic_unsigned.all;
architecture rtl of adder is
signal tmp : std_logic_vector(1 downto 0);
begin
   tmp <= ('0' & a) - b - cin;
   s <= tmp(0);
   cout <= tmp(1);
end rtl;

architecture structural of adder is
begin
   s <= a xor b xor cin;
   cout <= ((not a) and b and cin) or ( a and (not b) and cin) or
            (a and b and (not cin)) or ( a and b and cin);
end structural;

library IEEE;
use IEEE.std_logic_1164.all;
entity top is
   port ( a : in std_logic_vector(1 downto 0);
          b : in std_logic_vector(1 downto 0);
          c : in std_logic;
          cout : out std_logic;
          sum : out std_logic_vector(1 downto 0));
end top;
```

```
architecture top_a of top is
component myadder
   port ( a : in std_logic;
          b : in std_logic;
          cin : in std_logic;
          s : out std_logic;
          cout : out std_logic);
end component;

signal carry : std_logic;
begin
   s1 : myadder port map ( a(0), b(0), c, sum(0), carry);
   r1 : myadder port map ( a(1), b(1), carry, sum(1), cout);
end top_a;

library work;
configuration config_top of top is  -- configuration_declaration
   for top_a  -- block_configuration "for block_specification"
   -- component_configuration
      for s1: myadder -- component_specification
         use entity work.adder (structural); -- binding_indication
      end for;  -- component_configuration
   -- component_configuration
      for r1: myadder -- component_specification
         use entity work.adder (rtl); -- binding_indication
      end for;  -- component_configuration
   end for;  -- block_configuration
end config_top;
```

Results

```
for s1 : myadder use entity work.adder (structural); end for;
```



```
for r1 : myadder use entity work.adder (rtl); end for;
```

Unsupported Constructs for Configuration Declaration

The following are the configuration declaration constructs that are *not* supported by the synthesis tool. Appropriate messages are displayed in the log file if these constructs are used.

1. The VHDL-93 LRM defines the configuration declaration as:

configuration_declaration ::=

> **configuration** identifier **of** entity_name **is**
> configuration_declarative_part
> block_configuration
> **end** [**configuration**] [configuration_simple_name] **;**

configuration_declarative_part ::= { configuration_declarative_item }

configuration_declarative_item ::=

> use_clause **|** attribute_specification **|** group_declaration

The synthesis tool does not support the configuration_declarative_part. It parses the use_clause and attribute_specification without any warning message. The group_declaration is not supported and an error message is issued.

2. VHDL-93 LRM defines entity aspect in the binding indication as:

entity_aspect ::=

> **entity** entity_name [(architecture_identifier)] **|**
> **configuration** configuration_name **| open**

block_configuration ::=

> **for** block_specification
> { use_clause }
> { configuration_item }
> **end for ;**

The synthesis tool does not support use_clause in block_configuration. This construct is parsed and ignored.

Scalable Designs

This section describes creating and using scalable VHDL designs.

You can create a VHDL design that is scalable, meaning that it can handle a user-specified number of bits or components. Any of these methods can be used to create scalable designs:

- Unconstrained vector ports
- VHDL generics
- VHDL generate statements.

Creating a Scalable Design Using Unconstrained Vector Ports

Do not size (constrain) the ports until you need them. This first example is coding the adder using the - operator, and gives much better synthesized results than the second and third scalable design examples, which code the adders as random logic.

Example

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity addn is
-- Notice that a, b, and result ports are not constrained.
-- In VHDL, they automatically size to whatever is connected
-- to them.
port(result : out std_logic_vector;
   cout : out std_logic;
   a, b : in std_logic_vector;
   cin : in std_logic);
end addn;

architecture stretch of addn is
   signal tmp : std_logic_vector (a'length downto 0);
begin
-- The next line works because "-" sizes to the largest operand
-- (also, you need only pad one argument).
tmp <=  ('0' & a) - b - cin;
```

```
result <= tmp(a'length - 1 downto 0);
cout <= tmp(a'length);
assert result'length = a'length;
assert result'length = b'length;
end stretch;

-- Top level design
-- Here is where you specify the size for a, b,
-- and result. It is illegal to leave your top
-- level design ports unconstrained.

library ieee;
use ieee.std_logic_1164.all;
entity addntest is
   port (result : out std_logic_vector (7 downto 0);
         cout : out std_logic;
         a, b : in std_logic_vector (7 downto 0);
         cin : in std_logic);
end addntest;

architecture top of addntest is
component addn
   port (result : std_logic_vector;
         cout : std_logic;
         a, b : std_logic_vector;
         cin : std_logic);
end component;

begin
test : addn port map (
   result => result,
   cout => cout,
   a => a,
   b => b,
   cin => cin
               );
end;
```

Creating a Scalable Design Using VHDL Generics

Create a VHDL generic with default value. The generic is used to represent bus sizes inside a architecture, or a number of components. You can define more than one generic per declaration by separating the generic definitions with semicolons (;).

