Synopsys FPGA Synthesis
Synplify Pro for Microsemi Edition

Reference

December 2012

SYNOPSYS®

Copyright Notice and Proprietary Information

Right to Copy Documentation

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

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

"This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of ____ and its employees. This is copy number __________."

Destination Control Statement

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

Disclaimer

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

Registered Trademarks (®)

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

Trademarks (™)

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

Service Marks (sm)

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

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

ARM and AMBA are registered trademarks of ARM Limited.

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

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

Printed in the U.S.A
December 2012

Contents

Chapter 1: Product Overview

Chapter 2: User Interface Overview

Chapter 3: User Interface Commands

Chapter 4: HDL Analyst Tool

Chapter 5: Constraints

Chapter 6: SCOPE Constraints Editor

Chapter 7: Input and Result Files

Chapter 8: Verilog Language Support

Chapter 9: SystemVerilog Language Support

Chapter 10: VHDL Language Support

Chapter 11: VHDL 2008 Language Support

Chapter 8: Utilities

Chapter 11: Synthesis Attributes and Directives

Chapter 12: Batch Commands and Scripts

Appendix A: Designing with Microsemi

Chapter 1

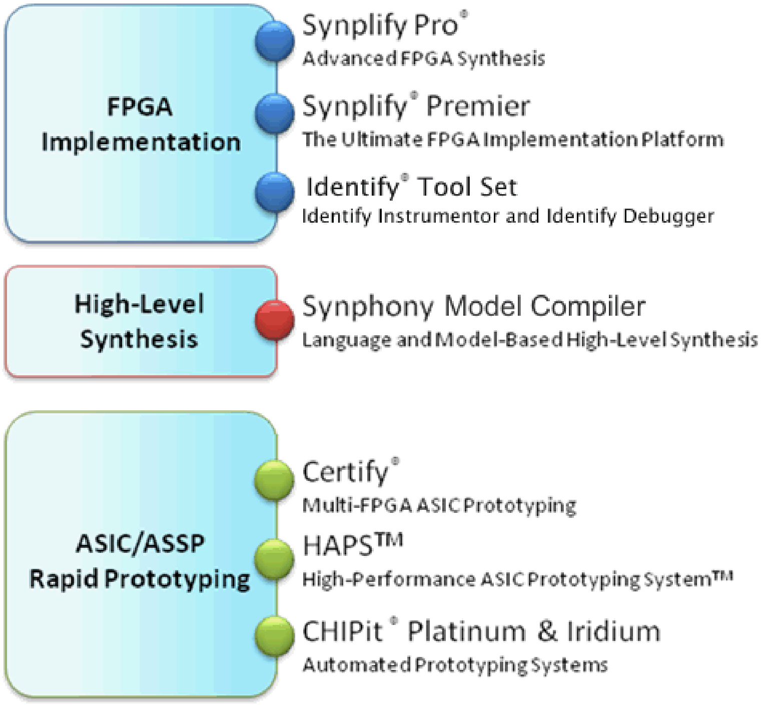
Product Overview

This document is part of a set that includes reference and procedural information for the Synplify Pro® synthesis tool. 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.

The following provide an introduction to the synthesis tools.

Synopsys FPGA and Prototyping Products

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


FPGA Implementation
Synplify Pro®
Advanced FPGA Synthesis
Synplify® Premier
The Ultimate FPGA Implementation Platform
Identify® Tool Set
Identify Instrumentor and Identify Debugger
High-Level Synthesis
Synphony Model Compiler
Language and Model-Based High-Level Synthesis
ASIC/ASSP Rapid Prototyping
Certify®
Multi-FPGA ASIC Prototyping
HAPS™
High-Performance ASIC Prototyping System™
CHIPit® Platinum & Iridium
Automated Prototyping Systems

FPGA Implementation Tools

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

Synplify Pro Product

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

Synplify Premier Product

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

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

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

Identify Tool Set

The Identify® tool set allows you to instrument and debug an operating FPGA directly in the source RTL code. The Identify software is used to verify your design in hardware as you would in simulation, however much faster and with in-system stimulus. Designers and verification engineers are able to navigate the design graphically and instrument signals directly in RTL with which they are familiar, as probes or sample triggers. After synthesis, results are viewed embedded in the RTL source code or in a waveform. Design iterations are rapidly performed using incremental place and route. Identify software is closely integrated with synthesis and routing tools to create a seamless development environment.

Synphony Model Compiler

Synphony Model Compiler is a language and model-based high-level synthesis technology that provides an efficient path from algorithm concept to silicon. Designers can construct high-level algorithm models from math languages and IP model libraries, then use the Synphony Model Compiler engine to synthesize optimized RTL implementations for FPGA and ASIC architectural exploration and rapid prototyping. In addition, Synphony Model Compiler generates high performance C-models for system validation and early software development in virtual platforms. Key features for this product include:

- MATLAB Language Synthesis
- Automated Fixed-point Conversion Tools
- Synthesizable Fixed-point High Level IP Model Library
- High Level Synthesis Optimizations and Transformations
- Integrated FPGA and ASIC Design Flows
- RTL Testbench Generation
- C-model Generation for Software Development and System Validation

Rapid Prototyping

The Certify® and Identify products are tightly integrated with the HAPS™ and ChipIT® hardware tools.

Certify Product

The Certify software is the leading implementation and partitioning tool for ASIC designers using FPGA-based prototypes to verify their designs. The tool provides a quick and easy method for partitioning large ASIC designs into multi-FPGA prototyping boards. Powerful features allow the tool to adapt easily to existing device flows, therefore, speeding up the verification process and helping with the time-to-market challenges. Key features include the following:

- Graphical User Interface (GUI) Flow Guide
- Automatic/Manual Partitioning
- Synopsys Design Constraints Support for Timing Management
- Multi-core Parallel Processing Support for Faster Runtimes
- Support for Most Current FPGA Devices
- Industry Standard Synplify Premier Synthesis Support
- Compatible with HAPS-5x and HAPS-6x Boards Including HSTDM

Overview of the Synthesis Tools

This section introduces the technology, main features, and user interface of the FPGA Synplify Pro synthesis tool. See the following for details:

- Synplify Pro Features, on page 30
- BEST Algorithms, on page 31
- Graphic User Interface, on page 31
- Projects, Implementations, and Workspaces, on page 34

Synplify Pro Features

The following features are specific to the Synplify Pro tool.

- The HDL Analyst® RTL analysis and debugging environment, a graphical tool for analysis and crossprobing. See RTL View, on page 68, Technology View, on page 69, and Analyzing With the HDL Analyst Tool, on page 301 in the *User Guide*.
- The Text Editor window, with a language-sensitive editor for writing and editing HDL code. See Text Editor View, on page 76.
- The SCOPE® (Synthesis Constraint Optimization Environment®) tool, which provides a spreadsheet-like interface for managing timing constraints and design attributes. See SCOPE User Interface, on page 362.
- FSM Compiler, a symbolic compiler that performs advanced finite state machine (FSM) optimizations. See FSM Compiler, on page 84.
- Integration with the Identify RTL Debugger.
- FSM Explorer, which tries different state machine optimizations before picking the best implementation. See FSM Explorer, on page 86.
- The FSM Viewer, for viewing state transitions in detail. See FSM Viewer Window, on page 74.
- The Tcl window, a command line interface for running TCL scripts. See Tcl Script Window, on page 62.

- The Timing Analyst window, which allows you to generate timing schematics and reports for specified paths for point-to-point timing analysis.
- Place-and-Route implementation(s) to automatically run placement and routing after synthesis. You can run place-and-route from within the tool or in batch mode. This feature is supported for the latest Microsemi technologies (see Running Place-and-Route after Synthesis, on page 356 in the *User Guide*).
- Other special windows, or *views*, for analyzing your design, including the Watch Window and Message Viewer (see The Project View, on page 44).
- Retiming optimizations are only available with this tool.
- Advanced analysis features like crossprobing and probe point insertion.

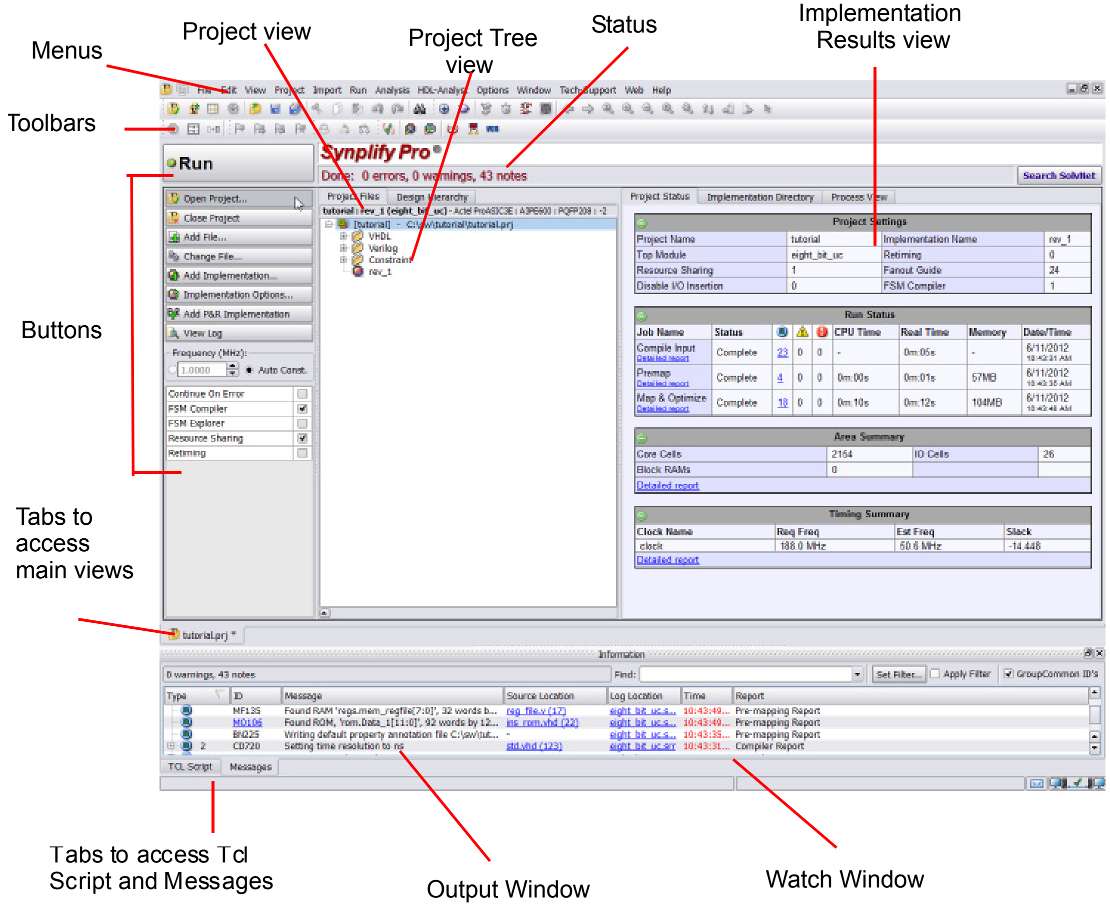
BEST Algorithms

The Behavior Extracting Synthesis Technology (BEST™) feature is the underlying proprietary technology that the synthesis tools use 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

The Synopsys FPGA family of products share a common graphical user interface (GUI), in order to ensure a cohesive look and feel across the different products. The following figure shows the graphical user interface for the Synplify Pro tool.



Done: 0 errors, 0 warnings, 43 notes

Frequency (MHz): ○ 1.0000 ● Auto Const.

Continue On Error	□
FSM Compiler	☑
FSM Explorer	□
Resource Sharing	☑
Retiming	□

tutorial : rev_1 (eight_bit_uc) - Actel ProASIC3E : A3PE600 : PQFP208 : -2

[tutorial] - C:\sw\tutorial\tutorial.prj
VHDL
Verilog
Constraint
rev_1

Project Settings

Project Name	tutorial	Implementation Name	rev_1
Top Module	eight_bit_uc	Retiming	0
Resource Sharing	1	Fanout Guide	24
Disable I/O Insertion	0	FSM Compiler	1

Run Status

Job Name	Status	●	⚠	●	CPU Time	Real Time	Memory	Date/Time
Compile Input Detailed report	Complete	23	0	0	-	0m:05s	-	6/11/2012 10:42:31 AM
Premap Detailed report	Complete	4	0	0	0m:00s	0m:01s	57MB	6/11/2012 10:42:35 AM
Map & Optimize Detailed report	Complete	18	0	0	0m:10s	0m:12s	104MB	6/11/2012 10:42:48 AM

Area Summary

Core Cells	2154	IO Cells	26
Block RAMs	0		

Detailed report

Timing Summary

Clock Name	Req Freq	Est Freq	Slack
clock	188.0 MHz	50.6 MHz	-14.448

Detailed report

0 warnings, 43 notes Find: □ Apply Filter ☑ GroupCommon ID's

```
Type      ID     Message                                          Source Location   Log Location       Time         Report
     ●    MF135  Found RAM 'regs.mem_regfile[7:0]', 32 words b...  reg_file.v (17)    eight_bit_uc.s...  10:43:49...  Pre-mapping Report
     ●    MO106  Found ROM, 'rom.Data_1[11:0]', 92 words by 12...  ins_rom.vhd (22)   eight_bit_uc.s...  10:43:49...  Pre-mapping Report
     ●    BN225  Writing default property annotation file C:\sw\tut...  -            eight_bit_uc.s...  10:43:35...  Pre-mapping Report
⊞    ● 2  CD720  Setting time resolution to ns                     std.vhd (123)      eight_bit_uc.srr   10:43:31...  Compiler Report
```

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*.

For information about...	See...
Project window	The Project View, on page 44
RTL view	RTL View, on page 68
Technology view	Technology View, on page 69
Text Editor view	Text Editor View, on page 76
FSM Viewer window	FSM Viewer Window, on page 74
Tcl window	Tcl Script Window, on page 62
Watch Window	Watch Window, on page 58
SCOPE spreadsheet	SCOPE Tabs, on page 363
Other views and windows	The Project View, on page 44
Menu commands and their dialog boxes	Menus, on page 115
Toolbars	Toolbars, on page 93
Buttons	Buttons and Options, on page 116
Context-sensitive popup menus and their dialog boxes	Popup Menus, on page 281
Online help	Use the F1 keyboard shortcut or click the Help button in a dialog box. See Help Menu, on page 279, for more information.

Projects, Implementations, and Workspaces

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 is highlighted. You can display multiple implementations in the same Project view. The output files generated for the active implementation are displayed in the Implementation Results view on the right.

A *Place and Route implementation*, located in the project implementation hierarchy, is created automatically for supported technologies. To view the P&R implementation, select the plus sign to expand the project implementation hierarchy. To add, remove, or set options, right-click on the P&R implementation. You can create multiple P&R implementations for each project implementation. Select a P&R implementation to activate it.

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. Before you can start the 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.

Starting the Synthesis Tool in Interactive Mode

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

- To start the synthesis tool from the Microsoft® Windows® operating system, choose
  - Start->Programs->Synopsys->Synplify Pro *version*
- To start the tool from a DOS command line, specify the executable:
  - *installDirectory*\bin\synplify_pro.exe

  The executable name is the name of the product followed by an exe file extension.
- To start the synthesis tool from a Linux platform, type the appropriate command at the system prompt:
  - `synplify_pro`

For information about using the synthesis tool in batch mode, see Starting the Tool from the Command Line, on page 36.

Starting the Tool from the Command Line

The command to start the synthesis tool from the command line includes a number of command line options. These options control tool action on startup and, in many cases, can be combined on the same command line. To start the synthesis tool, use the following syntax:

toolname [**-***option ...*] [*projectFile*]

In the syntax statement, *toolName* is the name of the synthesis tool:

- synplify_pro

ProjectFile is the name of the project (.prj) file to open and, if omitted, defaults to the last project file opened. *Option* is any of the following command line options:

-batch *projectFile* |*tclScriptName*
-compile
-evalhostid
-help
-history *historyLogFilename*
-impl *implementationName*
-licensetype *licenseFeatureName*
-license_wait *licenseWaitTime*
-log *logFilename*
-runall
-shell
-tcl *projectFile*|*tclScriptName*
-tclcmd *tclCommandName*
-verbose_log
-version

The following table describes the command line options.

Option	Description
toolname	Starts the synthesis tool: • synplify_pro *projectFile* is the name of the project (.prj) file to open and, if omitted, defaults to the last project file opened.
-batch	Starts the synthesis tool in batch mode from the specified project or Tcl file without opening the Project window. This option is not available for the Synplify Pro tool with node-locked licenses.
-compile	Compiles the project, but does not map it.
-evalhostid	Reports host ID for node-locked and floating licenses.
-help	Lists available command line options and descriptions.
-history	Records all Tcl commands and writes them to the specified history log file when the command exits.
-Identify_dir	Specifies the location of the Identify installation directory for launching the Identify tool set. The installation path specified appears in the Configure Identify Launch dialog box (Options->Configure Identify Launch).
-impl	Runs only the specified implementation. You can use this option in conjunction with the -batch keyword.
-licensetype	Specifies a license if you work in an environment with multiple Synopsys FPGA licenses. You can use this option in conjunction with the -batch keyword.
-license_wait	Specifies how long to wait for a Synopsys FPGA license. License queuing allows you to wait until a license becomes available or specify a wait time in seconds. You can use this option in conjunction with the -batch keyword.
-log	Writes all output to the specified log file.
-runall	Runs all the implementations in the project file.
-shell	Starts synthesis tool in shell mode. Note: The FPGA synthesis tools only support the -shell option on UNIX and Linux platforms.
-tcl	Starts the synthesis tool in the graphical user interface using the specified project or Tcl file.

Option	Description
-tclcmd	Specifies Tcl command to be executed on startup.
-verbose_log	Writes messages to stdout.log in verbose mode.
-version	Reports version of specified synthesis tool.

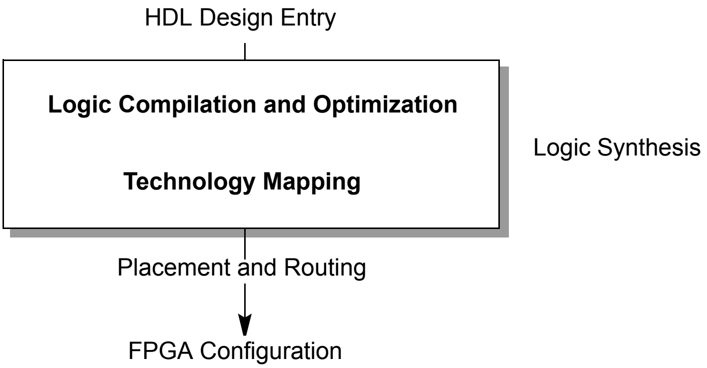
Logic Synthesis Overview

When you run the synthesis tool, it performs *logic synthesis*. This consists of two stages:

- logic compilation (HDL language synthesis) and optimization
- 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 to 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 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, VHDL, or IP files in formats such as EDIF, 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.

   See SCOPE Tabs, on page 363, 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 synthesis tool displays the status (Compiling... or Mapping...). You can monitor messages by checking the log file (View->View Log File) or the Tcl window (View->Tcl Window). 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 and the Tcl window. 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. From the log file, you can jump to the corresponding source code or display information on the specific error or warning. Correct all errors and any relevant warnings and then rerun synthesis.

Getting Help

Before calling Synopsys SolvNet Support, look through the documentation for 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.

Finding Information

For help with...	Refer to the...
How to...	*User Guide* and various application notes available on the Synplicity support web site
Flow information	*User Guide* and various application notes available on the Synopsys SolvNet support web site
FPGA Implementation Tools	Synopsys Web Page (Web->FPGA Implementation Tools menu command from within the software)
Synthesis features	*User Guide* and *Reference Manual*
Language and syntax	*Reference Manual*
Attributes and directives	*Reference Manual*
Tcl language	Online help (Help->Tcl Help)
Synthesis Tcl commands	*Reference Manual* or type help followed by the command name in the Tcl window
Using tool-specific features and attributes	*User Guide*
Error and warning messages	Click on the message ID code

CHAPTER 2

User Interface Overview

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

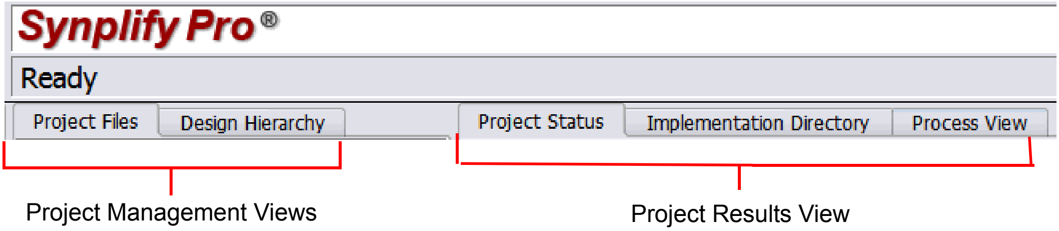
These are the topics in this chapter:

The Project View

The Project View is the main interface to the tool. The Project view consists of a Project Management View and Project Results View. Use the Project Management view to create or open projects, create new implementations, set device options, and initiate design synthesis. The Project Results view contains the results of the synthesis runs for the implementations of your design and allows you to control job process flows. For a description of the following Project views, see:

- Project Management Views
- The Project Results View
  - Project Status View
  - Implementation Directory
  - Process View

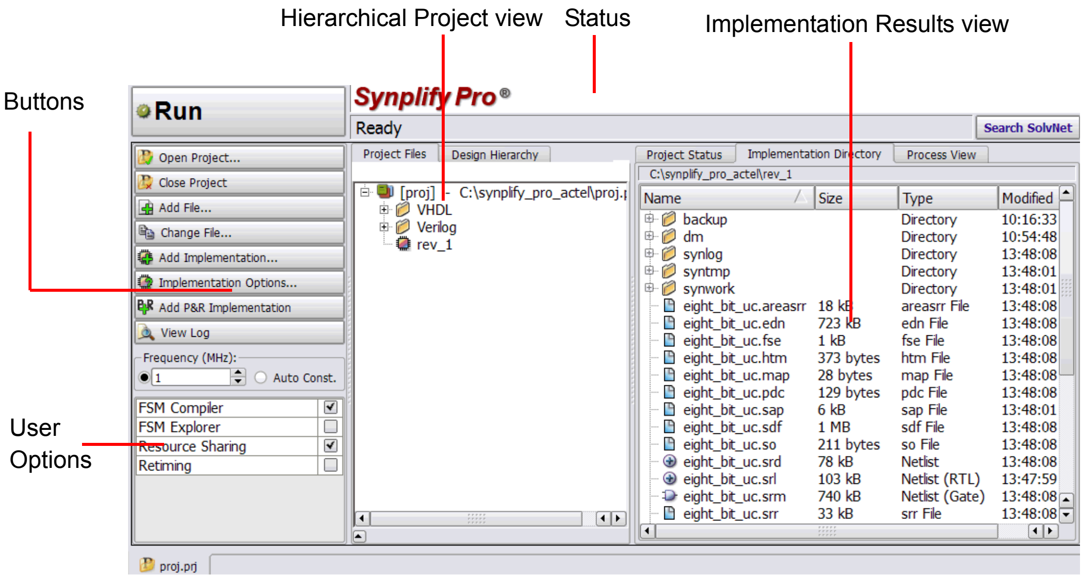
The Project view typically displays the following process view tabs, depending on the synthesis tool you use.



Project Management Views

The Project Management views appear on the left side of the Project view and are used to create or open projects, create new implementations, set device options, and initiate design synthesis.

The following figure shows the Project view as it appears in the Synplify Pro interface.



Frequency (MHz): ● 1 ○ Auto Const.

FSM Compiler ☑
FSM Explorer □
Resource Sharing ☑
Retiming □

[proj] - C:\synplify_pro_actel\proj.p
VHDL
Verilog
rev_1

C:\synplify_pro_actel\rev_1

Name	Size	Type	Modified
backup		Directory	10:16:33
dm		Directory	10:54:48
synlog		Directory	13:48:08
syntmp		Directory	13:48:01
synwork		Directory	13:48:01
eight_bit_uc.areasrr	18 kB	areasrr File	13:48:08
eight_bit_uc.edn	723 kB	edn File	13:48:08
eight_bit_uc.fse	1 kB	fse File	13:48:08
eight_bit_uc.htm	373 bytes	htm File	13:48:08
eight_bit_uc.map	28 bytes	map File	13:48:08
eight_bit_uc.pdc	129 bytes	pdc File	13:48:08
eight_bit_uc.sap	6 kB	sap File	13:48:01
eight_bit_uc.sdf	1 MB	sdf File	13:48:08
eight_bit_uc.so	211 bytes	so File	13:48:08
eight_bit_uc.srd	78 kB	Netlist	13:48:08
eight_bit_uc.srl	103 kB	Netlist (RTL)	13:47:59
eight_bit_uc.srm	740 kB	Netlist (Gate)	13:48:08
eight_bit_uc.srr	33 kB	srr File	13:48:08

The Project View Interface

The Project view has the following main parts:

Project View Interface	Description
Status	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 184).
Buttons and options	Allow immediate access to some of the more common commands. See Buttons and Options, on page 110 for details.
Project tree view	Lists the projects, workspaces, and implementations, and their associated HDL source files and constraint files.
Implementation Results view	Lists the result of the synthesis runs for the implementations of your design. 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 Project Results view includes the following: • Project Status View—provides an overview of the project settings and at-a-glance summary of synthesis messages and reports. • Implementation Directory—lists the names and types of the result files, and the dates they were last modified. • Process View—gives you instant visibility to the synthesis and place-and-route job flows. See The Project Results View, on page 47 for more information.

To customize the Project view display, use the Options->Project View Options command (Project View Options Command, on page 254).

The Project Results View

The Project Results view appears on the right side of the Project view and contains the results of the synthesis runs for the implementations of your design. The Project Results view includes the following:

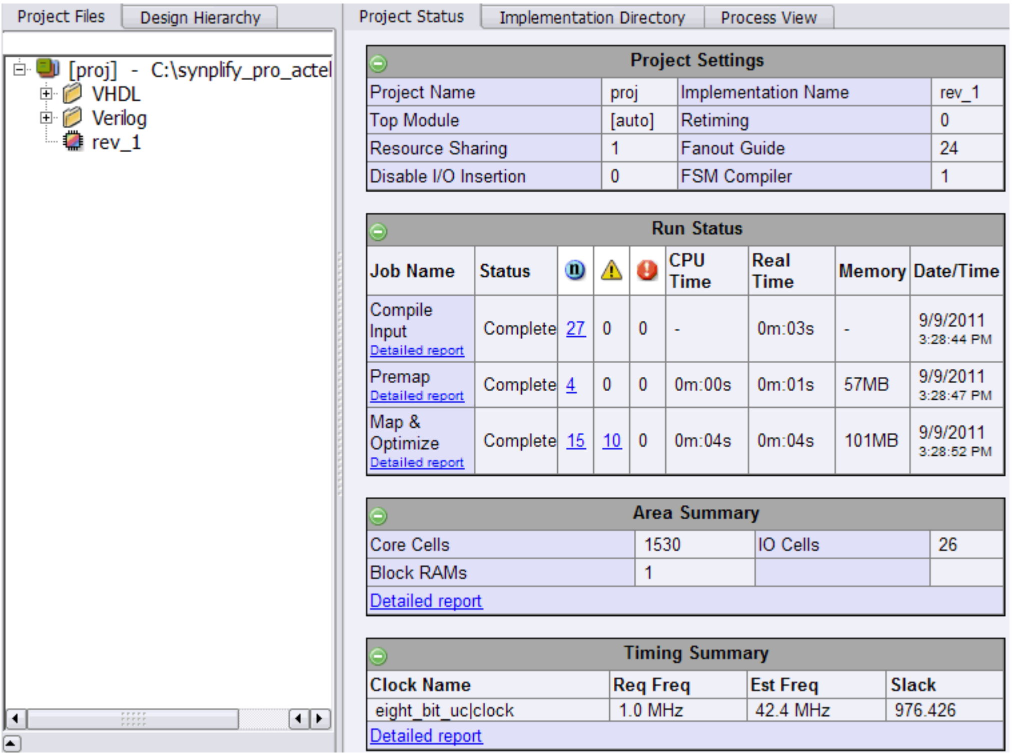
- Project Status View
- Implementation Directory
- Process View

Project Status View

The Project Status view provides an overview of the project settings and at-a-glance summary of synthesis messages and reports such as an area or optimization summary for the active implementation. You can track the status and settings for your design and easily navigate to reports and messages in the Project view.

To display this window, click on the Project Status tab in the Project view. An overview for the project is displayed in a spreadsheet format for each of the following sections:

- Project Settings
- Run Status
- Reports



Project Settings

Project Name	proj	Implementation Name	rev_1
Top Module	[auto]	Retiming	0
Resource Sharing	1	Fanout Guide	24
Disable I/O Insertion	0	FSM Compiler	1

Run Status

Job Name	Status	ⓘ	⚠	❗	CPU Time	Real Time	Memory	Date/Time
Compile Input Detailed report	Complete	27	0	0	-	0m:03s	-	9/9/2011 3:28:44 PM
Premap Detailed report	Complete	4	0	0	0m:00s	0m:01s	57MB	9/9/2011 3:28:47 PM
Map & Optimize Detailed report	Complete	15	10	0	0m:04s	0m:04s	101MB	9/9/2011 3:28:52 PM

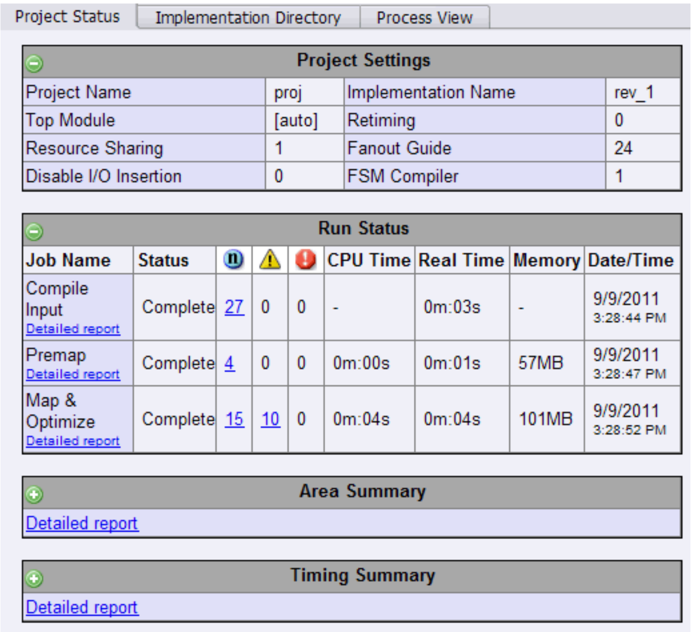
Area Summary

Core Cells	1530	IO Cells	26
Block RAMs	1		
Detailed report			

Timing Summary

Clock Name	Req Freq	Est Freq	Slack
eight_bit_uc|clock	1.0 MHz	42.4 MHz	976.426
Detailed report			

You can expand or collapse each section of the Project Status view by clicking on the + or - icon in the upper left-corner of each section.



Project Settings			
Project Name	proj	Implementation Name	rev_1
Top Module	[auto]	Retiming	0
Resource Sharing	1	Fanout Guide	24
Disable I/O Insertion	0	FSM Compiler	1

Run Status

Job Name	Status	ⓘ	⚠	❗	CPU Time	Real Time	Memory	Date/Time
Compile Input Detailed report	Complete	27	0	0	-	0m:03s	-	9/9/2011 3:28:44 PM
Premap Detailed report	Complete	4	0	0	0m:00s	0m:01s	57MB	9/9/2011 3:28:47 PM
Map & Optimize Detailed report	Complete	15	10	0	0m:04s	0m:04s	101MB	9/9/2011 3:28:52 PM

Area Summary

Detailed report

Timing Summary

Detailed report

Project Settings

The Project Settings table gets populated with the project settings from the run_options.txt file after a synthesis run. This section displays information, such as:

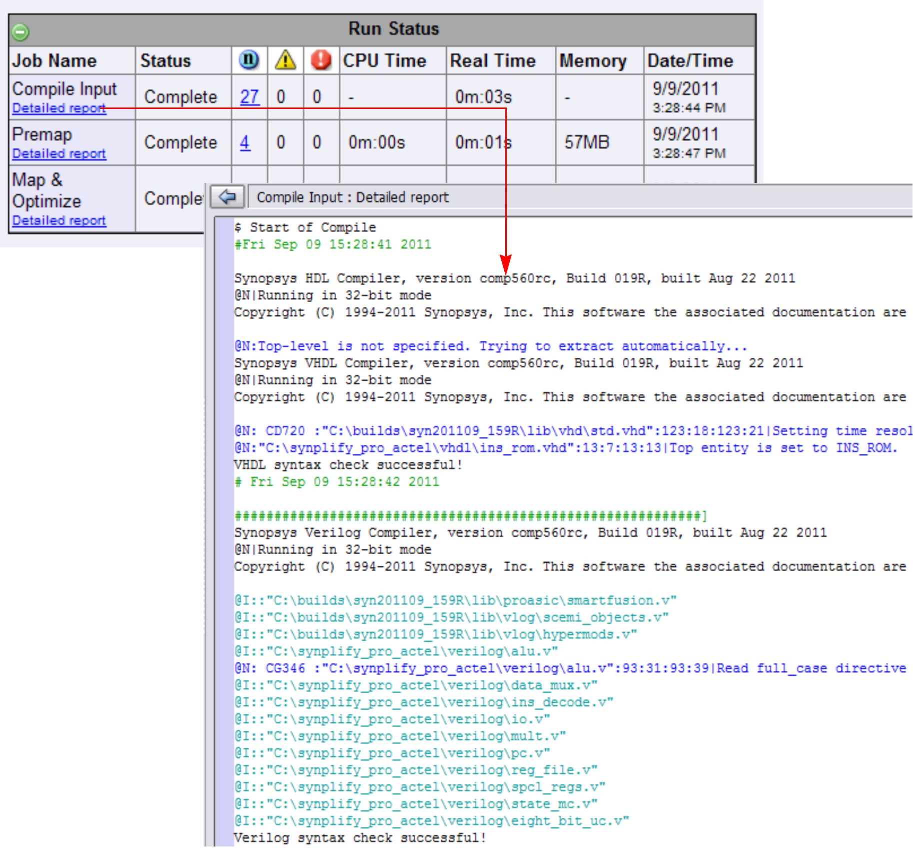
- Project name, top-level module, and implementation name
- Project options currently specified, such as Retiming, Resource Sharing, Fanout Guide, and Disable I/O Insertion.

Run Status

The Run Status table gets updated during and after a synthesis run. This section displays job status information for the compiler, premap job, mapper, and place-and-route runs, as needed. This section displays information, such as:

- Job name - For example, jobs include HDL Compiler, Premap Job, and Mapper. The job might have a Detailed Report link. When you click on this link, a Report tab is created that displays information for the selected job.

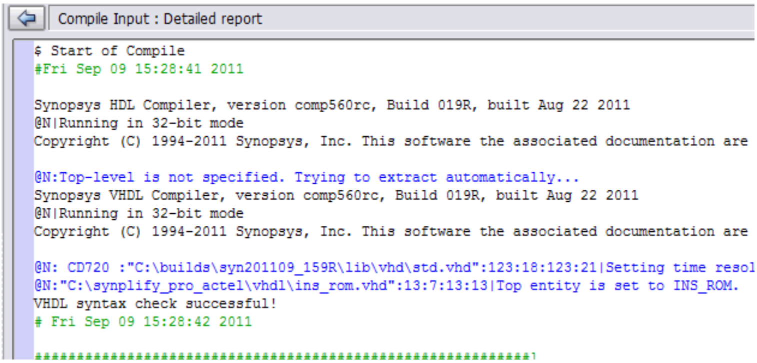
Thereafter, information is rewritten to the Report tab for subsequent links. Use the arrow icon (⇦) to get back to the main Project Status view.



Run Status								
Job Name	Status	ⓘ	⚠	❗	CPU Time	Real Time	Memory	Date/Time
Compile Input Detailed report	Complete	27	0	0	-	0m:03s	-	9/9/2011 3:28:44 PM
Premap Detailed report	Complete	4	0	0	0m:00s	0m:01s	57MB	9/9/2011 3:28:47 PM
Map & Optimize Detailed report	Comple							

```
$ Start of Compile
#Fri Sep 09 15:28:41 2011

Synopsys HDL Compiler, version comp560rc, Build 019R, built Aug 22 2011
@N|Running in 32-bit mode
Copyright (C) 1994-2011 Synopsys, Inc. This software the associated documentation are

@N:Top-level is not specified. Trying to extract automatically...
Synopsys VHDL Compiler, version comp560rc, Build 019R, built Aug 22 2011
@N|Running in 32-bit mode
Copyright (C) 1994-2011 Synopsys, Inc. This software the associated documentation are

@N: CD720 :"C:\builds\syn201109_159R\lib\vhd\std.vhd":123:18:123:21|Setting time resol
@N:"C:\synplify_pro_actel\vhdl\ins_rom.vhd":13:7:13:13|Top entity is set to INS_ROM.
VHDL syntax check successful!
# Fri Sep 09 15:28:42 2011

###########################################################]
Synopsys Verilog Compiler, version comp560rc, Build 019R, built Aug 22 2011
@N|Running in 32-bit mode
Copyright (C) 1994-2011 Synopsys, Inc. This software the associated documentation are

@I::"C:\builds\syn201109_159R\lib\proasic\smartfusion.v"
@I::"C:\builds\syn201109_159R\lib\vlog\scemi_objects.v"
@I::"C:\builds\syn201109_159R\lib\vlog\hypermods.v"
@I::"C:\synplify_pro_actel\verilog\alu.v"
@N: CG346 :"C:\synplify_pro_actel\verilog\alu.v":93:31:93:39|Read full_case directive
@I::"C:\synplify_pro_actel\verilog\data_mux.v"
@I::"C:\synplify_pro_actel\verilog\ins_decode.v"
@I::"C:\synplify_pro_actel\verilog\io.v"
@I::"C:\synplify_pro_actel\verilog\mult.v"
@I::"C:\synplify_pro_actel\verilog\pc.v"
@I::"C:\synplify_pro_actel\verilog\reg_file.v"
@I::"C:\synplify_pro_actel\verilog\spcl_regs.v"
@I::"C:\synplify_pro_actel\verilog\state_mc.v"
@I::"C:\synplify_pro_actel\verilog\eight_bit_uc.v"
Verilog syntax check successful!
```

- Job status - Job can be running or completed.
- Notes, warnings, and errors – A message count is displayed with a link to information about these messages. When you click on this link, a Report tab is created that displays these messages. Thereafter, subsequent links overwrite information to this Report tab. Use the arrow icon (⇦) to get back to the main Project Status view.



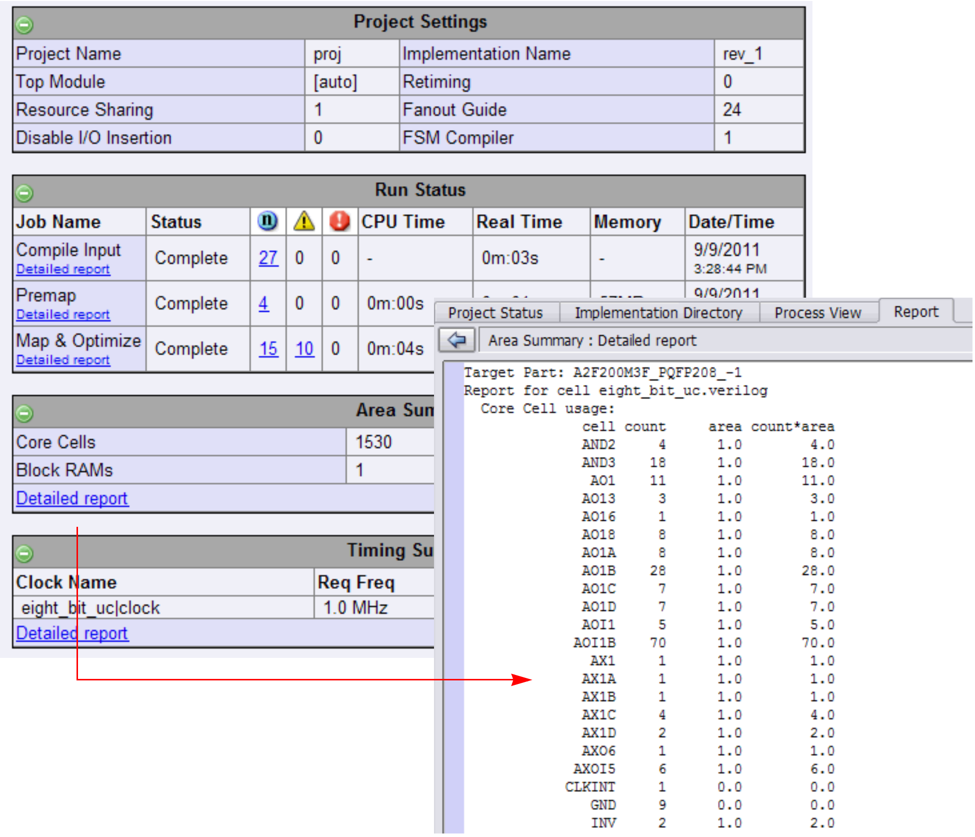
```
$ Start of Compile
#Fri Sep 09 15:28:41 2011

Synopsys HDL Compiler, version comp560rc, Build 019R, built Aug 22 2011
@N|Running in 32-bit mode
Copyright (C) 1994-2011 Synopsys, Inc. This software the associated documentation are

@N:Top-level is not specified. Trying to extract automatically...
Synopsys VHDL Compiler, version comp560rc, Build 019R, built Aug 22 2011
@N|Running in 32-bit mode
Copyright (C) 1994-2011 Synopsys, Inc. This software the associated documentation are

@N: CD720 :"C:\builds\syn201109_159R\lib\vhd\std.vhd":123:18:123:21|Setting time resol
@N:"C:\synplify_pro_actel\vhdl\ins_rom.vhd":13:7:13:13|Top entity is set to INS_ROM.
VHDL syntax check successful!
# Fri Sep 09 15:28:42 2011

############################################################1
```

- Real and CPU times, peak memory, and a timestamp

Reports

The mapper summary table generates various reports such as an Area Summary, Compile Point Summary, or Optimization Summary. Click on the Detailed Report link when applicable, then a Report tab is created that displays information about these summary tables. Thereafter, subsequent links overwrite information to the Report tab. These reports are written to the synlog folder for the active implementation.

For example, the Area Summary contains a resource usage count for components such as registers, LUTs, and I/O ports in the design. Click on the Detailed report link to display the usage count information in the design for this report.



Project Settings			
Project Name	proj	Implementation Name	rev_1
Top Module	[auto]	Retiming	0
Resource Sharing	1	Fanout Guide	24
Disable I/O Insertion	0	FSM Compiler	1

Run Status

Job Name	Status	ⓘ	⚠	❗	CPU Time	Real Time	Memory	Date/Time
Compile Input Detailed report	Complete	27	0	0	-	0m:03s	-	9/9/2011 3:28:44 PM
Premap Detailed report	Complete	4	0	0	0m:00s			9/9/2011
Map & Optimize Detailed report	Complete	15	10	0	0m:04s			

Area Sun

Core Cells	1530
Block RAMs	1

Detailed report

Timing Su

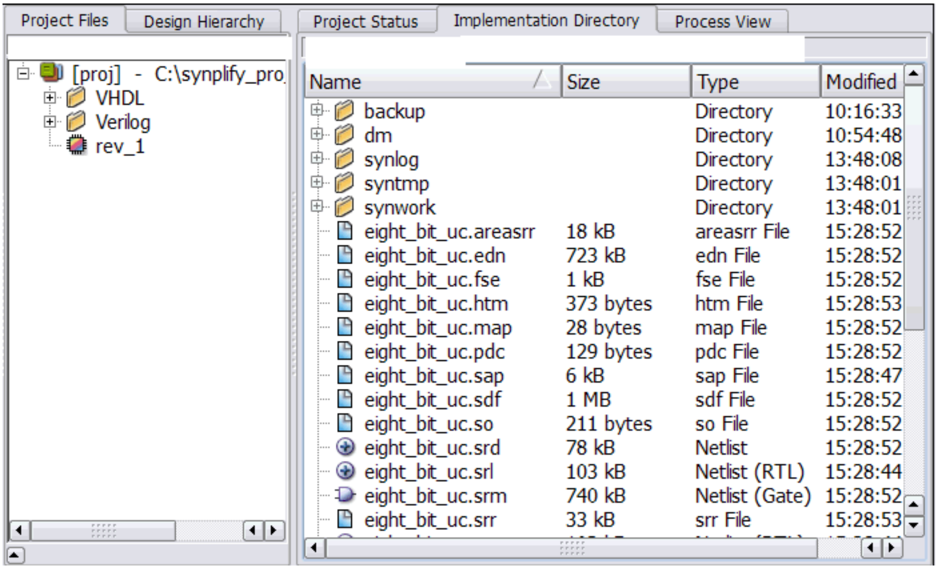
Clock Name	Req Freq
eight_bit_uc|clock	1.0 MHz

Detailed report

```
Target Part: A2F200M3F_PQFP208_-1
Report for cell eight_bit_uc.verilog
  Core Cell usage:
              cell  count     area  count*area
              AND2     4       1.0         4.0
              AND3    18       1.0        18.0
               AO1    11       1.0        11.0
              AO13     3       1.0         3.0
              AO16     1       1.0         1.0
              AO18     8       1.0         8.0
              AO1A     8       1.0         8.0
              AO1B    28       1.0        28.0
              AO1C     7       1.0         7.0
              AO1D     7       1.0         7.0
              AOI1     5       1.0         5.0
             AOI1B    70       1.0        70.0
               AX1     1       1.0         1.0
              AX1A     1       1.0         1.0
              AX1B     1       1.0         1.0
              AX1C     4       1.0         4.0
              AX1D     2       1.0         2.0
              AXO6     1       1.0         1.0
             AXOI5     6       1.0         6.0
            CLKINT     1       0.0         0.0
               GND     9       0.0         0.0
               INV     2       1.0         2.0
```

Implementation Directory

An implementation is one version 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 is highlighted. You can display multiple implementations in the same Project view. The output files generated for the active implementation are displayed in the Implementation Directory.

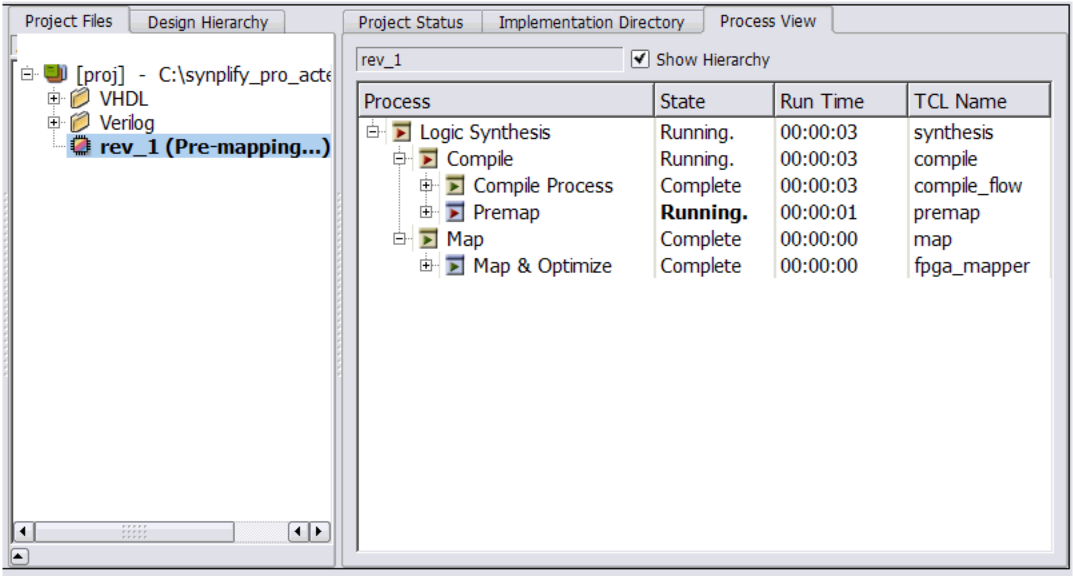


Name	Size	Type	Modified
backup		Directory	10:16:33
dm		Directory	10:54:48
synlog		Directory	13:48:08
syntmp		Directory	13:48:01
synwork		Directory	13:48:01
eight_bit_uc.areasrr	18 kB	areasrr File	15:28:52
eight_bit_uc.edn	723 kB	edn File	15:28:52
eight_bit_uc.fse	1 kB	fse File	15:28:52
eight_bit_uc.htm	373 bytes	htm File	15:28:53
eight_bit_uc.map	28 bytes	map File	15:28:52
eight_bit_uc.pdc	129 bytes	pdc File	15:28:52
eight_bit_uc.sap	6 kB	sap File	15:28:47
eight_bit_uc.sdf	1 MB	sdf File	15:28:52
eight_bit_uc.so	211 bytes	so File	15:28:52
eight_bit_uc.srd	78 kB	Netlist	15:28:52
eight_bit_uc.srl	103 kB	Netlist (RTL)	15:28:44
eight_bit_uc.srm	740 kB	Netlist (Gate)	15:28:52
eight_bit_uc.srr	33 kB	srr File	15:28:53

Process View

As process flow jobs become more complex, the benefits of exposing the underlying job flow is extremely valuable. The Process View gives you this visibility to track the design progress for the synthesis and place-and-route job flows.

Click the Process View tab on the right side of the Project Results view. This displays the job flow hierarchy run on the active implementation and is a function of this current implementation and its project settings.



Process	State	Run Time	TCL Name
Logic Synthesis	Running.	00:00:03	synthesis
Compile	Running.	00:00:03	compile
Compile Process	Complete	00:00:03	compile_flow
Premap	**Running.**	00:00:01	premap
Map	Complete	00:00:00	map
Map & Optimize	Complete	00:00:00	fpga_mapper

Process View Displays and Controls

The Process View shows the current state of a job and allows you to control the run. You can see various aspects of the synthesis process flow, such as logical synthesis, premap, map, and placement. If you run place and route, you can see its job processes as well.

Appropriate jobs of the process flow contains the following information:

- Job Input and Output Files
- Completion State

  Displays if the job generated an error, warning, or was canceled.
- Job State
  - Out-of-date – Job needs to be run.
  - Running – Job is active.
  - Complete – Job has completed and is up-to-date.
  - Complete * – Job is up-to-date, so the job is skipped.
- Run/File Time – Job process flow runtime in real time or file creation date timestamp.
- Job TCL Command – Job process name.

Each job has the following control commands that allows you to run jobs at any stage of the design process, for example map. Right-click any job icon and select one of the following commands from the popup menu:

- Cancel *jobProcess* that is running
- Disable *jobProcess* that you do not want to run
- Run this *jobProcess* only
- Run to this *jobProcess* from the beginning of run
- Run from this *jobProcess* to the end of run

Hierarchical Job Flows

A hierarchical job flow runs two or more subordinate jobs. Primitive jobs launch an executable, but have no subordinate jobs. The Logical Synthesis flow is a hierarchical job that runs the Compile and Map flows.

The state of a hierarchical job depends on the state of its subordinate jobs.

- If a subordinate job is out-of-date, then its parent job is out-of-date.
- If a subordinate job has an error, then its parent job terminates with this error.
- If a subordinate job has been canceled, then its parent job is canceled as well.
- If a subordinate job is running, then its parent job is also running.

The Process View is a hierarchical tree view. To collapse or expand the main hierarchical tree, enable or disable the Show Hierarchy option. Use the plus or minus icon to expand or collapse each process flow to show the details of the jobs. The icons below are used to show the information for the state of each process:

- Red arrow (▶) – Job is out-of-date and needs to be rerun.
- Green arrow (▶) – Job is up-to-date.
- Red Circle with! (❗) - Job encountered an error.

Other Windows and Views

Besides the Project view, the Synopsys FPGA synthesis tools provide other 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 described here:

- Dockable GUI Entities, on page 58
- Watch Window, on page 58
- Tcl Script and Messages Windows, on page 62
- Tcl Script Window, on page 62
- Message Viewer, on page 63
- Output Windows (Tcl Script and Watch Windows), on page 67
- RTL View, on page 68
- Technology View, on page 69
- Hierarchy Browser, on page 72
- FSM Viewer Window, on page 74
- Text Editor View, on page 76
- Context Help Editor Window, on page 79
- Search SolvNet, on page 81

See the following for descriptions of other views and windows that are not covered here:

Project view	The Project View, on page 44
SCOPE	SCOPE Tabs, on page 363

Dockable GUI Entities

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

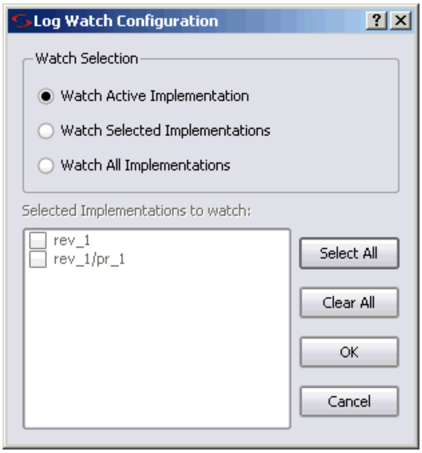
Watch Window

The Watch window displays selected information from the log file (see Log File, on page 435) as a spreadsheet of parameters that you select to monitor. The values are updated when synthesis finishes.

Watch Window Display

Display of the Watch window is controlled by the View ->Watch Window command. By default, the Watch window is below the Project view in the lower right corner of the main application window.

To access the Watch window configuration menu, right-click in any cell. Select Configure Watch to display the Log Watch Configuration dialog box.



Log Watch Configuration

Watch Selection

● Watch Active Implementation

○ Watch Selected Implementations

○ Watch All Implementations

Selected Implementations to watch:

□ rev_1

□ rev_1/pr_1

In the Watch window, indicate which implementations to watch under Watch Selection. The selected implementation(s) will display in the Watch window.

You can move the Watch window anywhere on the screen; you can make it float in its own window (named Watch Window) or dock it at a docking area (an edge) of the application window. Double-click in the banner to toggle between docked and floating.

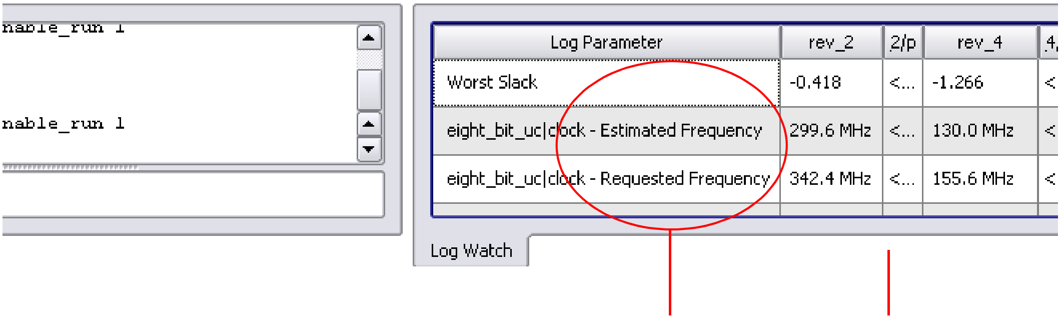
The Watch window has a special positioning popup menu that you access by right-clicking the window border. The following commands are in the menu:

Command	Description
Allow Docking	A toggle: when enabled, the window can be docked.
Hide	Hides the window; use View ->Watch Window to show it again.
Float in Main Window	A toggle: when enabled, the window is floated (undocked).

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

Using the Watch Window

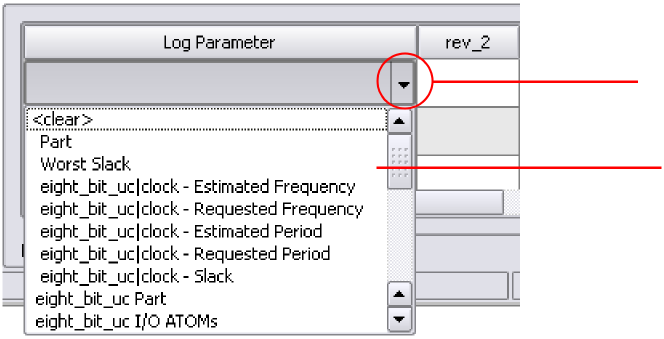
You can view and compare the results of multiple implementations in the Watch window.



Log Parameter	rev_2	2/p	rev_4	4,
Worst Slack	-0.418	<...	-1.266	<
eight_bit_uc|clock - Estimated Frequency	299.6 MHz	<...	130.0 MHz	<
eight_bit_uc|clock - Requested Frequency	342.4 MHz	<...	155.6 MHz	<

Log Parameters Watch Window

To choose log parameters from a pull-down menu, click in the Log Parameter section of the window. Click the pull-down arrow that appears to display the parameter list choices:

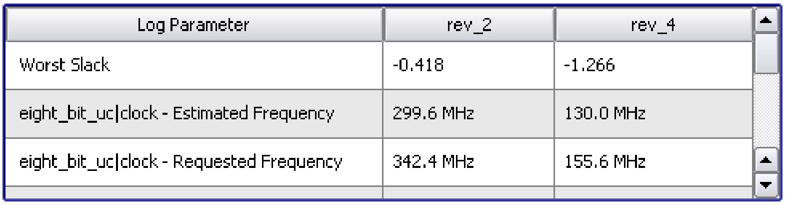

Log Parameter
rev_2
<clear>
Part
Worst Slack
eight_bit_uc|clock - Estimated Frequency
eight_bit_uc|clock - Requested Frequency
eight_bit_uc|clock - Estimated Period
eight_bit_uc|clock - Requested Period
eight_bit_uc|clock - Slack
eight_bit_uc Part
eight_bit_uc I/O ATOMs

Click pull-down arrow

to

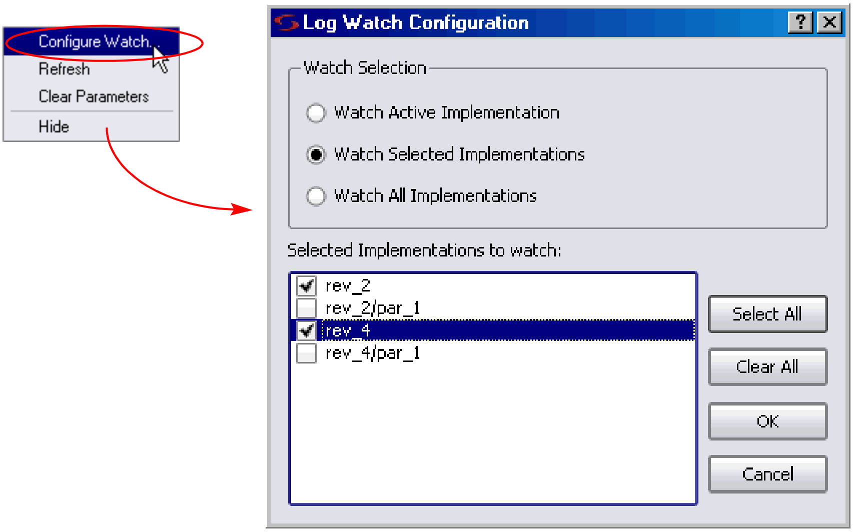
display list of choices

The Watch window creates an entry for each implementation of a project:



Log Parameter	rev_2	rev_4
Worst Slack	-0.418	-1.266
eight_bit_uc|clock - Estimated Frequency	299.6 MHz	130.0 MHz
eight_bit_uc|clock - Requested Frequency	342.4 MHz	155.6 MHz

To choose the implementations to watch, use the Log Watch Configuration dialog box. To display this box, right-click in the Watch window, then choose Configure Watch in the popup menu. Enable Watch Selected Implementations, then choose the implementations you want to watch in the list Selected Implementations to watch. The other buttons let you watch only the active implementation or all implementations.



Watch Selection

○ Watch Active Implementation

● Watch Selected Implementations

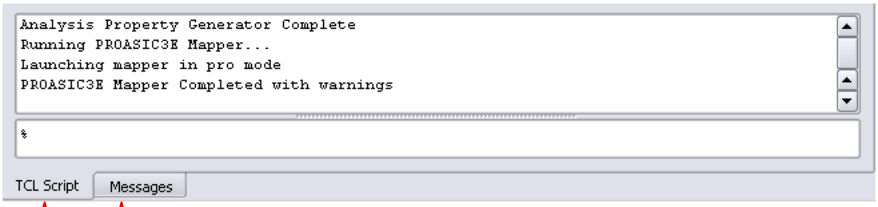
○ Watch All Implementations

Selected Implementations to watch:

☑ rev_2
□ rev_2/par_1
☑ rev_4
□ rev_4/par_1

Tcl Script and Messages Windows

The Tcl window has tabs for the Tcl Script and Messages windows. By default, the Tcl windows are located below the Project Tree view in the lower left corner of the main application window.



```
Analysis Property Generator Complete
Running PROASIC3E Mapper...
Launching mapper in pro mode
PROASIC3E Mapper Completed with warnings
```

```
%
```

Messages panel displays errors, warnings, and notes

Tcl Script panel to display and input Tcl commands

You can float the Tcl windows by clicking on a window edge while holding the Ctrl or Shift key. You can then drag the window to float it anywhere on the screen or dock it at an edge of the application window. Double-click in the banner to toggle between docked and floating.

Right-clicking the Tcl windows *title bar* when the window is floating displays a popup menu with commands Hide and Move. Hide removes the window (use View ->Tcl Window to redisplay the window). Move lets you position the window using either the arrow keys or the mouse.

For more information about the Tcl windows, see Tcl Script Window, on page 62 and Message Viewer, on page 63.

Tcl Script Window

The Tcl Script window is an interactive command shell that implements the Tcl command-line interface. You can type or paste Tcl commands at the prompt (“% ”). For a list of the available commands, type “help *” (without the quotes) at the prompt. For general information about Tcl syntax, choose Help ->TCL.

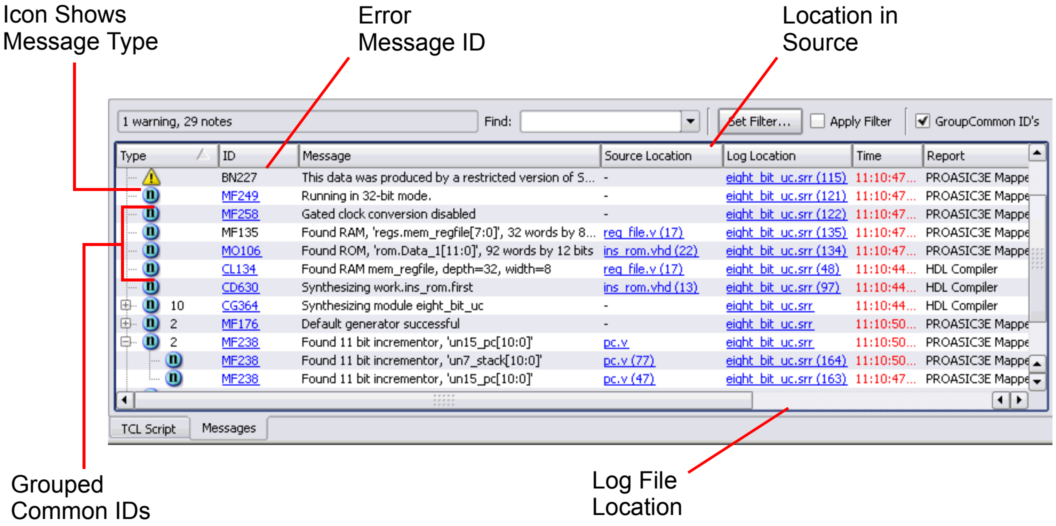
The Tcl script window also displays each command executed in the course of running the synthesis tool, regardless of whether it was initiated from a menu, button, or keyboard shortcut. Right-clicking inside the Tcl window displays a popup menu with the Copy, Paste, Hide, and Help commands.

See also

- Batch Commands for Synthesis, on page 1035, for information about the Tcl synthesis commands.
- Generating a Job Script, on page 457 in the *User Guide*.

Message Viewer

To display errors, warnings, and notes after running the synthesis tool, click the Messages tab in the Tcl Window. A spreadsheet-style interactive interface appears.



Type	ID	Message	Source Location	Log Location	Time	Report
⚠	BN227	This data was produced by a restricted version of S...	-	eight_bit_uc.srr (115)	11:10:47...	PROASIC3E Mappe
n	MF249	Running in 32-bit mode.	-	eight_bit_uc.srr (121)	11:10:47...	PROASIC3E Mappe
n	MF258	Gated clock conversion disabled	-	eight_bit_uc.srr (122)	11:10:47...	PROASIC3E Mappe
n	MF135	Found RAM, 'regs.mem_regfile[7:0]', 32 words by 8...	reg_file.v (17)	eight_bit_uc.srr (135)	11:10:47...	PROASIC3E Mappe
n	MO106	Found ROM, 'rom.Data_1[11:0]', 92 words by 12 bits	ins_rom.vhd (22)	eight_bit_uc.srr (134)	11:10:47...	PROASIC3E Mappe
n	CL134	Found RAM mem_regfile, depth=32, width=8	reg_file.v (17)	eight_bit_uc.srr (48)	11:10:44...	HDL Compiler
n	CD630	Synthesizing work.ins_rom.first	ins_rom.vhd (13)	eight_bit_uc.srr (97)	11:10:44...	HDL Compiler
⊞ n 10	CG364	Synthesizing module eight_bit_uc	-	eight_bit_uc.srr	11:10:44...	HDL Compiler
⊞ n 2	MF176	Default generator successful	-	eight_bit_uc.srr	11:10:50...	PROASIC3E Mappe
⊟ n 2	MF238	Found 11 bit incrementor, 'un15_pc[10:0]'	pc.v	eight_bit_uc.srr	11:10:50...	PROASIC3E Mappe
n	MF238	Found 11 bit incrementor, 'un7_stack[10:0]'	pc.v (77)	eight_bit_uc.srr (164)	11:10:50...	PROASIC3E Mappe
n	MF238	Found 11 bit incrementor, 'un15_pc[10:0]'	pc.v (47)	eight_bit_uc.srr (163)	11:10:47...	PROASIC3E Mappe

Interactive tasks in the Messages panel include:

- Drag the pane divider with the mouse to change the relative column size.
- Click on the ID entry to open online help for the error, warning, or note.
- Click on a Source Location entry to go to the section of code in the source HDL file that is causing the message.
- Click on a Log Location entry to go to its location in the log file.

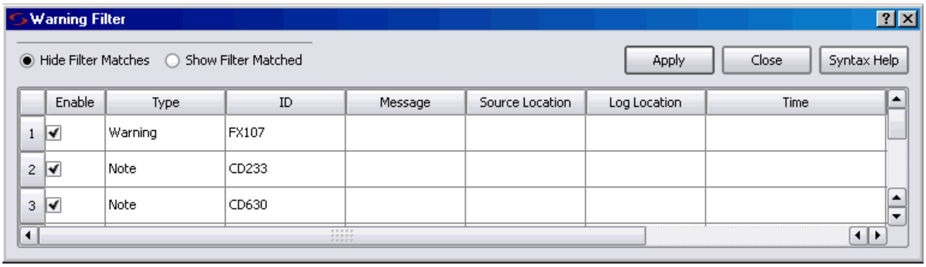
The following table describes the contents of the Messages panel. You can sort the messages by clicking the column headers. For further sorting, use Find and Filter. For details about using this window, see Checking Results in the Message Viewer, on page 245 in the *User Guide*.

Item	Description
Find	Type into this field to find errors, warnings, or notes.
Filter	Opens the Warning Filter dialog box. See Messages Filter, on page 66.
Apply Filter	Enable/disable the last saved filter.
Group Common ID's	Enable/disable grouping of repeated messages. Groups are indicated by a number next to the type icon. There are two types of groups: • The same warning or note ID appears in multiple source files indicated by a dash in the source files column. • Multiple warnings or notes in the same line of source code indicated by a bracketed number.
Type	The icons indicate the type of message: ❗ Error ⚠ Warning ⓝ Note ⓐ Advisory A plus sign next to an icon indicates that repeated messages are grouped together. Click the plus sign to expand and view the various occurrences of the message.
ID	This is the message ID. You can select an underlined ID to launch help on the message.
Message	The error, warning, or note message text.

Item	Description
Source Location	The HDL source file that generated the error, warning, or note message.
Log Location	The location of the error, warning, or note message in the log file.
Time	The time the error, warning or note message was recorded in the log file.
Report	Indicates which section of the Log File report the error appears, for example Compiler or Mapper.

Messages Filter

You filter which errors, warnings, and notes appear in the Messages panel of the Tcl Window using match criteria for each field. The selections are combined to produce the result. You can elect to hide or show the warnings that match the criteria you set. See Checking Results in the Message Viewer, on page 245 in the *User Guide*.

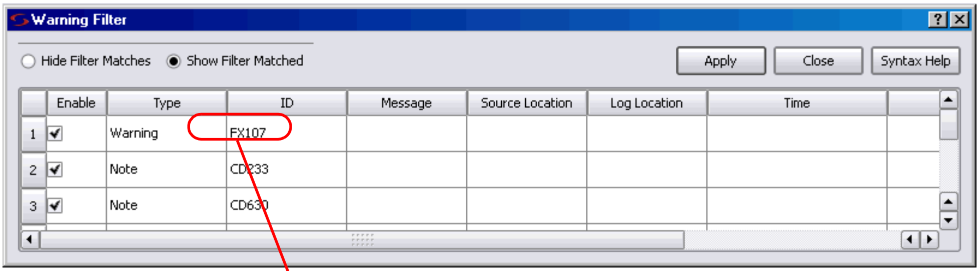


● Hide Filter Matches ○ Show Filter Matched

	Enable	Type	ID	Message	Source Location	Log Location	Time
1	☑	Warning	FX107				
2	☑	Note	CD233				
3	☑	Note	CD630				

Item	Description
Hide Filter Matches	Hides matched criteria in the Messages Panel.
Show Filter Matches	Shows matched criteria in the Messages Panel.
Syntax Help	Gives quick syntax descriptions.
Apply	Applies the filter criteria to the Messages Panel report, without closing the window.
Type, ID, Message, Source Location, Log Location, Time, Report	Log file report criteria to use when filtering.

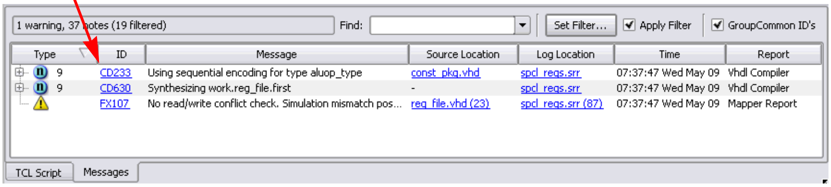
The following is a filtering example.



○ Hide Filter Matches ● Show Filter Matched

	Enable	Type	ID	Message	Source Location	Log Location	Time
1	☑	Warning	FX107				
2	☑	Note	CD233				
3	☑	Note	CD630				

Show Filter Matches

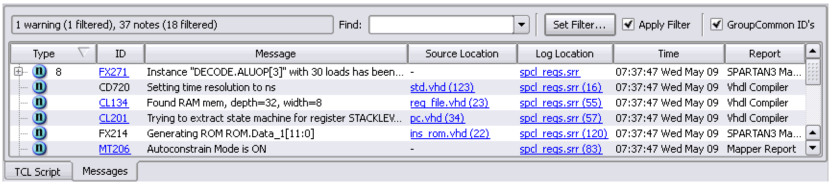


1 warning, 37 notes (19 filtered)

☑ Apply Filter ☑ GroupCommon ID's

Type	ID	Message	Source Location	Log Location	Time	Report
⊞ n 9	CD233	Using sequential encoding for type aluop_type	const_pkg.vhd	spcl_regs.srr	07:37:47 Wed May 09	Vhdl Compiler
⊞ n 9	CD630	Synthesizing work.reg_file.first	-	spcl_regs.srr	07:37:47 Wed May 09	Vhdl Compiler
⚠	FX107	No read/write conflict check. Simulation mismatch pos...	reg_file.vhd (23)	spcl_regs.srr (87)	07:37:47 Wed May 09	Mapper Report

Hide Filter Matches



1 warning (1 filtered), 37 notes (18 filtered)

☑ Apply Filter ☑ GroupCommon ID's

Type	ID	Message	Source Location	Log Location	Time	Report
⊞ n 8	FX271	Instance "DECODE.ALUOP[3]" with 30 loads has been...	-	spcl_regs.srr	07:37:47 Wed May 09	SPARTAN3 Ma...
n	CD720	Setting time resolution to ns	std.vhd (123)	spcl_regs.srr (16)	07:37:47 Wed May 09	Vhdl Compiler
n	CL134	Found RAM mem, depth=32, width=8	reg_file.vhd (23)	spcl_regs.srr (55)	07:37:47 Wed May 09	Vhdl Compiler
n	CL201	Trying to extract state machine for register STACKLEV...	pc.vhd (34)	spcl_regs.srr (57)	07:37:47 Wed May 09	Vhdl Compiler
n	FX214	Generating ROM ROM.Data_1[11:0]	ins_rom.vhd (22)	spcl_regs.srr (120)	07:37:47 Wed May 09	SPARTAN3 Ma...
n	MT206	Autoconstrain Mode is ON	-	spcl_regs.srr (83)	07:37:47 Wed May 09	Mapper Report

Output Windows (Tcl Script and Watch Windows)

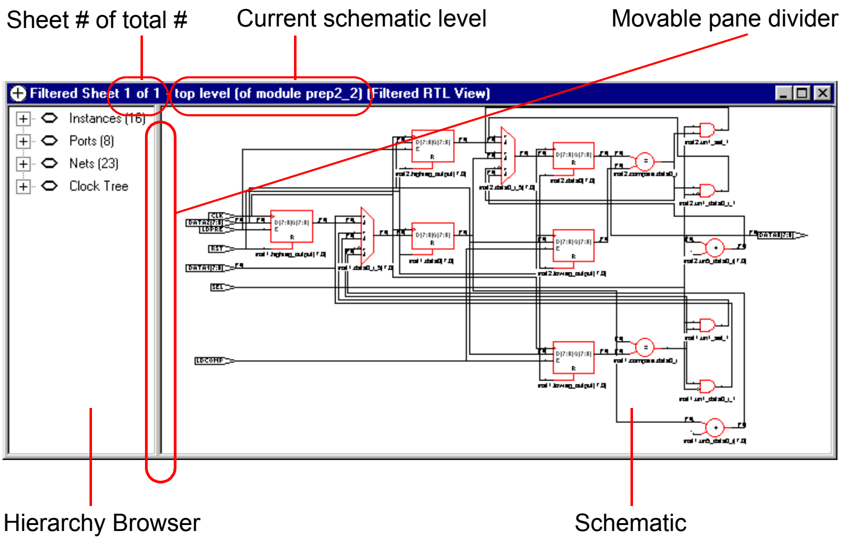
Output windows can display or remove the Tcl Script and Log Watch output windows simultaneously from the Project view, by selecting View->Output Windows from the main menu. Refer to Watch Window, on page 58 and Tcl Script and Messages Windows, on page 62 for more information.

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 69.

To display an 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 72. 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.



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:

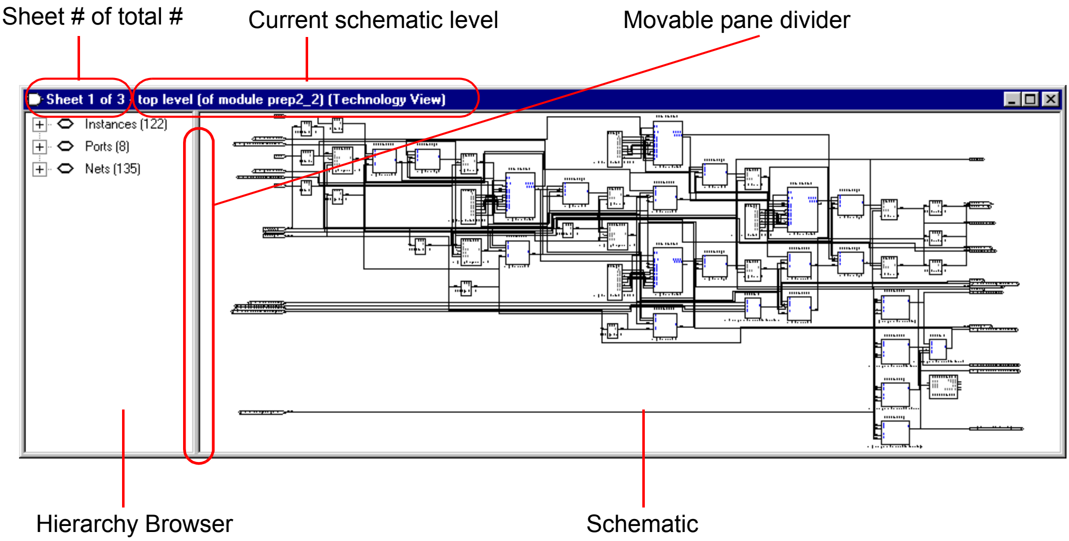
For information about...	See...
Hierarchy Browser	Hierarchy Browser, on page 72
Procedures for RTL view operations like crossprobing, searching, pushing/popping, filtering, flattening, etc.	Working in the Schematic Views, on page 258 of the *User Guide.*
Explanations or descriptions of features like object display, filtering, flattening, etc.	HDL Analyst Tool, on page 307
Commands for RTL view operations like filtering, flattening, etc.	Accessing HDL Analyst Commands, on page 309 HDL Analyst Menu, on page 238
Viewing commands like zooming, panning, etc.	View Menu: RTL and Technology Views Commands, on page 138
History commands: Back and Forward	View Menu: RTL and Technology Views Commands, on page 138
Search command	Find Command (HDL Analyst), on page 129

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 68.

To display a Technology view, first synthesize your design, and then either 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 72. 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.



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 72
Procedures for Technology view operations like crossprobing, searching, pushing/popping, filtering, flattening, etc.	Working in the Schematic Views, on page 258 of the *User Guide*
Explanations or descriptions of features like object display, filtering, flattening, etc.	HDL Analyst Tool, on page 307
Commands for Technology view operations like filtering, flattening, etc.	Accessing HDL Analyst Commands, on page 309 HDL Analyst Menu, on page 238
Viewing commands like zooming, panning, etc.	View Menu: RTL and Technology Views Commands, on page 138
History commands: Back and Forward	View Menu: RTL and Technology Views Commands, on page 138
Search command	Find Command (HDL Analyst), on page 129

Hierarchy Browser

The Hierarchy Browser is the left pane in the RTL and Technology views. (See RTL View, on page 68 and Technology View, on page 69.) The Hierarchy Browser categorizes the design objects in a series of trees, and lets you browse the design hierarchy or select objects. Selecting an object in the Browser selects that 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 73 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 you can use the scroll bar. Use the Shift or Ctrl keys to select multiple objects. See Navigating With a Hierarchy Browser, on page 332 for more information about using the Hierarchy Browser for navigation and crossprobing.

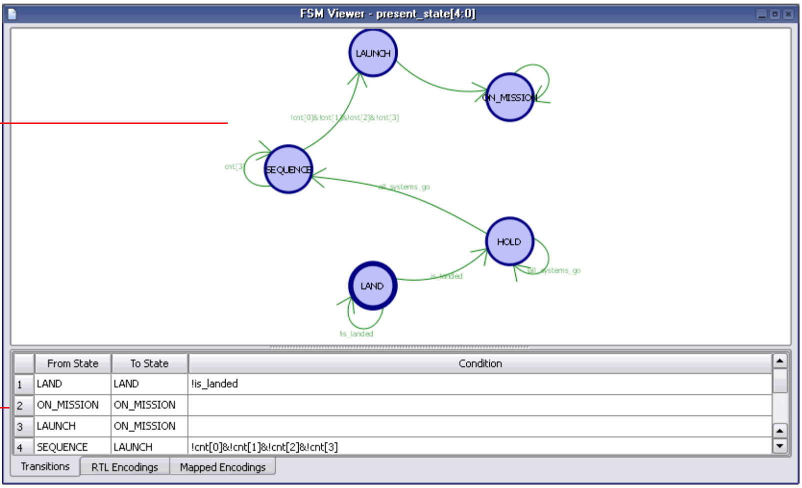
Hierarchy Browser Symbols

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

Symbol	Description	Symbol	Description
⬭	Folder	▷	Buffer
▶	Input port	D	AND gate
▷	Output port	Do	NAND gate
⬬	Bidirectional port	⊃	OR gate
o—o	Net	⊃o	NOR gate
□	Other primitive instance	⊃	XOR gate
■	Hierarchical instance	⊃o	XNOR gate
□	Technology-specific primitive or inferred ROM	⊕	Adder
□	Register or inferred state machine	⊛	Multiplier
▷	Multiplexer	⊜	Equal comparator
▷	Tristate	(<)	Less-than comparator
▷o	Inverter	(<=)	Less-than-or-equal comparator

FSM Viewer Window

Pushing down into a state machine primitive in the RTL view displays the FSM Viewer and enables the FSM toolbar. The FSM Viewer contains graphical information about the finite state machines (FSMs) in your design. The window has a state-transition diagram and tables of transitions and state encodings.



State-Transition Diagram

Transitions and Encodings Tables

	From State	To State	Condition
1	LAND	LAND	!is_landed
2	ON_MISSION	ON_MISSION	
3	LAUNCH	ON_MISSION	
4	SEQUENCE	LAUNCH	!cnt[0]&!cnt[1]&!cnt[2]&!cnt[3]

For the FSM Viewer to display state machine names for a Verilog design, you must use the Verilog parameter keyword. If you specify state machine names using the define keyword, the FSM Viewer displays the binary values for the state machines, rather than their names.

You can toggle display of the FSM tables on and off with the Toggle FSM Table icon ( ) on the FSM toolbar. The FSM tables are in the following panels:

- The Transitions panel describes, for each transition, the From State, To State, and Condition of transition.
- The RTL Encodings panel describes the correlation, in the RTL view, between the states (State) and the outputs (Register) of the FSM cell.
- The Mapped Encodings panel describes the correlation, in the Technology view, between the states (State) and their encodings into technology-specific registers. The information in this panel is available only after the design has been synthesized.

The following table describes FSM Viewer operations.

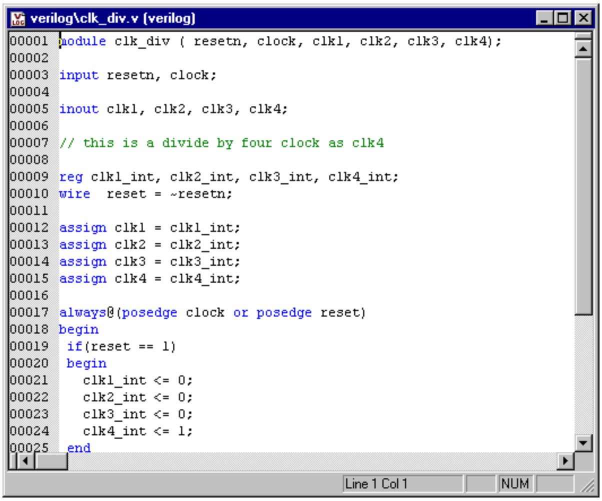
To accomplish this...	**Do this...**
Open the FSM Viewer	Run the FSM Compiler or the FSM Explorer. Use the push/pop mode in the RTL view to push down into the FSM and open the FSM Viewer window.
Hide/display the table	Use the FSM icons.
Filter selected states and their transitions	Select the states. Right-click and choose the filter criteria from the popup, or use the FSM icons.
Display the encoding properties of a state	Select a state. Right-click to display its encoding properties (RTL or Mapped).
Display properties for the state machine	Right-click the window, outside the state-transition diagram. The property sheet shows the selected encoding method, the number of states, and the total number of transitions among states.
Crossprobe	Double-click a register in an RTL or Technology view to see the corresponding code. Select a state in the FSM view to highlight the corresponding code or register in other open views.

See also:

- Pushing and Popping Hierarchical Levels, on page 329, for information on the operation of pushing into a state machine.
- FSM Viewer Toolbar, on page 99, for information on the FSM icons.
- See Using the FSM Viewer, on page 317 of the *User Guide* for more information on using the FSM viewer.

Text Editor View

The Text Editor view displays text files. These can be constraint files, source code 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 HDL Source Files with the Built-in Text Editor, on page 32 in the *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
```

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 (HDL) 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.

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 make the string unique and 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.

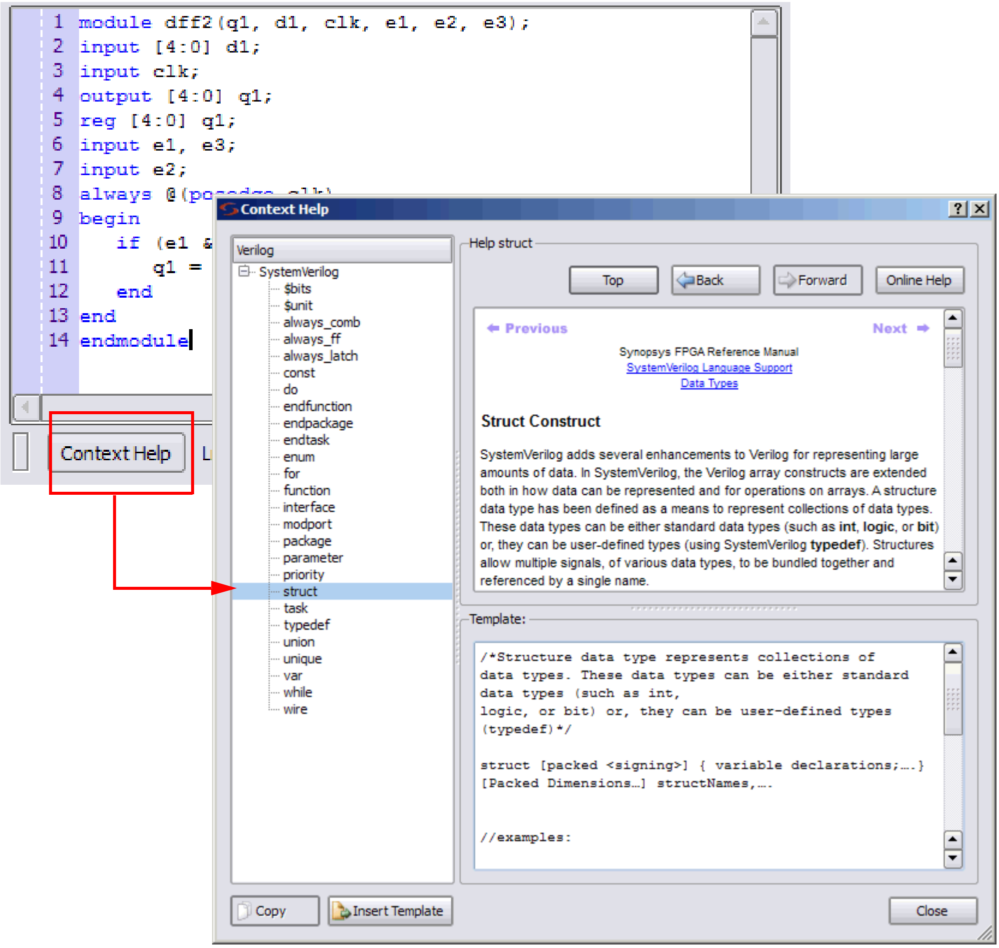
Feature	Description
Deleting a bookmark	Click the line with the bookmark. Choose Edit ->Toggle Bookmark, type Ctrl-F2, or click the Toggle Bookmark icon (⚑) on the Edit toolbar.
Deleting all bookmarks	Choose Edit ->Delete all Bookmarks, type Ctrl-Shift-F2, or click the Clear All Bookmarks icon (⚑) on the Edit toolbar.
Editing columns	Press and hold Alt, then drag the mouse down a column of text to select it.
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 259, for information on setting Text Editor preferences.
- File Menu, on page 117, for information on printing setup operations.
- Edit Menu Commands for the Text Editor, on page 125, for information on Text Editor editing commands.
- Text Editor Popup Menu, on page 282, for information on the Text Editor popup menu.
- Text Editor Toolbar, on page 100, for information on bookmark icons of the Edit toolbar.
- Keyboard Shortcuts, on page 102, for information on keyboard shortcuts that can be used in the Text Editor.

Context Help Editor Window

Use the Context Help button to copy SystemVerilog constructs into your source file. This feature is currently supported for some of the SystemVerilog constructs. When you load a SystemVerilog file into the UI, the Context Help button displays at the bottom of the window. Click this button to display the Context Help Editor.



```
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;
always @(po
begin
   if (e1 &
      q1 =
   end
end
endmodule
```

Struct Construct

SystemVerilog adds several enhancements to Verilog for representing large amounts of data. In SystemVerilog, the Verilog array constructs are extended both in how data can be represented and for operations on arrays. A structure data type has been defined as a means to represent collections of data types. These data types can be either standard data types (such as **int**, **logic**, or **bit**) or, they can be user-defined types (using SystemVerilog **typedef**). Structures allow multiple signals, of various data types, to be bundled together and referenced by a single name.

```
/*Structure data type represents collections of
data types. These data types can be either standard
data types (such as int,
logic, or bit) or, they can be user-defined types
(typedef)*/

struct [packed <signing>] { variable declarations;….}
[Packed Dimensions…] structNames,….