Syntax

generic (*generic_1_name* **:** *type* [**:=** *default_value*]**) ;**

Examples

```
generic (num : integer := 8) ;
generic (top : integer := 16; num_bits : integer := 32);
```

Using a Scalable Architecture with VHDL Generics

Instantiate the scalable architecture, and override the default generic value with the generic map statement.

Syntax

generic map (*list_of_overriding_values* **)**

Examples

Generic map construct

```
generic map (16)
-- These values will get mapped in order given.
generic map (8, 16)
```

Creating a scalable adder

```
library ieee;
use ieee.std_logic_1164.all;
entity adder is
   generic(num_bits : integer := 4);-- Default adder
              -- Size is 4 bits
   port (a : in std_logic_vector (num_bits downto 1);
         b : in std_logic_vector (num_bits downto 1);
         cin : in std_logic;
         sum : out std_logic_vector (num_bits downto 1);
         cout : out std_logic);
end adder;

architecture behave of adder is
begin
process (a, b, cin)
   variable vsum : std_logic_vector (num_bits downto 1);
   variable carry : std_logic;


begin
   carry := cin;
   for i in 1 to num_bits loop
      vsum(i) := (a(i) xor b(i)) xor carry;
      carry := (a(i) and b(i)) or (carry and (a(i) or b(i)));
   end loop;
   sum <= vsum;
   cout <= carry;
end process;
end behave;
```

Scaling the Adder by Overriding the generic Statement

```
library ieee;
use ieee.std_logic_1164.all;
entity adder16 is
   port (a : in std_logic_vector (16 downto 1);
         b : in std_logic_vector (16 downto 1);
         cin : in std_logic;
         sum : out std_logic_vector (16 downto 1);
         cout : out std_logic);
end adder16;

architecture behave of adder16 is
-- The component declaration goes here.
-- This allows you to instantiate the adder.
component adder
-- The default adder size is 4 bits.
```

```
   generic(num_bits : integer := 4);
   port (a : in std_logic_vector ;
         b : in std_logic_vector;
         cin : in std_logic;
         sum : out std_logic_vector;
         cout : out std_logic);
end component;

begin
my_adder : adder
   generic map (16) -- Use a 16 bit adder
   port map(a, b, cin, sum, cout);
end behave;
```

Creating a Scalable Design Using Generate Statements

A VHDL generate statement allows you to repeat logic blocks in your design without having to write the code to instantiate each one individually.

Creating a 1-bit Adder

```
library ieee;
use ieee.std_logic_1164.all;
entity adder is
   port (a, b, cin : in std_logic;
         sum, cout : out std_logic);
end adder;

architecture behave of adder is
begin
   sum <= (a xor b) xor cin;
cout <= (a and b) or (cin and a) or (cin and b);
end behave;
```

Instantiating the 1-bit Adder Many Times with a Generate Statement

```
library ieee;
use ieee.std_logic_1164.all;
entity addern is
   generic(n : integer := 8);
   port (a, b : in std_logic_vector (n downto 1);
         cin : in std_logic;
         sum : out std_logic_vector (n downto 1);
         cout : out std_logic);
end addern;
```

```
architecture structural of addern is
-- The adder component declaration goes here.
component adder
   port (a, b, cin : in std_logic;
         sum, cout : out std_logic);
end component;

signal carry : std_logic_vector (0 to n);
begin
-- Generate instances of the single-bit adder n times.
-- You need not declare the index 'i' because
-- indices are implicitly declared for all FOR
-- generate statements.

gen: for i in 1 to n generate
   add: adder port map(
      a => a(i),
      b => b(i),
      cin => carry(i - 1),
      sum => sum(i),
      cout => carry(i));
end generate;

carry(0) <= cin;
cout <= carry(n);

end structural;
```

RAM Inference

The synthesis tool can infer synchronous and synchronously resettable RAMs from your VHDL source code and, where appropriate, generate technology-specific single or dual-port RAMs. No special input such as attributes or directives in your source code is needed.

The RTL-level RAM inferred by the compiler always has an asynchronous READ. During synthesis, the mappers convert this to the appropriate technology-specific RAM primitives. Depending on the technology used, the synthesized design can include RAM primitives with either synchronous or asynchronous READs. See Synchronous READ RAMs, on page 10-83, for information on coding your Verilog description to ensure that technology-specific RAM primitives with synchronous READs are used.