//examples:
```

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

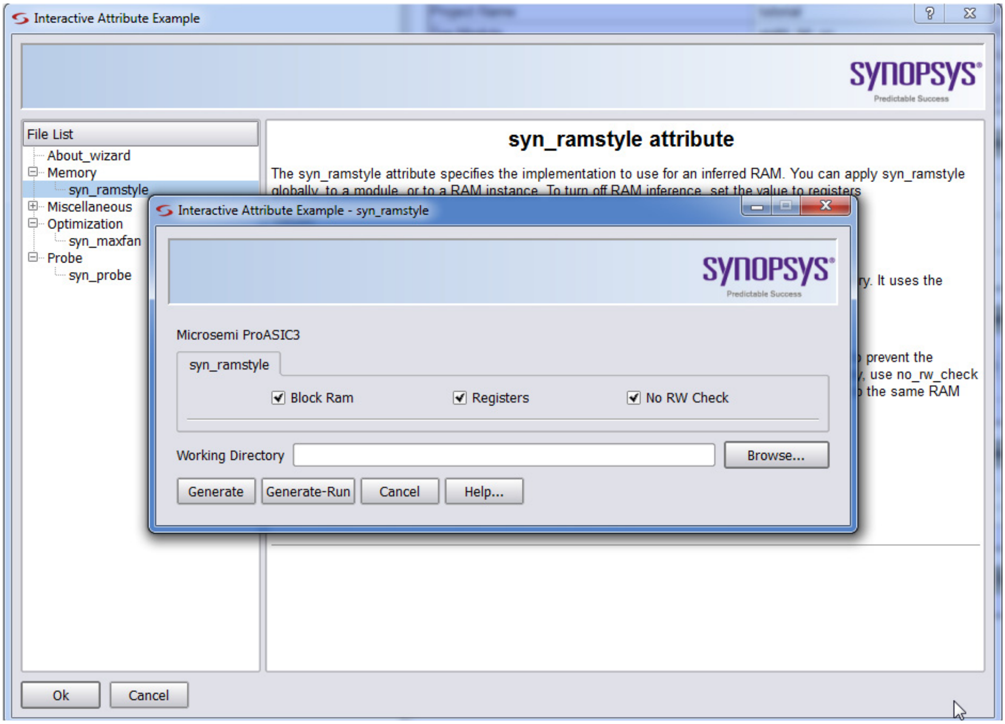
Insert Template button is also enabled. When you click the Insert Template button, the code shown in the template window is inserted into your SystemVerilog file at the location of the cursor. This allows you to easily insert code and modify it for the design that you are going to synthesize. If you want to copy only parts of the template, select the code you want to insert and click Copy. You can then paste it into your file.

Field/Option	Description
Top	Takes you to the top of the context help page for the selected construct.
Back	Takes you back to the last context help page previously viewed.
Forward	Once you have gone back to a context help page, use Forward to return to the original context help page from where you started.
Online Help	Brings up the interactive online help for the synthesis tool.
Copy	Allows you to copy selected code from the Template file and paste it into the editor file.
Insert Template	Automatically copies the code description in its entirety from the Template file to the editor file.

Interactive Attribute Examples

The Interactive Attribute Examples wizard lets you select pre-defined attributes to run in a project. To use this tool:

1. Click Help.
2. Click Interactive Attribute Examples.



syn_ramstyle attribute

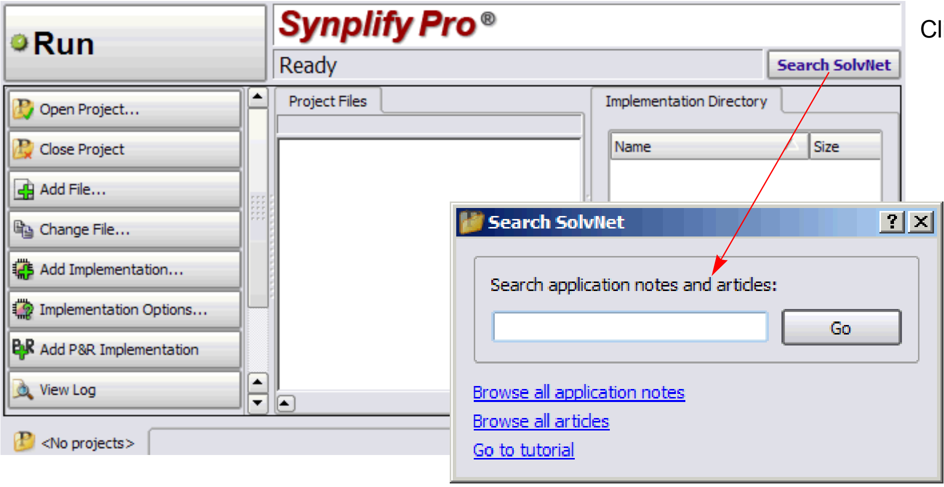
The syn_ramstyle attribute specifies the implementation to use for an inferred RAM. You can apply syn_ramstyle globally, to a module, or to a RAM instance. To turn off RAM inference, set the value to registers

ry. It uses the

prevent the
, use no_rw_check
the same RAM

Search SolvNet

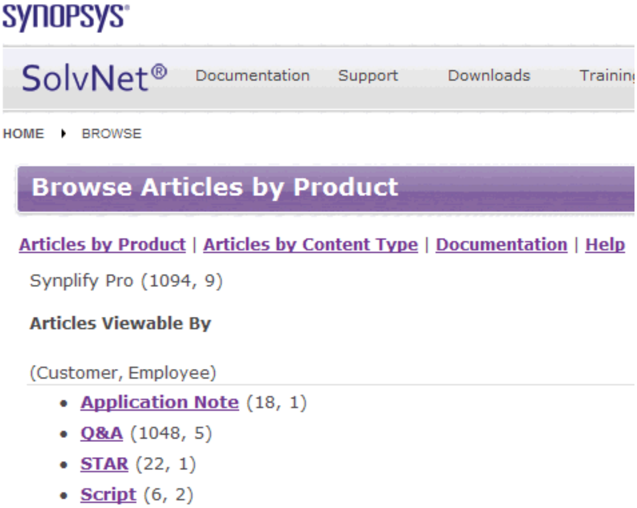
The Synopsys FPGA synthesis tools provide an easy way to access SolvNet from within the Project view. Click the Search SolvNet button in the GUI, a Search SolvNet dialog box appears.



Click

You can search the SolvNet database for Articles and Application Notes using the following methods:

- Specify a topic in the Search application notes and articles field, then click the Go button—takes you to Application Notes and Articles on SolvNet related to the topic.
- Click the Browse all application notes link—takes you to a SolvNet page that links to all the Synopsys FPGA products Application Notes.
- Click the Browse all articles link—takes you to the Browse Articles by Product SolvNet page.



SYNOPSYS

SolvNet® Documentation Support Downloads Trainin

HOME ▸ BROWSE

Browse Articles by Product

Articles by Product | Articles by Content Type | Documentation | Help

Synplify Pro (1094, 9)

Articles Viewable By

(Customer, Employee)

- Application Note (18, 1)
- Q&A (1048, 5)
- STAR (22, 1)
- Script (6, 2)

- Click the Go to tutorial link—takes you to the tutorial page for the Synopsys FPGA product you are using.

FSM Compiler

The FSM Compiler performs proprietary, state-machine optimization techniques (other synthesis tools treat state machines as regular logic). 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.

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 for FPGAs or for 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. For example, in a 16-state machine 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 method 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. You can view the same information graphically with the FSM viewer. The graphical description of a state machine makes it easier to verify behavior. For information on the FSM Viewer, see FSM Viewer Window, on page 74.

See also:

- Log File, on page 435, for information on the log file.
- RTL View and Technology View Popup Menu Commands, on page 301, 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 perform. For more information, see *Deciding when to Optimize State Machines, on page 221* of the *User Guide.*

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	Disable (enable) 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.

FSM Explorer

The FSM Explorer automatically explores different encoding styles for state machines and picks the style best suited to your design. The FSM explorer runs the FSM viewer to identify the finite state machines in a design, then analyzes the FSMs to select the optimum encoding style for each.

To enable the FSM Explorer, do one of the following:

- Turn on the FSM Explorer check box in the Project view
- Display the Implementation Options dialog box (Project ->Implementation Options) and enable the FSM Explorer option on the Options/Constraints panel.

The FSM Explorer runs during synthesis. The cost of running analysis is significant, so when analysis finishes, the encoding information is saved to a file. The synthesis tool reuses the file in subsequent synthesis iterations, which reduces overhead and saves runtime by not reanalyzing the design when you recompile. However, if you make changes to your design or your state machine, you must rerun the FSM Explorer (Run ->FSM Explorer or the F10 key) to reanalyze the encoding.

For more information about using the FSM Explorer, see Running the FSM Explorer, on page 226 in the *User Guide*.

Using the Mouse

The mouse button operations in Synopsys FPGA products are standard; refer to Mouse Operation Terminology for a summary of supported functions. The Synopsys FPGA tools also provide support for:

- Using Mouse Strokes, on page 87
- Using the Mouse Buttons, on page 89
- Using the Mouse Wheel, on page 91

Mouse Operation Terminology

The following terminology is used to refer to mouse operations:

Term	Meaning
Click	Click with the *left* mouse button: press then release it without moving the mouse.
Double-click	Click the left mouse button twice rapidly, without moving the mouse.
Right-click	Click with the right mouse button.
Drag	Press the left mouse button, hold it down while moving the mouse, then release it. Dragging an object moves the object to where the mouse is released; then, releasing is sometimes called "*dropping*". Dragging initiated when the mouse is not over an object often traces a selection rectangle, whose diagonal corners are at the press and release positions.
Press	Depress a mouse button; unless otherwise indicated, the left button is implied. It is sometimes used as an abbreviation for "press and hold".
Hold	Keep a mouse button depressed. It is sometimes used as an abbreviation for "press and hold".
Release	Stop holding a mouse button depressed.

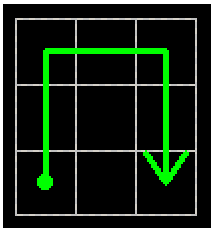
Using Mouse Strokes

Mouse strokes are used to quickly perform simple repetitive commands. Mouse strokes are drawn by pressing and holding the right mouse button as you draw the pattern. The stroke must be at least 16 pixels in width or height to be recognized. You will see a green mouse trail as you draw the stroke (the actual color depends on the window background color).

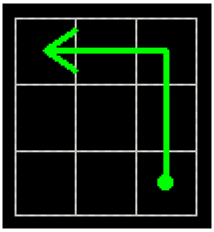
Some strokes are context sensitive. That is, the interpretation of the stroke depends upon the window in which the stroke is started. For example, in an Analyst view, the right stroke means "Next Sheet." In a dialog box, the right stroke means "OK."

For information on each of the available mouse strokes, consult the Mouse Stroke Tutor.

The strokes you draw are interpreted on a grid of one to three rows. 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 Last Operation

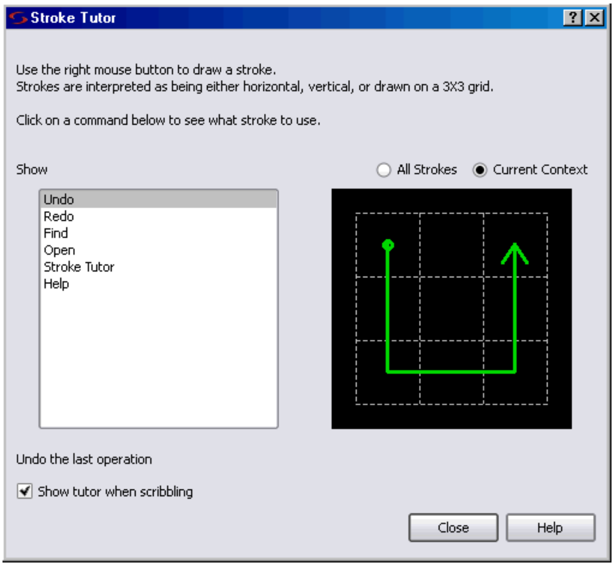


Back to Previous View

The Mouse Stroke Tutor

Do one of the following to access the Mouse Stroke Tutor:

- Help->Stroke Tutor
- Draw a question mark stroke ("?")
- Scribble (Show tutor when scribbling must be enabled on the Stroke Help dialog box)



Use the right mouse button to draw a stroke.
Strokes are interpreted as being either horizontal, vertical, or drawn on a 3X3 grid.

Click on a command below to see what stroke to use.

Show ○ All Strokes ● Current Context

Undo
Redo
Find
Open
Stroke Tutor
Help

Undo the last operation

☑ Show tutor when scribbling

The tutor displays the available strokes along with a description and a diagram of the stroke. You can draw strokes while the tutor is displayed.

Mouse strokes are context sensitive. When viewing the Stroke Tutor, you can choose All Strokes or Current Context to view just the strokes that apply to the context of where you invoked the tutor. For example, if you draw the "?" stroke in an Analyst window, the Current Context option in the tutor shows only those strokes recognized in the Analyst window.

You can display the tutor while working in a window such as the Analyst RTL view. However you cannot display the tutor while a modal dialog is displayed, as input is restricted to the modal dialog.

Using the Mouse Buttons

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

- You select an object by clicking it. You 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 white space (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 clicking 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 68, and Technology View, on page 69, for information on HDL Analyst views.

Most of the mouse button operations involve selecting and deselecting 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 the wheel 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.

User Interface Preferences

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

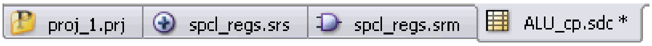
Preferences	Description	For option descriptions, see...
Text Editor	Fonts and colors	Editor Options Command
HDL Analyst tool (RTL/Technology views)	HDL Analyst options	HDL Analyst Menu
Project view	Organization and display of project files	Project View Options Command

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.
- IP Toolbar — Working with IPs and the VCS simulator.
- Analyst Toolbar — Manipulation of RTL and Technology views.
- View Toolbar — Viewing and hierarchy navigation.
- FSM Viewer Toolbar — Display of finite state machine (FSM) information.
- Text Editor Toolbar — Text Editor bookmark commands.

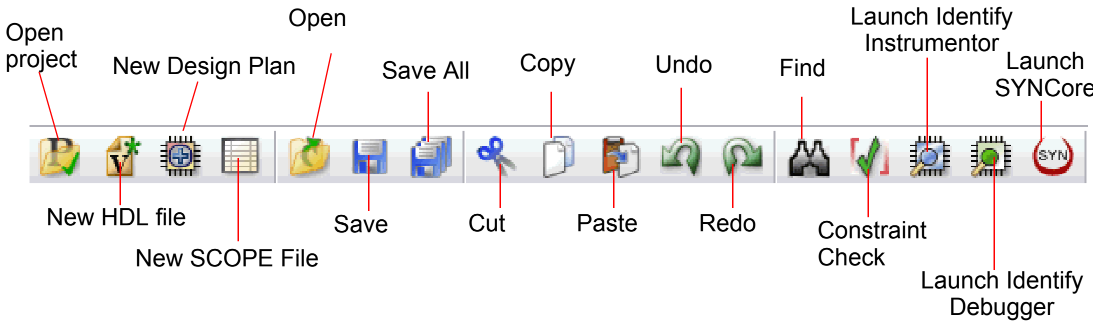
You can enable or disable the display of individual toolbars – see Toolbar Command, on page 140.

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:



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

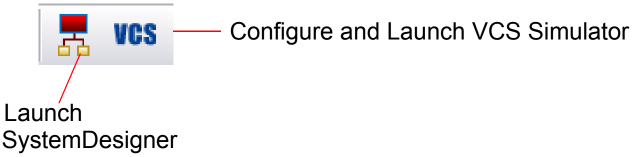
- File Menu, on page 117
- Edit Menu, on page 124

Icon	Description
Open Project	Displays the Open Project dialog box to create a new project or to open an existing project. 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.
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 All.

Icon	Description
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.
Constraint Check	Checks the syntax and applicability of the timing constraints in the fdc file for your project and generates a report (*project_name*_cck.rpt). Same as Run->Constraint Check.
Launch Identify Instrumentor	Launches the Synopsys Identify Instrumentor product. For more information, see Working with the Identify Tool Set, on page 357 of the User Guide.
Launch Identify Debugger	Launches the Synopsys Identify Debugger product. For more information, see Working with the Identify Tool Set, on page 357 of the User Guide.
Launch SYNCore	Launches the SYNCore IP wizard. This tool helps you build IP blocks such as memory models for your design. For more information, see Launch SYNCore Command, on page 187.
Launch SystemDesigner	Not applicable for Microsemi technologies.

IP Toolbar

The IP toolbar provides the following icons, by default:

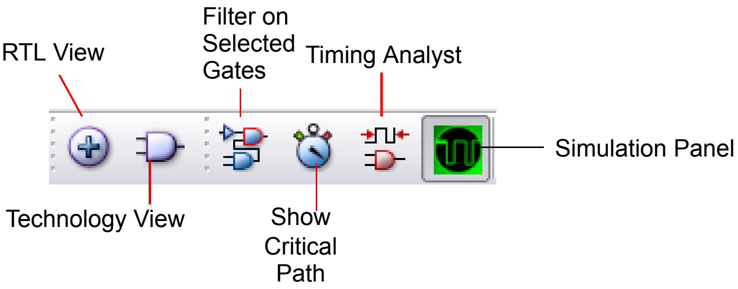


The following table describes the default IP icons.

Icon	Description
Launch SystemDesigner	Not applicable for Microsemi technologies.
VCS Simulator	Configures and launches the VCS simulator.

Analyst Toolbar

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

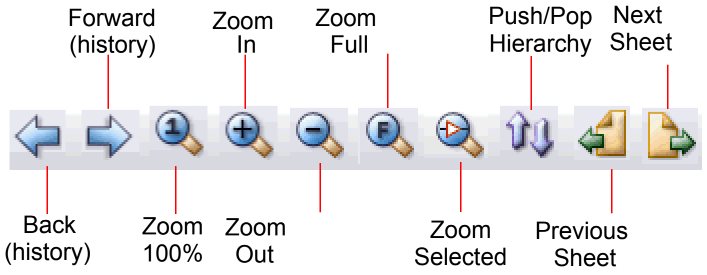


The following table describes the default Analyst icons. Each is equivalent to an HDL Analyst menu command – see HDL Analyst Menu, on page 238, for more information.

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.
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.
Timing Analyst...	Generates and displays a custom timing report and view. The timing report provides more information than the default report (specific paths or more than five paths) or one that provides timing based on additional analysis constraint files. See Analysis Menu, on page 226. Same as Analysis ->Timing Analyst.
Simulation Panel	Not applicable for Microsemi technologies.

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:



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, on page 137, for more information. Zoom icons are also available in some other views.

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]

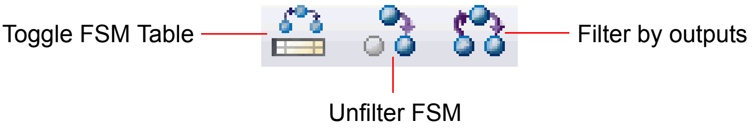
Icon	Description
Zoom Selected	When selected, zooms in on only the selected objects to the full window size.
Push/Pop Hierarchy	Toggles traversing the hierarchy using the push/pop mode. Same as View ->Push/Pop Hierarchy.
Previous Sheet	Displays the previous sheet of a multiple-sheet schematic. Same as 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, FSM Viewer, RTL views, and Technology views.

b. Available only in the FSM Viewer, RTL views, and Technology views.

FSM Viewer Toolbar

When you push down into a state machine primitive in an RTL view, the FSM Viewer displays and enables the FSM toolbar. The FSM Viewer graphically displays the states and transitions. It also lists them in table form. By default, the FSM toolbar provides the following icons, providing access to common FSM Viewer commands.

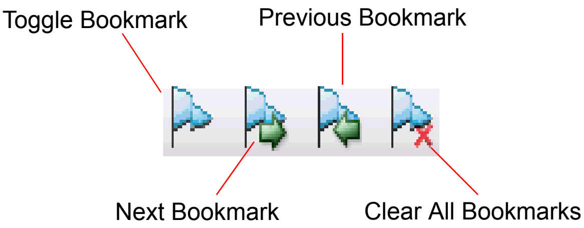


The following table describes the default FSM icons. Each is available in the FSM viewer, and each is equivalent to a View menu command available there – see View Menu, on page 137, for more information.

Icon	Description
Toggle FSM Table	Toggles the display of state-and-transition tables. Same as View->FSM Table.
Unfilter FSM	Restores a filtered FSM diagram so that all the states and transitions are showing. Same as View->Unfilter.
Filter by outputs	Hides all but the selected state(s), their output transitions, and the destination states of those transitions. Same as View->Filter->By output transitions.

Text Editor 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 93.) The Edit toolbar provides the following icons, by default:



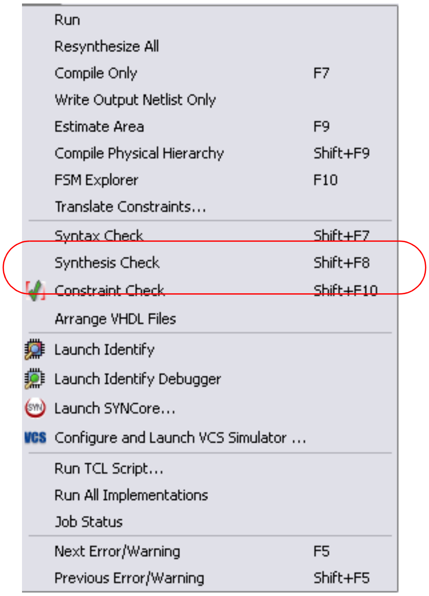
The following table describes the default Edit icons. Each is available in the Text Editor, and each is equivalent to an Edit menu command there – see Edit Menu Commands for the Text Editor, on page 125, for more information.

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.



The following table describes the 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 241).
Ctrl-++ (number pad)	In the FSM Viewer, hides all but the selected state(s), their output transitions, and the destination states of those transitions. Same as View ->Filter ->By output transitions.
Ctrl-+- (number pad)	In the FSM Viewer, hides all but the selected state(s), their input transitions, and the origin states of those transitions. Same as View ->Filter ->By input transitions.
Ctrl-+* (number pad)	In the FSM Viewer, hides all but the selected state(s), their input and output transitions, and their predecessor and successor states. Same as View ->Filter ->By any transition.
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.
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 239).
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 129.)
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 239).

Keyboard Shortcut	Description
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 239).
Ctrl-Enter (Return)	In the FSM Viewer, hides all but the selected state(s). Same as View->Filter->Selected (see View Menu, on page 137).
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 125).
Ctrl-F4	Closes the current window. Same as File ->Close.
Ctrl-F6	Toggles between active windows.
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 125). In an RTL or Technology view, selects the sheet number in a multiple-page schematic. Same as View->View Sheets (see View Menu: RTL and Technology Views Commands, on page 138).
Ctrl-h	In the Text Editor, replaces text. Same as Edit->Replace (see Edit Menu Commands for the Text Editor, on page 125).
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 239).
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 239).

Keyboard Shortcut	Description
Ctrl-l	In the FSM Viewer, or 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: All Views, on page 137).
Ctrl-m	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 239).
Ctrl-n	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-q	In an RTL or Technology view, toggles the display of visual properties of instances, pins, nets, and ports in a design.
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 239).
Ctrl-s	In the Project View, saves the file. Same as File ->Save.
Ctrl-t	Toggles display of the Tcl window. Same as View ->Tcl Window (see View Menu, on page 137).
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 125). In the FSM Viewer, restores a filtered FSM diagram so that all the states and transitions are showing. Same as View->Unfilter (see View Menu: FSM Viewer Commands, on page 139).
Ctrl-v	Pastes the last object copied or cut. Same as Edit ->Paste.

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: RTL and Technology Views Commands, on page 138). 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: RTL and Technology Views Commands, on page 138). 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 125).
Ctrl-Shift-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 245).
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 238).
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 238).
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 125).

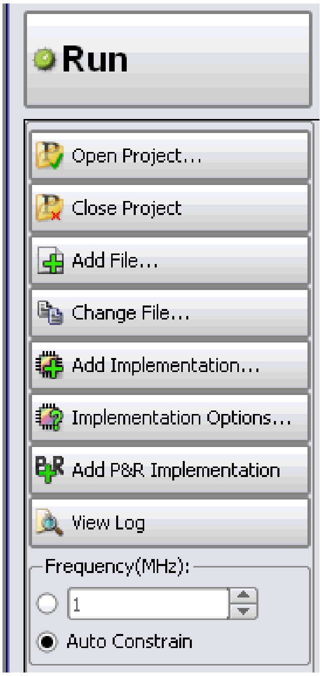
Keyboard Shortcut	Description
d	In an RTL or Technology view, selects the driver for the selected net. Limited to the current schematic. Same as HDL Analyst ->Current Level ->Select Net Driver (see HDL Analyst Menu, on page 238).
Delete (DEL)	Removes the selected files from the project. Same as Project->Remove Files From Project.
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 238).
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: RTL and Technology Views Commands, on page 138). 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 125).
F4	In the Project view, adds a file to the project. Same as Project->Add Source File (see Build Project Command, on page 121). In an RTL or Technology view, zooms the view so that it shows the entire design. Same as View->Full View (see View Menu: RTL and Technology Views Commands, on page 138).
F5	Displays the next source file error. Same as Run->Next Error/Warning (see Run Menu, on page 180).
F7	Compiles your design, without mapping it. Same as Run->Compile Only (see Run Menu, on page 180).
F8	Synthesizes (compiles and maps) your design. Same as Run->Synthesize (see Run Menu, on page 180).

Keyboard Shortcut	Description
F10	In the Project view, runs the FSM Explorer to determine optimum encoding styles for finite state machines. Same as Run ->FSM Explorer (see Run Menu, on page 180). 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: RTL and Technology Views Commands, on page 138).
F11	Toggles zooming in. Same as View->Zoom In (see View Menu: RTL and Technology Views Commands, on page 138).
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 241.
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 238).
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 238).
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 238).
Shift-F2	In the Text Editor, takes you to the previous bookmark.
Shift-F4	Allows you to add source files to your project (Project->Add Source Files). Same as File->New Workspace (see New Workspace Command, on page 122).
Shift-F5	Displays the previous source file error. Same as Run->Previous Error/Warning (see Run Menu, on page 180).

Keyboard Shortcut	Description
Shift-F7	Checks source file syntax. Same as Run->Syntax Check (see Run Menu, on page 180).
Shift-F8	Checks synthesis. Same as Run->Synthesis Check (see Run Menu, on page 180).
Shift-F10	Checks the timing constraints in the constraint files in your project and generates a report (*project_name*_cck.rpt). Same as Run->Constraint Check (see Run Menu, on page 180).
Shift-F11	Toggles zooming out. Same as View->Zoom Out (see View Menu, on page 137).
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 247.

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.



The following table describes the Project View buttons and options.

Button/Option	Action
Open Project...	Opens a new or existing project. Same as File->Open Project (see Open Project Command, on page 122).
Close Project	Closes the current project. Same as File->Close Project (see Run Menu, on page 180).
Add File...	Adds a source file to the project. Same as Project->Add Source File (see Build Project Command, on page 121).

Button/Option	Action
Change File...	Replaces one source file with another. Same as Project ->Change File (see Change File Command, on page 148).
Add Implementation	Creates a new implementation. Same as Project ->New Implementation (see New Workspace Command, on page 122).
Implementation Options/	Displays the Implementation Options dialog box, where you can set various options for synthesis. Same as Project ->Implementation Options (see Implementation Options Command, on page 158).
Add P&R Implementation	Creates a place-and-route implementation to control and run place and route from within the synthesis tool. See Add P&R Implementation Popup Menu Command, on page 298 for a description of the dialog box, and Running Place-and-Route after Synthesis, on page 356 in the *User Guide* for information about using this feature.
View Log	Displays the log file. Same as View ->View Log File (see View Menu, on page 137).
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 Implementation Options dialog box.
Auto Constrain	When Auto Constrain is enabled and no clocks are defined, the software automatically constrains the design to achieve best possible timing by reducing periods of individual clock and the timing of any timed I/O paths in successive steps. See Using Auto Constraints, on page 341 in the *User Guide* for detailed information about using this option. You can also set this option on the Constraints panel of the Implementation Options dialog box.
FSM Compiler	Turning on this option enables special FSM optimizations. Same as enabling the FSM Compiler option on the Options panel of the Implementation Options dialog box (see FSM Compiler, on page 84 and Optimizing State Machines, on page 220 in the *User Guide*).

Button/Option	Action
FSM Explorer	When enabled, the FSM Explorer selects an encoding style for the finite state machines in your design. Same as enabling the FSM Explorer option on the Options panel of the Implementation Options dialog box. For more information, see FSM Explorer, on page 86 and Running the FSM Compiler, on page 223 in the *User Guide*.
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 439). Same as enabling the Resource Sharing option on the Options panel of the Implementation Options dialog box. See Sharing Resources, on page 215 in the *User Guide*.
Retiming	When enabled, improves the timing performance of sequential circuits. The retiming process moves storage devices (flip-flops) across computational elements with no memory (gates/LUTs) to improve the performance of the circuit. This option also adds a retiming report to the log file. Same as enabling the Retiming option on the Options panel of the Implementation Options dialog box. Use the syn_allow_retiming attribute to enable or disable retiming for individual flip-flops. See syn_black_box Directive, on page 921 for syntax details.
Run	Runs synthesis (compilation and mapping). Same as the Run->Synthesize command (see Run Menu, on page 180).
✉ Technical Resource Center	Goes to the web page for the Synopsys Technical Resource Center, which contains Synplicity Business Group (SBG) product Messages.

CHAPTER 3

User Interface Commands

The following describe the commands and ways to access them in the graphical user interface (GUI).

- RTL and Technology Views Popup Menus, on page 301

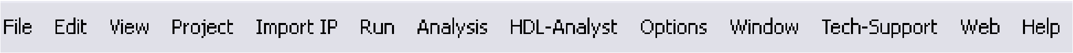
Command Access

The product interface provides access to the commands in the following ways:

- Menus, on page 115
- Context-sensitive Popup Menus, on page 116
- Toolbars, on page 116
- Keyboard Shortcuts, on page 116
- Buttons and Options, on page 116
- Tcl Command Equivalents, on page 116

Menus

The set of commands on the pull-down 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. The menu bar in the Project view is shown below:



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

- File Menu, on page 117
- Edit Menu, on page 124
- View Menu, on page 137
- Project Menu, on page 143
- Import Menu, on page 179
- Run Menu, on page 180
- Analysis Menu, on page 226
- HDL Analyst Menu, on page 238
- Options Menu, on page 250
- Tech-Support Menu, on page 270

- Web Menu, on page 275
- Help Menu, on page 279

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 281, Project View Popup Menus, on page 287, and RTL and Technology Views Popup Menus, on page 301 for information on individual popup menus.

Toolbars

Toolbars contain icons associated with commonly used commands. For more information about toolbars, see Toolbars, on page 93.

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 102 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 110 for details.

Tcl Command Equivalents

The Tcl (Tool Command Language) commands can be entered directly in the Tcl window or included in Tcl scripts that you can run in batch mode. For information about Tcl commands, see Chapter 12, *Batch Commands and Scripts*.

File Menu

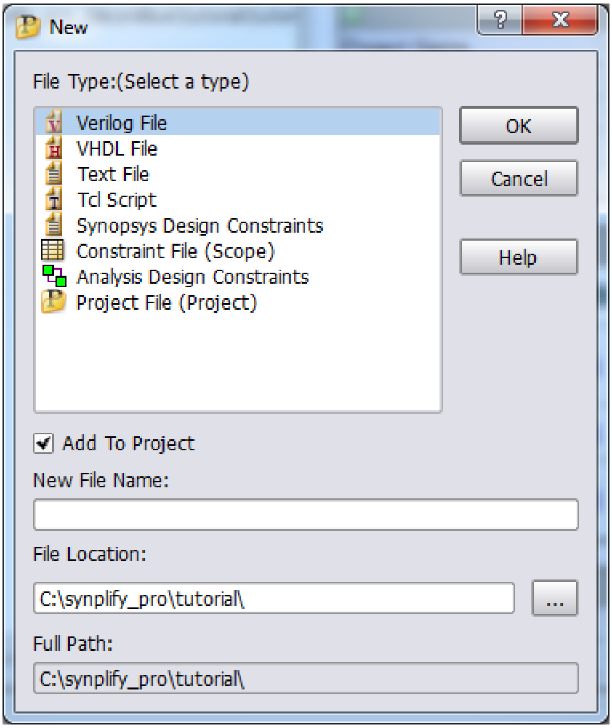
Use the File menu for opening, creating, saving, and closing projects and files. The following table describes the File menu commands.

Command	Description
New	Creates one of the following types of files: Text, Tcl Script, VHDL, Verilog, Synopsys Design Constraints, Constraint, Analysis Design Constraint, and Project. See New Command, on page 118.
Open	Opens a project or file.
Close	Closes a project or file.
Save	Saves a project or a file.
Save As	Saves a project or a file to a specified name.
Save All	Saves all projects or files.
Print	Prints a file. For more information about printing, see the operating system documentation.
Print Setup	Specify print options.
Create Image	This command is available in the following views: • HDL Analyst Views • FSM Viewer A camera pointer (📷) appears. Drag a selection rectangle around the region for which you want to create an image, then release the mouse button. You can also simply click in the current view, then the Create Image dialog appears. See Create Image Command, on page 120.
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 121.
Open Project	Opens a project. See Open Project Command, on page 122.
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 254.

Command	Description
New Workspace	Creates a new project workspace. Prompts you to select projects to add to the workspace. See New Workspace Command, on page 122.
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)	Lists the last six files you opened as separate menu items. Choose a file to open it.
Exit	Exits the session.

New Command

Select File->New to display the New dialog box, where you can select a file type to be created (Verilog, VHDL, text, Tcl script, Synopsys design constraints, constraint, analysis design constraints, project). For most file types, a text editor window opens to allow you to define the file contents. You must provide a file name. You can automatically add the new file to your project by enabling the Add To Project checkbox before clicking OK.



File Type:(Select a type)

- Verilog File
- VHDL File
- Text File
- Tcl Script
- Synopsys Design Constraints
- Constraint File (Scope)
- Analysis Design Constraints
- Project File (Project)

☑ Add To Project

New File Name:

File Location: C:\synplify_pro\tutorial\

Full Path: C:\synplify_pro\tutorial\

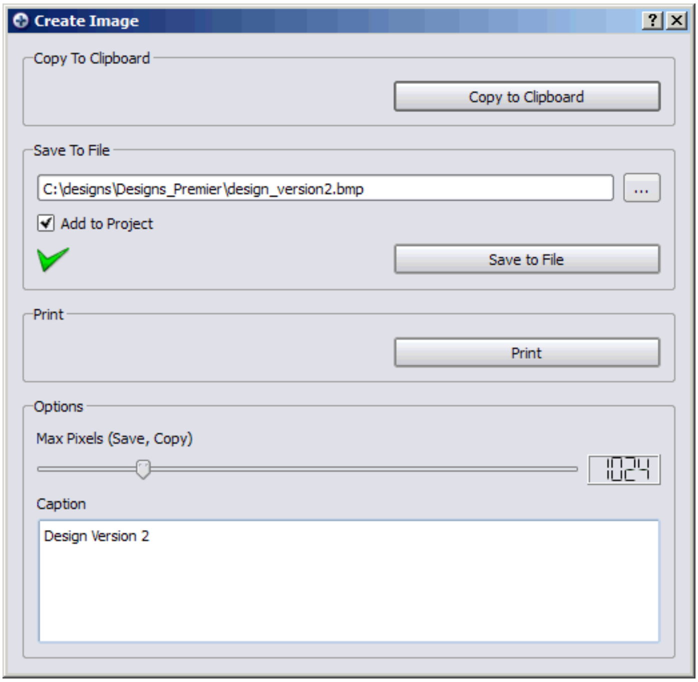
File Type	Opens Window	Directory Name	Extension
Verilog	Text Editor	Verilog	.v
VHDL	Text Editor	VHDL	.vhd
Text	Text Editor	Other	.txt
Tcl Script	Text Editor	Tcl Script	.tcl
FPGA Design Constraints	SCOPE	Constraint	.fdc
Constraint	SCOPE	Constraint	.sdc
Analysis Design Constraints	SCOPE	Analysis Design Constraint	.adc
Project	None	None	.prj

Create Image Command

Select File->Create Image to create a capture image from any of the following views:

- HDL Analyst Views
- FSM Viewer

Drag the camera cursor to define the area for the image. When you release the cursor, the Create Image dialog box appears. Use the dialog box to copy the image, save the image to a file, or to print the image.



C:\designs\Designs_Premier\design_version2.bmp

☑ Add to Project

Max Pixels (Save, Copy): 1024

Caption: Design Version 2

Field/Option	Description
Copy to Clipboard	Copies the image to the clipboard so you can paste it into a selected application (for example, a Microsoft Word file). When you copy an image to the clipboard, a green check mark appears in the Copy To Clipboard field.
Save to File	Saves the image to the specified file. You can save the file in a number of formats (platform dependent) including bmp, jpg, png, ppm, tif, xbm, and xpm.
Add to Project	Adds the saved image file to the Images folder in the Project view. This option is enabled by default.
Save to File button	You must click this button to save an image to the specified file. When you save the image, a green check mark appears in the Save To File field.
Print	Prints the image. When you print the image, a green check mark appears in the Print field.
Options	Allows you to select the resolution of the image saved to a file or copied to the clipboard. Use the Max Pixels slider to change the image resolution.
Caption	Allows you to enter a caption for a saved or copied image. The caption is overlayed at the top-left corner of the image.

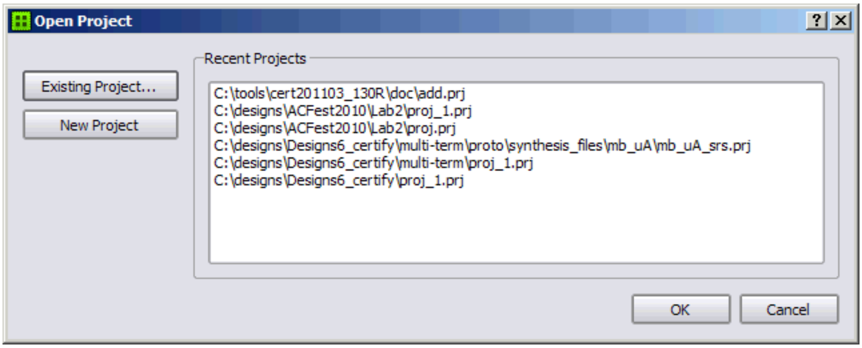
Build Project Command

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 New 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 144.

Open Project Command

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

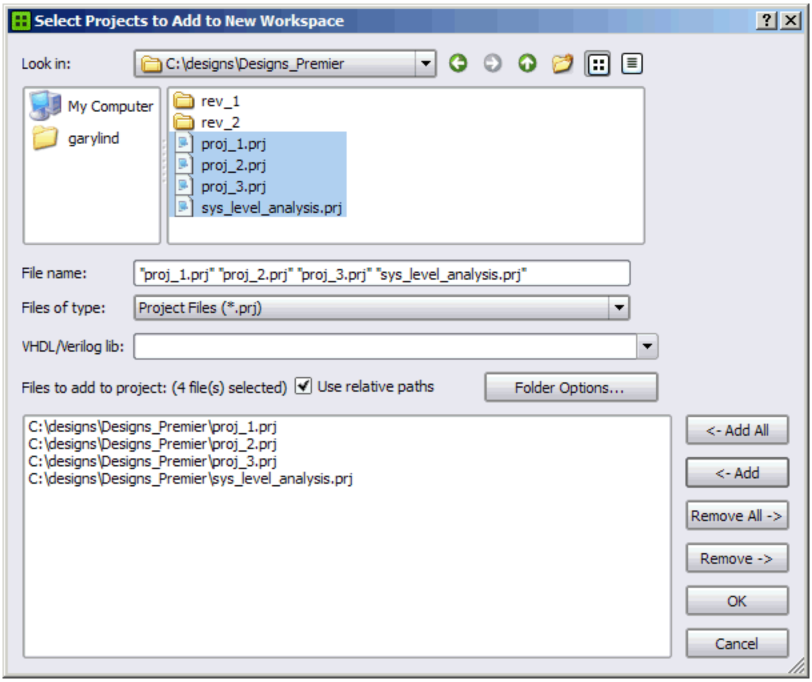


```
C:\tools\cert201103_130R\doc\add.prj
C:\designs\ACFest2010\Lab2\proj_1.prj
C:\designs\ACFest2010\Lab2\proj.prj
C:\designs\Designs6_certify\multi-term\proto\synthesis_files\mb_uA\mb_uA_srs.prj
C:\designs\Designs6_certify\multi-term\proj_1.prj
C:\designs\Designs6_certify\proj_1.prj
```

Field/Option	Description
Existing Project	Displays the Open Project dialog box for opening an existing project.
New Project	Creates a new project and places it in the Project view.

New Workspace Command

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



Select Projects to Add to New Workspace

Look in: C:\designs\Designs_Premier

- rev_1
- rev_2
- proj_1.prj
- proj_2.prj
- proj_3.prj
- sys_level_analysis.prj

File name: "proj_1.prj" "proj_2.prj" "proj_3.prj" "sys_level_analysis.prj"

Files of type: Project Files (*.prj)

VHDL/Verilog lib:

Files to add to project: (4 file(s) selected) ☑ Use relative paths

```
C:\designs\Designs_Premier\proj_1.prj
C:\designs\Designs_Premier\proj_2.prj
C:\designs\Designs_Premier\proj_3.prj
C:\designs\Designs_Premier\sys_level_analysis.prj
```

Field/Option	Description
File name	The name of a project file to add to the workspace. If you enter a name using the keyboard, 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.
VHDL/Verilog lib	The name of the VHDL or Verilog library (default is work).
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.
Use relative paths	When you add files to the project, you can specify either to use the relative path or full path names for the files.

Field/Option	Description
Folder Options	When you add files to folders, you can specify the folder name as either the: • Operating System (OS) folder name • Parent path name from a list provided in the display See Folder Options, on page 146.
<- 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 views).

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

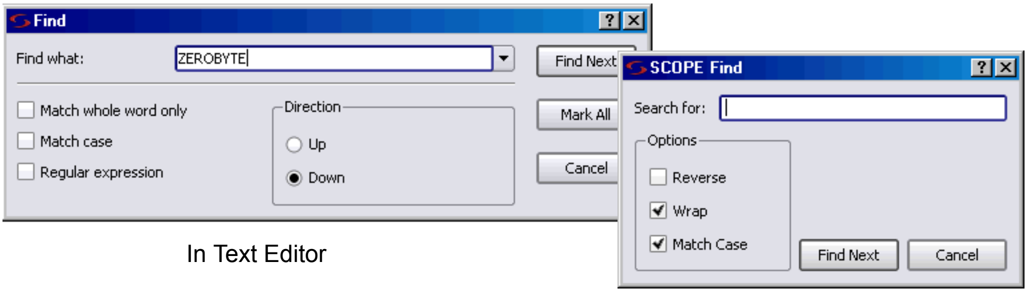
Command	Description
Basic Edit Menu Commands	
Undo	Cancels the last action.
Redo	Performs the action undone by Undo.
Cut	Removes the selected text and makes it available to Paste.
Copy	Duplicates the selected text and makes it available to Paste.
Paste	Pastes text that was cut (Cut) or copied (Copy).

Command	Description
Delete	Deletes the selected object.
Find	Searches the file for text matching a given search string. See Find Command (Text), on page 126. In the RTL view, opens 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 129. In the project view, searches files for text strings; see Find Command (In Project), on page 127.
Find Next	Continues the search initiated by the last Find.
Find in Files	Performs a string search of the target files (see Find in Files Command, on page 133).
Edit Menu Commands for the Text Editor	
All	Selects all text in the file.
Replace	Finds and replaces text. See Replace Command, on page 135.
Goto	Goes to a specific line number. See Goto Command, on page 136.
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.
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.
Select->All	Selects all text in the file (same as All).

Find Command (Text)

Select Edit->Find to display the Find dialog box. In the SCOPE window, the FSM Viewer and the Text Editor window, the command has basic text-based search capabilities. Some search features, like regular expressions and line-number highlighting, are available only in the Text Editor. See Find Command (In Project), on page 127, to search for files in the Project.

The HDL Analyst Find command is different; see Find Command (HDL Analyst), on page 129 for details.



Find

Find what: ZEROBYTE

☐ Match whole word only

☐ Match case

☐ Regular expression

Direction: ○ Up ● Down

SCOPE Find

Search for:

Options: ☐ Reverse ☑ Wrap ☑ Match Case

In Text Editor

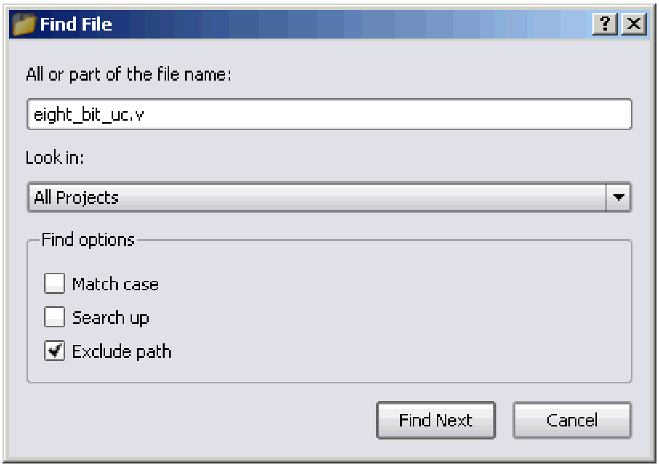
In SCOPE

Field/Option	Description
Find What/Search for	Search string matching the text to find. In the text editor, you can use the pull-down list to view and reuse search strings used previously in the current session.
Match whole word only (text editor only)	When enabled, matches the entire word rather than a portion of the word.
Match Case	When enabled, searching is case sensitive.
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 an empty string.
Direction/Reverse	Changes search direction. In the text editor, buttons select the search direction (Up or Down).

Field/Option	Description
Find Next	Initiates a search for the search string (see Find What/Search for). In the text editor, searching starts again after reaching the end (Down) or beginning (Up) of the file.
Wrap (SCOPE only)	When enabled, searching starts again after reaching the end or beginning (Reverse) of the spread sheet.
Mark All (Text editor only)	Highlights the line numbers of the text matching the search string and closes the Find dialog box.

Find Command (In Project)

Select Edit->Find to display the Find File dialog box. In the Project view, the command has basic text-based search capabilities to locate files in the project.


Find File

All or part of the file name: eight_bit_uc.v

Look in: All Projects

Find options

□ Match case

□ Search up

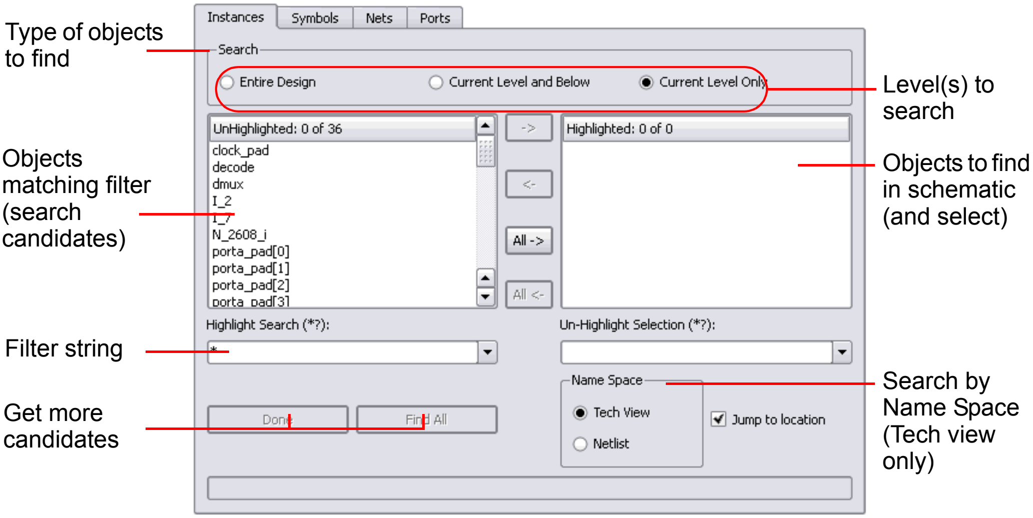
☑ Exclude path

Field/Option	Description
All or part of the file name	Search string matching the file to find. You can specify all or part of the file name.
Look in	Search for files in all projects or limit the search to files only in the specified project.
Match Case	When enabled, searching is case sensitive.

Field/Option	Description
Search up	Searches in the up direction (search terminates when an end of tree is reached in either direction).
Exclude path	Excludes the path name during the search.
Find Next	Initiates a search for the file name string.

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 Using Find for Hierarchical and Restricted Searches, on page 282 of the *User Guide*.



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

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 282 of the *User Guide*.

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 to 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 285 of the *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.

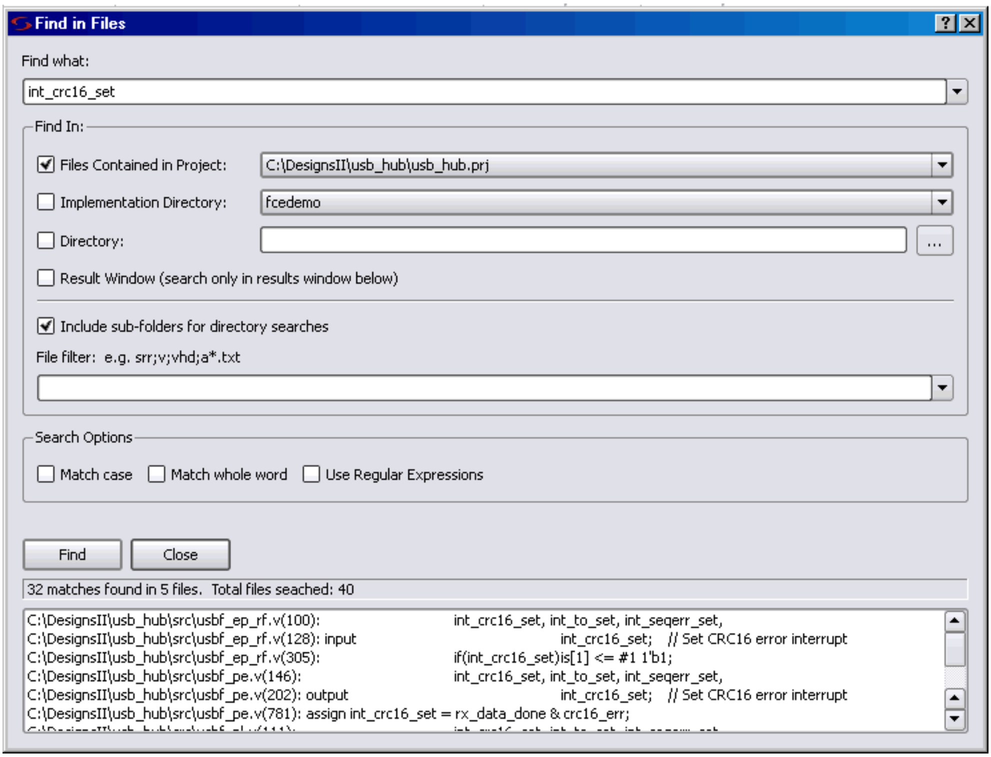
Field/Option	Description
All ->	Moves all names from the UnHighlighted to the Highlighted area, and highlights their objects in the RTL and Technology views.
<- All	Moves all names from the Highlighted to the UnHighlighted area, and unhighlights their objects in the RTL and Technology views.
Highlighted	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 which 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.
Name Space: Tech View	Searches for the specified name using the mapped (srm) database. For more information, see Using Find for Hierarchical and Restricted Searches, on page 282 of the *User Guide*.
Name Space: Netlist	Searches for the specified name using the output netlist file. For more information, see Using Find for Hierarchical and Restricted Searches, on page 282 of the *User Guide*.

Field/Option	Description
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 282 of the *User Guide*.

Find in Files Command

The Find in Files command searches the defined target for the occurrence of a specified search string. The list of files containing the string is reported in the display area at the bottom of the dialog box. For information on using this feature, see Searching Files, on page 152 of the *User Guide*.



Find what: int_crc16_set

☑ Files Contained in Project: C:\DesignsII\usb_hub\usb_hub.prj

☐ Implementation Directory: fcedemo

☐ Directory:

☐ Result Window (search only in results window below)

☑ Include sub-folders for directory searches

File filter: e.g. srr;v;vhd;a*.txt

☐ Match case ☐ Match whole word ☐ Use Regular Expressions

32 matches found in 5 files. Total files seached: 40

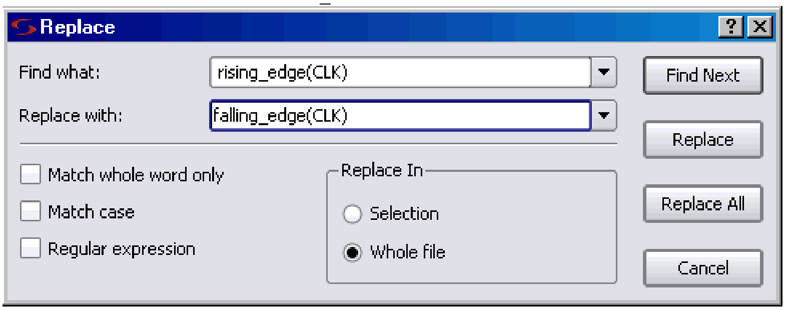
```
C:\DesignsII\usb_hub\src\usbf_ep_rf.v(100):              int_crc16_set, int_to_set, int_seqerr_set,
C:\DesignsII\usb_hub\src\usbf_ep_rf.v(128): input                   int_crc16_set;   // Set CRC16 error interrupt
C:\DesignsII\usb_hub\src\usbf_ep_rf.v(305):              if(int_crc16_set)is[1] <= #1 1'b1;
C:\DesignsII\usb_hub\src\usbf_pe.v(146):                 int_crc16_set, int_to_set, int_seqerr_set,
C:\DesignsII\usb_hub\src\usbf_pe.v(202): output                     int_crc16_set;   // Set CRC16 error interrupt
C:\DesignsII\usb_hub\src\usbf_pe.v(781): assign int_crc16_set = rx_data_done & crc16_err;
```

Field/Option	Description
Find what:	Text string object of search.
Files Contained in Project	Drop-down menu identifying the source project of the files to be searched.
Implementation Directory	Drop-down menu restricting project search to a specific implementation or all implementations.
Directory	Identifies directory for files to be searched.

Field/Option	Description
Result Window	Allows a secondary search string (Find what) to be applied to the targets reported from the initial search.
Include sub-folders for directory searches	When checked, extends the search to sub-directories of the target directory.
File filter	Excludes files from the search by filename extension.
Search Options	Standard string search options; check to enable.
Find	Initiates search.
Result Display	List of files containing search string. Status line lists the number of matches in each file and the number of files searched.

Replace Command

Use Edit->Replace to find and optionally replace text in the Text Editor.



Find what: rising_edge(CLK)

Replace with: falling_edge(CLK)

☐ Match whole word only

☐ Match case

☐ Regular expression

Replace In

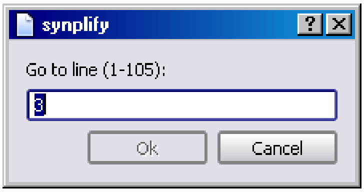
○ Selection

● Whole file

Feature	Description
Find What	Search string matching the text to find. You can use the pull-down 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 pull-down list to view and reuse replacement text used previously in the current session.
Match whole word only	Finds only occurrences of the exact string (strings longer than the Find what string are not recognized).
Match case	When enabled, searching is case sensitive.
Regular expression	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.
Selection	Replace All replaces only the matched occurrence.
Whole file	Replace All replaces all matching occurrences.
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.

Goto Command

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



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.

- View Menu Commands: All Views, on page 137
- View Menu: Zoom Commands, on page 138
- View Menu: RTL and Technology Views Commands, on page 138
- View Menu: FSM Viewer Commands, on page 139

View Menu Commands: All Views

Command	Description
Font Size	Changes the font size in the Project UI of the synthesis tools. You can select one of the following options: • Increase Font Size • Decrease Font Size • Reset Font Size (default size)
Toolbars	Displays the Toolbars dialog box, where you specify the toolbars to display. See Toolbar Command, on page 140.
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.
Output Windows	Displays or removes the Tcl Script/Messages and Watch windows simultaneously in the Project view. Refer to the Tcl Window and Watch Window options for more information.
Tcl Window	When enabled, displays the Tcl Script and Messages windows. All commands you execute in the Project view appear in the Tcl window. You can enter or paste Tcl commands and scripts in the Tcl window. Check for notes, warning, and errors in the Messages window.

Command	Description
Watch Window	When enabled, displays selected information from the log file in the Watch window.
View Log File	Displays a log file report that includes compiler, mapper, and timing information on your design. See View Log File Command, on page 142.
View Result File	Displays a detailed netlist report.

View Menu: Zoom Commands

Command	Description
Zoom In Zoom Out	Lets you Zoom in or out. When selected, a Z-shaped mouse pointer (Z) 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. If your mouse has a wheel feature, use the wheel to pan up and down. To pan left and right, use the Shift key with the wheel.
Full View	Zooms the active view so that it shows the entire design.
Normal View	Zooms the active view to normal size and centers it where you click. If the view is already normal size, clicking centers the view.

View Menu: RTL and Technology Views Commands

These commands are available when the RTL view or Technology view is active. These commands are available in addition to the commands described in View Menu Commands: All Views, on page 137 and View Menu: Zoom Commands, on page 138.

Command	Description
Push/Pop Hierarchy	Traverses design hierarchy using the push/pop mode – see Exploring Design Hierarchy, on page 272 of the *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 141.
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->HDL Analyst Options->Visual Properties. See Visual Properties Panel, on page 268.
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.
Filter	Filters the RTL/Technology view to display only the selected objects.

View Menu: FSM Viewer Commands

The following commands are available when the FSM viewer is active. These commands are in addition to the common commands described in View Menu Commands: All Views, on page 137 and View Menu: Zoom Commands, on page 138.

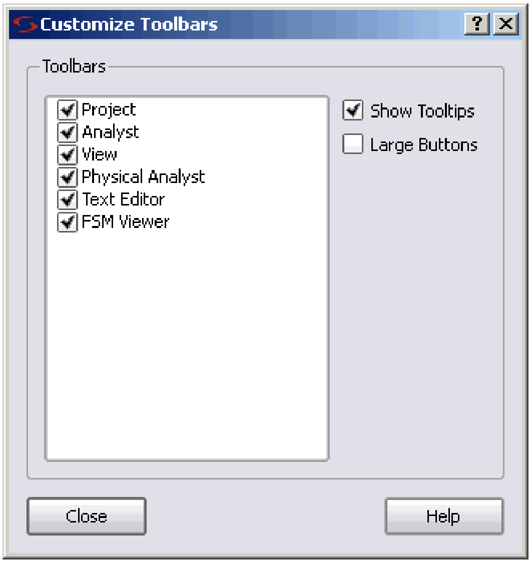
Command	Description
Filter->Selected	Hides all but the selected state(s).
Filter->By output transitions	Hides all but the selected state(s), their output transitions, and the destination states of those transitions.
Filter->By input transitions	Hides all but the selected state(s), their input transitions, and the origin states of those transitions.

Command	Description
Filter->By any transition	Hides all but the selected state(s), their input and output transitions, and their predecessor and successor states.
Unfilter	Restores a filtered FSM diagram so that all the states and transitions are showing.
Cross Probing	Enables cross probing between FSM nodes and RTL view schematic.
Select All States	Selects all the states.
FSM Table	Toggles display of the transition table.
FSM Graph	Toggles FSM state diagram on or off.
Annotate Transitions	Toggles display of state transitions on or off on FSM state diagram
Selection Transcription	
Tool Tips	Toggles state diagram tool tips on or off.
FSM Properties	Displays FSM Properties dialog box.
Unselect All	Unselects all states and transitions.

Toolbar Command

Select View->Toolbars to display the Toolbars dialog box, where you can:

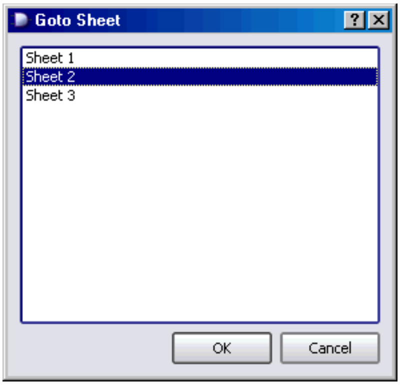
- Choose the toolbars to display
- Customize their appearance



Feature	Description
Toolbars	Lists the available toolbars. Select the toolbars that you want to display.
Show Tooltips	When selected, a descriptive tooltip appears whenever you position the pointer over an icon.
Large Buttons	When selected, large icons are used.

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.

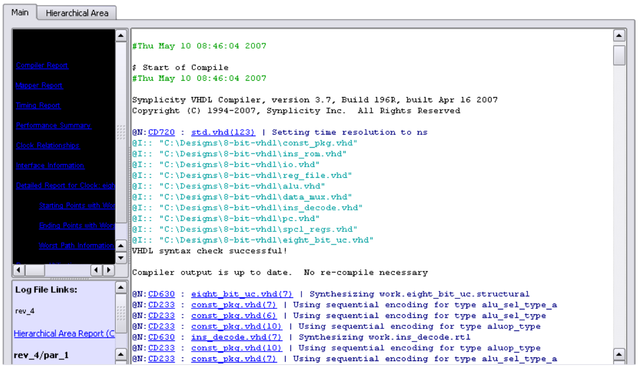


To see if your design has multiple sheets, check the sheet count display at the top of the schematic window.

View Log File Command

View->View Log File displays the log file report for your project. The log file is available in either text (*project_name*.srr) or HTML (*project_name*_srr.htm) format. To enable or disable the HTML file format for the log file, select the View log file in HTML option in the Options->Project View Options dialog box.

When opening the log file, a table of contents appears. Selecting an item from the table of contents takes you to the corresponding HTML page. To go back to the Table of Contents, right-click on the HTML page and select Back from the menu.