Example

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

entity ram_test is
port (
   d : in std_logic_vector(7 downto 0);
   a : in std_logic_vector(6 downto 0);
   we : in std_logic;
   clk : in std_logic;
   q : out std_logic_vector(7 downto 0));
end ram_test;

architecture rtl of ram_test is
type mem_type is array (127 downto 0) of
   std_logic_vector (7 downto 0);

signal mem: mem_type;
begin
process (clk)
   begin
      if rising_edge(clk) then
         if (we = '1') then
            mem(conv_integer (a)) <= d;
         end if;
      end if;
end process;

q <= mem(conv_integer (a));

end rtl ;
```

RAMs with Special Write Enables

The synthesis tool can infer RAMs when the Write Enable is tied to Vcc or when the Write Enable is nested within if statements.

always-enabled Write Enable

The RAM extraction code supports the inference of RAMs with their Write Enable tied permanently to Vcc rather than implementing the logic in flip-flops.

Example

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

entity WEtrue is
   generic (ram_depth : integer := 1024;
            addr_width : integer := 10;
            data_width : integer := 8);
   port (clk   : in std_logic;
         addr  : in std_logic_vector(addr_width-1 downto 0);
         data_in : in std_logic_vector(data_width-1 downto 0);
         data_out: out std_logic_vector(data_width-1 downto 0));
end WEtrue;

architecture rtl of WEtrue is
   type ram_type is array(ram_depth-1 downto 0)of
   std_logic_vector(data_width-1 downto 0);
signal ram : ram_type;
signal we : std_logic;

begin

   we <= '1';

   process(clk)
   begin
      if(clk'event and clk='1') then
         if(we = '1') then
            ram(CONV_INTEGER(addr)) <= data_in;
         end if;
      end if;
   end process;

   data_out <= ram(CONV_INTEGER(addr));

end rtl;
```

Nested Write Enable

The RAM extraction code can infer RAMs when the Write Enable is more complex as found in nested IF statements. The compilers extract common terms from the feedback MUX Enables to derive the common Write Enable signal.

Example

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

entity WEnestedif is
   generic (ram_depth : integer := 1024;
            addr_width : integer := 10;
            data_width : integer := 8);
   port( addr  : in std_logic_vector(addr_width-1 downto 0);
         data_in : in std_logic_vector(data_width-1 downto 0);
         we1, we2, clk: in std_logic;
         data_out : out std_logic_vector(data_width-1 downto 0));
end WEnestedif;

architecture rtl of WEnestedif is
   type ram_type is array(ram_depth-1 downto 0)of
   std_logic_vector(data_width-1 downto 0);
signal ram : ram_type;

begin
   data_out <= ram(conv_integer(addr));
      process(clk, addr, we1, we2)
      begin
         if(clk'event and clk='1') then
            if(we1 = '1') then
               if (we2 = '1') then
               ram(conv_integer(addr)) <= data_in;
               end if;
            end if;
         end if;
      end process;
end rtl;
```

Limited RAM Resources

If your RAM resources are limited, designate additional instances of inferred RAMs as flip-flops and logic using the syn_ramstyle attribute. This attribute takes the string argument of registers, placed on the RAM instance.

Additional Components Generated

After inferring a RAM for some technologies, you might notice that the synthesis tool has generated a few additional components adjacent to the RAM. These components assure accuracy in your post placement and routing simulation results.

Synchronous READ RAMs

All RAM primitives that the synthesis tool generates for inferred RAMs have asynchronous READs.

The following examples shows how the READ address or the data output of the RAM can be registered to ensure generation of a synchronous READ RAM.

Example: READ Address Registered

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity ram_test is
port (
   d : in std_logic_vector(7 downto 0);
   a : in std_logic_vector(6 downto 0);
   we : in std_logic;
   clk : in std_logic;
   q : out std_logic_vector(7 downto 0));
end ram_test;
architecture rtl of ram_test is
type mem_type is array (127 downto 0) of
   std_logic_vector (7 downto 0);
signal mem: mem_type;
signal read_add : std_logic_vector(6 downto 0);
begin
   process (clk)
   begin
      if rising_edge(clk) then
         if (we = '1') then
            mem(conv_integer(a)) <= d;
         end if;
         read_add <= a;
      end if;
   end process;
```

```
        q <= mem(conv_integer(read_add));
        end rtl ;
```

Example: Data Output Registered

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

entity ram_test is
   port ( d: in std_logic_vector(7 downto 0);
          a: in integer range 127 downto 0;
         we: in std_logic;
         clk: in std_logic;
         q: out std_logic_vector(7 downto 0));
end ram_test;

architecture rtl of ram_test is

type mem_type is array (127 downto 0) of std_logic_vector (7 downto 0);
signal mem: mem_type;

begin
     process(clk)
    begin
        if (clk'event and clk='1') then
            q <= mem(a);
            if (we='1') then
                mem(a) <= d;
            end if;
       end if;
     end process;

end rtl;
```

ROM Inference

As part of BEST (Behavioral Extraction Synthesis Technology) feature, the synthesis tool infers ROMs (read-only memories) from your HDL source code, and generates block components for them in the RTL view.

The data contents of the ROMs are stored in a text file named `rom.info`. To quickly view `rom.info` in read-only mode, synthesize your HDL source code, open an RTL view, then push down into the ROM component. For an example of the ROM data, refer to ROM Table Data (rom.info File), on page 9-48.

Generally, the synthesis tool infers ROMs from HDL source code that uses case statements, or equivalent if statements, to make 16 or more signal assignments using constant values (words). The constants must all be the same width.

Another requirement for ROM inference is that values must be specified for at least half of the address space. For example, if the ROM has 5 address bits, then the address space is 32 and at least 16 of the different addresses must be specified.

Example