```
#Thu May 10 08:46:04 2007

$ Start of Compile
#Thu May 10 08:46:04 2007

Synplicity VHDL Compiler, version 3.7, Build 196R, built Apr 16 2007
Copyright (C) 1994-2007, Synplicity Inc.  All Rights Reserved

@N:CD720 : std.vhd(123) | Setting time resolution to ns
@I:: "C:\Desigms\8-bit-vhdl\const_pkg.vhd"
@I:: "C:\Desigms\8-bit-vhdl\ins_rom.vhd"
@I:: "C:\Desigms\8-bit-vhdl\io.vhd"
@I:: "C:\Desigms\8-bit-vhdl\reg_file.vhd"
@I:: "C:\Desigms\8-bit-vhdl\alu.vhd"
@I:: "C:\Desigms\8-bit-vhdl\data_mux.vhd"
@I:: "C:\Desigms\8-bit-vhdl\ins_decode.vhd"
@I:: "C:\Desigms\8-bit-vhdl\pc.vhd"
@I:: "C:\Desigms\8-bit-vhdl\spcl_regs.vhd"
@I:: "C:\Desigms\8-bit-vhdl\eight_bit_uc.vhd"
VHDL syntax check successful!

Compiler output is up to date.  No re-compile necessary

@N:CD630 : eight_bit_uc.vhd(7) | Synthesizing work.eight_bit_uc.structural
@N:CD233 : const_pkg.vhd(7) | Using sequential encoding for type alu_sel_type_a
@N:CD233 : const_pkg.vhd(6) | Using sequential encoding for type alu_sel_type
@N:CD233 : const_pkg.vhd(10) | Using sequential encoding for type aluop_type
@N:CD630 : ins_decode.vhd(7) | Synthesizing work.ins_decode.rtl
@N:CD233 : const_pkg.vhd(10) | Using sequential encoding for type aluop_type
@N:CD233 : const_pkg.vhd(7) | Using sequential encoding for type alu_sel_type_a
```

Project Menu

You use the Project menu to set implementation options, add or remove files from a project, change project filenames, create new implementations, and archive or copy the project. The Project menu commands change, depending on the view you are in. For example, the HDL Analyst RTL and Technology views only include a subset of the Project menu commands.

The Synplify Pro tools provide a graphical user interface (GUI) with views that help you manage hierarchical designs that can be synthesized independently and imported back to the top-level project in a team design flow called Hierarchical Project Management. This feature is not available for Microsemi technologies.

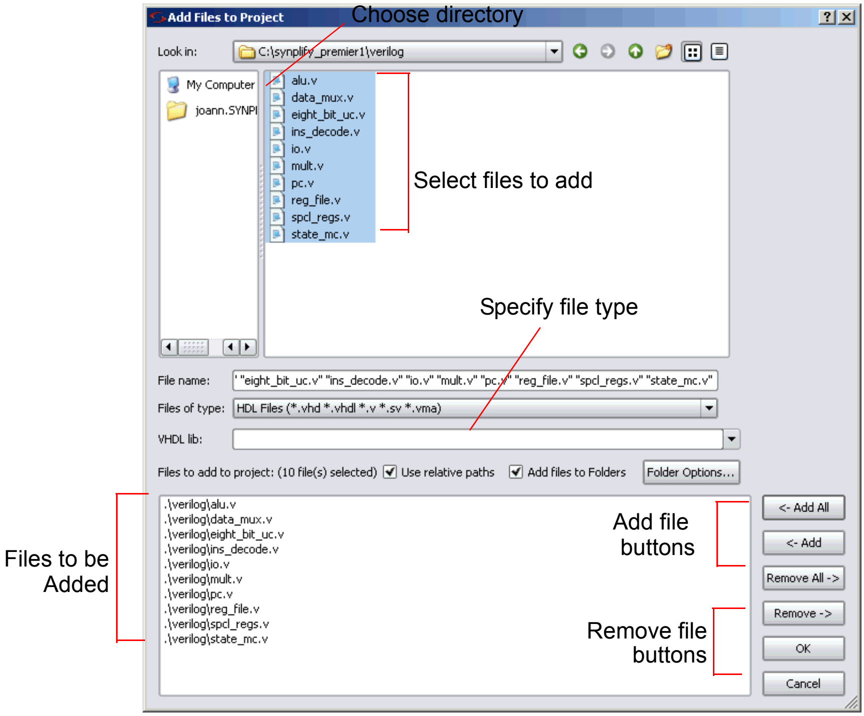
The following table describes the Project menu commands.

Command	Description
Implementation Options	Displays the Implementation Options dialog box, where you set options for implementing your design. See Implementation Options Command, on page 158.
Add Source File	Displays the Select Files to Add to Project dialog box. See Add Source File Command, on page 144.
Remove File 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 148.
Set VHDL 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 148.
Add 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 Add Implementation Command, on page 149).
New Identify Implementation	Creates a new Identify implementation for a current design. To launch the Identify tool set, see the Launch Identify Instrumentor Command, on page 184 and Launch Identify Debugger Command, on page 186.

Command	Description
Convert Vendor Constraints	Not applicable for Microsemi technologies.
Archive Project	Archives a design project. Use this command to archive a full or partial project, or to add files to or remove files from an archived project. See Archive Project Command, on page 150 for a description of the utility wizard options.
Un-Archive Project	Loads an archived project file to the specified directory. See Un-Archive Project Command, on page 152 for a description of the utility wizard options.
Copy Project	Creates a copy of a design project. Use this command to create a copy of a full or partial project. See Copy Project Command, on page 154 for a description of the utility wizard options.
Hierarchical Project Options	Not applicable for Microsemi technologies.
Add SubProject Implementation	Not applicable for Microsemi technologies.

Add Source File Command

Select Project->Add Source File to add files, such as HDL source files, to your project. This selection displays the Select Files to Add to Project dialog box.



Choose directory

Look in: C:\synplify_premier1\verilog

alu.v
data_mux.v
eight_bit_uc.v
ins_decode.v
io.v
mult.v
pc.v
reg_file.v
spcl_regs.v
state_mc.v

Select files to add

Specify file type

File name: " "eight_bit_uc.v" "ins_decode.v" "io.v" "mult.v" "pc.v" "reg_file.v" "spcl_regs.v" "state_mc.v"

Files of type: HDL Files (*.vhd *.vhdl *.v *.sv *.vma)

VHDL lib:

Files to add to project: (10 file(s) selected) ☑ Use relative paths ☑ Add files to Folders

.\verilog\alu.v
.\verilog\data_mux.v
.\verilog\eight_bit_uc.v
.\verilog\ins_decode.v
.\verilog\io.v
.\verilog\mult.v
.\verilog\pc.v
.\verilog\reg_file.v
.\verilog\spcl_regs.v
.\verilog\state_mc.v

Files to be Added

Add file buttons

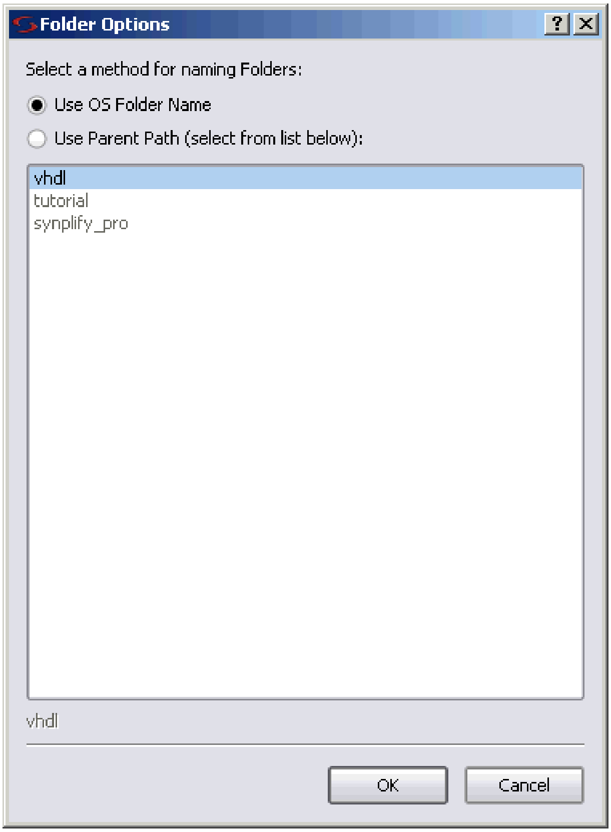
Remove file buttons

Feature	Description
Look in	The directory of the file to add. You can use the pull-down 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	The type (extension) of files to be added to the project. Only files in the active directory that match the file type selected from the drop-down menu are displayed in the list of files. Use All Files to list all files in the directory.
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. For information about adding files to custom folders, see Creating Custom Folders, on page 120.

Feature	Description
Use relative paths	When you add files to the project, you can specify either to use the relative path or full path names for the files.
Add files to Folders	When you add files to the project, you can specify whether or not to automatically add the files to folders. See the Folder Options described below.
Folder Options	When you add files to folders, you can specify the folder name as either the: • Operating System (OS) folder name • Parent path name from a list provided in the display For details, see Folder Options, on page 146.

Folder Options

When you click the Folder Options button of the Select Projects to Add to New Workspace dialog box, the Folders Options dialog box is displayed. Select a method for naming folders here.



Select a method for naming Folders:

● Use OS Folder Name

○ Use Parent Path (select from list below):

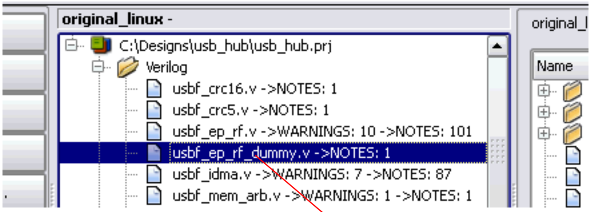
vhdl
tutorial
synplify_pro

vhdl

Feature	Description
Use OS Folder Name	The directory of the file to add. Use this option to specify the folder containing the file as the custom folder name for the directory.
Use Parent Path (select from list below)	The directory of the file to add. Select the folder from the display window to determine the level of hierarchy reflected in the custom folder path name.

Change File Command

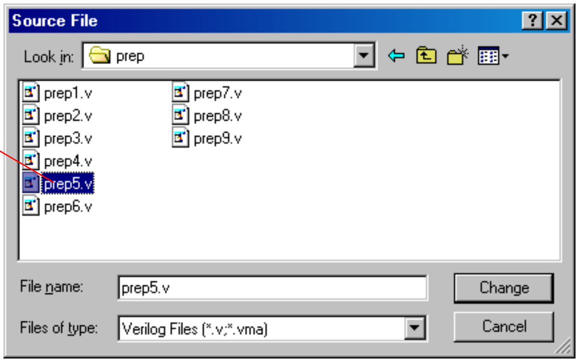
Select Project->Change File to replace a file in the project files list with another of the same type. This displays the Source File dialog box, where you specify the replacement file. You must first select the file to replace, in the Project view, before you can use this command.



C:\Designs\usb_hub\usb_hub.prj
Verilog
usbf_crc16.v ->NOTES: 1
usbf_crc5.v ->NOTES: 1
usbf_ep_rf.v ->WARNINGS: 10 ->NOTES: 101
usbf_ep_rf_dummy.v ->NOTES: 1
usbf_idma.v ->WARNINGS: 7 ->NOTES: 87
usbf_mem_arb.v ->WARNINGS: 1 ->NOTES: 1

First select a file in the Project view

Then choose the replacement file



Look in: prep

prep1.v
prep2.v
prep3.v
prep4.v
prep5.v
prep6.v
prep7.v
prep8.v
prep9.v

File name: prep5.v

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

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 Popup Menu Command, on page 292. This is the same dialog box as that displayed by right-clicking a VHDL filename in the Project view and choosing File Options.

Add Implementation Command

Select Project->Add Implementation to create a new implementation for the selected project. This selection displays the Implementation Options dialog box, where you define the implementation options for the project – see Implementation Options Command, on page 158. 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.

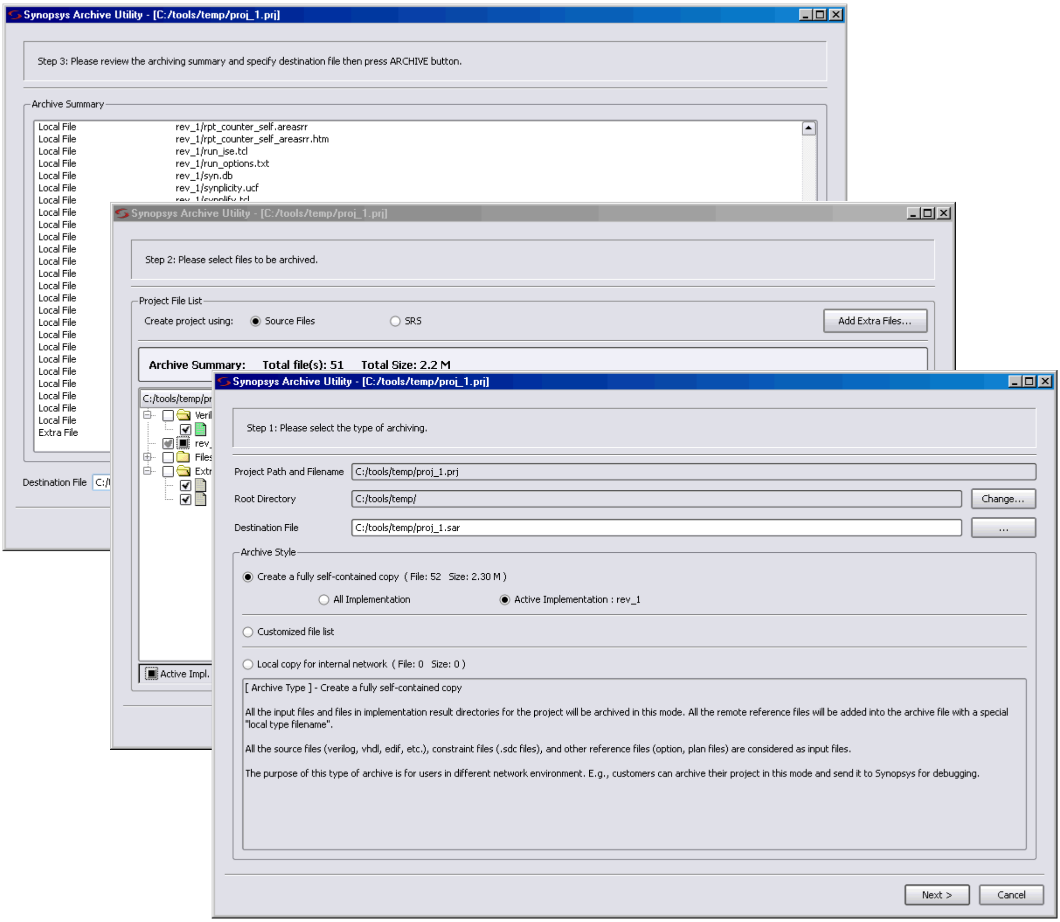
Convert Vendor Constraints Command

The Project->Convert Vendor Constraints is not available for Microsemi technologies.

Archive Project Command

Use the Project->Archive Project command to store files for a design project into a single archive file in Synopsys Proprietary Format (`sar`). You can archive an entire project or selected files from the project.

The Archive Project command displays the Synopsys Archive Utility wizard consisting of either two (all files archived) or three (custom file selection) tabs.



Synopsys Archive Utility - [C:/tools/temp/proj_1.prj]

Step 3: Please review the archiving summary and specify destination file then press ARCHIVE button.

Archive Summary

Local File rev_1/rpt_counter_self.areasrr
Local File rev_1/rpt_counter_self_areasrr.htm
Local File rev_1/run_ise.tcl
Local File rev_1/run_options.txt
Local File rev_1/syn.db
Local File rev_1/synplicity.ucf
Local File rev_1/synplify.tcl
Local File
Local File
Local File
Local File
Local File
Local File
Local File
Local File
Local File
Local File
Local File
Local File
Local File
Local File
Local File
Local File
Local File
Local File
Extra File

Destination File C:/

Synopsys Archive Utility - [C:/tools/temp/proj_1.prj]

Step 2: Please select files to be archived.

Project File List

Create project using: ● Source Files ○ SRS

Archive Summary: Total file(s): 51 Total Size: 2.2 M

C:/tools/temp/pr

Synopsys Archive Utility - [C:/tools/temp/proj_1.prj]

Step 1: Please select the type of archiving.

Project Path and Filename: C:/tools/temp/proj_1.prj

Root Directory: C:/tools/temp/

Destination File: C:/tools/temp/proj_1.sar

Archive Style

● Create a fully self-contained copy (File: 52 Size: 2.30 M)

○ All Implementation ● Active Implementation : rev_1

○ Customized file list

○ Local copy for internal network (File: 0 Size: 0)

[Archive Type] - Create a fully self-contained copy

All the input files and files in implementation result directories for the project will be archived in this mode. All the remote reference files will be added into the archive file with a special "local type filename".

All the source files (verilog, vhdl, edif, etc.), constraint files (.sdc files), and other reference files (option, plan files) are considered as input files.

The purpose of this type of archive is for users in different network environment. E.g., customers can archive their project in this mode and send it to Synopsys for debugging.

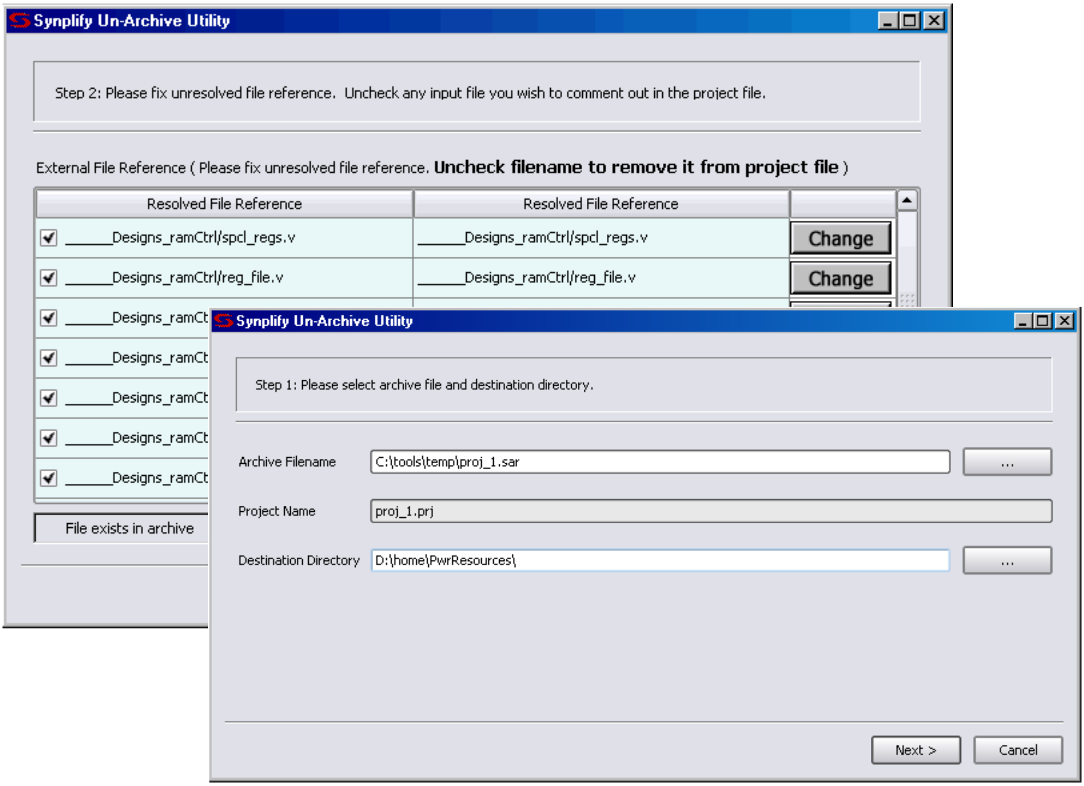
Option	Description
Project Path and Filename	Path and filename of the .prj file.
Root Directory	Top-level directory that contains the project files.
Destination Directory	Pathname of the directory to store the archive .sar file.
Archive Style	The type of archive: • Create a fully self-contained copy – all project files are archived; includes project input files and result files. • If the project contains more than one implementation: - All Implementation includes all implementations in the project. - Active Implementation includes only the active implementation. • Customized file list – only project files that you select are included in the archive. • Local copy for internal network – only project input files are archived, no result files will be included.
Create Project using	If you select the Customized file list option in the wizard, you can choose one the following options on the second tab: • Source Files – Includes all design files in the archive. You cannot enable the SRS option if this option is enabled. • SRS – Includes all .srs files (RTL schematics) in the archive. You cannot enable the Source Files option when this option is enabled.
Add Extra Files	If you select the Customized file list option in the wizard, you can use this button on the second tab to add additional files to the archive.

For step-by-step details on how to use the archive utility, see Archive Project Command, on page 150.

Un-Archive Project Command

Use the Project->Un-Archive Project command to extract the files from an archived design project.

This command displays a Synplicity Un-Archive Utility wizard.



Synplify Un-Archive Utility

Step 2: Please fix unresolved file reference. Uncheck any input file you wish to comment out in the project file.

External File Reference (Please fix unresolved file reference. **Uncheck filename to remove it from project file**)

Resolved File Reference	Resolved File Reference	
☑ ______Designs_ramCtrl/spcl_regs.v	______Designs_ramCtrl/spcl_regs.v	Change
☑ ______Designs_ramCtrl/reg_file.v	______Designs_ramCtrl/reg_file.v	Change
☑ ______Designs_ramCt		
☑ ______Designs_ramCt		
☑ ______Designs_ramCt		
☑ ______Designs_ramCt		
☑ ______Designs_ramCt		

File exists in archive

Synplify Un-Archive Utility

Step 1: Please select archive file and destination directory.

Archive Filename: C:\tools\temp\proj_1.sar

Project Name: proj_1.prj

Destination Directory: D:\home\PwrResources\

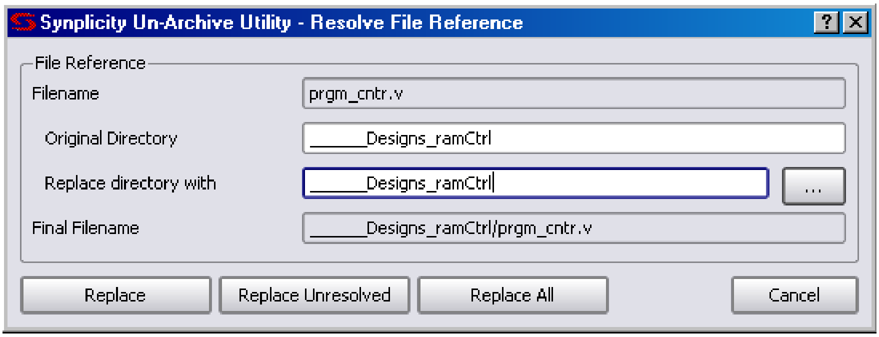
Option	Description
Archive Filename	Path and filename of the .prj file.

Option	Description
Project Name	Top-level directory that contains the project files.
Destination Directory	Pathname of the directory to store the archive .sar file.
Original File Reference/ Resolved File Reference	Displays the files in the archive that will be extracted. You can exclude files from the .sar by unchecking the file in the Original File Reference list. Any unchecked files are commented out in the .prj file. If there are unresolved reference files in the .sar file, you must fix (Resolve button) or uncheck them. Or, if there are files that you want the change when project files are extracted, use the Change button and select files, as appropriate. See *Resolve File Reference*, next for more details.

For step-by-step details on how to use the un-archive utility, see Un-Archive Project Command, on page 152.

Resolve File Reference

When you use the Un-Archive Utility wizard to extract a project, if there are unresolved file references, use the Resolve button next to the file to point to a new file location. You can also optionally replace project files in the destination directory by clicking the Change button next to the file you want to replace. The Change and Resolve buttons bring up the following dialog box:



Filename: prgm_cntr.v

Original Directory: ______Designs_ramCtrl

Replace directory with: ______Designs_ramCtrl

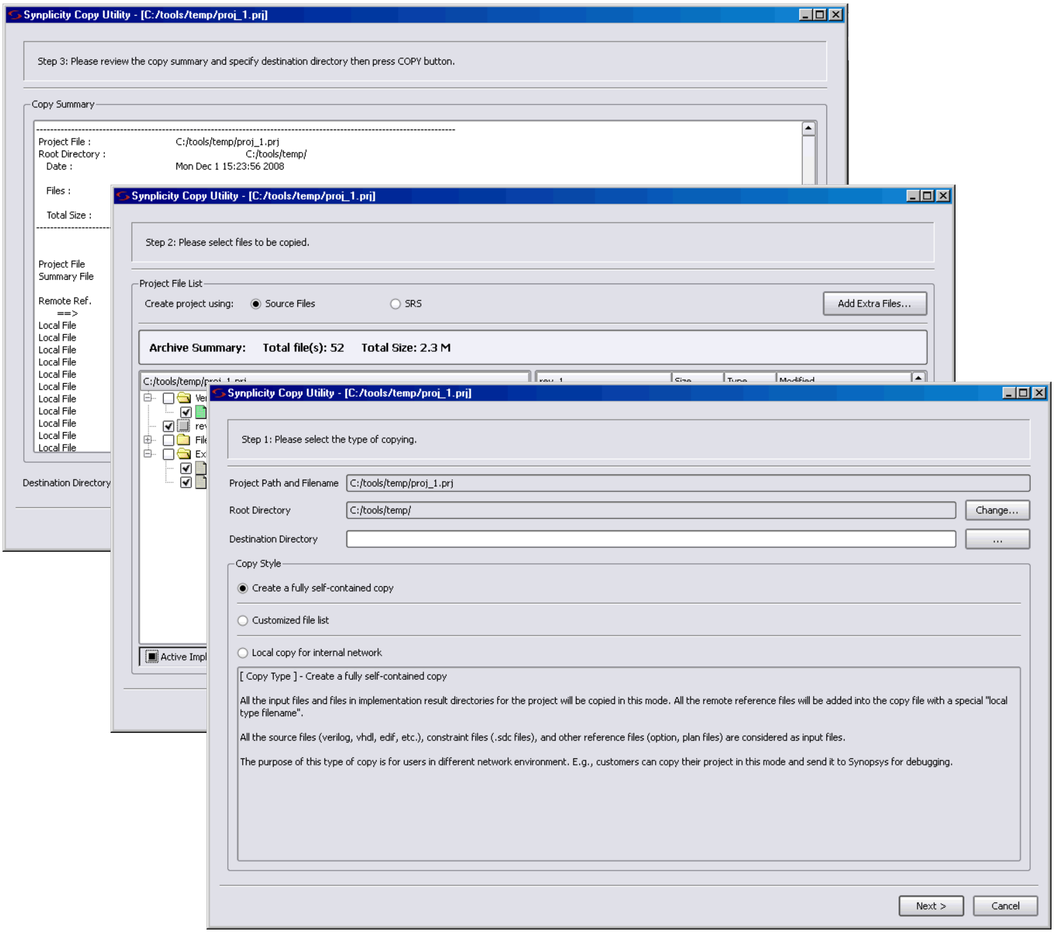
Final Filename: ______Designs_ramCtrl/prgm_cntr.v

Option	Description
Filename	Specifies the path and name of the file you want to change or resolve.
Original Directory	Specifies the location of the project at the time it was archived.
Replace directory with	Specifies the new location of the project files you want to use to replace files.
Final Filename	Specifies the path name of the directory and the file name of the replace file.
Replace buttons	• Replace – replaces only the file specified in the Filename field when the project is extracted. • Replace Unresolved – replaces any unresolved files in the project, with files of the same name from the Replace directory. • Replace All – replaces all files in the archived project with files of the same name from the Replace directory. • To undo any replace-file references, clear the Replace directory with field, then click Replace. This causes the utility to point back to the Original Directory and filenames.

Copy Project Command

Use the Project->Copy Project command to create a copy of a design project. You can copy an entire project or selected files from the project.

The Copy Project command displays the Synopsys Copy Utility wizard consisting of either two (all files copied) or three (custom file selection) tabs.



Synplicity Copy Utility - [C:/tools/temp/proj_1.prj]

Step 3: Please review the copy summary and specify destination directory then press COPY button.

Copy Summary

```
------------------------------------------------------------------------------------------------------------
Project File :              C:/tools/temp/proj_1.prj
Root Directory :                        C:/tools/temp/
  Date :                    Mon Dec 1 15:23:56 2008

  Files :

  Total Size :
---------------------

Project File
Summary File

Remote Ref.
    ==>
Local File
Local File
Local File
Local File
Local File
Local File
Local File
Local File
Local File
Local File
Local File
```

Destination Directory

Synplicity Copy Utility - [C:/tools/temp/proj_1.prj]

Step 2: Please select files to be copied.

Project File List

Create project using: ● Source Files ○ SRS

Archive Summary: **Total file(s): 52** **Total Size: 2.3 M**

Synplicity Copy Utility - [C:/tools/temp/proj_1.prj]

Step 1: Please select the type of copying.

Project Path and Filename: C:/tools/temp/proj_1.prj

Root Directory: C:/tools/temp/

Destination Directory:

Copy Style

● Create a fully self-contained copy

○ Customized file list

○ Local copy for internal network

[Copy Type] - Create a fully self-contained copy

All the input files and files in implementation result directories for the project will be copied in this mode. All the remote reference files will be added into the copy file with a special "local type filename".

All the source files (verilog, vhdl, edif, etc.), constraint files (.sdc files), and other reference files (option, plan files) are considered as input files.

The purpose of this type of copy is for users in different network environment. E.g., customers can copy their project in this mode and send it to Synopsys for debugging.

Option	**Description**
Project Path and Filename	Path and filename of the .prj file.
Root Directory	Top-level directory that contains the project files.
Destination Directory	Pathname of the directory to store the archive .sar file.

Option	Description
Copy Style	The type of archive: • Create a fully self-contained copy – all project files are archived; includes project input files and result files. • If the project contains more than one implementation: - All Implementation includes all implementations in the project. - Active Implementation includes only the active implementation. • Customized file list – only project files that you select are included in the archive. • Local copy for internal network – only project input files are archived, no result files will be included.
Create Project using	If you select the Customized file list option in the wizard, you can choose one the following options on the second tab: • Source Files – Includes all design files in the archive. You cannot enable the SRS option if this option is enabled. • SRS – Includes all .srs files (RTL schematics) in the archive. You cannot enable the Source Files option if this option is enabled.
Add Extra Files	If you select the Customized file list option in the wizard, you can use this button on the second tab to add additional files to the archive.

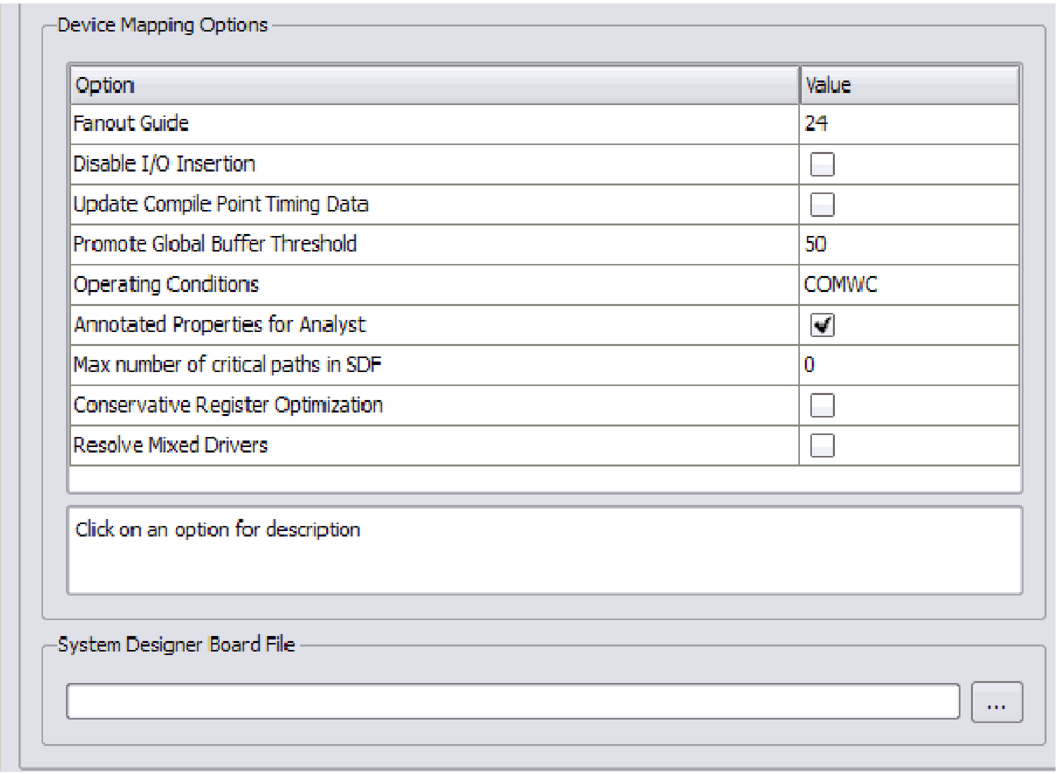
For step-by-step details on how to use the copy utility, see Copy Project Command, on page 154.

Hierarchical Project Options Command

The Project->Hierarchical Project Options command is not available for Microsemi technologies.

Implementation Options Command

You use the Implementation Options 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.



Device Mapping Options

Option	Value
Fanout Guide	24
Disable I/O Insertion	□
Update Compile Point Timing Data	□
Promote Global Buffer Threshold	50
Operating Conditions	COMWC
Annotated Properties for Analyst	☑
Max number of critical paths in SDF	0
Conservative Register Optimization	□
Resolve Mixed Drivers	□

Click on an option for description

System Designer Board File

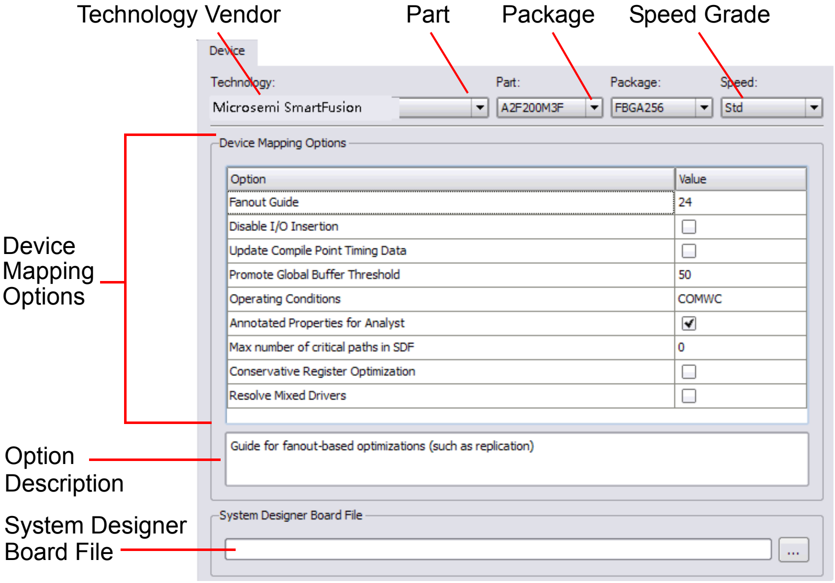
This section describes the following:

- Device Panel, on page 159. For device-specific details of the options, refer to the appropriate vendor chapter.
- Options Panel, on page 160
- Constraints Panel, on page 162
- Implementation Results Panel, on page 163
- Timing Report Panel, on page 165

- VHDL Panel, on page 166
- Verilog Panel, on page 169
- Place and Route Panel, on page 178

Device Panel

You use the Device panel to set mapping options for the selected technology.



Technology: Microsemi SmartFusion

Part: A2F200M3F

Package: FBGA256

Speed: Std

Device Mapping Options

Option	Value
Fanout Guide	24
Disable I/O Insertion	□
Update Compile Point Timing Data	□
Promote Global Buffer Threshold	50
Operating Conditions	COMWC
Annotated Properties for Analyst	☑
Max number of critical paths in SDF	0
Conservative Register Optimization	□
Resolve Mixed Drivers	□

Guide for fanout-based optimizations (such as replication)

System Designer Board File

The mapping options vary, depending on the technology. See Setting Device Options, on page 131 in the *User Guide* for a procedure, and the relevant vendor sections in this reference manual for technology-specific descriptions of the options.

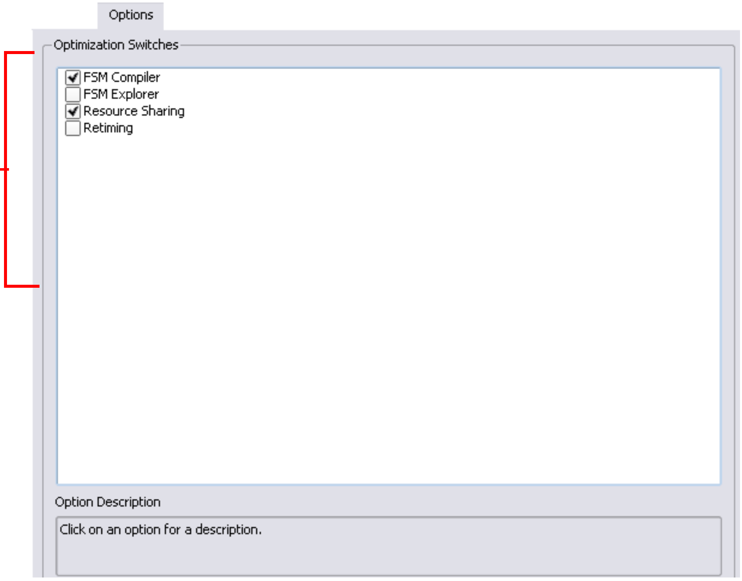
The table below lists the following category of options. Not all options are available for all tools and technologies.

Option	Description
Technology Vendor	Specify the device technology you want to synthesize. You can also select the part, package, and speed grade to use. For more information, see the appropriate vendor appendix in the *Reference* manual.
Device Mapping Options	The device mapping options vary depending on the device technology you select. For more information, see the appropriate vendor appendix in the *Reference* manual.
Option Description	Click on a device mapping option to display its description in this field. Refer to the relevant vendor sections for technology-specific descriptions of the options.
System Designer Board File	Not applicable for Microsemi technologies.

Options Panel

You use the Options panel of the Implementation Options dialog box to define general options for synthesis optimization. See Setting Optimization Options, on page 133 of the *User Guide* for details.

*



Synthesis optimization options

Optimization Switches

☑ FSM Compiler
□ FSM Explorer
☑ Resource Sharing
□ Retiming

Option Description

Click on an option for a description.

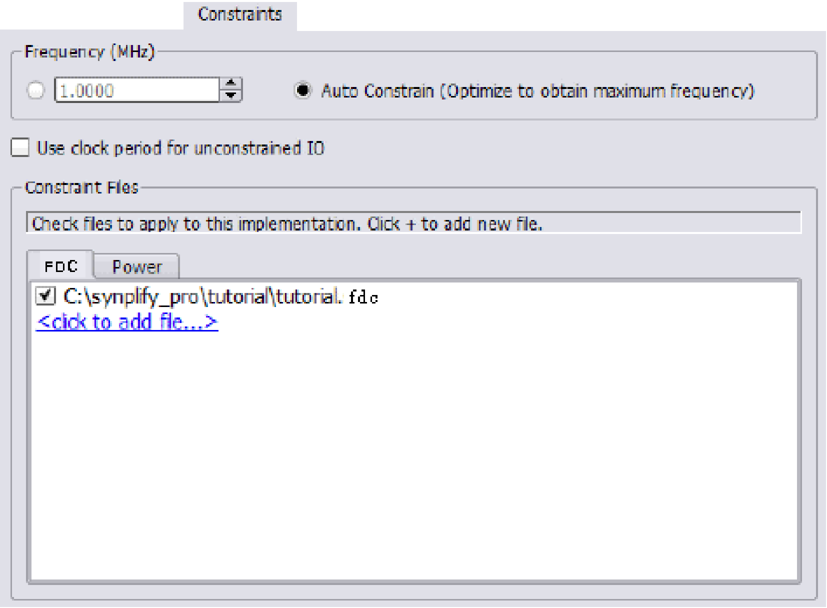
The following table lists the options alphabetically. Not all options are available for all tools and technologies.

Option	Description
Enable 64-bit Synthesis	Enables/disables the 64-bit mapping switch. When enabled, this switch allows you to run client programs in 64-bit mode, if available on your system. For batch mode, use this Tcl command in your project file: set_option -enable64bit 1 This option is supported on the Windows and Linux platforms.
FSM Compiler	Determines whether the FSM Compiler is run. See FSM Compiler, on page 84 and Optimizing State Machines, on page 220 in the *User Guide.*

Option	Description
FSM Explorer	Determines whether the FSM Explorer is run. See Running the FSM Explorer, on page 226 in the *User Guide*.
Resource Sharing	Controls whether you optimize area by sharing resources. See Sharing Resources, on page 215 in the *User Guide*.
Retiming	Determines whether the tool moves storage devices across computational elements to improve timing performance in sequential circuits. Note that the tool might retime registers associated with RAMs and DSPs regardless of the Retiming setting. For details about using this feature, see Retiming, on page 198 in the *User Guide*.

Constraints Panel

You use the Constraints panel of the Implementation Options dialog box to specify target frequency and timing constraint files for design synthesis. See Specifying Global Frequency and Constraint Files, on page 135, in the *User Guide* for details.



Constraints

Frequency (MHz)

○ 1.0000 ● Auto Constrain (Optimize to obtain maximum frequency)

☐ Use clock period for unconstrained IO

Constraint Files

Check files to apply to this implementation. Click + to add new file.

FDC Power

☑ C:\synplify_pro\tutorial\tutorial.fdc

<click to add file...>

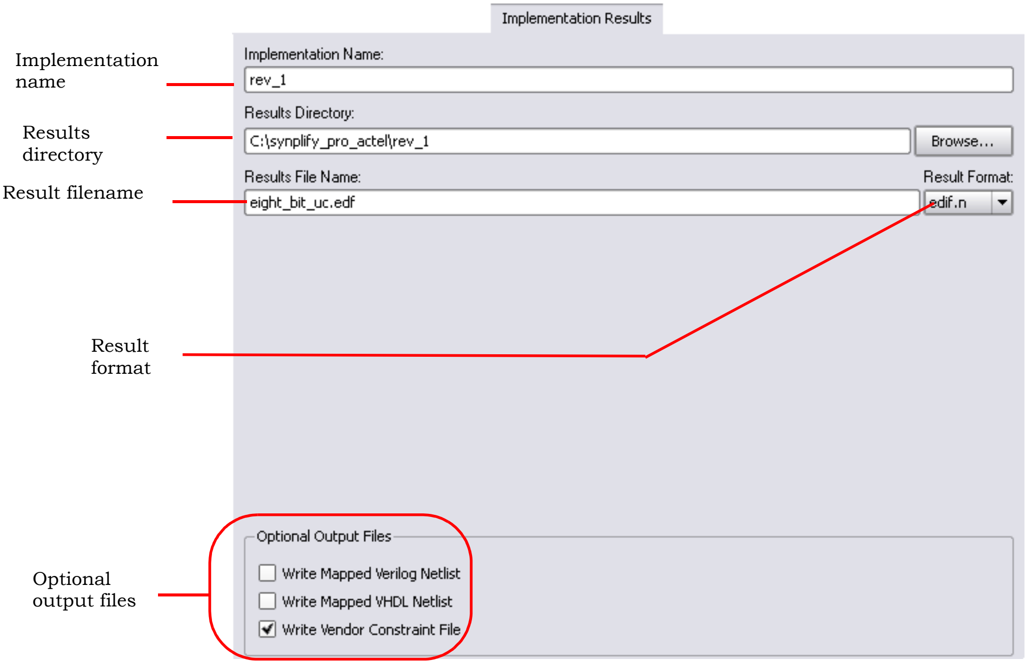
Option	Description
Frequency	Sets the default global frequency. You can either set the global frequency here or in the Project view. To override the default you set here, set individual clock constraints from the SCOPE interface.
Auto Constrain	When enabled and no clocks are defined, the software automatically constrains the design to achieve the best possible timing. It does this by reducing periods of each individual clock and the timing of any timed I/O paths in successive steps. See Auto Constraints, on page 359 for an explanation, and Using Auto Constraints, on page 341 in the *User Guide* for information about using this option. You can also set this option in the Project view.
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 set_input_delay or set_output_delay constraints are considered during synthesis or forward-annotated after synthesis. When enabled, the software considers any explicit set_input_delay or set_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.
Constraint Files SDC, FDC	Specifies which timing constraints files to use for the implementation. Enable the checkbox to select a file. For the Synplify Pro tool, block-level files in the compile-point flows, the Module column shows the name of the module or compile point.

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 137 of the *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_1

Results Directory: C:\synplify_pro_actel\rev_1

Results File Name: eight_bit_uc.edf

Result Format: edif.n

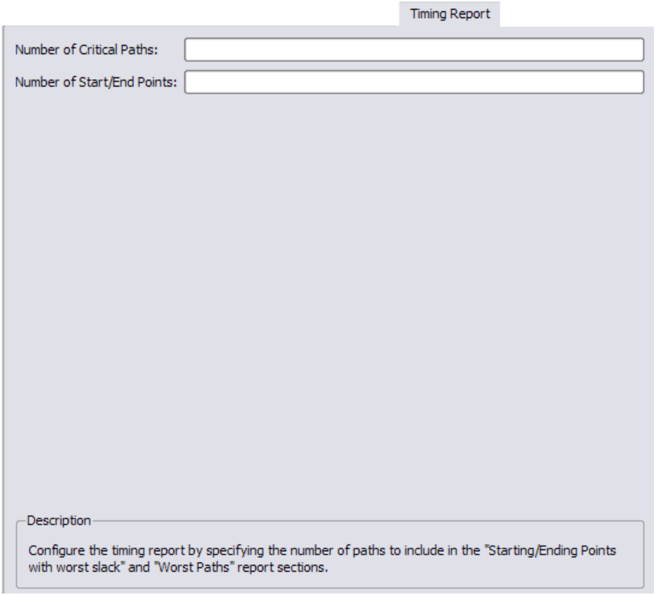
Optional Output Files

- ☐ Write Mapped Verilog Netlist
- ☐ Write Mapped VHDL Netlist
- ☑ Write Vendor Constraint File

Option	Description
Implementation Name Results Directory Result Base Name	Displays implementation name, directory path for results, and the base name for the result files.
Result Format	Select the output that corresponds to the technology you are using. See Generating Vendor-Specific Output, on page 352 in the *User Guide* for a list of netlist formats.
Write Mapped Verilog Netlist Write Mapped VHDL Netlist	Generates mapped Verilog or VHDL netlist files.
Write Vendor Constraint File	Generates a vendor-specific constraint file for forward annotation.

Timing Report Panel

Use the Timing Report panel (Implementation Options dialog box) to set criteria for the (default) output timing report. 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 138 in the *User Guide* for details. For a description of the report, see Timing Reports, on page 441.



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.

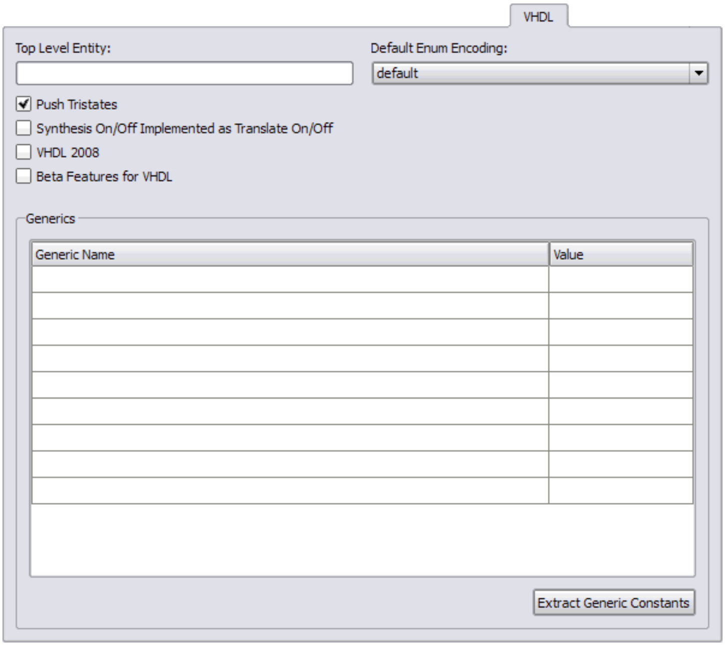
Option	Description
Number of Critical Paths	Set the number of critical paths you want the software to report.
Number of Start/End Points	Specify the number of start and end points you want to see reported in the critical path sections.

See also:

- Timing Reports, on page 441, for more information on the default timing report, which is affected by the Timing Report panel settings.
- Analysis Menu, on page 226, information on creating additional custom timing reports for certain device technologies.

VHDL Panel

You use the VHDL panel in the Implementation Options 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 139, of the *User Guide* for details.



VHDL

Top Level Entity:

Default Enum Encoding: default

☑ Push Tristates

□ Synthesis On/Off Implemented as Translate On/Off

□ VHDL 2008

□ Beta Features for VHDL

Generics

Generic Name	Value

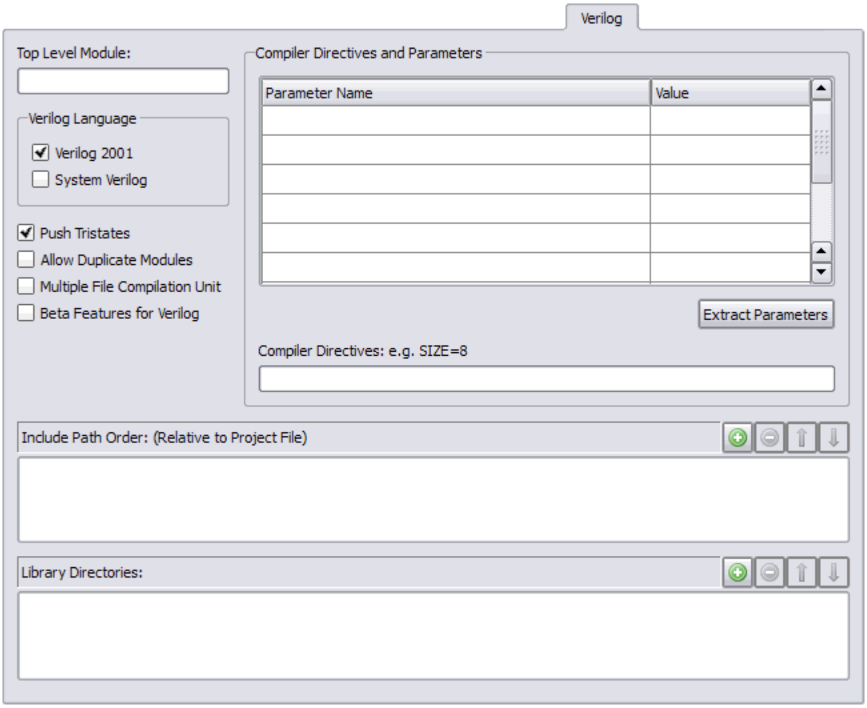
The following table describes the options available:

Feature	Description
Top Level Entity	The name of the top-level entity of your design. If the top-level entity does not use the default work library to compile the VHDL files, you must specify the library file where the top-level entity can be found. To do this, the top-level entity name must be preceded by the VHDL library followed by a period (.). To specify VHDL library files, see Project Menu, on page 143 for the Set VHDL Library command, or the File Options Popup Menu Command, on page 292.
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 177.

Feature	Description
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 Directive, on page 1030 for details).
VHDL 2008	When enabled, allows you to use VHDL 2008 language standards.
Implicit Initial Value Support	When enabled, the compiler passes init values through a syn_init property to the mapper. For more information, see VHDL Implicit Data-type Defaults, on page 668.
Beta Features for VHDL	Enables use of any VHDL beta features included in the release. Enabling this checkbox is equivalent to including a set_option -hdl_define -set _BETA_FEATURES_ON_ directive in the project file.
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 Implementation Options 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 139 of the *User Guide* for details.


Verilog
Top Level Module:
Verilog Language
Compiler Directives and Parameters
Parameter Name
Value
Extract Parameters
Compiler Directives: e.g. SIZE=8
Include Path Order: (Relative to Project File)
Library Directories:

☑ Verilog 2001
□ System Verilog
☑ Push Tristates
□ Allow Duplicate Modules
□ Multiple File Compilation Unit
□ Beta Features for Verilog

Feature	Description
Top Level Module	The name of the top-level module of your design.
Compiler Directives and Parameters	Shows design parameters extracted with Extract Parameters. You can override the default and set a new value for the parameter. The value is valid for the current implementation.
Extract Parameters	Extracts design parameters from the top-level module and displays them in the table. See Compiler Directives and Design Parameters, on page 172.

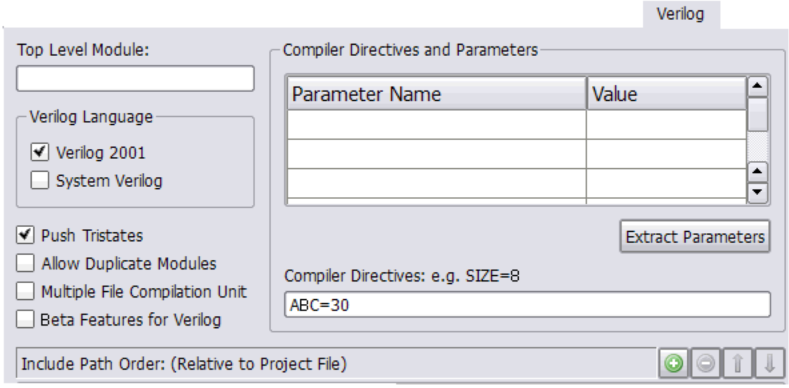
Feature	Description
Compiler Directives	Provides an interface where you can enter compiler directives that you would normally enter in the code with ‘ifdef and ‘define statements. See Compiler Directives and Design Parameters, on page 172.
Verilog Language – Verilog 2001	When enabled, the default Verilog standard for the project is Verilog 2001. When both Verilog 2001 and SystemVerilog are disabled, the default standard is Verilog 95. For information about Verilog 2001, see Verilog 2001 Support, on page 463. 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 Popup Menu Command, on page 292).
Verilog Language – SystemVerilog	When enabled, the default Verilog standard for the project is SystemVerilog which is the default standard for all new projects. Enabling SystemVerilog automatically enables Verilog 2001.
Push Tristates	When enabled (default), tristates are pushed across process/block boundaries. For details, see Push Tristates Option, on page 177.
Allow Duplicate Modules	Allows the use of duplicate modules in your design. When enabled, the last definition of the module is used by the software and any previous definitions are ignored. You should not use duplicate module names in your Verilog design, therefore, this option is disabled by default. However, if you need to, you can allow for duplicate modules by enabling this option.
Multiple File Compilation Unit	When enabled (the default), the Verilog compiler uses the compilation unit for modules defined in multiple files. See SystemVerilog Compilation Units, on page 603 for additional information.
Beta Features for Verilog	Enables use of any Verilog beta features included in the release. Enabling this checkbox is equivalent to including a set_option -hdl_define -set _BETA_FEATURES_ON_ directive in the project file.

Feature	Description
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. The include paths are relative. See Updating Verilog Include Paths in Older Project Files, on page 119 in the *User Guide* for additional information.
Library Directories	Specifies all the paths to the directories which contain the Verilog library files to be included in your design for the project.

Compiler Directives and Design Parameters

When you click the Extract Parameters button in the Verilog panel (Implementation Options dialog box), parameter values from the top-level module are displayed in the table. You can also override the default by setting a new value for the parameter. The value is valid for the current implementation only.

The Compiler Directives field provides an interface where you can enter compiler directives that you would normally enter in the code using ‘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 the directive shown below to the Compiler Directives field of the Verilog panel:


Verilog
Top Level Module:
Verilog Language
Compiler Directives and Parameters
Parameter Name
Value
Extract Parameters
Include Path Order: (Relative to Project File)

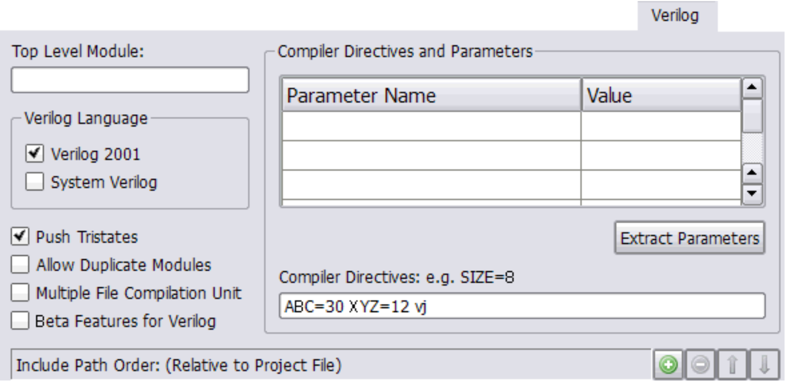
☑ Verilog 2001
☐ System Verilog
☑ Push Tristates
☐ Allow Duplicate Modules
☐ Multiple File Compilation Unit
☐ Beta Features for Verilog

Compiler Directives: e.g. SIZE=8
ABC=30

the software writes the following statement to the project file:

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

To define multiple variables in the GUI, use a space delimiter. For example:



Top Level Module:

Verilog Language

☑ Verilog 2001

□ System Verilog

☑ Push Tristates

□ Allow Duplicate Modules

□ Multiple File Compilation Unit

□ Beta Features for Verilog

Compiler Directives: e.g. SIZE=8

ABC=30 XYZ=12 vj

Include Path Order: (Relative to Project File)

The software writes the following statement to the prj file:

```
set_option hdl_define -set "ABC=30 XYZ=12 vj"
```

More information is provided for the following Verilog compiler directives:

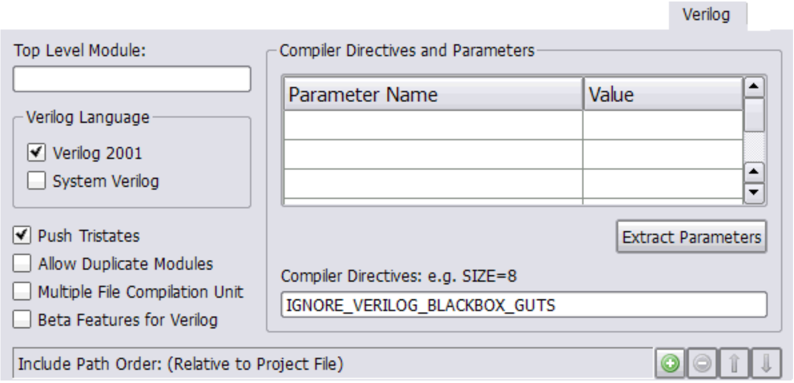
- IGNORE_VERILOG_BLACKBOX_GUTS Directive
- _BETA_FEATURES_ON_ Directive
- _SEARCHFILENAMEONLY_ Directive

IGNORE_VERILOG_BLACKBOX_GUTS Directive

When you use the syn_black_box directive, the compiler parses the contents of the black box and can determine whether illegal syntax or incorrect code is defined within it. Whenever this occurs, an error message is generated.

However, if you do not want the tool to check for illegal syntax in your black box, set the:

- Built-in compiler directive IGNORE_VERILOG_BLACKBOX_GUTS in the Compiler Directives field of the Verilog panel on the Implementation Options dialog box.


Verilog
Top Level Module:
Verilog Language
☑ Verilog 2001
□ System Verilog
☑ Push Tristates
□ Allow Duplicate Modules
□ Multiple File Compilation Unit
□ Beta Features for Verilog
Compiler Directives and Parameters
Parameter Name
Value
Extract Parameters
Compiler Directives: e.g. SIZE=8
Include Path Order: (Relative to Project File)

IGNORE_VERILOG_BLACKBOX_GUTS

The software writes the following command to the project file:

```
set_option -hdl_define -set "IGNORE_VERILOG_BLACKBOX_GUTS"
```

- `` `define IGNORE_VERILOG_BLACKBOX_GUTS `` directive in the Verilog file.

This option is implemented globally for the project file.

Example of the IGNORE_VERILOG_BLACKBOX_GUTS Directive

The IGNORE_VERILOG_BLACKBOX_GUTS directive ignores the contents of the black box. However, whenever you use this directive, you must first define the ports for the black box correctly. Otherwise, the IGNORE_VERILOG_BLACKBOX_GUTS directive generates an error. See the following valid Verilog example:

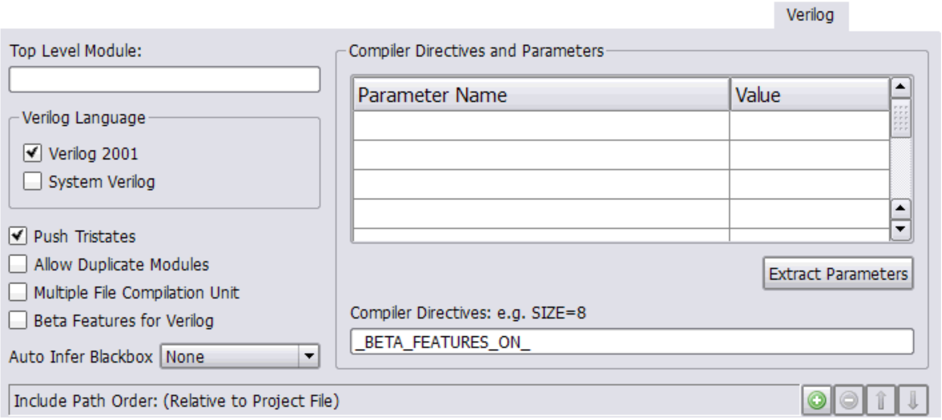
```
`define IGNORE_VERILOG_BLACKBOX_GUTS
module b1_fpga1 (A,B,C,D) /* synthesis syn_black_box */;
input B;
output A;
input [2:0] D;
output [2:0] C;
temp;
assign A = B;
assign C = D;

endmodule

module b1_fpga1_top (inout A, B, inout [2:0] C, D);
b1_fpga1 b1_fpga1_inst(A,B,C,D);
endmodule
```

_BETA_FEATURES_ON_ Directive

Beta features for the Verilog, SystemVerilog, or VHDL language must be explicity enabled. In the UI, a Beta Features checkbox is included on the VHDL or Verilog tab of the Implementations Options dialog box. A _BETA_FEATURES_ON_ compiler directive is also available. This directive is specified with a set_option -hdl_define command added to the project file as shown below:

```
set_option -hdl_define -set _BETA_FEATURES_ON_
```

The directive can also be added to the Compiler Directives field of the Verilog panel.



Verilog

Top Level Module:

Verilog Language

☑ Verilog 2001

□ System Verilog

☑ Push Tristates

□ Allow Duplicate Modules

□ Multiple File Compilation Unit

□ Beta Features for Verilog

Auto Infer Blackbox: None

Compiler Directives and Parameters

Parameter Name	Value

Compiler Directives: e.g. SIZE=8

_BETA_FEATURES_ON_

Include Path Order: (Relative to Project File)

SEARCHFILENAMEONLY Directive

This directive provides a workaround for some known limitations of the archive utility.

If Verilog 'include files belong in any of the following categories, you may encounter problems when compiling a design after un-archiving:

1. The include paths have relative paths to the project file.

   ```
   `include "../../../defines.h"
   ```

2. The include paths have absolute paths to the project file.

   ```
   `include "c:/temp/params.h"
   ```

   ```
   `include "/remote/sbg_home/user/params.h"
   ```

3. The include paths have the same file names, but are located in different directories relative to the project file.

```
`include "../myflop.v"
…
`include "../../myflop.v"
```

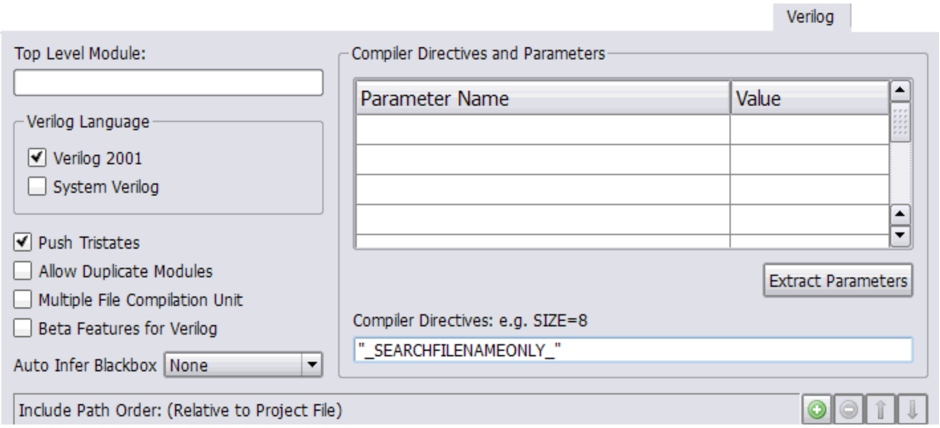
Use the _SEARCHFILENAMEONLY_ directive to resolve categories 1 and 2 above. Category 3 is a known limitation; in this case it is recommended that you adopt standard coding practices to avoid files with the same name and different content.

When you un-archive a `sar` file that contains relative or absolute include paths for the files in the project, you can add the _SEARCHFILENAMEONLY_ compiler directive to the unarchived project; this has the compiler remove the relative/absolute paths from the `include and search only for the file names.

This directive is specified with a set_option -hdl_define command added to an implementation within the project file as shown below:

```
set_option -hdl_define -set "_SEARCHFILENAMEONLY_"
```

The directive can also be added to the Compiler Directives field of the Verilog panel as shown below.



Top Level Module:

Verilog Language

☑ Verilog 2001

□ System Verilog

☑ Push Tristates

□ Allow Duplicate Modules

□ Multiple File Compilation Unit

□ Beta Features for Verilog

Auto Infer Blackbox None

Compiler Directives and Parameters

Parameter Name	Value

Compiler Directives: e.g. SIZE=8 "_SEARCHFILENAMEONLY_"

Include Path Order: (Relative to Project File)

The compiler generates the following warning message whenever it extracts include files using this directive:

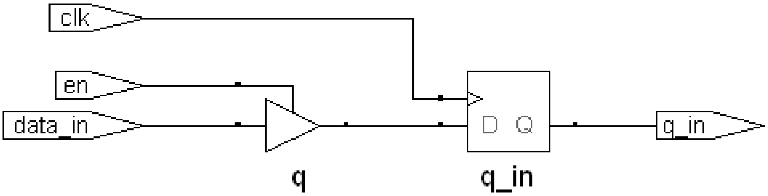
```
@W: | Macro _SEARCHFILENAMEONLY_ is set: fileName not found
attempting to search for base file name fileName
```

Push Tristates Option

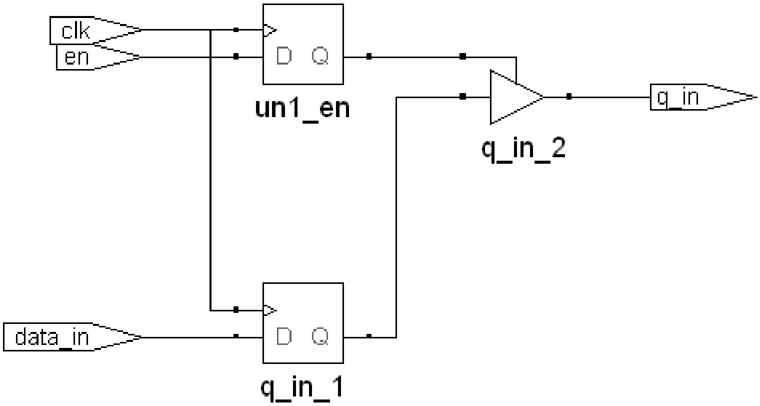
Pushing tristates is a synthesis optimization option you set with Project->Implementation Options->Verilog or VHDL.

Description

When the Push Tristates option is enabled, the Synopsys FPGA 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 Synopsys FPGA 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 Synopsys FPGA synthesis tool propagates the high-impedance states instead of producing tristate drivers at the outputs of process (VHDL) or always (Verilog) blocks.

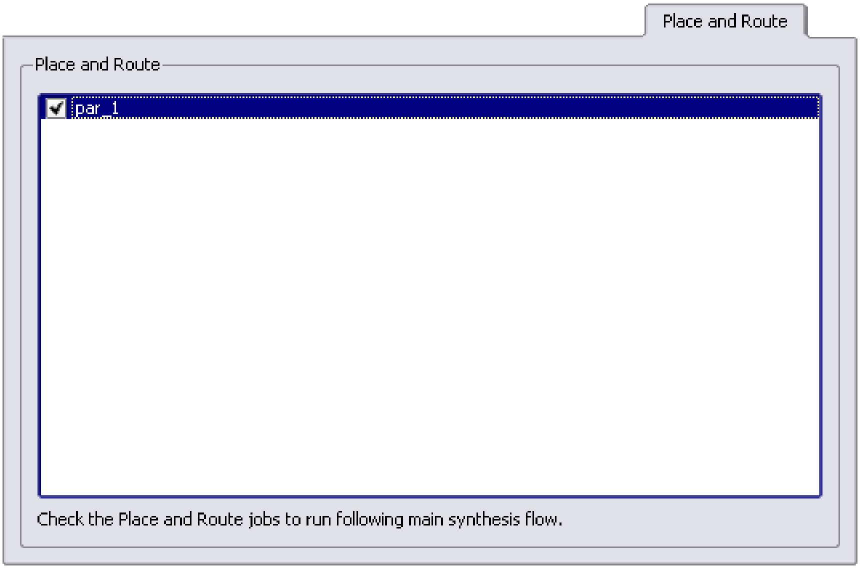
The disadvantage to pushing tristates is that you might use more design resources.

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.

Place and Route Panel

The Place and Route Jobs panel allows you to run selected place-and-route jobs after design synthesis. To create a place-and-route job, see Add P&R Implementation Popup Menu Command, on page 298 or Options for Place & Route Jobs Popup Menu Command, on page 299 for details. The Place and Route Panel is only available with certain technologies.



☑ par_1

Check the Place and Route jobs to run following main synthesis flow.

Import Menu

The Import option is not available for Microsemi technologies.

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.
- Check constraint syntax and how/if constraints are applied to the design.
- Run Tcl scripts.
- Run all implementations at once.
- Check the status of the current job.

The following table describes the Run menu commands.

Command	Description
Run	Synthesizes (compiles and maps) the top-level design. For the compile point flow, this command also synthesizes any compile points whose constraints, implementation options, or source code changed since the last synthesis run. 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->Run instead.
Compile Only	Compiles the design into technology-independent high-level structures. You can view the result in the RTL view.
Write Output Netlist Only	Generates an output netlist after synthesis has been run. This command generates the netlists you specify on the Implementation Results tab of the Implementation Options dialog box. You can also use this command in an incremental timing analysis flow. See Generating Custom Timing Reports with STA, on page 331 for details.

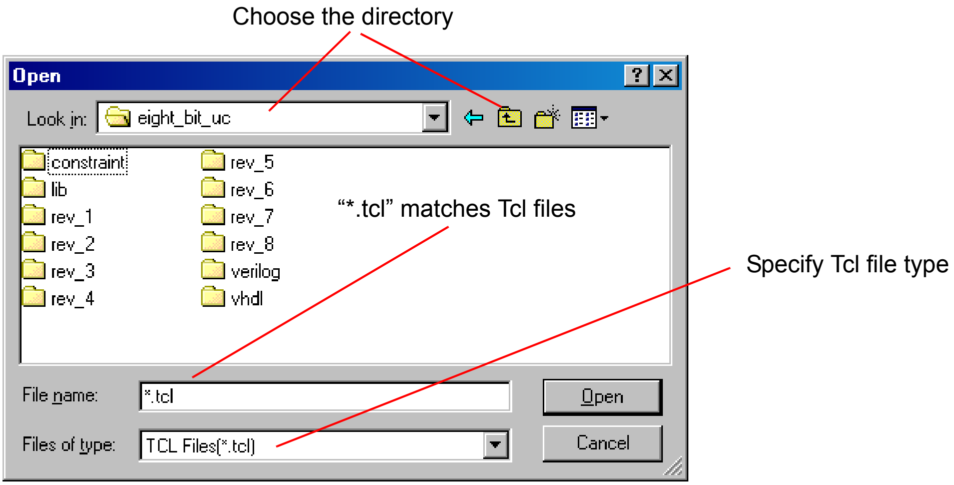
Command	Description
FSM Explorer	Analyzes finite state machines contained in a design, and selects the optimum encoding style. This menu command is not available in some views.
Syntax Check	Runs a syntax check on design code. The status bar at the bottom of the window displays any error messages. If 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.
Constraint Check	Checks the syntax and applicability of the timing constraints in the .fdc file for your project and generates a report (*projectName*_cck.rpt). The report contains information on the constraints that can be applied, cannot be applied because objects do not exist, and wildcard expansion on the constraints. See Constraint Checking Report, on page 450.
Arrange VHDL files	Reorders the VHDL source files for synthesis.
Launch Identify Instrumentor	Launches the Identify instrumentor. For more information, see: Working with the Identify Tool Set, on page 357 of the *User Guide*. To launch the Identify instrumentor in batch mode, use the set_option -identify_debug_mode 1 Tcl command.
Launch Identify Debugger	Launches the Identify debugger tool. For more information, see: Working with the Identify Tool Set, on page 357 of the *User Guide*. To launch the Identify debugger in batch mode, use the set_option -identify_debug_mode 1 Tcl command.
Launch SYNCore	Opens the Synopsys FPGA IP Core Wizard. This tool helps you build IP blocks such as memory or FIFO models for your design. See the Launch SYNCore Command, on page 187 for details.

Command	Description
Configure and Launch VCS Simulator	Allows you to configure and launch the VCS simulator. See Configure and Launch VCS Simulator Command, on page 216.
Run Tcl Script	Displays the Open dialog box, where you choose a Tcl script to run. See Run Tcl Script Command, on page 182.
Run All Implementations	Runs all implementations of one project at the same time.
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 184. Clicking in the status area of the Project view is a shortcut for this command.
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.

Run Tcl Script Command

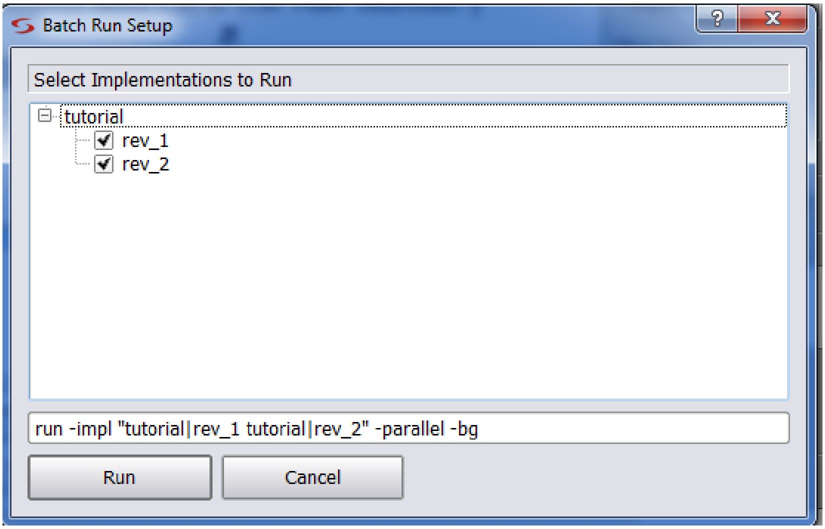
Select Run->Run Tcl Script to display the Open dialog box, where you specify the Tcl script file to execute. The File name area is filled automatically with the wildcard string “*.tcl”, corresponding to Tcl files.

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 122, for an explanation of the features in the Open dialog box.



Run All Implementations Command

Select Run->Run All Implementations to run selected implementations in batch mode. To use the Batch Run Setup dialog box, check one or more implementations from the list displayed and click the Run button.

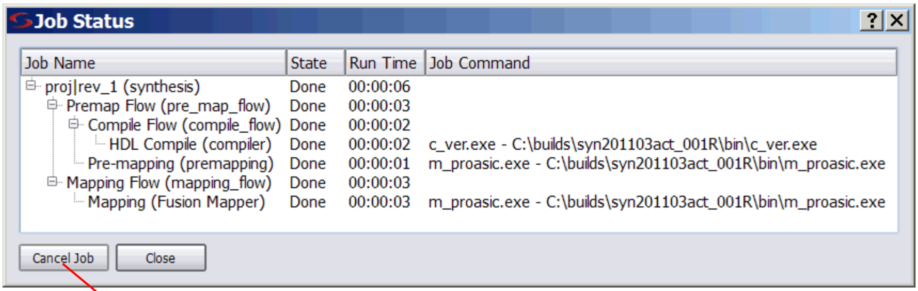