```
library ieee;
use ieee.std_logic_1164.all;
entity rom4 is
   port (a : in std_logic_vector(4 downto 0);
         z : out std_logic_vector(3 downto 0)
         );
end rom4;

architecture behave of rom4 is
begin
   process(a)
   begin
      if a = "00000" then
         z <= "0001";
      elsif a = "00001" then
         z <= "0010";
      elsif a = "00010" then
         z <= "0110";
      elsif a = "00011" then
         z <= "1010";
      elsif a = "00100" then
```

```
         z <= "1000";
      elsif a = "00101" then
         z <= "1001";
      elsif a = "00110" then
         z <= "0000";
      elsif a = "00111" then
         z <= "1110";
      elsif a = "01000" then
         z <= "1111";
      elsif a = "01001" then
         z <= "1110";
      elsif a = "01010" then
         z <= "0001";
      elsif a = "01011" then
         z <= "1000";
      elsif a = "01100" then
         z <= "1110";
      elsif a = "01101" then
         z <= "0011";
      elsif a = "01110" then
         z <= "1111";
      elsif a = "01111" then
         z <= "1100";
      elsif a = "10000" then
         z <= "1000";
      elsif a = "10001" then
         z <= "0000";
      elsif a = "10010" then
         z <= "0011";
      else
         z <= "0111";
      end if;
   end process;
end behave;
```

ROM Table Data (rom.info file)

```
Note: This data is for viewing only

ROM work.rom4(behave)-z_1[3:0]
address width: 5
data width: 4
inputs:
0: a[0]
1: a[1]
2: a[2]
3: a[3]
4: a[4]
outputs:
0: z_1[0]
1: z_1[1]
2: z_1[2]
3: z_1[3]