```
run -impl "tutorial|rev_1 tutorial|rev_2" -parallel -bg
```

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 The Project View, on page 44).

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



Job Name	State	Run Time	Job Command
proj|rev_1 (synthesis)	Done	00:00:06	
Premap Flow (pre_map_flow)	Done	00:00:03	
Compile Flow (compile_flow)	Done	00:00:02	
HDL Compile (compiler)	Done	00:00:02	c_ver.exe - C:\builds\syn201103act_001R\bin\c_ver.exe
Pre-mapping (premapping)	Done	00:00:01	m_proasic.exe - C:\builds\syn201103act_001R\bin\m_proasic.exe
Mapping Flow (mapping_flow)	Done	00:00:03	
Mapping (Fusion Mapper)	Done	00:00:03	m_proasic.exe - C:\builds\syn201103act_001R\bin\m_proasic.exe

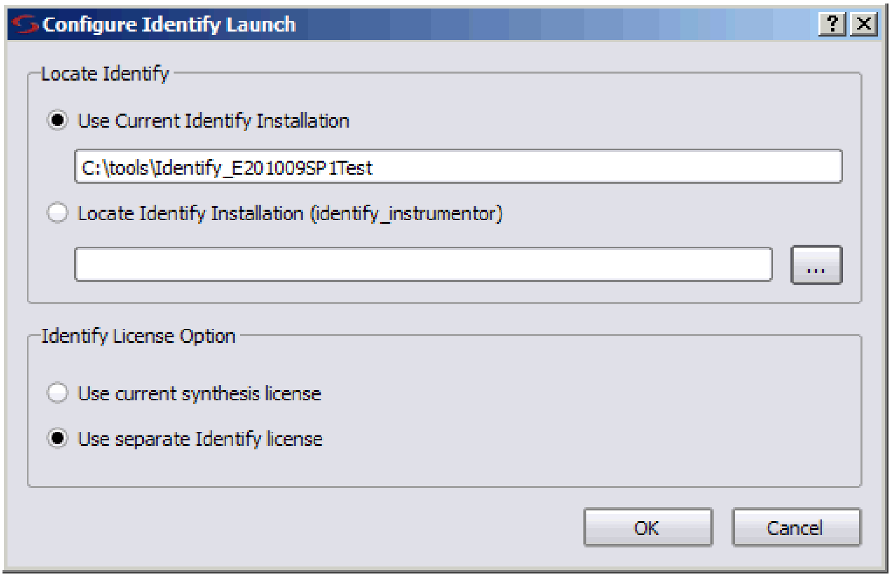
To cancel a job, select it,
then click the Cancel button

Launch Identify Instrumentor Command

The Launch Identify Instrumentor command lets you start the Identify instrumentor from within the synthesis interface. Before you can use this command, you must define an Identify implementation in the project view. For a description of the work flow using the Identify debugger, see Working with the Identify Tool Set, on page 357 in the *User Guide*.

Configure Identify Launch Dialog Box

The Configure Identify Launch dialog box is automatically displayed when the location of the Identify executable has not been previously defined.



Configure Identify Launch

Locate Identify

● Use Current Identify Installation

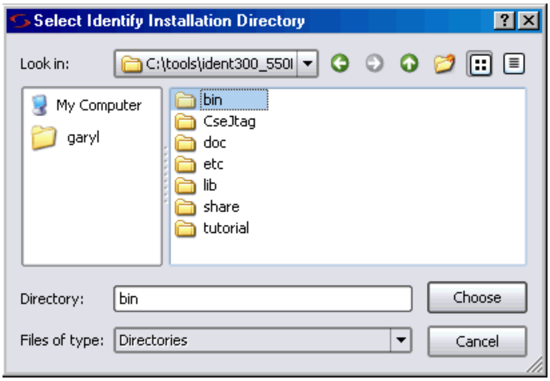
C:\tools\Identify_E201009SP1Test

○ Locate Identify Installation (identify_instrumentor)

Identify License Option

○ Use current synthesis license

● Use separate Identify license

Command	Description
Use current Identify installation:	A pointer to the current installation of the Identify software. Click the radio button to use the displayed version.
Locate Identify Installation (identify_instrumentor)	A pointer to the Identify install directory. Use the (...) button to navigate to the directory location.  Select Identify Installation Directory Look in: C:\tools\ident300_550I Directory: bin Files of type: Directories My Computer garyl bin CseJtag doc etc lib share tutorial Choose Cancel
Identify License Option	Radio buttons to select the Identify license option. Select Use current synthesis license when only a single TSL license is available; select Use separate Identify Instrumentor license when multiple licenses are available. With a single TSL license, you are prohibited from compiling or mapping in the synthesis tool while the Identify instrumentor is open.

Launch Identify Debugger Command

The Launch Identify Debugger command lets you start the Identify Debugger software from within the synthesis interface. Before you can use this command, you must have an active Identify implementation and an instrumented design. For a description of the work flow using the Identify/Identify RTL Debugger software, see Working with the Identify Tool Set, on page 357 in the *User Guide*.

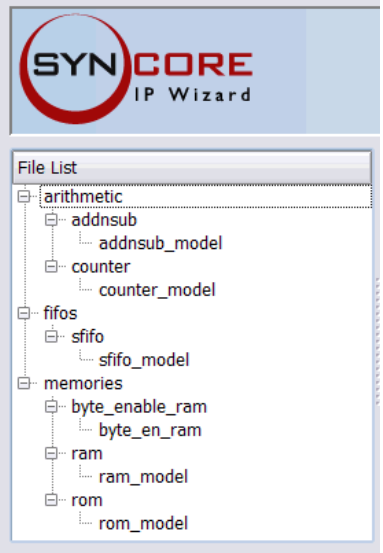
Launch SYNCore Command

The SYNCore wizard helps you build IP cores. Currently, the wizard can compile RAM and ROM memories including a byte-enable RAM, a FIFO, an adder/subtractor, and a counter. The resulting Verilog models can be synthesized and simulated. For details about using the wizard to build these models, see the following sections in the user guide:

- Specifying FIFOs with SYNCore, on page 370
- Specifying RAMs with SYNCore, on page 376
- Specifying Byte-Enable RAMs with SYNCore, on page 383
- Specifying ROMs with SYNCore, on page 389
- Specifying Adder/Subtractors with SYNCore, on page 394
- Specifying Counters with SYNCore, on page 401

To start the SYNCore wizard, select Run->Launch SYNCore and:

- Select sfifo_model and click Ok to start the FIFO wizard described in SYNCore FIFO Wizard, on page 189.
- Select ram_model and click Ok to start the RAM wizard described in SYNCore RAM Wizard, on page 198.
- Select byte_en_ram and click Ok to start the byte-enable RAM wizard described in SYNCore Byte-Enable RAM Wizard, on page 202.
- Select rom_model and click Ok to start the ROM wizard described in SYNCore ROM Wizard, on page 205.
- Select addnsub_model and click Ok to start the adder/subtractor wizard, described in SYNCore Adder/Subtractor Wizard, on page 209.
- Select counter_model and click Ok to start the counter wizard described in SYNCore Counter Wizard, on page 213.



Each SYNCore wizard has three tabs at the top, and some buttons below, which are described here:

Parameters	Consists of a multiple-screen wizard that lets you set parameters for that model. See SYNCore FIFO Wizard, on page 189, SYNCore RAM Wizard, on page 198, SYNCore Byte-Enable RAM Wizard, on page 202, SYNCore ROM Wizard, on page 205, SYNCore Adder/Subtractor Wizard, on page 209, or SYNCore Counter Wizard, on page 213 for details about the options you can set.
Core Overview	Contains basic information about the kind of model you are creating.
Generate	Generates the model with the parameters you specify in the wizard.
Sync FIFO Info, RAM Info, BYTE ENABLE RAM Info, ROM Info, ADDnSUB Info, COUNTER Info	Opens a window with technical information about the corresponding model.

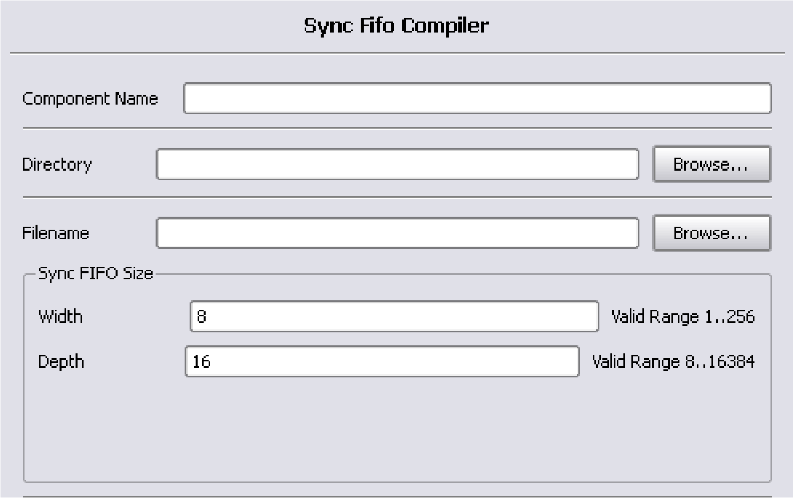
SYNCore FIFO Wizard

The following describe the parameters you can set in the FIFO wizard, which opens when you select sfifo_model:

- SYNCore FIFO Parameters Page 1, on page 189
- SYNCore FIFO Parameters Page 2, on page 190
- SYNCore FIFO Parameters Page 3, on page 192
- SYNCore FIFO Parameters Page 4, on page 194
- SYNCore FIFO Parameters Page 5, on page 196

SYNCore FIFO Parameters Page 1

The page 1 parameters define the FIFO. Data is written/read on the rising edge of the clock.



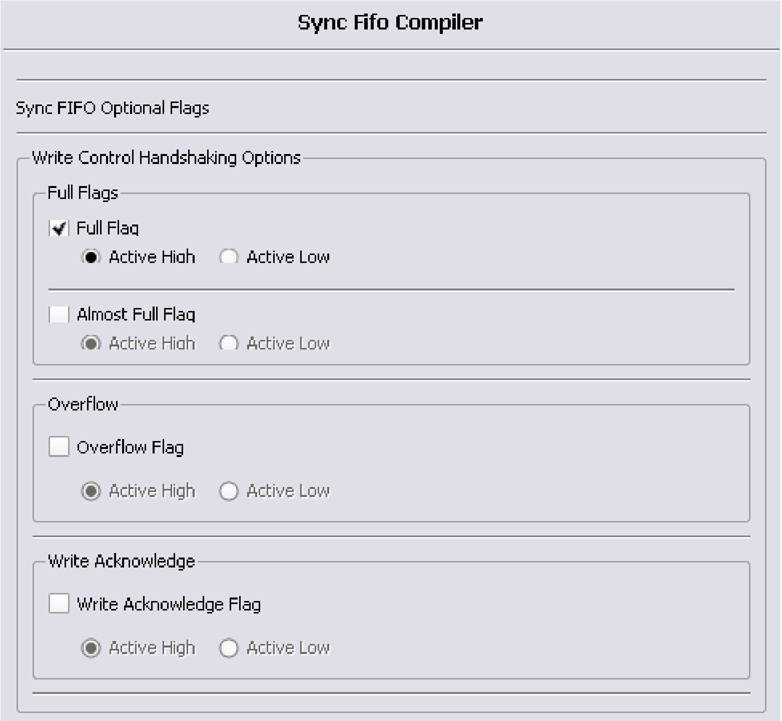
Width 8 Valid Range 1..256

Depth 16 Valid Range 8..16384

Parameter	Function
Component Name	Specifies a name for the FIFO. This is the name that you instantiate in your design file to create an instance of the SYNCore FIFO in your design. Do not use spaces.
Directory	Indicates the directory where the generated files will be stored. Do not use spaces.

Parameter	Function
Filename	Specifies the name of the generated file containing the HDL description of the generated FIFO. Do not use spaces.
Width	Specifies the width of the FIFO data input and output. It must be within the valid range.
Depth	Specifies the depth of the FIFO. It must be within the valid range.

SYNCore FIFO Parameters Page 2



Sync Fifo Compiler

Sync FIFO Optional Flags

Write Control Handshaking Options

Full Flags

☑ Full Flag

● Active High ○ Active Low

□ Almost Full Flag

● Active High ○ Active Low

Overflow

□ Overflow Flag

● Active High ○ Active Low

Write Acknowledge

□ Write Acknowledge Flag

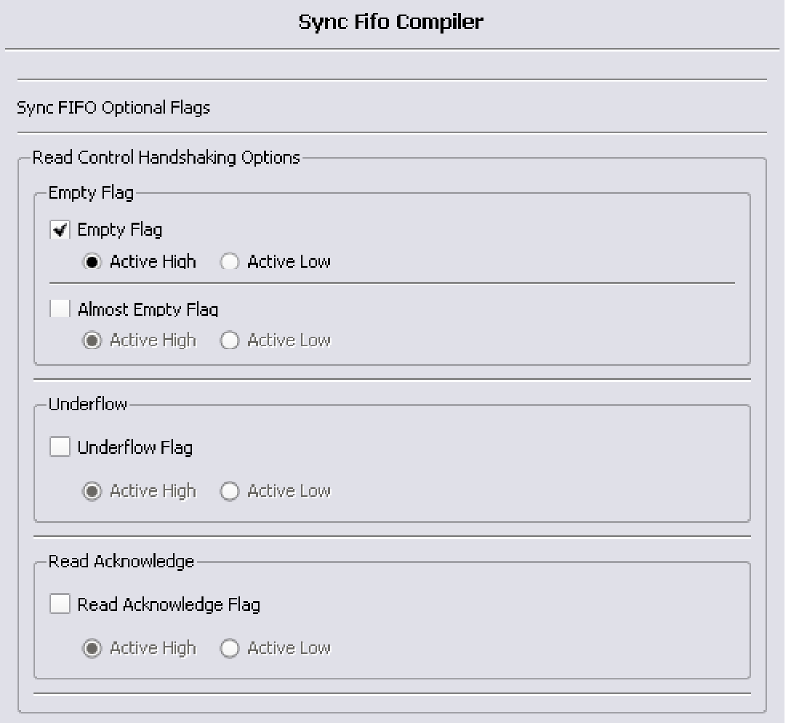
● Active High ○ Active Low

The page 2 parameters let you specify optional handshaking flags for FIFO write operations. When you specify a flag, the symbol on the left reflects your choice. Data is written/read on the rising edge of the clock.

Parameter	Function
Full Flag	Specifies a Full signal, which is asserted when the FIFO memory queue is full and no more writes can be performed until data is read. Enabling this option makes the Active High and Active Low options (FULL_FLAG_SENSE parameter) available for the signal. See Full/Almost Full Flags, on page 784 and FIFO Parameters, on page 782 for descriptions of the flag and parameter.
Almost Full Flag	Specifies an Almost_full signal, which is asserted to indicate that there is one location left and the FIFO will be full after one more write operation. Enabling this option makes the Active High and Active Low options available for the signal (AFULL_FLAG_SENSE parameter. See Full/Almost Full Flags, on page 784 and FIFO Parameters, on page 782 for descriptions of the flag and parameter.
Overflow Flag	Specifies an Overflow signal, which is asserted to indicate that the write operation was unsuccessful because the FIFO was full. Enabling this option makes the Active High and Active Low options available for the signal (OVERFLOW_FLAG_SENSE parameter). See Handshaking Flags, on page 786 f and FIFO Parameters, on page 782 for descriptions of the flag and parameter.
Write Acknowledge Flag	Specifies a Write_ack signal, which is asserted at the completion of a successful write operation. Enabling this option makes the Active High and Active Low options (WACK_FLAG_SENSE parameter) available for the signal. See Handshaking Flags, on page 786 and FIFO Parameters, on page 782 for descriptions of the flag and parameter.
Active High	Sets the specified signal to active high (1).
Active Low	Sets the specified signal to active low (0).

SYNCore FIFO Parameters Page 3

The page 3 parameters let you specify optional handshaking flags for FIFO read operations. Data is written/read on the rising edge of the clock.



Sync Fifo Compiler

Sync FIFO Optional Flags

Read Control Handshaking Options

Empty Flag

☑ Empty Flag

● Active High ○ Active Low

□ Almost Empty Flag

● Active High ○ Active Low

Underflow

□ Underflow Flag

● Active High ○ Active Low

Read Acknowledge

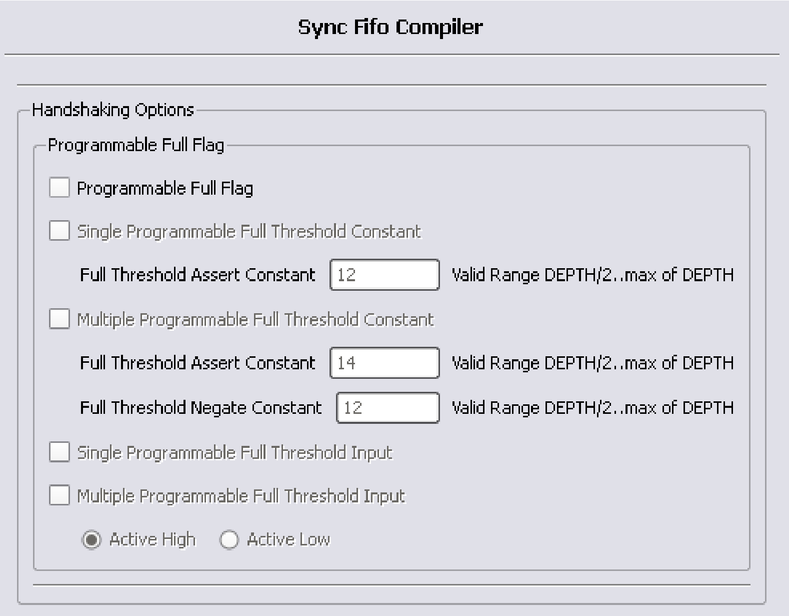
□ Read Acknowledge Flag

● Active High ○ Active Low

Parameter	Function
Empty Flag	Specifies an Empty signal, which is asserted when the memory queue for the FIFO is empty and no more reads can be performed until data is written. Enabling this option makes the Active High and Active Low options (EMPTY_FLAG_SENSE parameter) available for the signal. See Empty/Almost Empty Flags, on page 785 and FIFO Parameters, on page 782 for descriptions of the flag and parameter.

Parameter	Function
Almost Empty Flag	Specifies an Almost_empty signal, which is asserted when there is only one location left to be read. The FIFO will be empty after one more read operation. Enabling this option makes the Active High and Active Low options (AEMPTY_FLAG_SENSE parameter) available for the signal. See Empty/Almost Empty Flags, on page 785 and FIFO Parameters, on page 782 for descriptions of the flag and parameter.
Underflow Flag	Specifies an Underflow signal, which is asserted to indicate that the read operation was unsuccessful because the FIFO was empty. Enabling this option makes the Active High and Active Low options (UNDRFLW_FLAG_SENSE parameter) available for the signal. See Handshaking Flags, on page 786 and FIFO Parameters, on page 782 for descriptions of the flag and parameter.
Read Acknowledge Flag	Specifies a Read_ack signal, which is asserted at the completion of a successful read operation. Enabling this option makes the Active High and Active Low options (RACK_FLAG_SENSE parameter) available for the signal. See Handshaking Flags, on page 786 and FIFO Parameters, on page 782 for descriptions of the flag and parameter.
Active High	Sets the specified signal to active high (1).
Active Low	Sets the specified signal to active low (0).

SYNCore FIFO Parameters Page 4



Sync Fifo Compiler

Handshaking Options

Programmable Full Flag

□ Programmable Full Flag

□ Single Programmable Full Threshold Constant

Full Threshold Assert Constant: 12 Valid Range DEPTH/2..max of DEPTH

□ Multiple Programmable Full Threshold Constant

Full Threshold Assert Constant: 14 Valid Range DEPTH/2..max of DEPTH

Full Threshold Negate Constant: 12 Valid Range DEPTH/2..max of DEPTH

□ Single Programmable Full Threshold Input

□ Multiple Programmable Full Threshold Input

● Active High ○ Active Low

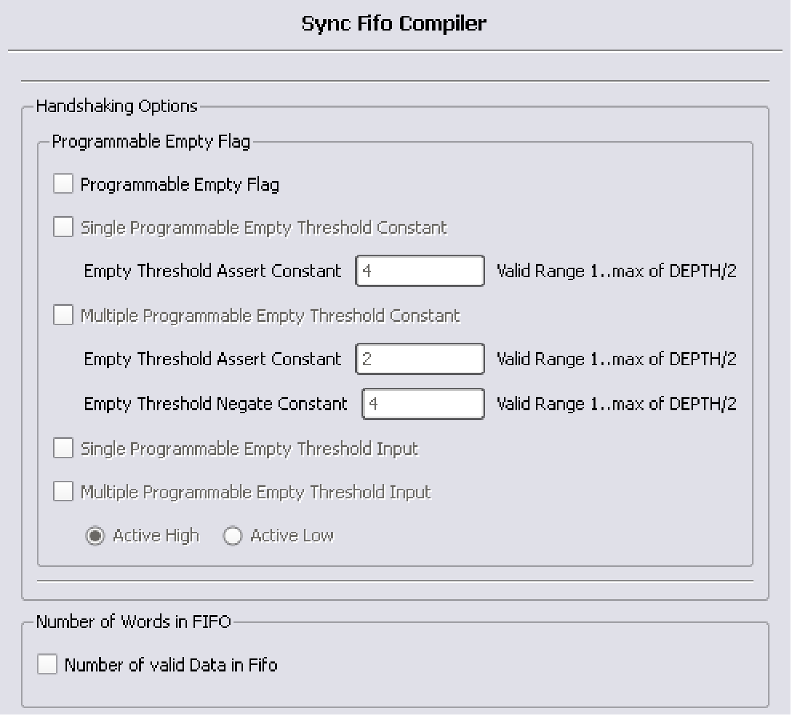
The page 4 parameters let you specify optional handshaking flags for FIFO programmable full operations. To use these options, you must have a Full signal specified. See FIFO Programmable Flags, on page 787 for details and FIFO Parameters, on page 782 for a list of the FIFO parameters. Data is written/read on the rising edge of the clock.

Parameter	Function
Programmable Full Flag	Specifies a Prog_full signal, which indicates that the FIFO has reached a user-defined full threshold. You can only enable this option if you set Full Flag on page 2. When it is enabled, you can specify other options for the Prog_Full signal (PFULL_FLAG_SENSE parameter). See Programmable Full, on page 788 and FIFO Parameters, on page 782 for descriptions of the flag and parameter.

Parameter	Function
Single Programmable Full Threshold Constant	Specifies a Prog_full signal with a single constant defining the assertion threshold (PGM_FULL_TYPE=1 parameter). See Programmable Full with Single Threshold Constant, on page 788 for details. Enabling this option makes Full Threshold Assert Constant available.
Multiple Programmable Full Threshold Constant	Specifies a Prog_full signal (PGM_FULL_TYPE=2 parameter), with multiple constants defining the assertion and de-assertion thresholds. See Programmable Full with Multiple Threshold Constants, on page 789 for details. Enabling this option makes Full Threshold Assert Constant and Full Threshold Negate Constant available.
Full Threshold Assert Constant	Specifies a constant that is used as a threshold value for asserting the Prog_full signal It sets the PGM_FULL_THRESH parameter for PGM_FULL_TYPE=1 and the PGM_FULL_ATHRESH parameter for PGM_FULL_TYPE=2.
Full Threshold Negate Constant	Specifies a constant that is used as a threshold value for de-asserting the Prog_full signal (PGM_FULL_NTHRESH parameter).
Single Programmable Full Threshold Input	Specifies a Prog_full signal (PGM_FULL_TYPE=3 parameter), with a threshold value specified dynamically through a Prog_full_thresh input port during the reset state. See Programmable Full with Single Threshold Input, on page 789 for details. Enabling this option adds the Prog_full_thresh input port to the FIFO.
Multiple Programmable Full Threshold Input	Specifies a Prog_full signal (PGM_FULL_TYPE=4 parameter), with threshold assertion and deassertion values specified dynamically through input ports during the reset state. See Programmable Full with Multiple Threshold Constants, on page 789 for details. Enabling this option adds the Prog_full_thresh_assert and Prog_full_thresh_negate input ports to the FIFO.
Active High	Sets the specified signal to active high (1).
Active Low	Sets the specified signal to active low (0).

SYNCore FIFO Parameters Page 5

These options specify optional handshaking flags for FIFO programmable empty operations. To use these options, you first specify an Empty signal on page 3. See FIFO Programmable Flags, on page 787 for details and FIFO Parameters, on page 782 for a list of the FIFO parameters. Data is written/read on the rising edge of the clock.



Sync Fifo Compiler

Handshaking Options

Programmable Empty Flag

□ Programmable Empty Flag

□ Single Programmable Empty Threshold Constant

Empty Threshold Assert Constant: 4 Valid Range 1..max of DEPTH/2

□ Multiple Programmable Empty Threshold Constant

Empty Threshold Assert Constant: 2 Valid Range 1..max of DEPTH/2

Empty Threshold Negate Constant: 4 Valid Range 1..max of DEPTH/2

□ Single Programmable Empty Threshold Input

□ Multiple Programmable Empty Threshold Input

● Active High ○ Active Low

Number of Words in FIFO

□ Number of valid Data in Fifo

Parameter	Function
Programmable Empty Flag	Specifies a Prog_empty signal (PEMPTY_FLAG_SENSE parameter), which indicates that the FIFO has reached a user-defined empty threshold. See Programmable Empty, on page 791 and FIFO Parameters, on page 782 for descriptions of the flag and parameter. Enabling this option makes the other options available to specify the threshold value, either as a constant or through input ports. You can also specify single or multiple thresholds for each of these options.

Parameter	Function
Single Programmable Empty Threshold Constant	Specifies a Prog_empty signal (PGM_EMPTY_TYPE=1 parameter), with a single constant defining the assertion threshold. See Programmable Empty with Single Threshold Input, on page 792 for details. Enabling this option makes Empty Threshold Assert Constant available.
Multiple Programmable Empty Threshold Constant	Specifies a Prog_empty signal (PGM_EMPTY_TYPE=2 parameter), with multiple constants defining the assertion and de-assertion thresholds. See Programmable Empty with Multiple Threshold Constants, on page 791 for details. Enabling this option makes Empty Threshold Assert Constant and Empty Threshold Negate Constant available.
Empty Threshold Assert Constant	Specifies a constant that is used as a threshold value for asserting the Prog_empty signal. It sets the PGM_EMPTY_THRESH parameter for PGM_EMPTY_TYPE=1 and the PGM_EMPTY_ATHRESH parameter for PGM_EMPTY_TYPE=2.
Empty Threshold Negate Constant	Specifies a constant that is used as a threshold value for de-asserting the Prog_empty signal (PGM_EMPTY_NTHRESH parameter).
Single Programmable Empty Threshold Input	Specifies a Prog_empty signal (PGM_EMPTY_TYPE=3 parameter), with a threshold value specified dynamically through a Prog_empty_thresh input port during the reset state. See Programmable Empty with Single Threshold Input, on page 792 for details. Enabling this option adds the Prog_full_thresh input port to the FIFO.
Multiple Programmable Empty Threshold Input	Specifies a Prog_empty signal (PGM_EMPTY_TYPE=4 parameter), with threshold assertion and deassertion values specified dynamically through Prog_empty_thresh_assert and Prog_empty_thresh_negate input ports during the reset state. See Programmable Empty with Multiple Threshold Inputs, on page 793 for details. Enabling this option adds the input ports to the FIFO.
Active High	Sets the specified signal to active high (1).
Active Low	Sets the specified signal to active low (0).

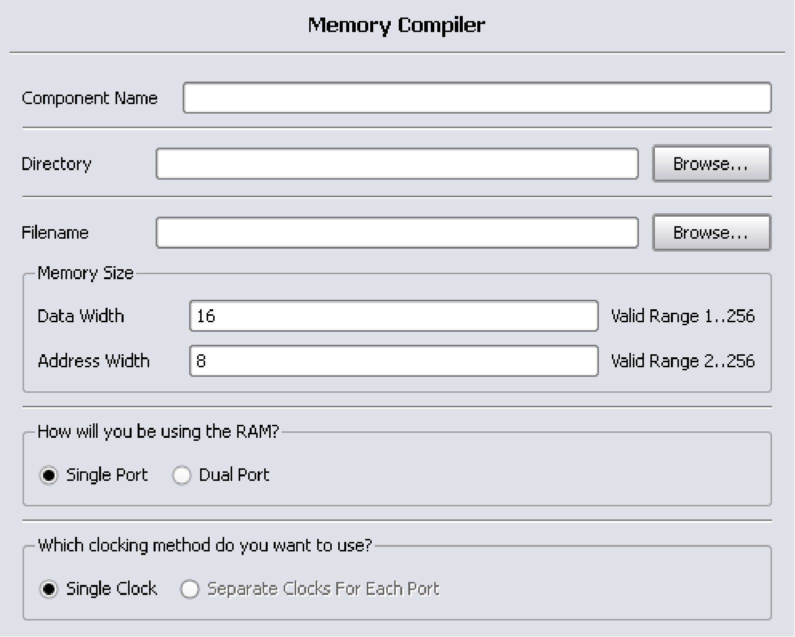
Parameter	Function
Number of Valid Data in FIFO	Specifies the Data_cnt signal for the FIFO output. This signal contains the number of words in the FIFO in the read domain.

SYNCore RAM Wizard

The following describe the parameters you can set in the RAM wizard, which opens when you select ram_model:

- SYNCore RAM Parameters Page 1, on page 198
- SYNCore RAM Parameters Pages 2 and 3, on page 200

SYNCore RAM Parameters Page 1



Memory Compiler

Component Name

Directory

Filename

Memory Size

Data Width 16 Valid Range 1..256

Address Width 8 Valid Range 2..256

How will you be using the RAM?

● Single Port ○ Dual Port

Which clocking method do you want to use?

● Single Clock ○ Separate Clocks For Each Port

Component Name

Specifies the name of the component. This is the name that you instantiate in your design file to create an instance of the SYNCore RAM in your design. Do not use spaces. For example:

```
ram101 <ComponentName> (
   .PortAClk(PortAClk)
   , .PortAAddr(PortAAddr)
   , .PortADataIn(PortADataIn)
   , .PortAWriteEnable(PortAWriteEnable)
   , .PortBDataIn(PortBDataIn)
   , .PortBAddr(PortBAddr)
   , .PortBWriteEnable(PortBWriteEnable)
   , .PortADataOut(PortADataOut)
   , .PortBDataOut(PortBDataOut)
);
```

Directory

Specifies the directory where the generated files are stored. Do not use spaces. The following files are created:

- filelist.txt – lists files written out by SYNCore
- options.txt – lists the options selected in SYNCore
- readme.txt – contains a brief description and known issues
- syncore_ram.v – Verilog library file required to generate RAM model
- testbench.v – Verilog testbench file for testing the RAM model
- instantiation_file.vin – describes how to instantiate the wrapper file
- *component*.v – RAM model wrapper file generated by SYNCore

Note that running the Memory Compiler wizard in the same directory overwrites the existing files.

Filename

Specifies the name of the generated file containing the HDL description of the compiled RAM. Do not use spaces.

Data Width

Is the width of the data you need for the memory. The unit used is the number of bits.

Address Width

Is the address depth you need for the memory. The unit used is the number of bits.

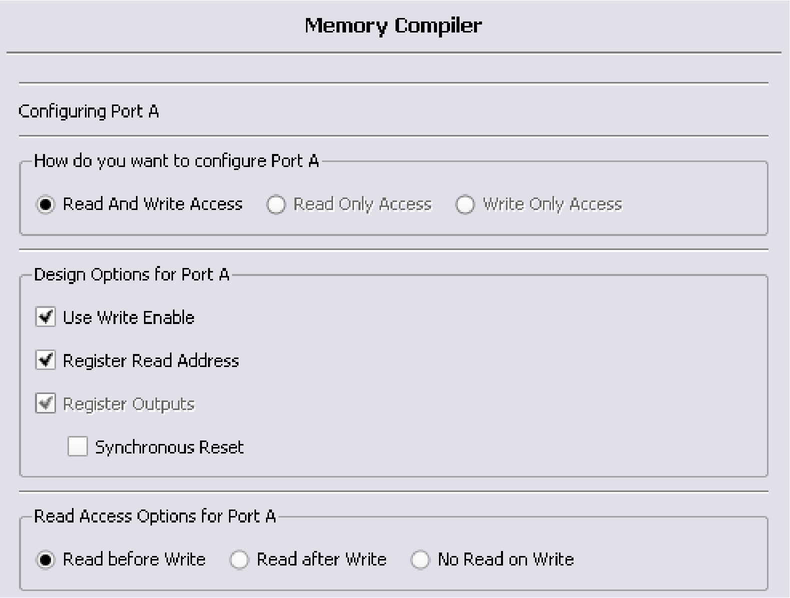
Single Port

When enabled, generates a single-port RAM.

Dual Port	When enabled, generates a dual-port RAM.
Single Clock	When enabled, generates a RAM with a single clock for dual-port configurations.
Separate Clocks for Each Port	When enabled, generates separate clocks for each port in dual-port RAM configurations.

SYNCore RAM Parameters Pages 2 and 3

The port implementation parameters on pages 2 and 3 are identical, but page 2 applies to Port A (single- and dual-port configurations), and page 3 applies to Port B (dual-port configurations only). The following figure shows the parameters on page 2 for Port A.



Memory Compiler

Configuring Port A

How do you want to configure Port A

● Read And Write Access ○ Read Only Access ○ Write Only Access

Design Options for Port A

☑ Use Write Enable

☑ Register Read Address

☑ Register Outputs

□ Synchronous Reset

Read Access Options for Port A

● Read before Write ○ Read after Write ○ No Read on Write

Read and Write Access	Specifies that the port can be accessed by both read and write operations
Read Only Access	Specifies that the port can only be accessed by read operations.
Write Only Access	Specifies that the port can only be accessed by write operations
Use Write Enable	Includes write-enable control. The RAM symbol on the left reflects the selections you make.

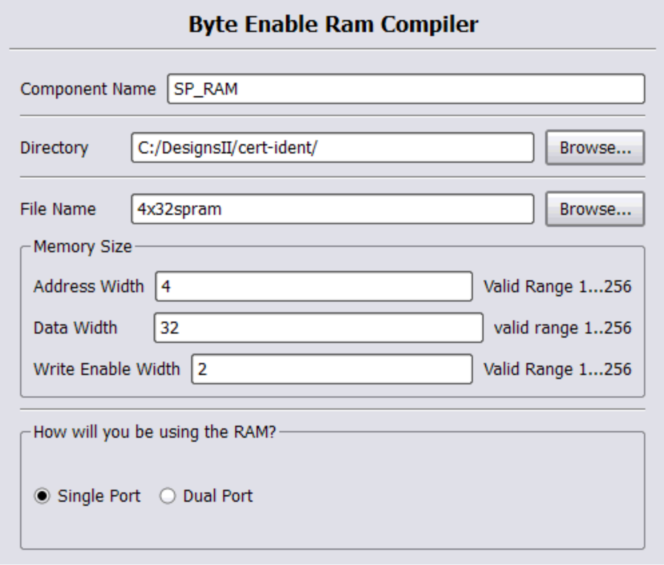
Register Read Address	Adds registers to the read address lines. The RAM symbol on the left reflects the selections you make.
Register Outputs	Adds registers to the write address lines when you specify separate read/write addressing. The register outputs are always enabled. The RAM symbol on the left reflects the selections you make.
Synchronous Reset	Individually synchronizes the reset signal with the clock when you enable Register Outputs. The RAM symbol on the left reflects the selections you make.
Read before Write	Specifies that the read operation takes place before the write operation for port configurations with both read and write access (Read And Write Access is enabled). For a timing diagram, see Read Before Write, on page 801.
Read after Write	Specifies that the read operation takes place after the write operation for port configurations with both read and write access (Read And Write Access is enabled). For a timing diagram, see Write Before Read, on page 802.
No Read on Write	Specifies that no read operation takes place when there is a write operation for port configurations with both read and write access (Read And Write Access is enabled). For a timing diagram, see No Read on Write, on page 803.

SYNCore Byte-Enable RAM Wizard

The following describes the parameters you can set in the byte-enable RAM wizard, which opens when you select byte_en_ram.

- SYNCore Byte-Enable RAM Parameters Page 1, on page 202
- SYNCore Byte-Enable RAM Parameters Pages 2 and 3, on page 204

SYNCore Byte-Enable RAM Parameters Page 1



Byte Enable Ram Compiler

Component Name: SP_RAM

Directory: C:/DesignsII/cert-ident/

File Name: 4x32spram

Memory Size

Address Width: 4 Valid Range 1...256

Data Width: 32 valid range 1..256

Write Enable Width: 2 Valid Range 1...256

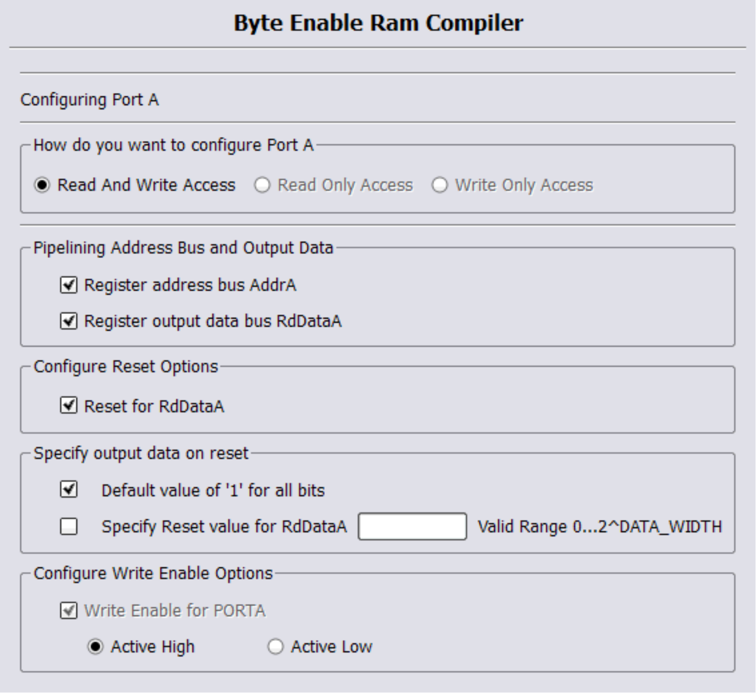
How will you be using the RAM?

● Single Port ○ Dual Port

Component Name	Specifies the name of the component. This is the name that you instantiate in your design file to create an instance of the SYNCore byte-enable RAM in your design. Do not use spaces.
Directory	Specifies the directory where the generated files are stored. Do not use spaces. The following files are created: • filelist.txt – lists files written out by SYNCore • options.txt – lists the options selected in SYNCore • readme.txt – contains a brief description and known issues • syncore_be_ram_sdp.v – SystemVerilog library file required to generate single or simple dual-port, byte-enable RAM model • syncore_be_ram_tdp.v – SystemVerilog library file required to generate true dual-port byte-enable RAM model • testbench.v – Verilog testbench file for testing the byte-enable RAM model • instantiation_file.vin – describes how to instantiate the wrapper file • *component*.v – Byte-enable RAM model wrapper file generated by SYNCore Note that running the byte-enable RAM wizard in the same directory overwrites the existing files.
Filename	Specifies the name of the generated file containing the HDL description of the compiled byte-enable RAM. Do not use spaces.
Address Width	Specifies the address depth for Ports A and B. The unit used is the number of bits; the default is 2
Data Width	Specifies the width of the data for Ports A and B. The unit used is the number of bits; the default is 2
Write Enable Width	Specifies the write enable width for Ports A and B. The unit used is the number of byte enables; the default is 2, the maximum is 4.
Single Port	When enabled, generates a single-port, byte-enable RAM (automatically enables single clock).
Dual Port	When enabled, generates a dual-port, byte-enable RAM (automatically enables separate clocks for each port).

SYNCore Byte-Enable RAM Parameters Pages 2 and 3

The port implementation parameters on pages 2 and 3 are identical, but page 2 applies to Port A (single- and dual-port configurations), and page 3 applies to Port B (dual-port configurations only). The following figure shows the parameters on page 2 for Port A.



Byte Enable Ram Compiler

Configuring Port A

How do you want to configure Port A

● Read And Write Access ○ Read Only Access ○ Write Only Access

Pipelining Address Bus and Output Data

☑ Register address bus AddrA

☑ Register output data bus RdDataA

Configure Reset Options

☑ Reset for RdDataA

Specify output data on reset

☑ Default value of '1' for all bits

☐ Specify Reset value for RdDataA [ ] Valid Range 0...2^DATA_WIDTH

Configure Write Enable Options

☑ Write Enable for PORTA

● Active High ○ Active Low

Read and Write Access	Specifies that the port can be accessed by both read and write operations (only mode allowed for single-port configurations).
Read Only Access	Specifies that the port can only be accessed by read operations (dual-port mode only).
Write Only Access	Specifies that the port can only be accessed by write operations (dual-port mode only).
Register address bus AddrA/B	Adds registers to the read address lines.
Register output data bus RdDataA/B	Adds registers to the read data lines. By default, the read data register is enabled.

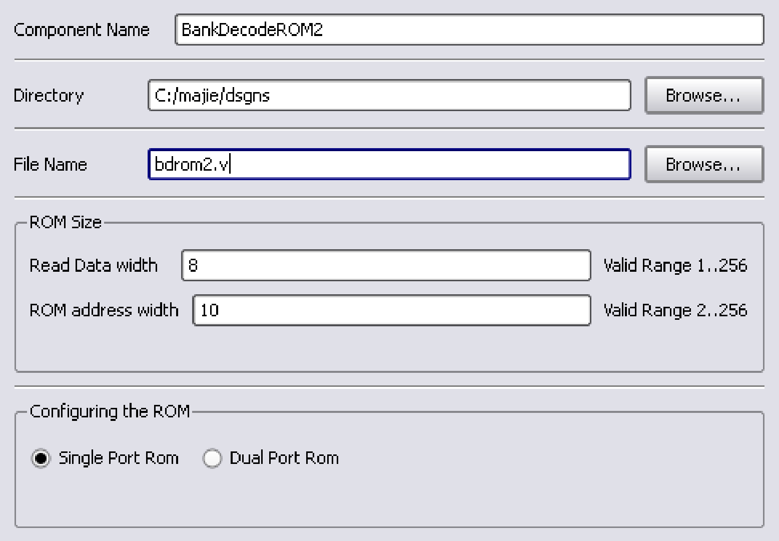
Reset for RdDataA/B	Specifies the reset type for registered read data: • Reset type is synchronous when Reset for RdDataA/B is enabled • Reset type is no reset when Reset for RdDataA/B is disabled
Specify output data on reset	Specifies reset value for registered read data (applies only when RdDataA/B is enabled): • Default value of '1' for all bits – sets read data to all 1's on reset • Specify Reset value for RdDataA/B – specifies reset value for read data; when enabled, value is entered in adjacent field.
Write Enable for Port A/B	Specifies the write enable level for Port A/B. Default is Active High.

SYNCore ROM Wizard

The following describe the parameters you can set in the ROM wizard, which opens when you select rom_model:

- SYNCore ROM Parameters Page 1, on page 206
- SYNCore ROM Parameters Pages 2 and 3, on page 207
- SYNCore ROM Parameters Page 4, on page 209

SYNCore ROM Parameters Page 1



Component Name: BankDecodeROM2

Directory: C:/majie/dsgns

File Name: bdrom2.v

ROM Size

Read Data width: 8 — Valid Range 1..256

ROM address width: 10 — Valid Range 2..256

Configuring the ROM

● Single Port Rom ○ Dual Port Rom

Component Name

Specifies the name of the component. This is the name that you instantiate in your design file to create an instance of the SYNCore ROM in your design. Do not use spaces.

Directory

Specifies the directory where the generated files are stored. Do not use spaces. The following files are created:

filelist.txt – lists files written out by SYNCore

options.txt – lists the options selected in SYNCore

readme.txt – contains a brief description and known issues

syncore_rom.v – Verilog library file required to generate ROM model

testbench.v – Verilog testbench file for testing the ROM model

instantiation_file.vin – describes how to instantiate the wrapper file

component.v – ROM model wrapper file generated by SYNCore

Note that running the ROM wizard in the same directory overwrites the existing files.

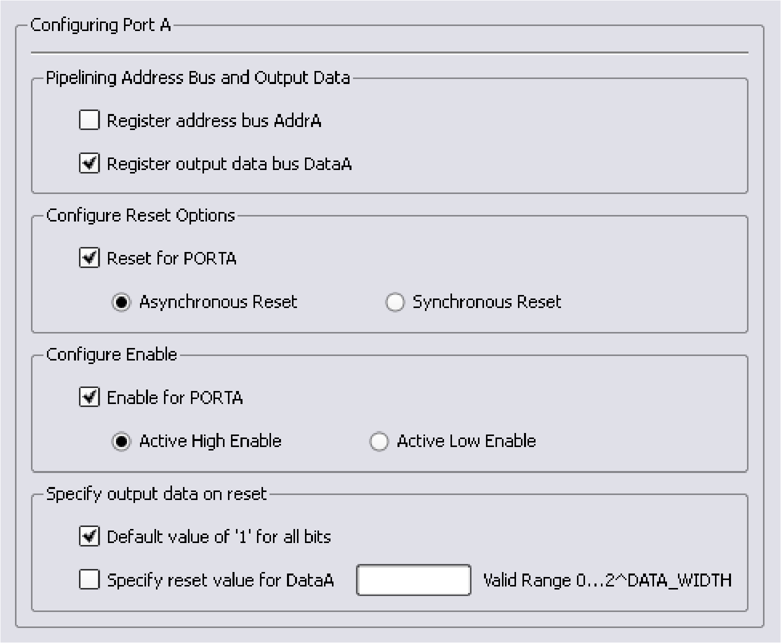
File Name

Specifies the name of the generated file containing the HDL description of the compiled ROM. Do not use spaces.

Read Data Width	Specifies the read data width of the ROM. The unit used is the number of bits and ranges from 2 to 256. Default value is 8. The read data width is common to both Port A and Port B. The corresponding file parameter is DATA_WIDTH=n.
ROM address width	Specifies the address depth for the memory. The unit used is the number of bits. Default value is 10. The corresponding file parameter is ADD_WIDTH=n.
Single Port Rom	When enabled, generates a single-port ROM. The corresponding file parameter is CONFIG_PORT="single".
Dual Port Rom	When enabled, generates a dual-port ROM. The corresponding file parameter is CONFIG_PORT="dual".

SYNCore ROM Parameters Pages 2 and 3

The port implementation parameters on pages 2 and 3 are the same; page 2 applies to Port A (single- and dual-port configurations), and page 3 applies to Port B (dual-port configurations only).



Configuring Port A

Pipelining Address Bus and Output Data

☐ Register address bus AddrA

☑ Register output data bus DataA

Configure Reset Options

☑ Reset for PORTA

● Asynchronous Reset ○ Synchronous Reset

Configure Enable

☑ Enable for PORTA

● Active High Enable ○ Active Low Enable

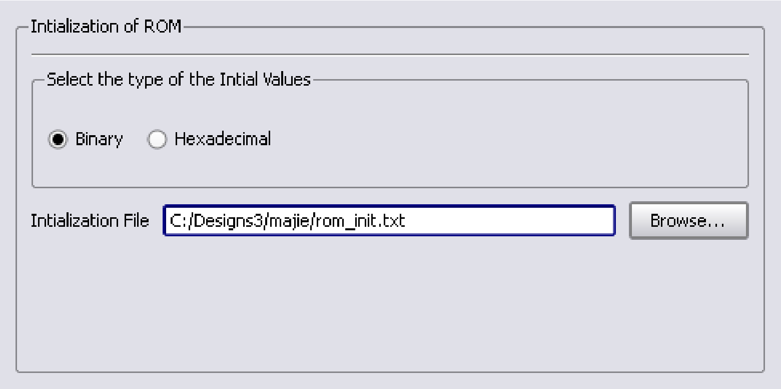
Specify output data on reset

☑ Default value of '1' for all bits

☐ Specify reset value for DataA Valid Range 0...2^DATA_WIDTH

Register address bus AddrA	Used with synchronous ROM configurations to register the read address. When checked, also allows chip enable to be configured.
Register output data bus DataA	Used with synchronous ROM configurations to register the data outputs. When checked, also allows chip enable to be configured.
Asynchronous Reset	Sets the type of reset to asynchronous (Configure Reset Options must be checked). Configuring reset also allows the output data pattern on reset to be defined. The corresponding file parameter is RST_TYPE_A=1/RST_TYPE_B=1.
Synchronous Reset	Sets the type of reset to synchronous (Configure Reset Options must be checked). Configuring reset also allows the output data pattern on reset to be defined.The corresponding file parameter is RST_TYPE_A=0/RST_TYPE_B=0.
Active High Enable	Sets the level of the chip enable to high for synchronous ROM configurations. The corresponding file parameter is EN_SENSE_A=1/EN_SENSE_B=1.
Active Low Enable	Sets the level of the chip enable to low for synchronous ROM configurations. The corresponding file parameter is EN_SENSE_A=0/EN_SENSE_B=0.
Default value of '1' for all bits	Specifies an output data pattern of all 1's on reset. The corresponding file parameter is RST_DATA_A={n{1'b1} }/RST_DATA_B={n{1'b1} }.
Specify reset value for DataA/DataB	Specifies a user-defined output data pattern on reset. The pattern is defined in the adjacent field. The corresponding file parameter is RST_TYPE_A=pattern/RST_TYPE_B=pattern.

SYNCore ROM Parameters Page 4



Intialization of ROM

Select the type of the Intial Values

● Binary ○ Hexadecimal

Intialization File C:/Designs3/majie/rom_init.txt

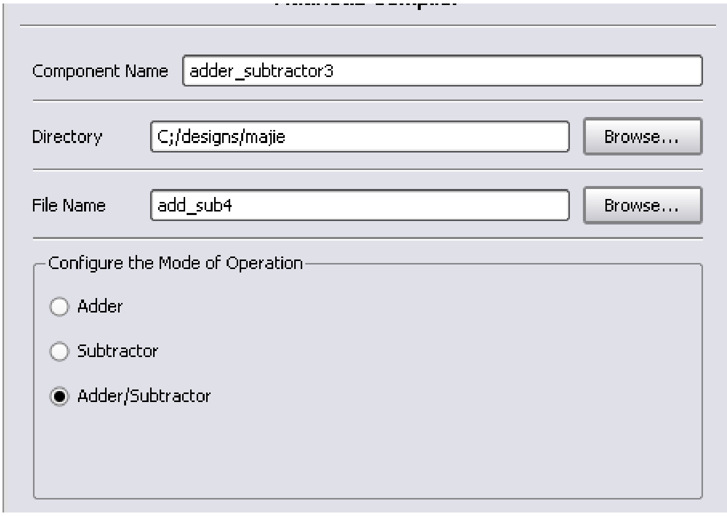
Binary	Specifies binary-formatted initialization file.
Hexadecimal	Specifies hexadecimal-formatted initial file.
Initialization File	Specifies path and filename of initialization file. The corresponding file parameter is INIT_FILE="*filename*".

SYNCore Adder/Subtractor Wizard

The following describe the parameters you can set in the adder/subtractor wizard, which opens when you select addnsub_model:

- SYNCore Adder/Subtractor Parameters Page 1, on page 210
- SYNCore Adder/Subtractor Parameters Page 2, on page 211

SYNCore Adder/Subtractor Parameters Page 1



Component Name adder_subtractor3

Directory C;/designs/majie

File Name add_sub4

Configure the Mode of Operation

○ Adder

○ Subtractor

● Adder/Subtractor

Component Name

Specifies a name for the adder/subtractor. This is the name that you instantiate in your design file to create an instance of the SYNCore adder/subtractor in your design. Do not use spaces.

Directory

Indicates the directory where the generated files will be stored. Do not use spaces. The following files are created:

- filelist.txt – lists files written out by SYNCore
- options.txt – lists the options selected in SYNCore
- readme.txt – contains a brief description and known issues
- syncore_ADDnSUB.v – Verilog library file required to generate adder/subtractor model
- testbench.v – Verilog testbench file for testing the adder/subtractor model
- instantiation_file.vin – describes how to instantiate the wrapper file
- *component*.v – adder/subtractor model wrapper file generated by SYNCore

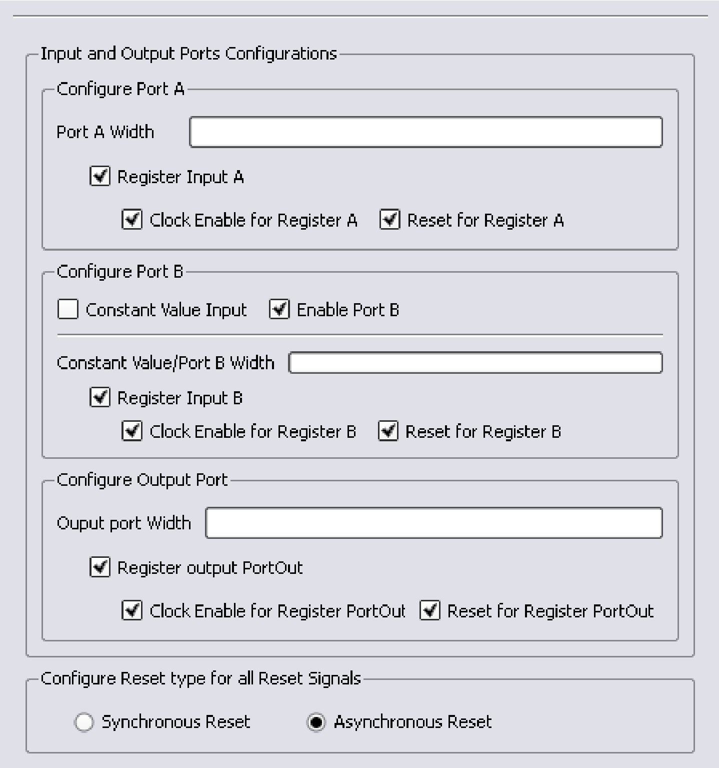
Note that running the wizard in the same directory overwrites any existing files.

Filename

Specifies the name of the generated file containing the HDL description of the generated adder/subtractor. Do not use spaces.

Adder	When enabled, generates an adder (the corresponding file parameter is ADD_N_SUB ="ADD")
Subtractor	When enabled, generates a subtractor (the corresponding file parameter is ADD_N_SUB ="SUB")
Adder/Subtractor	When enabled, generates a dynamic adder/subtractor (the corresponding file parameter is ADD_N_SUB ="DYNAMIC")

SYNCore Adder/Subtractor Parameters Page 2



Input and Output Ports Configurations

Configure Port A

Port A Width

☑ Register Input A

☑ Clock Enable for Register A ☑ Reset for Register A

Configure Port B

□ Constant Value Input ☑ Enable Port B

Constant Value/Port B Width

☑ Register Input B

☑ Clock Enable for Register B ☑ Reset for Register B

Configure Output Port

Ouput port Width

☑ Register output PortOut

☑ Clock Enable for Register PortOut ☑ Reset for Register PortOut

Configure Reset type for all Reset Signals

○ Synchronous Reset ● Asynchronous Reset

Port A Width	Specifies the width of port A (the corresponding file parameter is PORT_A_WIDTH=n)
Register Input A	Used with synchronous adder/subtractor configurations to register port A. When checked, also allows clock enable and reset to be configured (the corresponding file parameter is PORTA_PIPELINE_STAGE='0' or '1')
Clock Enable for Register A	Specifies the enable for port A register
Reset for Register A	Specifies the reset for port A register
Constant Value Input	Specifies port B as a constant input when checked and allows you to enter a constant value in the Constant Value/Port B Width field (the corresponding file parameter is CONSTANT_PORT ='0')
Enable Port B	Specifies port B as an input when checked and allows you to enter a port B width in the Constant Value/Port B Width field (the corresponding file parameter is CONSTANT_PORT ='1')
Constant Value/Port B Width	Specifies either a constant value or port B width depending on Constant Value Input and Enable Port B selection (the corresponding file parameters are CONSTANT_VALUE= n or PORT_B_WIDTH=n)
Register Input B	Used with synchronous adder/subtractor configurations to register port B. When checked, also allows clock enable and reset to be configured (the corresponding file parameter is PORTB_PIPELINE_STAGE='0' or '1')
Clock Enable for Register B	Specifies the enable for the port B register
Reset for Register B	Specifies the reset for the port B register
Output port Width	Specifies the width of the output port (the corresponding file parameter is PORT_OUT_WIDTH=n)
Register output PortOut	Used with synchronous adder/subtractor configurations to register the output port. When checked, also allows clock enable and reset to be configured (the corresponding file parameter is PORTOUT_PIPELINE_STAGE='0' or '1'
Clock Enable for Register PortOut	Specifies the enable for the output port register

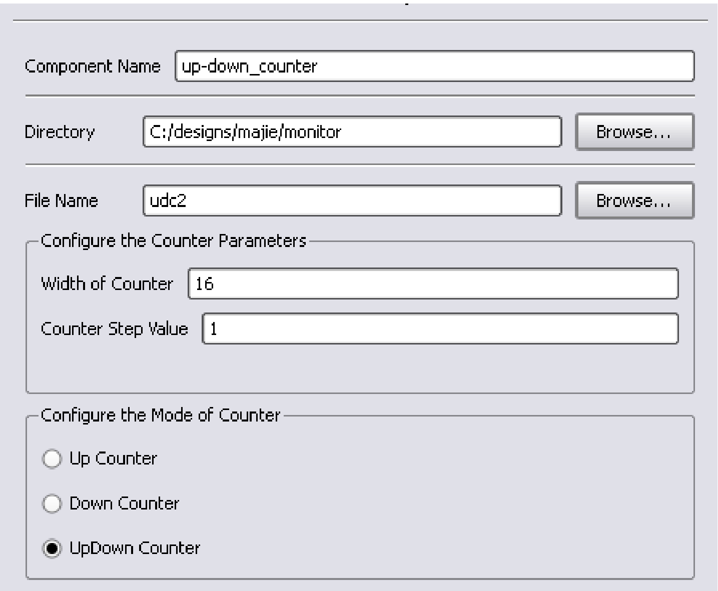
Reset for Register PortOut	Specifies the reset for the output port register
Synchronous Reset	Sets the type of reset to synchronous (the corresponding file parameter is RESET_TYPE='0')
Asynchronous Reset	Sets the type of reset to asynchronous (the corresponding file parameter is RESET_TYPE='1')

SYNCore Counter Wizard

The following describe the parameters you can set in the ROM wizard, which opens when you select counter_model:

- SYNCore Counter Parameters Page 1, on page 213
- SYNCore Counter Parameters Page 2, on page 215

SYNCore Counter Parameters Page 1



Component Name: up-down_counter

Directory: C:/designs/majie/monitor

File Name: udc2

Configure the Counter Parameters

Width of Counter: 16

Counter Step Value: 1

Configure the Mode of Counter

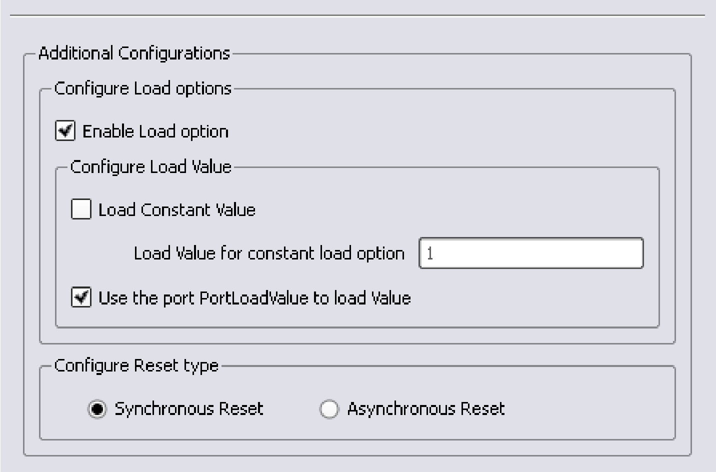
○ Up Counter

○ Down Counter

● UpDown Counter

Component Name	Specifies a name for the counter. This is the name that you instantiate in your design file to create an instance of the SYNCore counter in your design. Do not use spaces.
Directory	Indicates the directory where the generated files will be stored. Do not use spaces. The following files are created: • filelist.txt – lists files written out by SYNCore • options.txt – lists the options selected in SYNCore • readme.txt – contains a brief description and known issues • syncore_counter.v – Verilog library file required to generate counter model • testbench.v – Verilog testbench file for testing the counter model • instantiation_file.vin – describes how to instantiate the wrapper file • *component*.v – counter model wrapper file generated by SYNCore Note that running the wizard in the same directory overwrites any existing files.
Filename	Specifies the name of the generated file containing the HDL description of the generated counter. Do not use spaces.
Width of Counter	Determines the counter width (the corresponding file parameter is COUNT_WIDTH=n).
Counter Step Value	Determines the counter step value (the corresponding file parameter is STEP=n).
Up Counter	Specifies an up counter (the default) configuration (the corresponding file parameter is MODE=Up).
Down Counter	Specifies an down counter configuration (the corresponding file parameter is MODE=Down).
UpDown Counter	Specifies a dynamic up/down counter configuration (the corresponding file parameter is MODE=Dynamic).

SYNCore Counter Parameters Page 2



Additional Configurations

Configure Load options

☑ Enable Load option

Configure Load Value

☐ Load Constant Value

Load Value for constant load option: 1

☑ Use the port PortLoadValue to load Value

Configure Reset type

● Synchronous Reset ○ Asynchronous Reset

Enable Load option	Enables the load options
Load Constant Value	Load the constant value specified in the Load Value for constant load option field; (the corresponding file parameter is LOAD=1).
Load Value for constant load option	The constant value to be loaded.
Use the port PortLoadValue to load Value	Loads variable value from PortLoadValue (the corresponding file parameter is LOAD=2).
Synchronous Reset	Specifies a synchronous (the default) reset input (the corresponding file parameter is MODE=0).
Asynchronous Reset	Specifies an asynchronous reset input (the corresponding file parameter is MODE=1).

Configure and Launch VCS Simulator Command

The Configure and Launch VCS Simulator command enables you to launch VCS simulation from within the Synopsys FPGA synthesis tools. Additionally, configuration information, such as libraries and options can be specified on the Run VCS Simulator dialog box before running VCS simulation. You can launch this simulation tool from the synthesis tools on Linux platforms only.

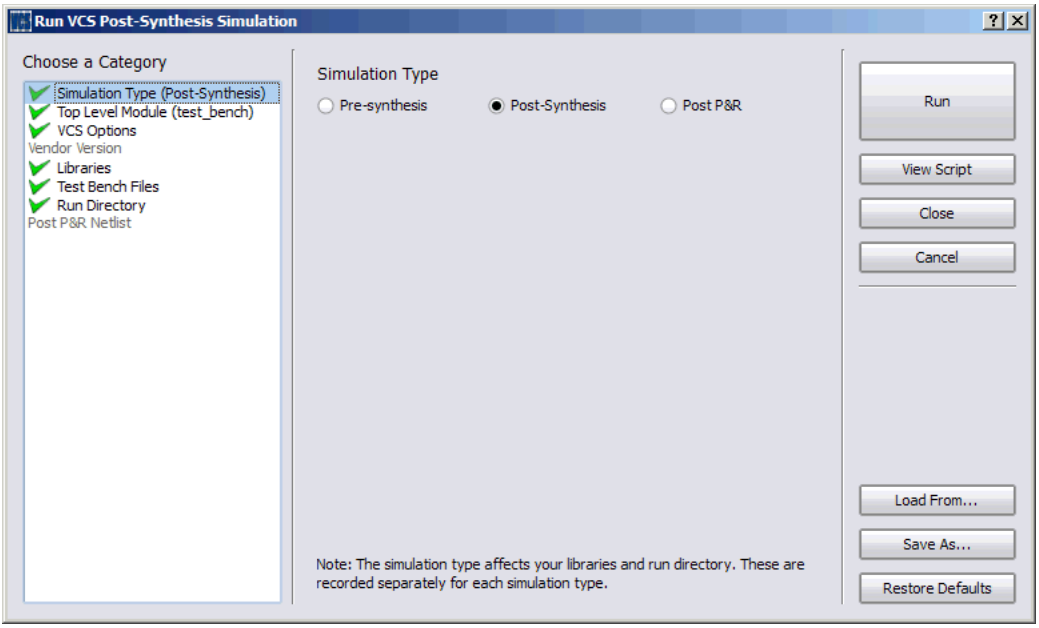
For a step-by-step procedure on setting up and launching this tool, see Simulating with the VCS Tool, on page 364 in the *User Guide*.

The Run VCS *SimulationType* Simulation dialog box contains unique pages for specific tasks, such as specifying simulation type, VCS options, and libraries or test bench files. From this dialog box:

- Choose a category, which simplifies the data input for each task.
- A task marked with (✓) means that data has automatically been filled in; however, an (⊗) requires that data must be filled in.
- You are prompted to save, after cancelling changes made in the dialog box.

Simulation Type

The following dialog box displays the Simulation Type task.



Simulation Type

○ Pre-synthesis ● Post-Synthesis ○ Post P&R

Note: The simulation type affects your libraries and run directory. These are recorded separately for each simulation type.

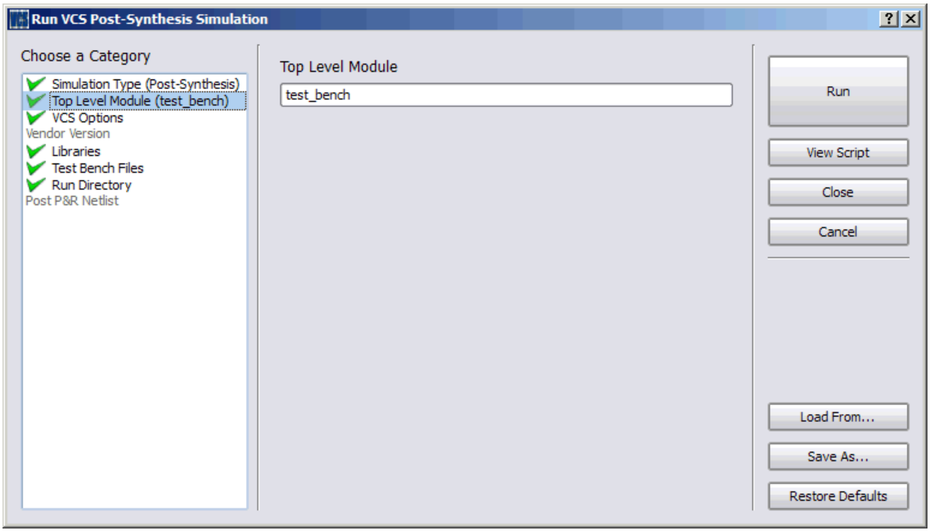
The Run VCS Simulator dialog box contains the following options:

Command	Description
Choose a Category Simulation Type	Select Simulation Type and choose the type of simulation to run: • Pre-synthesis – RTL simulation • Post-synthesis – Post-synthesis netlist simulation • Post-P&R – Post-P&R netlist simulation See Simulation Type, on page 216 to view the dialog box.
Choose a Category Top Level Module	Select Top Level Module and specify the top-level VCS module or modules for simulation. You can use any combination of the semi-colon (;), comma (,), or a space to separate multiple top-level modules. See Top Level Module, on page 220 to view the dialog box.
Choose a Category VCS Options	Select VCS Options and specify options for each VCS step: • Verilog compiler – VLOGAN command options for compiling and analyzing Verilog, like the -q option • VHDL compiler – VHDLAN options for compiling and analyzing VHDL • Elaboration – VCS command options. The default setting is -debug_all. • Simulation – SIMV command options. The default setting is -gui. The default settings use the FPGA version of VCS and open the VCS GUI for the debugger (DBE) and the waveform viewer. See VCS Options, on page 220 to view the dialog box.
Choose a Category Libraries	Select Libraries and specify library files typically used for Post-synthesis or Post-P&R simulation. These library files are automatically populated in the display window. You can choose to: • Add a library • Edit the selected library • Remove the selected library See Libraries, on page 221 and Changing Library and Test Bench Files, on page 223 for more information.

Command	Description
Choose a Category Test Bench Files	Select Test Bench Files and specify the test bench files typically used for Post-synthesis or Post-P&R simulation. These test bench files are automatically populated in the display window. You can choose to: • Add a test bench file • Edit the selected test bench file • Remove the selected test bench file See Test Bench Files, on page 222 and Changing Library and Test Bench Files, on page 223 for more information.
Choose a Category Run Directory	Select Run Directory and specify the results directory to run the VCS simulation. See Run Directory, on page 222 to view the dialog box.
Choose a Category Post P&R Netlist	Select Post P&R Netlist and specify the post place-and-route netlist to run the VCS simulation. See Post P&R Netlist, on page 223 to view the dialog box.
Run	Runs VCS simulation.
View Script	View the script file with the specified VCS commands and options before generating it. For an example, see VCS Script File, on page 225.
Load From	Use this option to load an existing VCS script.
Save As	Generates the VCS script. The tool generates the XML script in the directory specified.
Restore Defaults	Restores all the default VCS settings.

Top Level Module

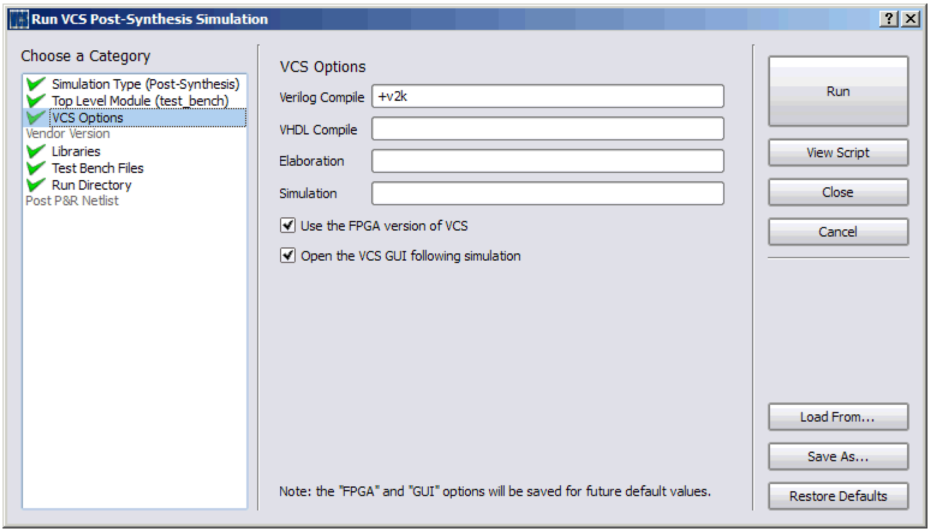
The following dialog box displays the Top Level Module task.



Top Level Module: test_bench

VCS Options

The following dialog box displays the VCS Options task.



VCS Options

Verilog Compile: +v2k

VHDL Compile:

Elaboration:

Simulation:

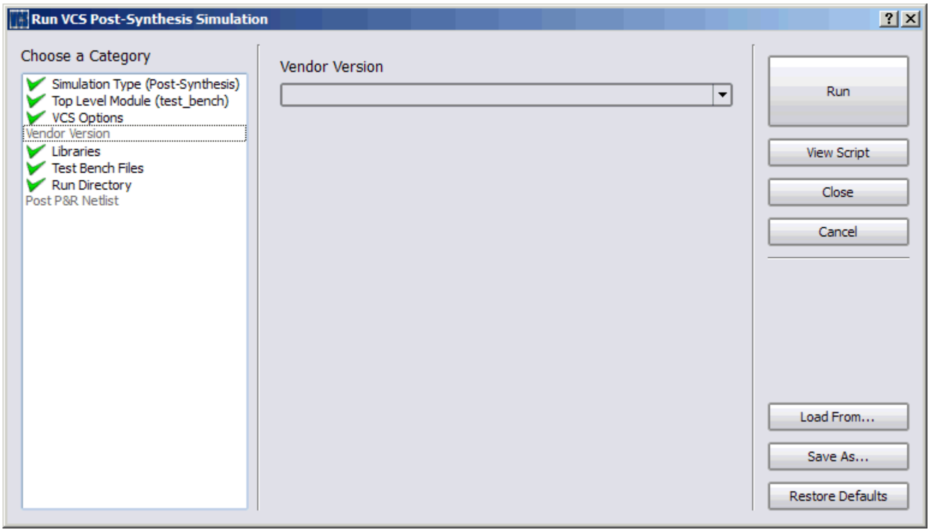
☑ Use the FPGA version of VCS

☑ Open the VCS GUI following simulation

Note: the "FPGA" and "GUI" options will be saved for future default values.

Vendor Version

The following dialog box displays the Vendor Versions task.



Run VCS Post-Synthesis Simulation

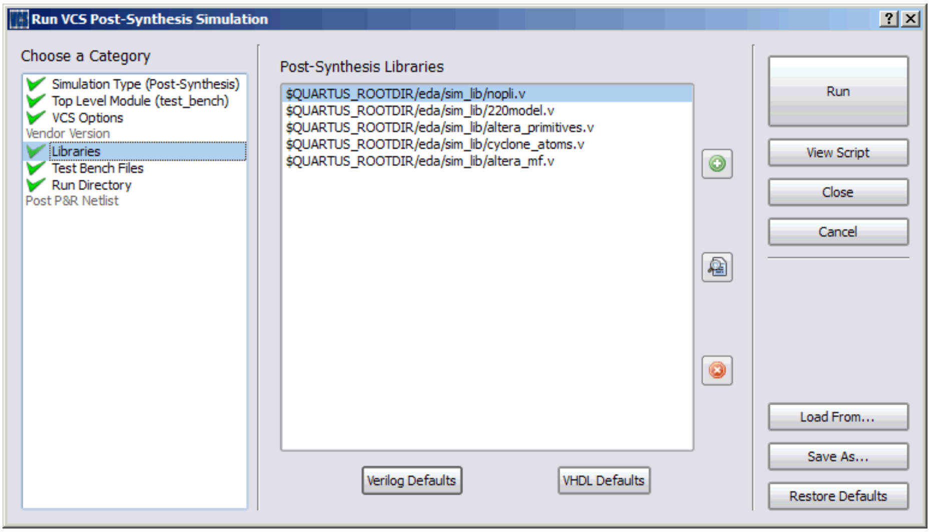
Choose a Category

- ✓ Simulation Type (Post-Synthesis)
- ✓ Top Level Module (test_bench)
- ✓ VCS Options
- Vendor Version
- ✓ Libraries
- ✓ Test Bench Files
- ✓ Run Directory
- Post P&R Netlist

Vendor Version

Libraries

The following dialog box displays the Libraries task.



Run VCS Post-Synthesis Simulation

Choose a Category

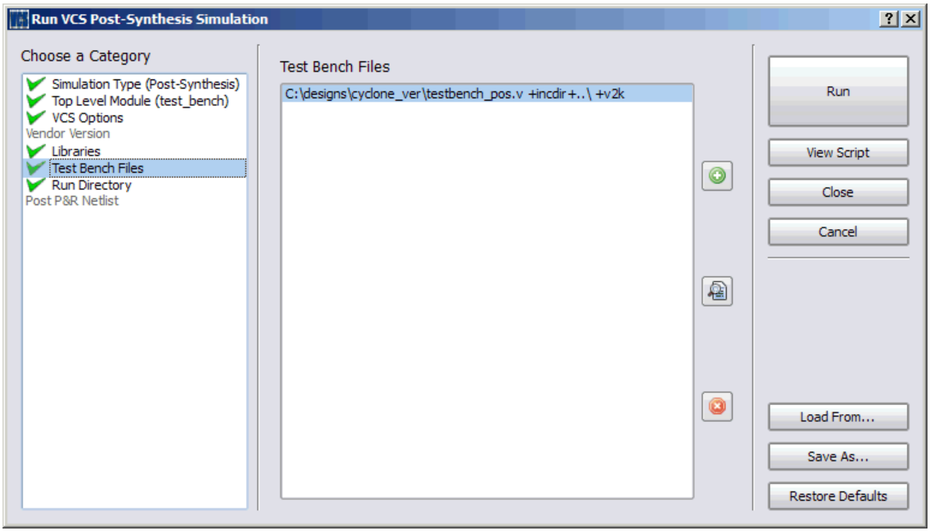
- ✓ Simulation Type (Post-Synthesis)
- ✓ Top Level Module (test_bench)
- ✓ VCS Options
- Vendor Version
- ✓ Libraries
- ✓ Test Bench Files
- ✓ Run Directory
- Post P&R Netlist

Post-Synthesis Libraries

```
$QUARTUS_ROOTDIR/eda/sim_lib/nopli.v
$QUARTUS_ROOTDIR/eda/sim_lib/220model.v
$QUARTUS_ROOTDIR/eda/sim_lib/altera_primitives.v
$QUARTUS_ROOTDIR/eda/sim_lib/cyclone_atoms.v
$QUARTUS_ROOTDIR/eda/sim_lib/altera_mf.v
```

Test Bench Files

The following dialog box displays the Test Bench Files task.

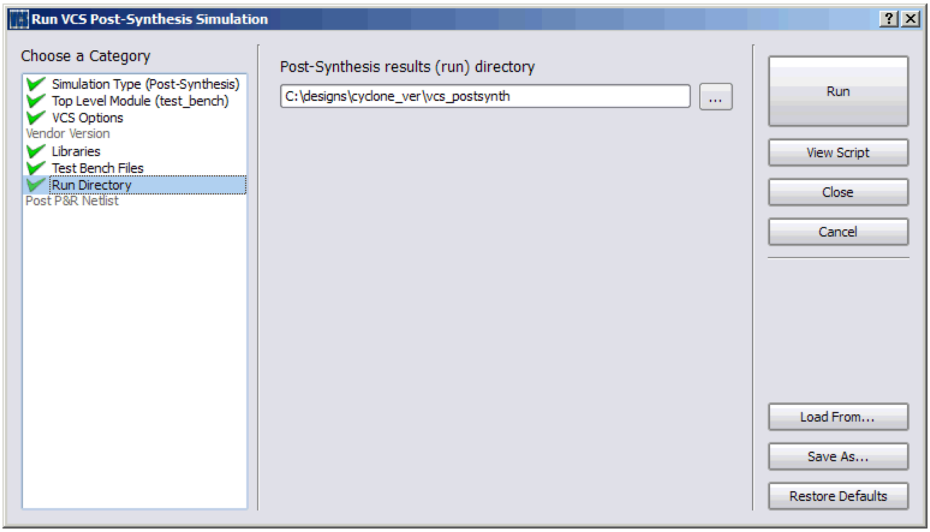


Test Bench Files

C:\designs\cyclone_ver\testbench_pos.v +incdir+..\ +v2k

Run Directory

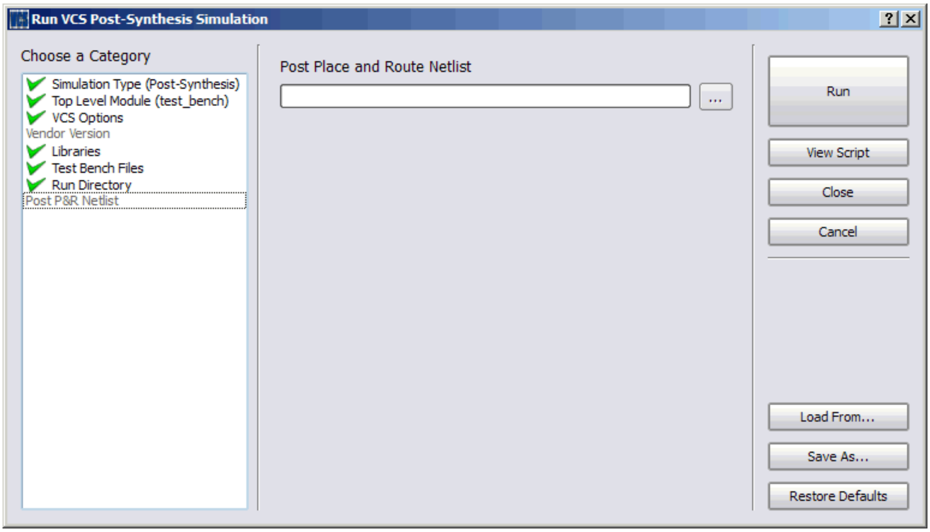
The following dialog box displays the Run Directory task.



Post-Synthesis results (run) directory: C:\designs\cyclone_ver\vcs_postsynth

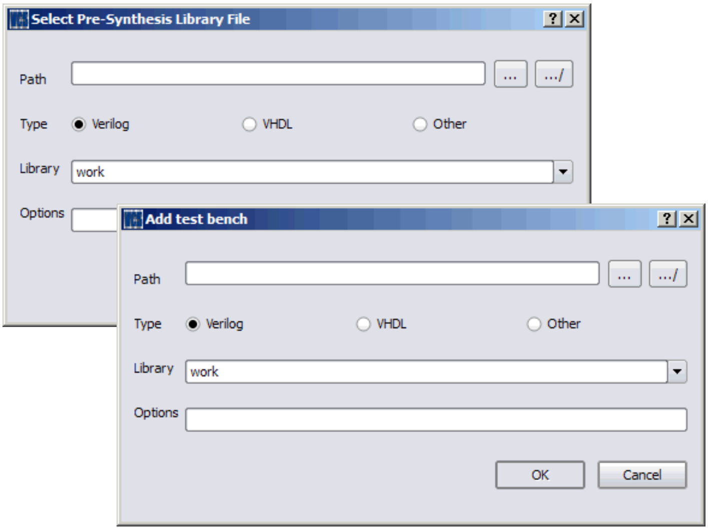
Post P&R Netlist

The following dialog box displays the Post P&R Netlist task.

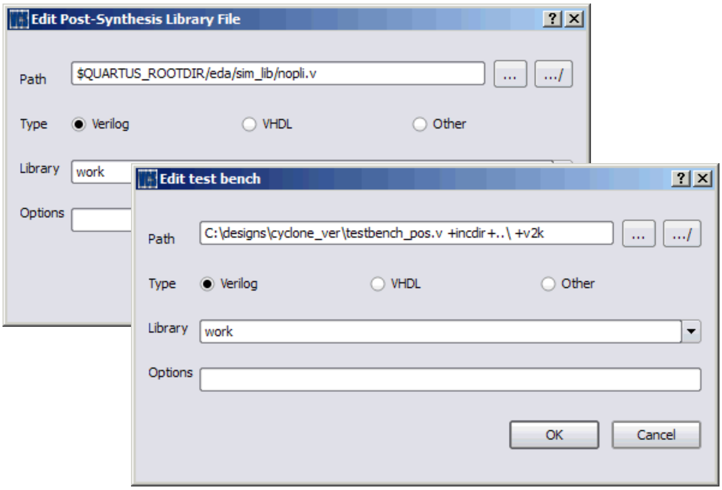


Changing Library and Test Bench Files

You can add Post-synthesis or Post place-and-route library files and test bench files before you launch the VCS simulator. For example, specify options on the following dialog box.



You can also edit library files and test bench files before you launch the VCS simulator. For example: specify options on the following dialog box.


Edit Post-Synthesis Library File

Path: $QUARTUS_ROOTDIR/eda/sim_lib/nopli.v

Type: ● Verilog ○ VHDL ○ Other

Library: work

Options:

Edit test bench

Path: C:\designs\cyclone_ver\testbench_pos.v +incdir+..\ +v2k

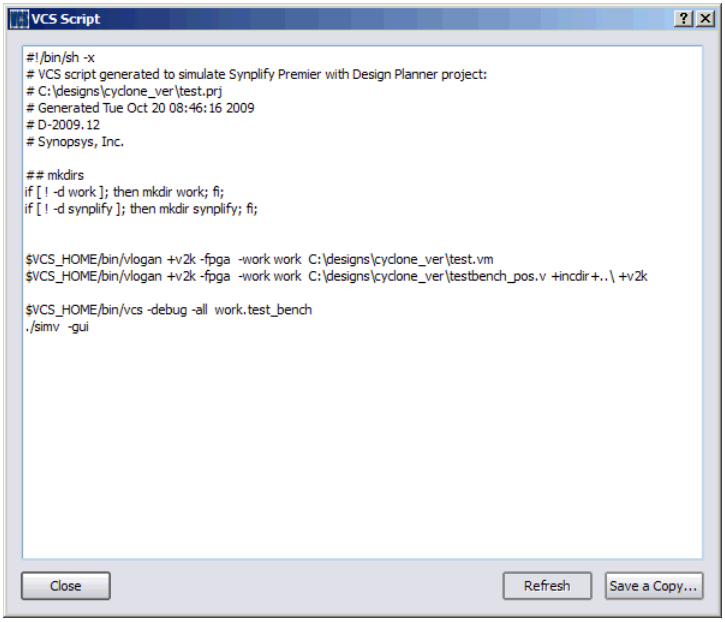
Type: ● Verilog ○ VHDL ○ Other

Library: work

Options:

VCS Script File

When you select the VCS Script button on the Run VCS Simulator dialog box, you can view the VCS script generated by the synthesis software for this VCS run. You can also save this VCS script to a file by clicking on Save a Copy.



```
#!/bin/sh -x
# VCS script generated to simulate Synplify Premier with Design Planner project:
# C:\designs\cyclone_ver\test.prj
# Generated Tue Oct 20 08:46:16 2009
# D-2009.12
# Synopsys, Inc.

## mkdirs
if [ ! -d work ]; then mkdir work; fi;
if [ ! -d synplify ]; then mkdir synplify; fi;


$VCS_HOME/bin/vlogan +v2k -fpga  -work work  C:\designs\cyclone_ver\test.vm
$VCS_HOME/bin/vlogan +v2k -fpga  -work work  C:\designs\cyclone_ver\testbench_pos.v +incdir+..\ +v2k

$VCS_HOME/bin/vcs -debug -all  work.test_bench
./simv  -gui
```

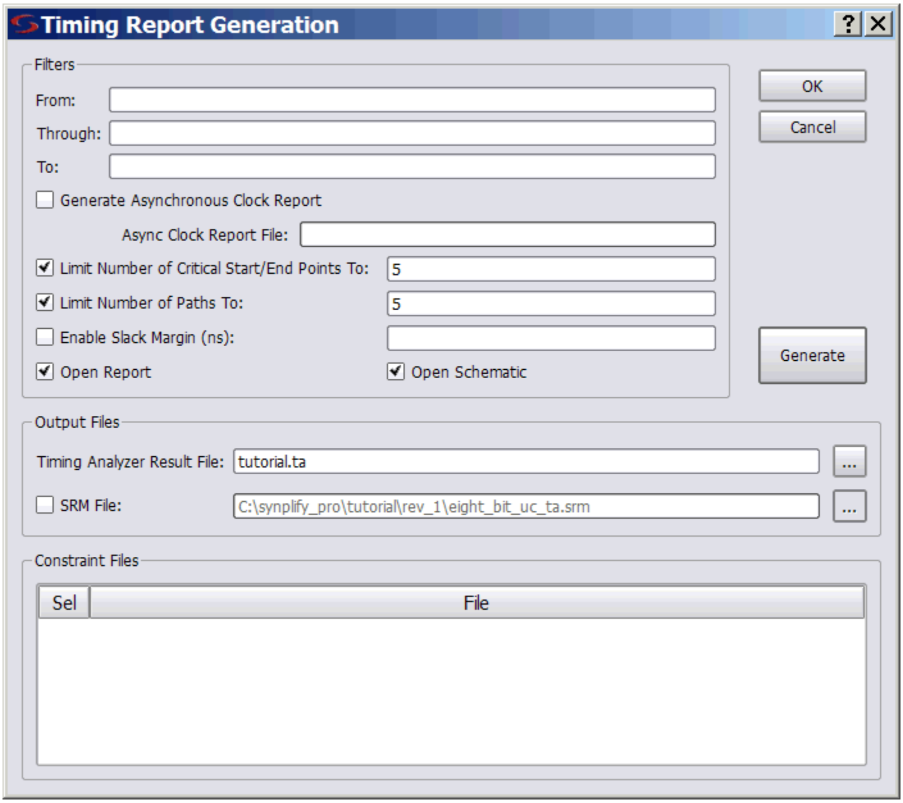
Analysis Menu

When you synthesize a design, a default timing report is automatically written to the log file (*projectName*.srr), located in the results directory. This report provides a clock summary, I/O timing summary, and detailed critical path information for the design. However, you can also generate a custom timing report that provides more information than the default report (specific paths or more than five paths) or one that provides timing based on additional analysis constraint files without rerunning synthesis.

Command	Description
Timing Analyst	Displays the Timing Report Generation dialog box to specify parameters for a stand-alone customized report. See Timing Report Generation Parameters, on page 227 for information on setting these options, and Analyzing Timing in Schematic Views, on page 324 in the *User Guide* for more information. If you click OK in the dialog box, the specified parameters are saved to a file. To run the report, click Generate. The report is created using your specified parameters.
Generate Timing	Generates and displays a report using the timing option parameters specified above. See the following: • Generating Custom Timing Reports with STA, on page 331 for specifics on how to run this report. • Timing Report Generation Parameters, on page 227 for information on setting parameters for the report. This includes information on filtering and options for running backannotation data and power consumption reports.

Timing Report Generation Parameters

You can use the Analysis->Timing Analyst command to specify parameters for a stand-alone timing report. See Timing Reports, on page 441 for information on the file contents.



Filters

From:

Through:

To:

□ Generate Asynchronous Clock Report

Async Clock Report File:

☑ Limit Number of Critical Start/End Points To: 5

☑ Limit Number of Paths To: 5

□ Enable Slack Margin (ns):

☑ Open Report ☑ Open Schematic

Output Files

Timing Analyzer Result File: tutorial.ta

□ SRM File: C:\synplify_pro\tutorial\rev_1\eight_bit_uc_ta.srm

Constraint Files

Sel	File

The following table provides brief descriptions of the parameters for running a stand-alone timing report.

Timing Report Option	Description
From or To	Specifies the starting (From) or ending (To) point of the path for one or more objects. It must be a timing start point (From) or end (To) point for each object. Use this option in combination with the others in the Filters section of the dialog box. See Combining Path Filters for the Timing Analyzer, on page 232 for examples of using filters.

Timing Report Option	Description
Through	Reports all paths through the specified point or list of objects. See for more information on using this filter. Use this option in combination with the others in the Filters section of the dialog box. See the following for additional information: • Timing Analyzer Through Points, on page 229 • Combining Path Filters for the Timing Analyzer, on page 232.
Generate Asynchronous Clock Report	Generates a report for paths that cross between clock groups. Generally paths in different clock groups are automatically handled as false paths. This option provides a file that contains information on each of the paths and can be viewed in a spreadsheet. This file is in the results directory (*projectName*_async_clk.rpt.csv). For details on the report, see Asynchronous Clock Report, on page 448.
Limit Number of Critical Start/End points	Specifies the maximum number of start/end paths to display for critical paths in the design. The default is 5. Use this option in combination with the others in the Filters section of the dialog box.
Limit Number of Paths to	Specifies the maximum number of paths to report. The default is 5. If you leave this field blank, all paths in the design are reported. Use this option in combination with the others in the Filters section of the dialog box.
Enable Slack Margin (ns)	Limits the report to paths within the specified distance of the critical path. Use this option in combination with the others in the Filters section of the dialog box.
Open Report	When enabled, clicking the Generate button opens the Text Editor on the generated custom timing report specified in the timing report file (ta).
Open Schematic	When enabled, clicking the Generate button opens a Technology view showing the netlist specified in the timing report netlist file (srm).

Timing Report Option	Description
Output Files	Displays the name of the generated report: • Async Clock Report File contains the spreadsheet data for the asynchronous clock report. This file is not automatically opened when report generation is complete. You can locate this file in the results directory. Default name is *projectName*_async_clk.rpt.csv (name cannot be changed). • Timing Analyst Results File is the standard timing report file, located in the Implementation Results directory. The file is also listed in the Project view. Default filename is *projectName*.ta. • SRM File updates the Technology view so that you can display the results of the timing updates in the HDL and Physical Analyst tools. The file is also listed in the Project view. For more details on any of these reports, see Timing Reports, on page 441.
Constraint Files	Enables analysis design constraint files (adc) to be used for stand-alone timing analysis only. See Input Files, on page 426 for information on this file.
Generate	Clicking this button generates the specified timing report file and timing view netlist file (srm) if requested, saves the current dialog box entries for subsequent use, then closes the dialog box.

Timing Analyzer Through Points

You can specify through points for nets (n:), hierarchical ports (t:), or instantiated cell pins (t:). You can specify the through points in two ways:

OR list

Enter the points as a space-separated list. The points are treated as an OR list and paths are reported if they crosses any of the points in the list. For example, when you type the following, the tool reports paths that pass through points b or c:

```
{n:b n:c}
```

See Filtering Points: OR List of Through Points, on page 230.

AND list

Enter the points in a product of sums (POS) format. The tool treats them as an AND list, and only reports the path if it passes through all the points in the list. The POS format for the timing report is the same as for timing constraints. The POS format is as follows:

```
{n:b n:c},{n:d n:e}
```

This constraint translates as follows:

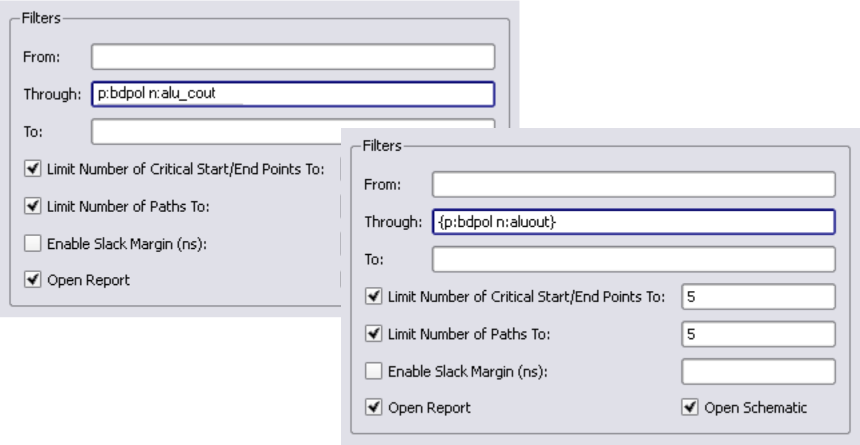
```
b AND d
OR b AND e
OR c AND d
OR c AND e
```

See Filtering Points: AND List of Through Points, on page 231.

See Defining From/To/Through Points for Timing Exceptions, on page 70 in the *User Guide* for more information about specifying through points.

Filtering Points: OR List of Through Points

This example reports the five worst paths through port bdpol or net aluout. You can enter the through points as a space-separated list (enclosing the list in braces is optional.)



Filters

From:

Through: p:bdpol n:alu_cout

To:

☑ Limit Number of Critical Start/End Points To:

☑ Limit Number of Paths To:

□ Enable Slack Margin (ns):

☑ Open Report

Filters

From:

Through: {p:bdpol n:aluout}

To:

☑ Limit Number of Critical Start/End Points To: 5

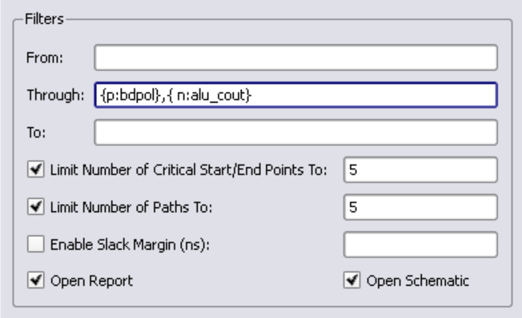
☑ Limit Number of Paths To: 5

□ Enable Slack Margin (ns):

☑ Open Report ☑ Open Schematic

Filtering Points: AND List of Through Points

This example reports the five worst paths passing through port bdpol and net aluout. Enclose each list in braces { } and separate the lists with a comma.



Filters

From:

Through: {p:bdpol},{ n:alu_cout}

To:

☑ Limit Number of Critical Start/End Points To: 5

☑ Limit Number of Paths To: 5

□ Enable Slack Margin (ns):

☑ Open Report ☑ Open Schematic

Combining Path Filters for the Timing Analyzer

This section describes how to use a combination of path filters to specify what you need and how to specify start and end points for path filtering.

Number and Slack Path Filters

The Limit Number of Paths To option specifies the maximum number of paths to report and the Enable Slack Margin option limits the report to output only paths that have a slack value that is within the specified value. When you use these two options together, the tighter constraint applies, so that the actual number of paths reported is the minimum of the option with the smallest value. For example, if you set the number of paths to report to 10 and the slack margin for 1 ns, if the design has only five paths within 1 ns of critical, then only five paths are reported (not the 10 worst paths). But if, for example, the design has 15 paths within a 1 ns of critical, only the first 10 are reported.

From/To/Through Filters

You can specify the from/to points for a path. You can also specify just a from point or just a to point. The from and to points are one or more hierarchical names that specify a port, register, pin on a register, or clock as object (clock alias). Ports and instances can have the same names, so prefix the name with p: for top-level port, i: for instance, or t: for hierarchical port or instance pin. However, the c: prefix for clocks is required for paths to be reported.

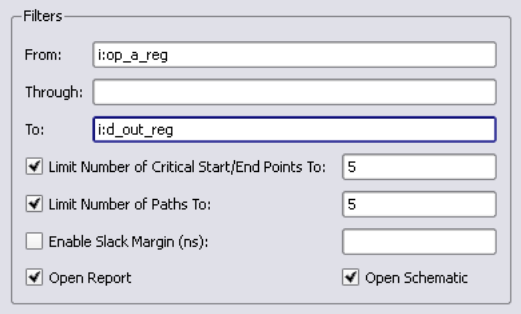
The timing analyst searches for the from/to objects in the following order: clock, port, bit port, cell (instance), net, and pin. Always use the prefix qualifier to ensure that all expected paths are reported. Remember that the timing analyst stops at the first occurrence of an object match. For buses, all possible paths from the specified start to end points are considered.

You can specify through points for nets, cell pins, or hierarchical ports.

You can simply type in from/to or through points. You can also cut-and-paste or drag-and-drop valid objects from the RTL or Technology views into the appropriate fields on the Timing Report Generation dialog box. Timing analysis requires that constraints use the Tech View name space. Therefore, it is recommended that you cut-and-paste or drag-and-drop objects from the Technology view rather than the RTL view.

This following examples show how to specify start, end or through point combinations for path filtering.

Filtering Points: Single Register to Single Register



Filters

From: i:op_a_reg

Through:

To: i:d_out_reg

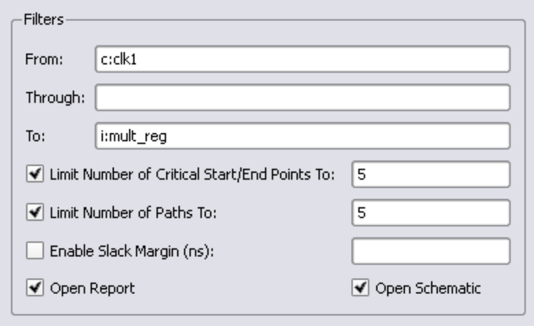
☑ Limit Number of Critical Start/End Points To: 5

☑ Limit Number of Paths To: 5

□ Enable Slack Margin (ns):

☑ Open Report ☑ Open Schematic

Filtering Points: Clock Object to Single Register



Filters

From: c:clk1

Through:

To: i:mult_reg

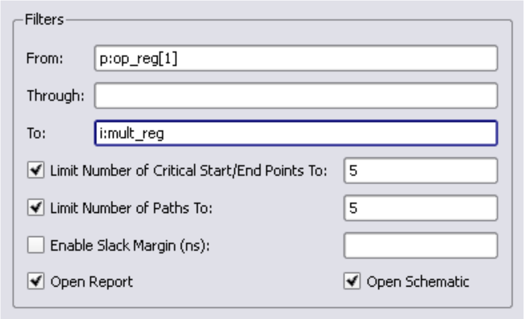
☑ Limit Number of Critical Start/End Points To: 5

☑ Limit Number of Paths To: 5

□ Enable Slack Margin (ns):

☑ Open Report ☑ Open Schematic

Filtering Points: Single Bit of a Bus to Single Register



Filters

From: p:op_reg[1]

Through:

To: i:mult_reg

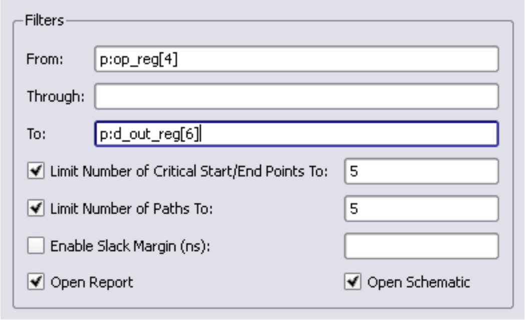
☑ Limit Number of Critical Start/End Points To: 5

☑ Limit Number of Paths To: 5

□ Enable Slack Margin (ns):

☑ Open Report ☑ Open Schematic

Filtering Points: Single Bit of a Bus to Single Bit of a Bus



Filters

From: p:op_reg[4]

Through:

To: p:d_out_reg[6]

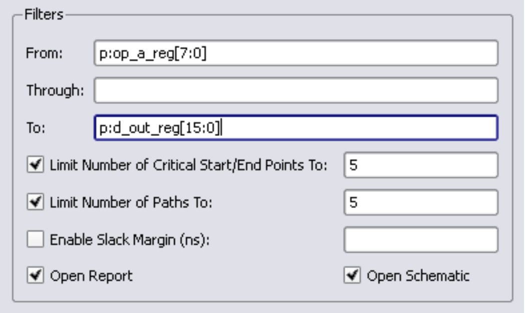
☑ Limit Number of Critical Start/End Points To: 5

☑ Limit Number of Paths To: 5

☐ Enable Slack Margin (ns):

☑ Open Report ☑ Open Schematic

Filtering Points: Multiple Bits of a Bus to Multiple Bits of a Bus



Filters

From: p:op_a_reg[7:0]

Through:

To: p:d_out_reg[15:0]

☑ Limit Number of Critical Start/End Points To: 5

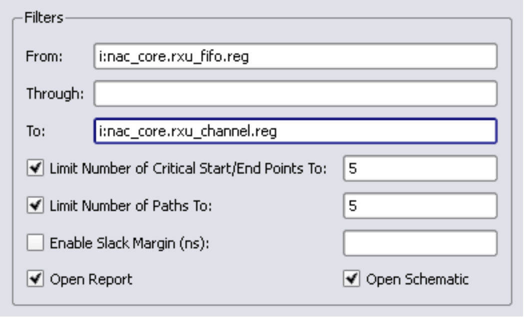
☑ Limit Number of Paths To: 5

☐ Enable Slack Margin (ns):

☑ Open Report ☑ Open Schematic

Filtering Points: With Hierarchy

This example reports the five worst paths for the net foo:



Filters

From: i:nac_core.rxu_fifo.reg

Through:

To: i:nac_core.rxu_channel.reg

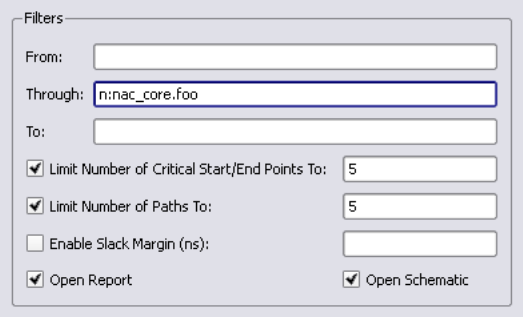
☑ Limit Number of Critical Start/End Points To: 5

☑ Limit Number of Paths To: 5

☐ Enable Slack Margin (ns):

☑ Open Report ☑ Open Schematic

Filtering Points: Through Point for a Net



Filters

From:

Through: n:nac_core.foo

To:

☑ Limit Number of Critical Start/End Points To: 5

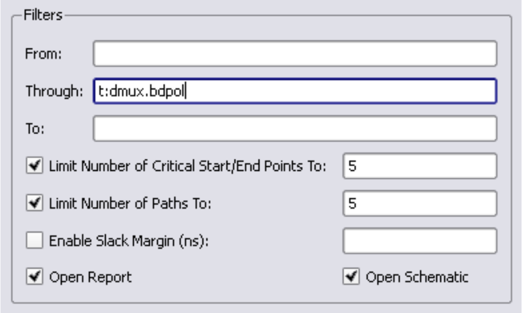
☑ Limit Number of Paths To: 5

□ Enable Slack Margin (ns):

☑ Open Report ☑ Open Schematic

Filtering Points: Through Point for a Hierarchical Port

This example reports the five worst paths for the hierarchical port bdpol:



Filters

From:

Through: t:dmux.bdpol

To:

☑ Limit Number of Critical Start/End Points To: 5

☑ Limit Number of Paths To: 5

□ Enable Slack Margin (ns):

☑ Open Report ☑ Open Schematic

Examples Using Wildcards

You can use the question mark (?) or asterisk (*) wildcard characters for object searching and name substitution. These characters work the same way in the synthesis tool environment as in the Linux environment.

The ? Wildcard

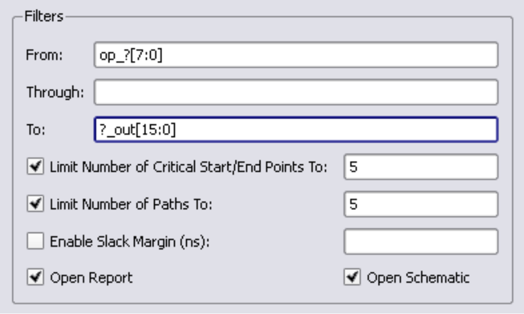
The ? matches single characters. If a design has buses op_a[7:0], op_b[7:0], and op_c[7:0], and you want to filter the paths starting at each of these buses, specify the start points as op_?[7:0]. See Example: ? Wildcard in the Name, on page 236 for another example.

The * Wildcard

The * matches a string of characters. In a design with buses op_a2[7:0], op_b2[7:0], and op_c2[7:0], where you want to filter the paths starting at each of these objects, specify the start points as op_*[*]. The report shows all paths beginning at each of these buses and for all of the bits of each bus. See Example: * Wildcard in the Name (With Hierarchy), on page 237 and Example: * Wildcard in the Bus Index, on page 237 for more examples.

Example: ? Wildcard in the Name

The ? is not supported in bus indices.



Filters

From: op_?[7:0]

Through:

To: ?_out[15:0]

☑ Limit Number of Critical Start/End Points To: 5

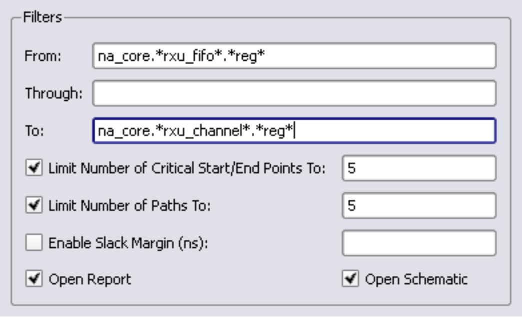
☑ Limit Number of Paths To: 5

□ Enable Slack Margin (ns):

☑ Open Report ☑ Open Schematic

Example: * Wildcard in the Name (With Hierarchy)

This example reports the five worst paths, starting at block rxu_fifo and ending at block rxu_channel within module nac_core. Each register in the design has the characters reg in the name.



Filters

From: na_core.*rxu_fifo*.*reg*

Through:

To: na_core.*rxu_channel*.*reg*

☑ Limit Number of Critical Start/End Points To: 5

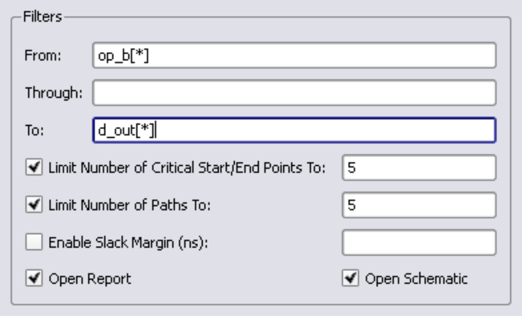
☑ Limit Number of Paths To: 5

☐ Enable Slack Margin (ns):

☑ Open Report ☑ Open Schematic

Example: * Wildcard in the Bus Index

This example reports the five worst paths, starting at op_b, and ending at d_out, taking into account all bits on these buses.



Filters

From: op_b[*]

Through:

To: d_out[*]

☑ Limit Number of Critical Start/End Points To: 5

☑ Limit Number of Paths To: 5

☐ Enable Slack Margin (ns):

☑ Open Report ☑ Open Schematic

HDL Analyst Menu

In the Project View, the HDL Analyst menu contains commands that provide project analysis in the following views:

- RTL View
- Technology View

This section describes the HDL Analyst menu commands for the RTL and Technology 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, such as filtered and unfiltered, transparent and opaque, see Filtered and Unfiltered Schematic Views, on page 308 and Transparent and Opaque Display of Hierarchical Instances, on page 314. For procedures on using the HDL Analyst tool, see Analyzing With the HDL Analyst Tool, on page 301 of the *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 View Submenus, on page 238
- HDL Analyst Menu: Hierarchical and Current Level Submenus, on page 239
- HDL Analyst Menu: Filtering and Flattening Commands, on page 241
- HDL Analyst Menu: Timing Commands, on page 245
- HDL Analyst Menu: Analysis Commands, on page 245
- HDL Analyst Menu: Selection Commands, on page 249
- HDL Analyst Menu: FSM Commands, on page 249

HDL Analyst Menu: RTL and Technology View Submenus

This table describes the commands that appear on the HDL Analyst->RTL and HDL Analyst->Technology submenus when the RTL or Technology View is active. For procedures on using these commands, see Analyzing With the HDL Analyst Tool, on page 301 of the *User Guide.*

HDL Analyst Command	Description
RTL->Hierarchical View	Opens a new, hierarchical RTL view. The schematic is unfiltered.
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 243 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 243 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 243 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 243 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 Analyzing With the HDL Analyst Tool, on page 301 of the *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 Analyzing With the HDL Analyst Tool, on page 301 of the *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 305 of the *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 243 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 243 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 312 of the *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 Analyzing With the HDL Analyst Tool, on page 301 of the *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.
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 Viewing Timing Information, on page 324 on the *User Guide*.

HDL Analyst Menu: Analysis Commands

This table describes the analysis commands on the HDL Analyst menu. For procedures on using the analysis commands, see Analyzing With the HDL Analyst Tool, on page 301 of the *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 248 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 263). For usage tips, see Usage Notes for Dissolving Instances, on page 248.
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 263). 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 312 of the *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 129) 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 312 of the *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	Displays the selected finite state machine in the FSM Viewer. Available only in an RTL view.
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.

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.

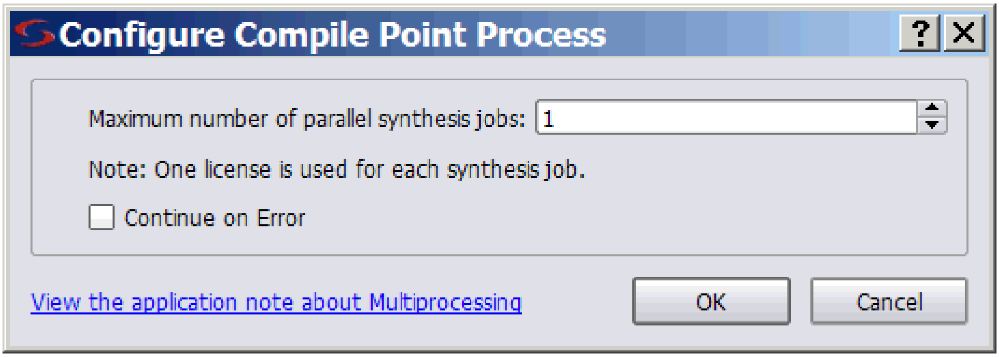
Command	Description
Basic Options Menu Commands for all Views	
Configure VHDL Compiler	Opens the Implementation Options dialog box where you can set the top-level entity and the encoding method for enumerated types. State-machine encoding is automatically determined by the FSM compiler or FSM explorer, or you can specify it explicitly using the syn_encoding attribute. See Implementation Options Command, on page 158 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 158.
Configure Compile Point Process	Lets you specify the maximum number of parallel synthesis jobs that can be run and how errors in compile points are treated. See Configure Compile Point Process Command, on page 251.
Toolbars	Lets you customize your toolbars.
Project View Options	Sets options for organizing files in the Project view. See Project View Options Command, on page 254.
Editor Options	Sets your Text Editor syntax coloring, font, and tabs. See Editor Options Command, on page 259.
P&R Environment Options	Displays the environmental variable options set for the place-and-route tool. See Place and Route Environment Options Command, on page 262.
HDL Analyst Options	Sets display preferences for HDL Analyst schematics (RTL and Technology views). See HDL Analyst Options Command, on page 263.

Command	Description
Configure External Programs	Lets you set browser and Acrobat Reader options on Linux platforms. See Configure External Programs Command, on page 268 for details.
Options Menu Commands Specifically for the Project View	
Configure Identify Launch	If Identify software is not properly installed, you might run into problems when you try to launch it from the synthesis tools. Use the Configure Identify Launch dialog box to help you resolve these issues. For guidelines to follow, see Handling Problems with Launching Identify, on page 359 in the *User Guide*.

Configure Compile Point Process Command

Use the Configure Compile Point Process command to let you run multiprocessing with compile points. This option allows the synthesis software to run multiple, independent compile point jobs simultaneously, providing additional runtime improvements for the compile point synthesis flow.

This feature is supported on Windows and Linux for certain technologies only. This command is greyed out for technologies that are not supported.



Configure Compile Point Process

Maximum number of parallel synthesis jobs: 1

Note: One license is used for each synthesis job.

☐ Continue on Error

Field/Option	Description
Maximum Number of Parallel Synthesis Jobs	Sets the maximum number of synthesis jobs that can run in parallel. It displays the current value from the ini file, and allows you to reset it. Use this option for multiprocessing by running compile point jobs in parallel. Set a value based on the number of available licenses. Note that one license is used for each job. See License Utilization for Multiprocessing, on page 253 for details. When you set this option, it resets the MaxParallelJobs value in the .ini file. See Maximum Parallel Jobs, on page 253 for other ways to specify this value.
Continue on Error	Allows the software to continue on error and synthesize the rest of the design, even when there might be problems with a portion of the design. The Continue on Error mode automatically enables the MultiProcessing option to run with compile points using one license; this is the default. For additional runtime improvements, you can specify multiple synthesis jobs that run in parallel. See Chapter 17, *Using Multiprocessing* for details. For more information about Continue on Error mode, see Continue on Error Mode, on page 431 in the *User Guide*.

Maximum Parallel Jobs

There are three ways to specify the maximum number of parallel jobs:

ini File	Set this variable in the MaxParallelJobs variable in the product ini file:
	`[JobSetting]`
	`MaxParallelJobs=<n>`
	This value is used by the UI as well as in batch mode, and is effective until you specify a new value. You can change it with the Options->Configure Compile Point Process command.

Tcl Variable	Set the following variable in a Tcl file, the project files, or from the Tcl window: `set_option -max_parallel_jobs=<n>` This is a global option that is applied to all project files and their implementations. This value takes effect immediately. If you set it in the Tcl file or project file, it remains in effect until you specify a new value. If you set it from the Tcl window, the max_parallel_jobs value is only effective for the session and will be lost when you exit the application.
Configure Compile Point Process Command	The Maximum Number of Parallel Synthesis Jobs option displays the current ini file value and allows you to reset it.

License Utilization for Multiprocessing

When you decide to run parallel synthesis jobs, a license is used for each compile point job that runs. For example, if you set the Maximum number of parallel synthesis jobs to 4, then the synthesis tool consumes one license and three additional licenses are utilized to run the parallel jobs if they are available for your computing environment. Licenses are released as jobs complete, and then consumed by new jobs which need to run.

The actual number of licenses utilized depends on the following:

1. Synthesis software scheme for the compile point requirements used to determine the maximum number of parallel jobs or licenses a particular design tries to use.

2. Value set on the Configure Compile Point Process dialog box.

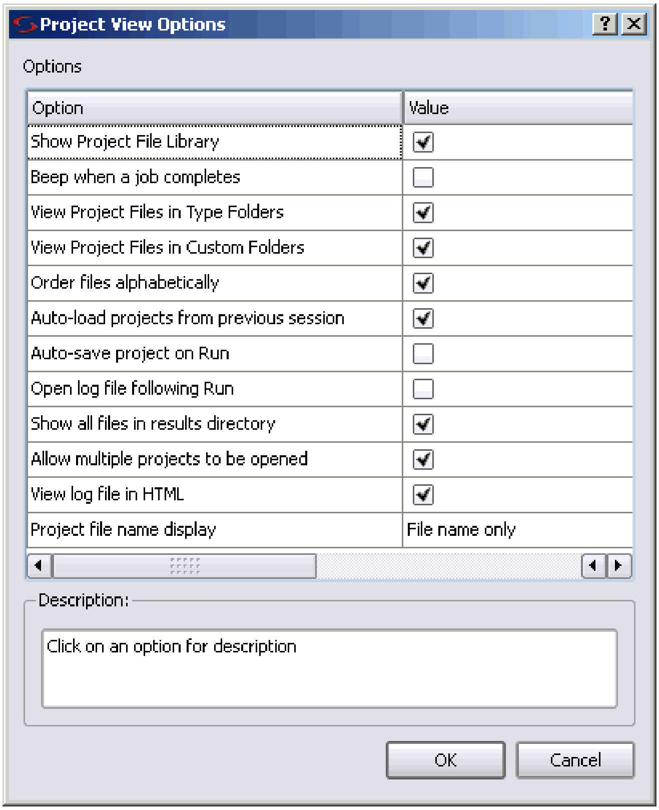
3. Number of licenses actually available. You can use Help->Preferred License Selection to check the number of available license. If you need to increase

the number of available licenses, you can specify multiple license types. For more information, see Specifying License Types, on page 470.

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

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.



Option	Value
Show Project File Library	☑
Beep when a job completes	□
View Project Files in Type Folders	☑
View Project Files in Custom Folders	☑
Order files alphabetically	☑
Auto-load projects from previous session	☑
Auto-save project on Run	□
Open log file following Run	□
Show all files in results directory	☑
Allow multiple projects to be opened	☑
View log file in HTML	☑
Project file name display	File name only

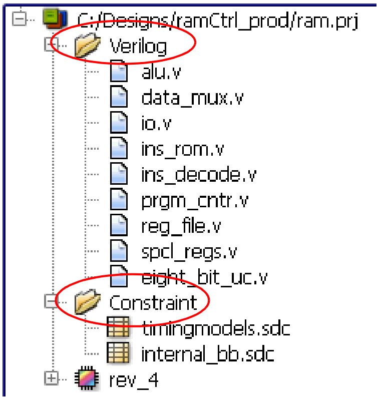
Click on an option for description

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 148, 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 Type Folders	When enabled, organizes project files into separate folders by type. See View Project Files in Type Folders Option, on page 256 and add_file, on page 1036.
View Project Files in Custom Folders	When enabled, allows you to view files contained within the custom folders created for the project. See View Project Files in Custom Folders Option, on page 257.
Order files alphabetically	When enabled, the software orders the files within folders alphabetically instead of in project order. You can also use the Sort Files option in the Project view.
Autoload projects from previous session	Enable/Disable automatically loading projects from the previous session. Otherwise, projects will not be loaded automatically. This option is enabled by default. See Loading Projects With the Run Command, on page 257.
Auto-save project on Run	Enable/Disable automatically saving projects when the Run button is selected. See Automatically Save Project on Run, on page 258.
Open Log file following Run	Enable/Disable automatically opening and displaying log file after a synthesis run.
Show all files in results directory	When enabled, shows all files in the Implementation Results view. When disabled, the results directory shows only files generated by the synthesis tool itself.

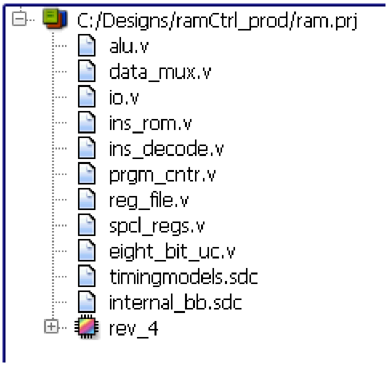
Field/Option	Description
Allow multiple projects to be opened	When enabled, multiple projects are displayed at the same time. See Allow Multiple Projects to be Opened Option, on page 257.
View log file in HTML	Enable/Disable viewing of log file report in HTML format versus text format. See Log File, on page 435.
Project file name display	From the drop-down menu, select one the following ways to display project files: • File name only • Relative file path • Full file path
Use links in SRR log file to individual job logs	Determines if individual job logs use links in the srr log file. You can select: • off—appends individual job logs to the srr log file. • on—always link to individual job logs. • if_up_to_date—only link to individual job logs if the module is up-to-date.

View Project Files in Type Folders Option