data:
00000 -> 0001
00001 -> 0010
00010 -> 0110
00011 -> 1010
00100 -> 1000
00101 -> 1001
00110 -> 0000
00111 -> 1110
01000 -> 1111
01001 -> 1110
01010 -> 0001
01011 -> 1000
01100 -> 1110
01101 -> 0011
01110 -> 0010
01111 -> 0010
10000 -> 0010
10001 -> 0010
10010 -> 0010
default -> 0111
```

Instantiating Black Boxes in VHDL

Black boxes are design units with just the interface specified; internal information is ignored by the synthesis tool. Black boxes can be used to directly instantiate:

- Technology-vendor primitives and macros (including I/Os).
- User-defined macros whose functionality was defined in a schematic editor, or another input source (when the place-and-route tool can merge design netlists from different sources).

Black boxes are specified with the syn_black_box synthesis directive, in conjunction with other directives. If the black box is a technology-vendor I/O pad, use the black_box_pad_pin directive instead.

Here is a list of the directives that you can use to specify modules as black boxes, and to define design objects on the black box for consideration during synthesis:

- syn_black_box
- black_box_pad_pin
- black_box_tri_pins
- syn_isclock
- syn_tco<*n*>
- syn_tpd<*n*>
- syn_tsu<*n*>

For descriptions of the black-box attributes and directives, see Chapter 8, *Synthesis Attributes and Directives*.

For information on how to instantiate black boxes and technology-vendor I/Os, see Defining Black Boxes for Synthesis, on page 5-22 of the *Synplify User Guide*.

Black-Box Timing Constraints

You can provide timing information for your individual black box instances. The following are the three predefined timing constraints available for black boxes.

- syn_tpd<*n*> – Timing propagation for combinational delay through the black box.
- syn_tsu<*n*> – Timing setup delay required for input pins (relative to the clock).
- syn_tco<*n*>– Timing clock to output delay through the black box.

Here, *n* is an integer from 1 through 10, inclusive. See Black-box Timing Models, on page 7-44, for details about constraint syntax.

Synthesis Attributes and Directives

VHDL synthesis attributes and directives allow you to associate information with your design to control the way it is analyzed, compiled, and mapped.

- *Attributes* direct the way your design is optimized and mapped during synthesis. Although you can place synthesis attributes directly in your source code, specify them in a constraint file instead, with the Attributes panel of the SCOPE spreadsheet. That way, changes can be made (to the constraint file), without requiring you to recompile.
- *Directives* control the way your design is analyzed prior to synthesis. Because of this, they must be included directly in your VHDL source code and cannot be specified in a constraint file.

Directives must be specified in source code; attributes can be specified in a constraint file or source code.

In addition to the general attributes and directives described in Chapter 8, *Synthesis Attributes and Directives*, the Synplify synthesis tool also supports vendor-specific directives and attributes that apply to the specific programmable logic vendor you are targeting for your design.

The VHDL attributes and directives are predefined in a package provided in the synthesis tool library (`*installation_dir*/lib/vhd/synattr.vhd`). You typically use this package, however, you can redefine the attributes and directives each time you include them in source code. The library contains the built-in attributes, along with declarations for timing constraints (including black-box timing constraints) and vendor-specific attributes. To access the library, add the following to the beginning of each of the VHDL design units that uses the attributes:

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

VHDL Synthesis Examples

This section describes the VHDL examples that are provided with the synthesis tool.

Combinational Logic Examples

The following combinational logic synthesis examples are included in the `installation_dir/examples/vhdl/combinat` directory:

- Adders
- ALU
- Bus Sorter (illustrates using procedures in VHDL)
- 3-to-8 Decoders
- 8-to-3 Priority Encoders
- Comparator
- Interrupt Handler (coded with an if-then-else statement for the desired priority encoding)
- Multiplexers (concurrent signal assignments, case statements, or if-then-else statements can be used to create multiplexers; the synthesis tool automatically creates parallel multiplexers when the conditions in the branches are mutually exclusive)
- Parity Generator
- Tristate Drivers

Sequential Logic Examples

The following sequential logic synthesis examples are included in the `*installation_dir*/examples/vhdl/sequentl` directory:

- Flip-flops and level-sensitive latches
- Counters (up, down, and up/down)
- Register file
- Shift register
- State machines

For additional information on synthesizing flip-flops and latches, see:

- Creating Flip-flops and Registers Using VHDL Processes, on page 10-37
- Level-sensitive Latches Using Concurrent Signal Assignments, on page 10-41
- Level-sensitive Latches Using VHDL Processes, on page 10-42
- Asynchronous Sets and Resets, on page 10-46
- Synchronous Sets and Resets, on page 10-47

PREP VHDL Benchmarks

PREP (Programmable Electronics Performance) Corporation distributes benchmark results that show how FPGA vendors compare with each other in terms of device performance and area.

The following PREP benchmarks are included in the `*installation_dir*/examples/vhdl/prep` directory:

- PREP VHDL Benchmark 1. Data Path
- PREP VHDL Benchmark 2. Timer/Counter
- PREP VHDL Benchmark 3. Small State Machine
- PREP VHDL Benchmark 4. Large State Machine
- PREP VHDL Benchmark 5. Arithmetic Circuit
- PREP VHDL Benchmark 6. 16-Bit Accumulator
- PREP VHDL Benchmark 7. 16-Bit Counter
- PREP VHDL Benchmark 8. 16-Bit Pre-scaled Counter
- PREP VHDL Benchmark 9. Memory Map

The source code for the benchmarks can be used for design examples for synthesis or for doing your own FPGA vendor comparisons.

PREP Corp. has disbanded, but you can still obtain information from their Web site: www.prep.org.

CHAPTER 11

Designing with Actel

The following topics describe how to design and synthesize with the Actel technology:

Using Actel

This section describes the use of the synthesis tool with Actel devices. It describes

Actel Device-specific Support

The Synplify synthesis tool creates technology-specific netlists for a number of Actel families of FPGAs. New devices are added on an ongoing basis. For the most current list of supported devices, check the Device panel of the Options for implementation dialog box (see Device Panel, on page 3-30).

After synthesis, the synthesis tool generates EDIF netlists ready for the Actel Designer Series place-and-route tool.

Actel Features

The Synplify synthesis tool contains the following Actel-specific features:

- Direct mapping to Actel c-modules and s-modules
- Timing-driven mapping, replication, and buffering
- Inference of counters, adders, and subtractors; module generation
- Automatic use of clock buffers for clocks and reset signals
- Automatic I/O insertion. See *I/O Insertion*, on page 11-7 for more information.

Constraints and Attributes

Optimize your design by using constraint files. Constraint files can contain timing constraints, general attributes, and Actel-specific attributes. Use the SCOPE spreadsheet to create and manage the files. Add the constraint files to the project list along with your HDL source files. For more information about constraint files, refer to the chapter on timing constraints. For a list of the Actel attributes, see *Actel Attribute and Directive Summary*, on page 11-11.

Synthesis Reports

The Synplify synthesis tool generates a resource usage report, a timing report, and a net buffering report for the Actel designs that you synthesize.To view the synthesis reports, click View Log.

Instantiating Macros and Black Boxes in Actel Designs

You can instantiate ACTgen macros or other Actel macros like gates, counters, flip-flops, or I/Os by using the supplied Actel macro libraries to predefine the Actel macro black boxes.

For general information on instantiating black boxes, see Instantiating Black Boxes in VHDL, on page 10-88, and Instantiating Black Boxes in Verilog, on page 9-49. For specific procedures about instantiating macros and black boxes and using Actel black boxes, see the following sections in the *User Guide*:

- Defining Black Boxes for Synthesis, on page 5-22
- Using Predefined Actel Black Boxes, on page 6-5
- Using ACTGen Macros, on page 6-5

Actel Device Mapping Options

You select device mapping options for Actel technologies, select Project -> Implementation Options->Device and set the options.

Option	For details, see...
Fanout Guide	*Fanout Limits*, on page 11-4
Hard Limit to Fanout	*Fanout Limits*, on page 11-4
Disable I/O Insertion	*I/O Insertion*, on page 11-7
Maximum Number of Critical Paths in SDF (certain technologies)	*Number of Critical Paths*, on page 11-7
Set Operating Condition Device options (certain technologies)	*Operating Condition Device Option*, on page 11-7

Fanout Limits

Large fanouts can cause large delays and routability problems. During technology mapping, the Synplify synthesis tool automatically maintains reasonable fanout limits, keeping the fanout under the limit you specify. You specify fanout limits for the whole design with Project->Implementation Options-> Device (*Fanout Implementation Options*, on page 11-5). Additionally, you can use the syn_maxfan attribute (*The syn_maxfan Attribute in Actel Designs*, on page 11-5) in certain Actel technologies to set lower-level fanout limits that override the global limit. For a procedure, see Setting Fanout Limits, on page 5-7 of the *Synplify User Guide*.

The synthesis tool first reduces fanout by replicating the driver of the high fanout net and splitting the net into segments. Replication can affect the number of register bits in your design. If replication is not possible, the algorithm buffers the signals. Buffering is more expensive in terms of intrinsic delay and resource consumption, and is therefore not used unless the fanout exceeds the guideline. (See *Buffering vs Replication*, on page 11-6 for further details.) Timing also affects the fanout limit for a net. Critical nets can be replicated or buffered more aggressively.

The log file contains the net buffering report that shows how many nets were buffered or had their sources replicated and the number of segments created for each net.

Fanout limits can be hard or soft. The software does not exceed a hard fanout limit. The software treats a soft limit as a guideline that it tries to stay under, but it might not honor the limit if there are excessive speed or area costs.

Fanout Implementation Options

When you select Project->Implementation Options->Device, you can set two global fanout options for the whole design:

Fanout Guide	This value sets a soft limit for the number of fanouts for a given driver. It is used globally, and the software tries to honor the limit for the whole design. The default varies depending on which Actel technology you use.
Hard Limit to Fanout	Checking this option makes the value you set in Fanout Guide a hard fanout limit that the software will not exceed.

The syn_maxfan Attribute in Actel Designs

For the ProASIC (500K), and ProASIC PLUS (PA), and ProASIC3/3E technologies, the syn_maxfan attribute is used to control the maximum fanout of the design, or an instance, net, or port. The limit specified by this attribute is treated as a hard or soft limit depending on where it is specified. The following rules described the behavior:

- Global fanout limits are usually specified with the fanout guide options (Project->Implementation Options->Device), but you can also use the syn_maxfan attribute on a top-level module or view to set a global soft limit. This limit may not be honored if the limit degrades performance. To set a global hard limit, you must use the Hard Limit to Fanout option.

- A syn_maxfan attribute can be applied locally to a module or view. In this case, the limit specified is treated as a soft limit for the scope of the module. This limit overrides any global fanout limits for the scope of the module.
- When a syn_maxfan attribute is specified on an instance that is not of primitive type inferred by Synplicity compiler, the limit is considered a soft limit which is propagated down the hierarchy. This attribute overrides any global fanout limits.
- When a syn_maxfan attribute is specified on a port, net, or register (or any primitive instance), the limit is considered a hard limit. This attribute overrides any other global fanout limits. Note that the syn_maxfan attribute does not prevent the instance from being optimized away and that design rule violations resulting from buffering or replication are the responsibility of the user.

Buffering vs Replication

The fanout of an instance or a net can be reduced by either replicating the driver of the net that violates the maxfan limit or by creating a buffer tree. Replication or buffering depends on where the syn_maxfan attribute is specified and also on other attributes such as syn_replicate. The set of rules given below describes when to replicate and when to create buffer tree.

- When a syn_maxfan attribute is set on a port or on a net driven by a port (or an I/O pad), a buffer tree is always created to honor the maxfan limit.
- When the value of a syn_replicate attribute is '0', a buffer tree is created. Note that the syn_replicate attribute must be used in conjunction with the syn_maxfan attribute. Otherwise it has no meaning for the Actel family. The syn_replicate attribute is used only to turn off the replication.
- When a syn_keep or syn_preserve attribute is specified on a net driver, a buffer tree is created.
- When a net driver is not a primitive gate or register, a buffer tree is created. (only registers are replicated).

I/O Insertion

The Synplify synthesis tool inserts I/O pads for inputs, outputs, and bidirectionals in the output netlist unless you disable I/O insertion. You can control I/O insertion with the Disable I/O Insertion option (Project->Implementation Options->Device).

If you do not want to automatically insert any I/O pads, check the Disable I/O Insertion box (Project->Implementation Options->Device). This is useful to see how much area your blocks of logic take up, before synthesizing an entire FPGA. If you disable automatic I/O insertion, you will not get any I/O pads in your design unless you manually instantiate them yourself.

If you disable I/O insertion, you can instantiate the Actel I/O pads you need directly. If you manually insert I/O pads, you only insert them for the pins that require them .

Number of Critical Paths

The Max number of critical paths in SDF option (Project->Implementation Options->Device) is only available only for the ProASIC (500K), ProASIC PLUS (PA), and ProASIC3/3E technologies. It lets you set the maximum number of critical paths in a forward-annotated constraint file (`.sdf`). The `.sdf` file displays a prioritized list of the worst-case paths in a design. Actel Designer prioritizes routing to ensure that the worst-case paths are routed efficiently.

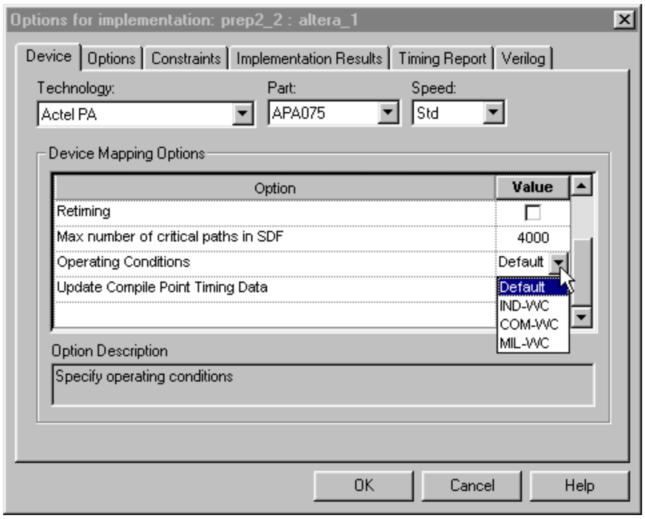
The default value for the number of critical paths that are forward-annotated is 4000. Various design characteristics affect this number, so experiment with a range of values to achieve the best circuit performance possible.

Operating Condition Device Option

You can now specify an operating condition for ProASIC (500K) and ProASIC Plus (PA) technologies. Different operating conditions cause differences in device performance. The operating condition affects the following:

- optimization, if you have timing constraints
- timing analysis
- timing reports

To set an operating condition, select the value for Operating Conditions from the menu on the Device tab of the Implementation Options dialog box.



Technology: Actel PA
Part: APA075
Speed: Std

Device Mapping Options

Option	Value
Retiming	☐
Max number of critical paths in SDF	4000
Operating Conditions	Default
Update Compile Point Timing Data	

Default
IND-WC
COM-WC
MIL-WC

Option Description
Specify operating conditions

To set an operating condition in a project or Tcl file, use the command:

```
set_option -opcond value
```

where *value* can be one of the following:

Default	Typical timing
MIL-WC	Worst-case Military timing
IND-WC	Worst-case Industrial timing
COM-WC	Worst-case Commercial timing
Automotive-WC	Worst-case Automotive timing

Note that even when a particular operating condition is valid for a family, it may not be applicable to every part/package/speed-grade combination in that family. Consult Actel's documentation or software for information on valid combinations and more information on the meaning of each operating condition.

Actel-specific Tcl Command Options

This section describes the use of Tcl command options with Actel devices.

set_option Command for Actel

You can use the set_option Tcl command to specify the same device mapping options as are available through the Options for implementation dialog box displayed in the Project view with Project -> Implementation Options (see Implementation Options Command, on page 3-29).

This section describes the Actel-specific set_option Tcl command options. These include the target technology, device architecture, and synthesis styles.

The table below provides information on specific options for Actel architectures. For a complete list of options for this command, refer to set_option, on page 5-13.

Note that you cannot specify a package (-package option) in the Synplify synthesis tool environment. You must use the Actel back end tool for this.

.

Option	Description
-technology *keyword*	Sets the target technology for the implementation. Keyword must be one of the following Actel architecture names 3200DX, 40MX, 42MX, 500K, 54SX, 54SXA, ACT1, ACT2, ACT3, AXCELERATOR, EX, and PA, and PROASIC3E. For the 1200XL architecture, use ACT2
-part *part_name*	Specifies a part for the implementation. Refer to the Options for implementation dialog box for available choices.
-speed_grade *value*	Sets the speed grade for the implementation. Refer to the Options for implementation dialog box for available choices. This option is not supported by the Actel 500K and ProAsic3E architectures.

Option	Description
-fanout_guide *value*	Sets the fanout limit guideline for the current project. If you want to set a hard limit, you must also set the -maxfan_hard option to true. For more information about fanout limits, see *Fanout Limits,* on page 11-4.
-maxfan_hard 1	Specifies that the specified -fanout_guide value is a hard fanout limit that the Synplify synthesis tool must not exceed. To set a guideline limit, see the -fanout_guide option. For more information about fanout limits, see *Fanout Limits,* on page 11-4.
-disable_io_insertion 1 | 0	Prevents (1) or allows (0) insertion of I/O pads during synthesis. The default value is false (enable I/0 pad insertion). For additional information about disabling I/O pads, see *I/O Insertion,* on page 11-7.
-opcond *value*	Sets the operating condition for device performance in the areas of optimization, timing analysis, and timing reports. This option applies only to the ProASIC (500K) and ProASIC Plus (PA) technologies. Values are Default, MIL-WC, IND-WC, COM-WC, and Automotive-WC. See *Operating Condition Device Option,* on page 11-7 for more information.
-report_path *value*	Sets the maximum number of critical paths in a forward-annotated SDF constraint file. This option applies only to the ProASIC (500K) and ProASIC Plus (PA and ProAsic3E) technologies. For information about setting critical paths, see *Number of Critical Paths,* on page 11-7.

Actel Attribute and Directive Summary

The following table summarizes the synthesis and Actel-specific attributes and directives available with the Actel technology. Complete descriptions and examples are in Chapter 8, *Synthesis Attributes and Directives*.

Attribute/Directive	Description
alsloc	Forward-annotates the relative placements of macros and IP blocks to Actel Designer. This attribute does not apply to ProASIC (500K) and ProASIC Plus (PA) designs.
alspin	Assigns scalar or bus ports to Actel I/O pin numbers. This attribute does not apply to ProASIC (500K) and ProASIC Plus (PA) designs.
alspreserve	Specifies that a net be preserved, and prevents it from being removed during place-and-route optimization. This attribute does not apply to ProASIC (500K) and ProASIC PLUS (PA) designs.
black_box_pad_pin (D)	Specifies that a pin on a black box is an I/O pad. It is applied to a component, architecture, or module, with a value that specifies the set of pins on the module or entity.
black_box_tri_pins (D)	Specifies that a pin on a black box is a tristate pin. It is applied to a component, architecture, or module, with a value that specifies the set of pins on the module or entity.
full_case (D)	Specifies that a Verilog case statement has covered all possible cases.
loop_limit (D)	Specifies a loop iteration limit for for loops.
parallel_case (D)	Specifies a parallel multiplexed structure in a Verilog case statement, rather than a priority-encoded structure.
syn_black_box (D)	Defines a black box for synthesis.
syn_direct_enable	When used as a compiler directive, this attribute marks the flip-flops with clock enables for the compiler to infer.

(D) indicates directives; all others are attributes.

Attribute/Directive	Description
syn_encoding	Specifies the encoding style for state machines.
syn_enum_encoding (D)	Specifies the encoding style for enumerated types (VHDL only).
syn_global_buffers	Sets the number of global buffers to use in a ProAsic3E design.
syn_hier	Controls the handling of hierarchy boundaries of a module or component during optimization and mapping.
syn_isclock (D)	Specifies that a black-box input port is a clock, even if the name does not indicate it is one.
syn_keep (D)	Prevents the internal signal from being removed during synthesis and optimization.
syn_maxfan	Controls the maximum fanout of an instance, net, or port.
syn_netlist_hierarchy	Determines whether the EDIF output netlist is flat or hierarchical.
syn_noarrayports	Prevents the ports in the EDIF output netlist from being grouped into arrays, and leaves them as individual signals.
syn_noclockbuf	Turns off the automatic insertion of clock buffers.
syn_noprune (D)	Controls the automatic removal of instances that have outputs that are not driven.
syn_preserve (D)	Prevents sequential optimizations across a flip-flop boundary during optimization, and preserves the signal.
syn_preserve_sr_priority	Forces set/reset flip-flops to honor the coded priority for the set or reset. ACT1 and 40MX architectures only.
syn_radhardlevel	Specifies the radiation-resistant design technique to apply to a module, architecture, or register.
syn_replicate	Controls replication.

(D) indicates directives; all others are attributes.

Attribute/Directive	Description
syn_reference_clock	Specifies a clock frequency other than that implied by the signal on the clock pin of the register.
syn_sharing (D)	Specifies resource sharing of operators.
syn_state_machine (D)	Determines if the FSM Compiler extracts a structure as a state machine.
syn_tco<n> (D)	Defines timing clock to output delay through the black box. The *n* indicates a value between 1 and 10.
syn_tpd<n> (D)	Specifies timing propagation for combinational delay through the black box. The *n* indicates a value between 1 and 10.
syn_tristate (D)	Specifies that a black-box pin is a tristate pin.
syn_tsu<n> (D)	Specifies the timing setup delay for input pins, relative to the clock. The *n* indicates a value between 1 and 10.
translate_off/translate_on (D)	Specifies sections of code to exclude from synthesis, such as simulation-specific code.
(D) indicates directives; all others are attributes.	

Index