```
C:/Designs/ramCtrl_prod/ram.prj
  Verilog
    alu.v
    data_mux.v
    io.v
    ins_rom.v
    ins_decode.v
    prgm_cntr.v
    reg_file.v
    spcl_regs.v
    eight_bit_uc.v
  Constraint
    timingmodels.sdc
    internal_bb.sdc
  rev_4
```

View project files in type folders *enabled*

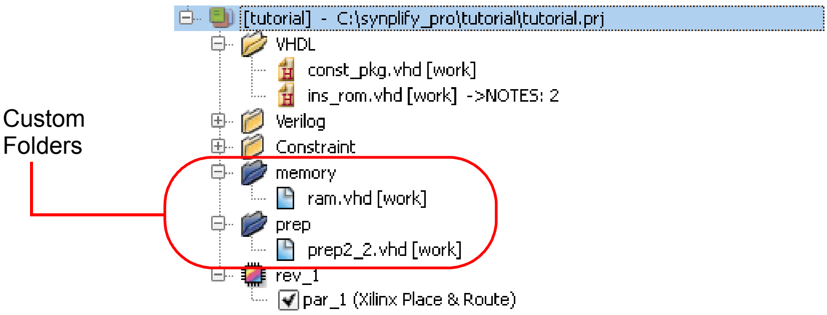


```
C:/Designs/ramCtrl_prod/ram.prj
  alu.v
  data_mux.v
  io.v
  ins_rom.v
  ins_decode.v
  prgm_cntr.v
  reg_file.v
  spcl_regs.v
  eight_bit_uc.v
  timingmodels.sdc
  internal_bb.sdc
  rev_4
```

View project files in type folders *disabled*

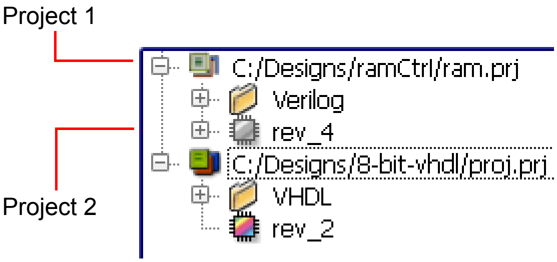
View Project Files in Custom Folders Option

Selecting this option enables you to view user-defined custom folders that contain a predefined subset of project files in various hierarchy groupings or organizational structures. Custom folders are distinguished by their blue color. For information on creating custom folders, see Creating Custom Folders, on page 120 in the *User Guide*.


[tutorial] - C:\synplify_pro\tutorial\tutorial.prj
VHDL
const_pkg.vhd [work]
ins_rom.vhd [work] ->NOTES: 2
Custom
Folders
Verilog
Constraint
memory
ram.vhd [work]
prep
prep2_2.vhd [work]
rev_1
par_1 (Xilinx Place & Route)

Allow Multiple Projects to be Opened Option

The following figure shows multiple projects open.



Loading Projects With the Run Command

When you load a project that includes the project -run command, a dialog box appears in the Project view with the following message:

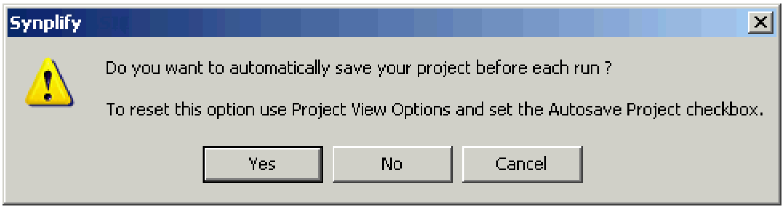
```
Project run command encountered during project load. Are you sure
you want to run?
```

You can reply with either yes or no.

Automatically Save Project on Run

If you have modified your project on the disk directory since being loaded into the Project view and you run your design, a message is generated that infers the UI is out-of-date.

The following dialog box appears with a message to which you must reply.



Do you want to automatically save your project before each run ?

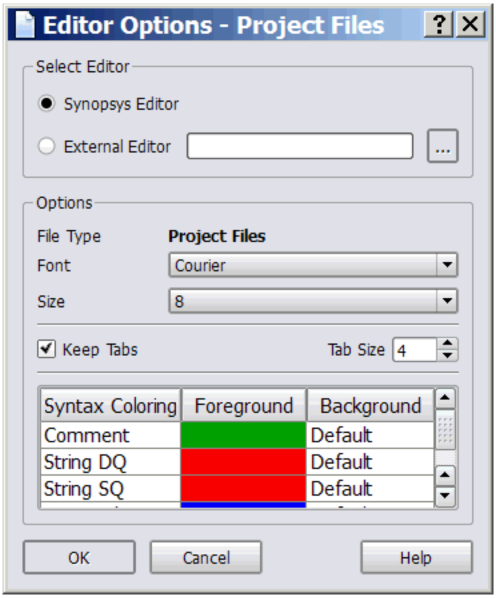
To reset this option use Project View Options and set the Autosave Project checkbox.

You can specify one of the following:

- Yes — The Auto-save project on Run switch on the Project View Options dialog box is automatically enabled, and then your design is run.
- No — The Auto-save project on Run switch on the Project View Options dialog box is not enabled, but your design is run.
- Cancel — Closes this message dialog box and does not run your design.

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 text editor.



Editor Options - Project Files

Select Editor

● Synopsys Editor

○ External Editor

Options

File Type Project Files

Font Courier

Size 8

☑ Keep Tabs

Tab Size 4

Syntax Coloring	Foreground	Background
Comment	🟩	Default
String DQ	🟥	Default
String SQ	🟥	Default

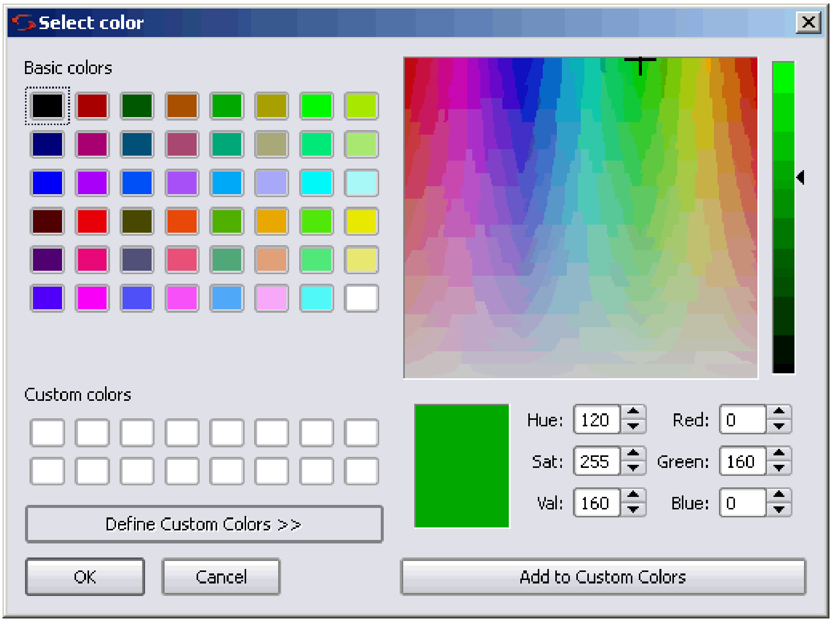
The following table describes the Editor Options dialog box features.

Feature	Description
Select Editor	Select an internal or external editor.
• Synopsys Editor	Sets the Synopsys text editor as the default text editor.
• External Editor	Uses the specified external text editor program to view text files from within the Synopsys FPGA tool. The executable specified must open its own window for text editing. See Using an External Text Editor, on page 38 of the *User Guide* for a procedure. *Note:* Files opened with an external editor *cannot* be crossprobed.
Options	Set text editing preferences.
• File Type	You can define text editor preferences for the following file types: project files, HDL files, log files, constraint files, and default files.

Feature	Description
• Font	Lets you define fonts to use with the text editor.
• Font Size	Lets you define font size to use with the text editor.
• Keep Tabs • Tab Size	Lets you define whether to use tab settings with the text editor.
• Syntax Coloring	Lets you define foreground or background syntax coloring to use with the text editor. See Color Options, on page 260.

Color Options

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



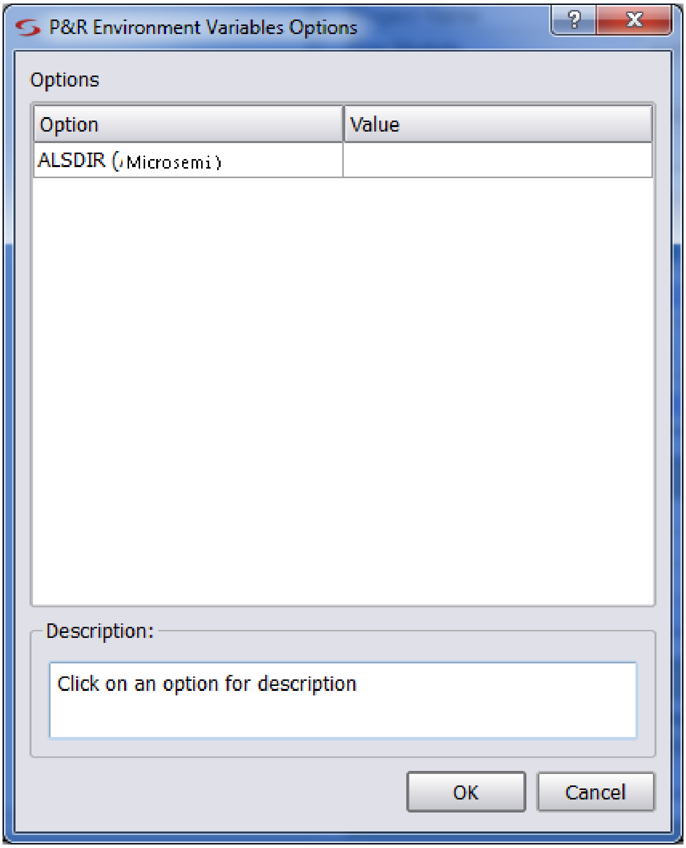
Hue: 120 Sat: 255 Val: 160
Red: 0 Green: 160 Blue: 0

You can set syntax colors for some common syntax options listed in the following table.

Syntax	Description
Comment	Comment strings contained in all file types.
Error	Error messages contained in the log file.
Gates	Gates contained in HDL source files.
Info	Informational messages contained in the log file.
Keywords	Generic keywords contained in the project, HDL source, constraint, and log files.
Line Comment	Line comments contained in the HDL source, C, C++, and log files.
Note	Notes contained in the log file.
FDCKeyword	Constraint-specific keywords contained in the .fdc file.
Strength	Strength values contained in HDL source files.
String DQ	String values within double quotes contained in the project, HDL source, constraint, C, C++, and log files.
String SQ	String values within single quotes contained in the project, HDL source, constraint, C, C++, and log files.
SVKeyword	SystemVerilog keywords contained in the Verilog file.
Types	Type values contained in HDL source files.
Warning	Warning messages contained in the log file.

Place and Route Environment Options Command

Select Options->P&R Environment Options to display the environmental variable options set for the place-and-route tool. Allows you to change the specified location of the selected place-and-route tool and is used during run to locate this version of the P&R tool.



Option	Value
ALSDIR (Microsemi)	

Click on an option for description

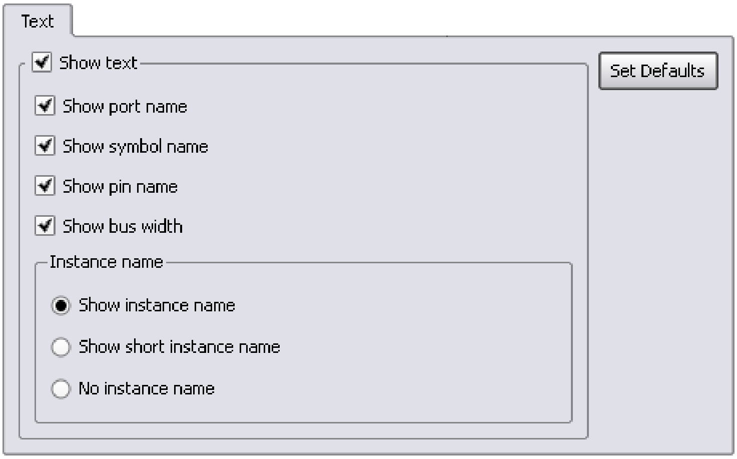
HDL Analyst Options Command

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 269 in the *User Guide*.

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

- Text Panel, on page 263
- General Panel, on page 264
- Sheet Size Panel, on page 266
- Visual Properties Panel, on page 268

Text Panel



☑ Show text

☑ Show port name

☑ Show symbol name

☑ Show pin name

☑ Show bus width

Instance name

● Show instance name

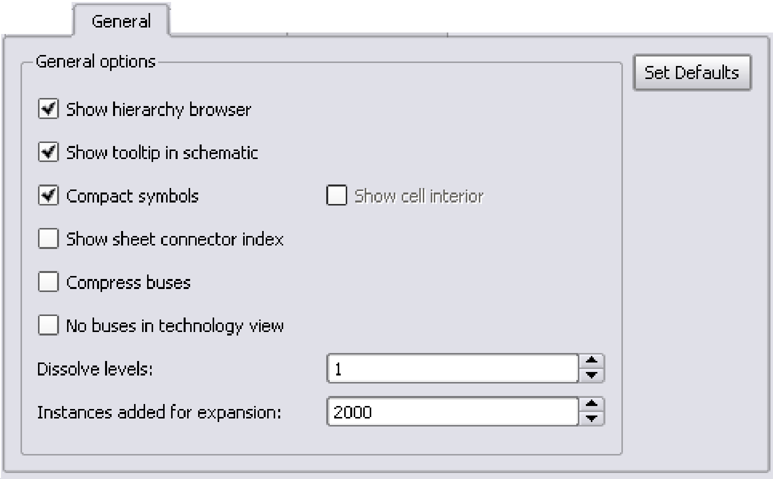
○ Show short instance name

○ 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 • Show 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

☑ Show tooltip in schematic

☑ Compact symbols ☐ Show cell interior

☐ Show sheet connector index

☐ Compress buses

☐ No buses in technology view

Dissolve levels: 1

Instances added for expansion: 2000

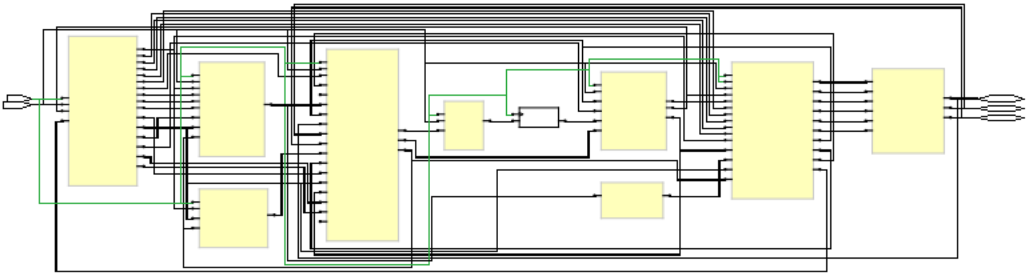
Set Defaults

The following options are in the General panel.

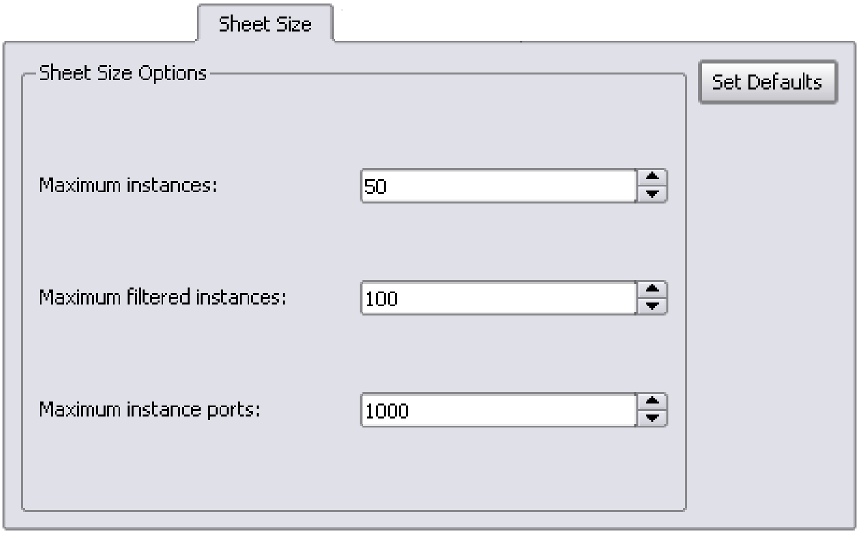
Field/Option	Description
Show hierarchy browser	When enabled, a hierarchy browser is present as the left pane of RTL and Technology views.
Show tooltip in schematic	When enabled, displays tooltips that hover objects as you move over them in the RTL and Technology schematic views.
Compact symbols	When 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 311.
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 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).
Display color-coded clock nets	Displays clock nets in the HDL Analyst View with the color green.
Dissolve levels	The number of levels to dissolve, during HDL Analyst->Dissolve Instances. See Dissolve Instances, on page 247
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.

Color-coded Clock Nets

Clock nets are displayed with the color green in the RTL and Technology views.



Sheet Size Panel



Maximum instances: 50

Maximum filtered instances: 100

Maximum instance ports: 1000

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 326.

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 Maximum filtered instances value, increasing its effective value. This means that you can see more child instances than Maximum filtered instances itself implies.

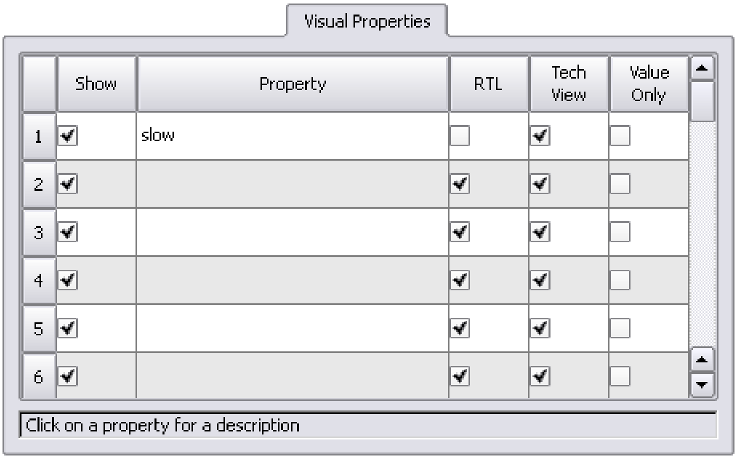
The Maximum filtered instances value must be at least the Maximum instances value. See Multiple-sheet Schematics, on page 326.

Maximum Instance Ports

Defines the maximum number of instance pins to display on a schematic sheet.

Visual Properties Panel

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 260 in the *User Guide*.



Visual Properties

	Show	Property	RTL	Tech View	Value Only
1	☑	slow	□	☑	□
2	☑		☑	☑	□
3	☑		☑	☑	□
4	☑		☑	☑	□
5	☑		☑	☑	□
6	☑		☑	☑	□

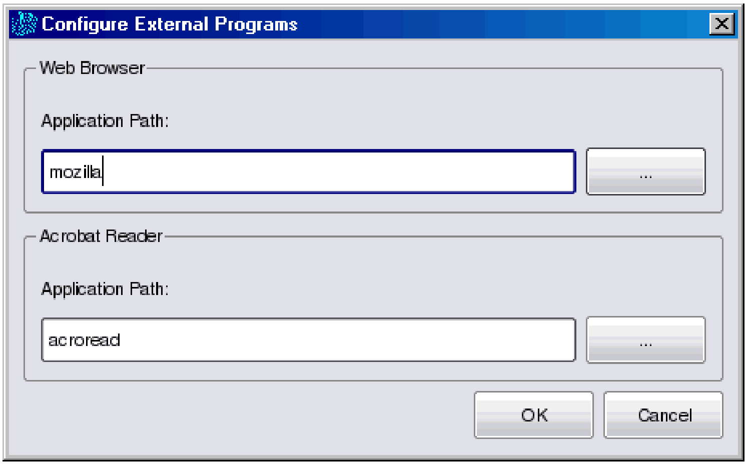
Click on a property for a description

The following options are in the Visual Properties panel.

Show	Toggles 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 Linux platforms only. It lets you specify the web browser and PDF reader for accessing Synopsys support (see Web Menu, on page 275 for details) and online documents.



Web Browser Application Path: mozilla

Acrobat Reader Application Path: 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.

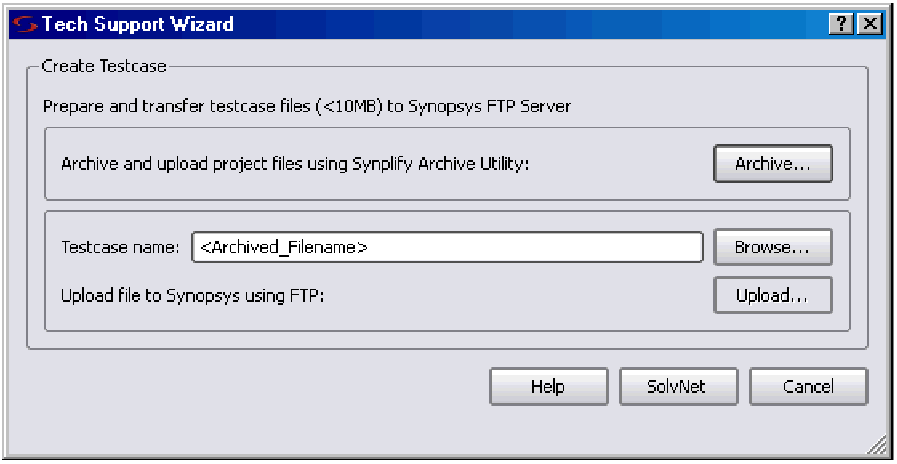
Tech-Support Menu

The Tech-Support menu contains information and the actions you can take when you encounter problems running your designs or working with the Synopsys FPGA Implementation products.

Command	Description
Submit Support Request	Opens the Technical Support wizard, which allows you to submit online support requests via SolvNet. The wizard includes provisions for attaching a testcase. See Submit Support Request Command, on page 270 for more information.
Web Support	Opens the Synopsys SolvNet Support page from where you can: • Log on to SolvNet to request Synopsys technical support. • Access the Synopsys Products, Downloads, Training, and Documentation pages that have links to product information. See Web Support Command, on page 273 for more information.

Submit Support Request Command

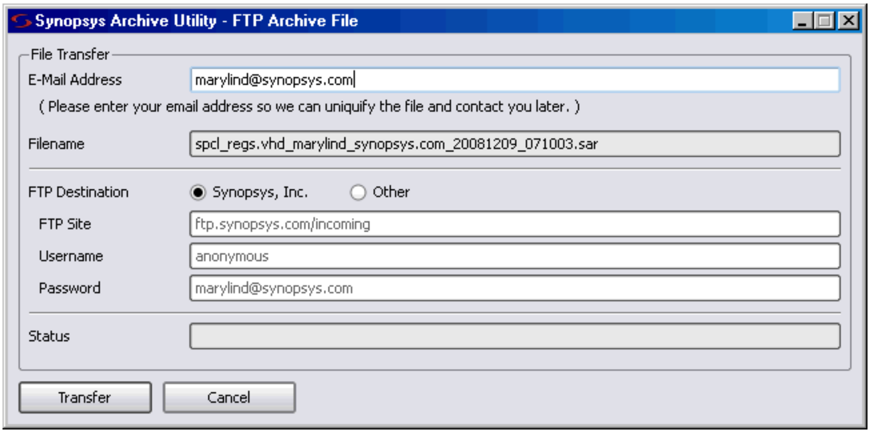
To open a request for Synopsys Technical Support, select Submit Support Request from the Tech-Support menu. This command brings up the web-based technical support wizard that helps you prepare the information required to provide technical support for your request through SolvNet.


Tech Support Wizard
Create Testcase
Prepare and transfer testcase files (<10MB) to Synopsys FTP Server
Archive and upload project files using Synplify Archive Utility:
Archive...
Testcase name:
<Archived_Filename>
Browse...
Upload file to Synopsys using FTP:
Upload...
Help
SolvNet
Cancel

Command	Description
Archive	Brings up the Synopsys Archive Utility to create an archive of your design. Note that designs are limited to 10 MBytes.
Testcase name	The name of the testcase or file to be transferred. To transfer multiple files, use the Synopsys Archive utility to create a single .sar file for the transfer.
Upload	Displays the FTP Archive File form to initiate the transfer of the testcase to the FTP file server. See FTP Archive File Form, on page 272.
SolvNet	Displays the Synopsys Sign In screen to access protected Synopsys applications. Signing in opens the SolvNet application which allows you to submit an online support request.

FTP Archive File Form

The FTP Archive File form is displayed when transferring a testcase or file to an FTP file server.



Synopsys Archive Utility - FTP Archive File

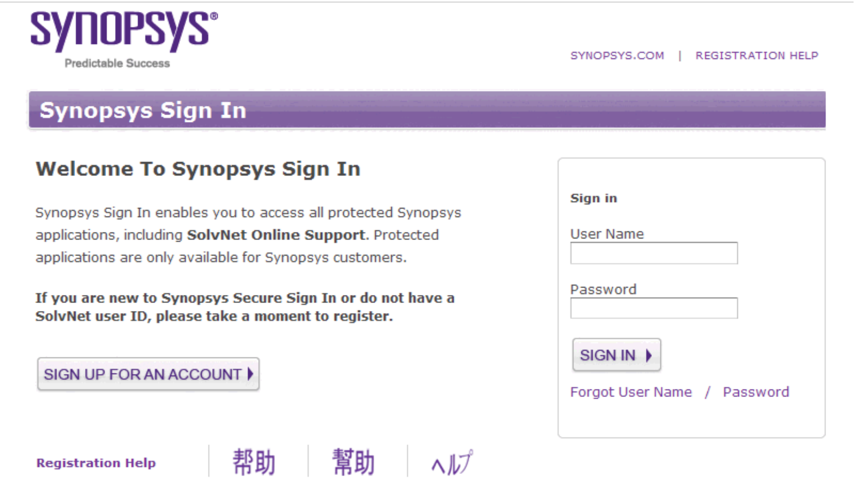
File Transfer

E-Mail Address	marylind@synopsys.com
(Please enter your email address so we can uniquify the file and contact you later.)	
Filename	spcl_regs.vhd_marylind_synopsys.com_20081209_071003.sar
FTP Destination	● Synopsys, Inc. ○ Other
FTP Site	ftp.synopsys.com/incoming
Username	anonymous
Password	marylind@synopsys.com
Status	

Command	Description
E-Mail Address	Your e-mail address; the name entered is used to make the FTP filename unique and is automatically entered into the Password field when using the Synopsys site.
Filename	A read-only field displaying the name of the testcase or file to be transferred. A .sar extension is added for single files (archive files are automatically assigned a .sar extension), and the filename is made unique with the e-mail address.
FTP Destination	Radio buttons that select either the Synopsys site or an alternate site. When Other is specified, you must supply FTP Site, Username, and Password entries.
Status	Reports the status of the transfer.
Transfer	Initiates the FTP file transfer; the results of the transfer are displayed in the Status field.

Web Support Command

Through the Synopsys SolvNet Support page, you can access Products, Downloads, Training, and Documentation pages on the Synopsys website as well as submit requests for technical support through SolvNet. First sign in to SolvNet.



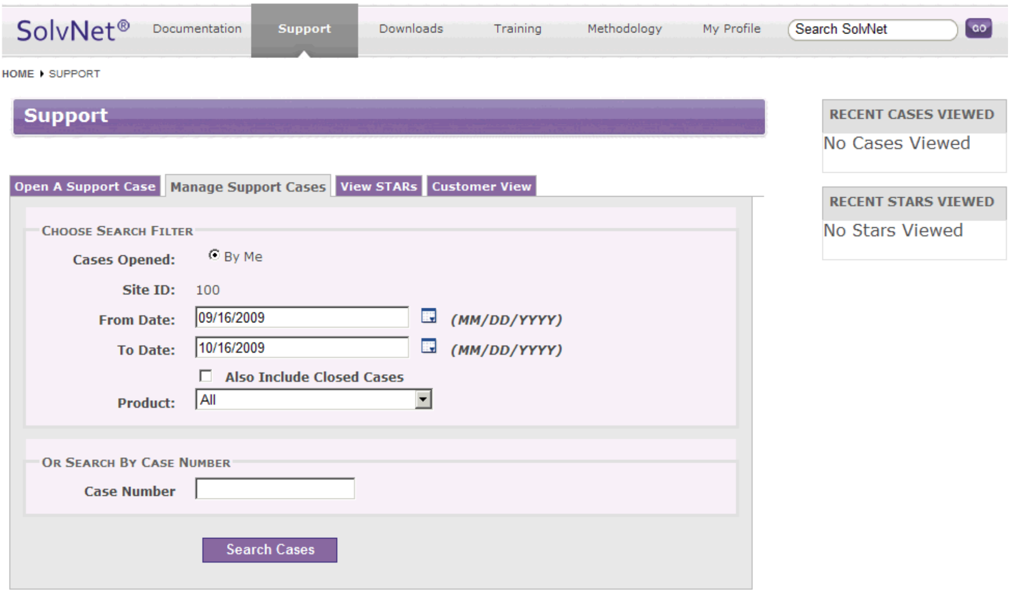
Synopsys Sign In

Welcome To Synopsys Sign In

Synopsys Sign In enables you to access all protected Synopsys applications, including **SolvNet Online Support**. Protected applications are only available for Synopsys customers.

If you are new to Synopsys Secure Sign In or do not have a SolvNet user ID, please take a moment to register.

Once you logon, the SolvNet Support page is displayed.



RECENT CASES VIEWED

No Cases Viewed

RECENT STARS VIEWED

No Stars Viewed

CHOOSE SEARCH FILTER

Cases Opened: ● By Me

Site ID: 100

From Date: 09/16/2009 (MM/DD/YYYY)

To Date: 10/16/2009 (MM/DD/YYYY)

□ Also Include Closed Cases

Product: All

OR SEARCH BY CASE NUMBER

Case Number

Web Menu

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

Command	Description
Go to SolvNet	Opens the home page for the Synopsys SolvNet Search support. This is a website that contains links to useful technical information; search for new or updated articles or documentation, like application notes, white papers, release notes, and other user-oriented documentation.
✉ Check Resource Center Messages	Opens a web page that contains updated 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 Check Resource Center Messages Command, on page 276 for additional information.
Configure Resource Center	Lets you set options for Technical Resource Center (TRC) updates. See Configure Resource Center Command, on page 278 for details.
Go to Training Center	Opens the Synopsys training web page for Synopsys products. Synopsys offers both online web-based training courses, as well as, classroom training courses taught by Synopsys personnel. Scroll down this page for Synplicity FPGA Implementation courses.
Synopsys Home	Opens the Synopsys home web page for Synopsys products.
FPGA Implementation Tools	Opens the Synopsys FPGA design solution web page for Synopsys FPGA products. You can find information about the full line of Synopsys FPGA Implementation products here.

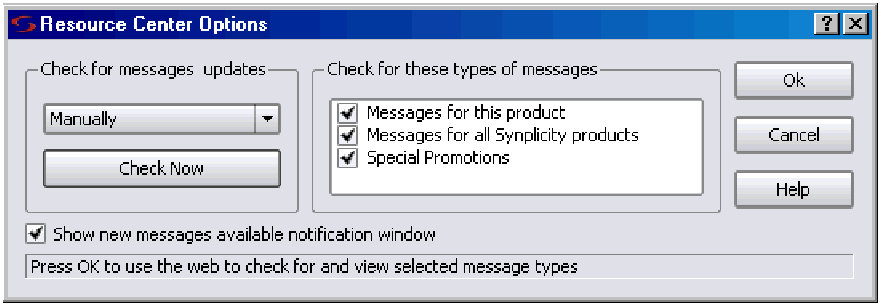
Check Resource Center Messages Command

This command lets you set options for messages from the Resource Center. SeUsing the Resource Center, on page 276 and Resource Center Messages, on page 277 for details.

Using the Resource Center

The following procedure explains how to set preferences and check the Technical Resource Center (TRC) for information.

1. To go to the TRC, select Web->Check Resource Center Messages.
2. To set preferences for accessing the Resource Center, select Web->Configure Resource Center and do the following:



Resource Center Options

Check for messages updates: Manually

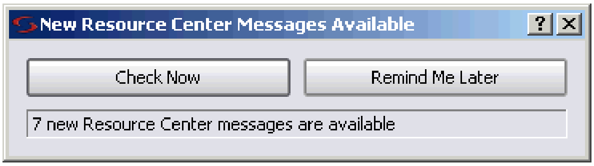
Check for these types of messages:
☑ Messages for this product
☑ Messages for all Synplicity products
☑ Special Promotions

☑ Show new messages available notification window

Press OK to use the web to check for and view selected message types

   - Set the frequency at which you wish to be notified.
   - Select the type of information you want to receive.
   - Select the Show new messages available notification window if you wish to receive immediate notification of any new messages.

     If enabled, the following window appears when a new message is added.



New Resource Center Messages Available

7 new Resource Center messages are available

- To go to the resource center immediately, click Check Now.

If the TRC has never been configured, the envelope icon in the lower right of the application window appears 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:
   - Click the envelope icon in the status bar (lower right of the application window) when it's yellow and blinking to view new information.
   - Select Web->Check Resource Center Messages.

     If you have not set your TRC preferences, the software opens the Resource Center Options dialog box described in the previous step.
4. For other Synopsys FPGA product information, select the following:

Web->Go to SolvNet	http://solvnet.synopsys.com
Web->Go to Training Center	Synopsys training page
Web->Synopsys Home page	Synopsys home page
Web->FPGA Implementation Tools	Synopsys FPGA design solution information page
Tech-Support>Web Support	SolvNet technical support

Resource Center Messages

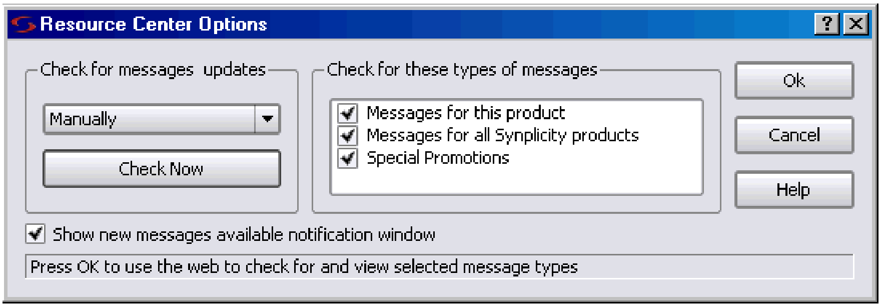
The envelope icon in the status bar (lower right of the application window) indicates when new messages are available:

Appearance	Signifies
Yellow or blinking yellow	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 Technical Resource Center.

Configure Resource Center Command

Sets options for automatic notification from the Technical Resource Center. From the Resource Center Options dialog box, you determine if you want to receive notification on updates for your product, all products, and/or promotional offers. You also determine how often you want the tool to check for new messages and alerts.



Check for messages updates: Manually

Check for these types of messages:
☑ Messages for this product
☑ Messages for all Synplicity products
☑ Special Promotions

☑ Show new messages available notification window

Press OK to use the web to check for and view selected message types

Option	Description
Check for messages and updates	Determines the frequency at which the software checks for updates. Select a setting from the menu: • Each invocation – checks for new messages and alerts each time you start the tool. • Weekly • Daily • Manually – does not automatically check for updates and/or alerts. With this setting, you must remember to go to the Resource Center to check for pertinent information on your product.
Check Now	Opens the home page for the Technical Resource Center (same as the Web->Check Resource Center Messages command).
Check for these types of messages	Determines the type of notification to receive. Check the boxes, as desired. • Messages for this product – Updates and critical bulletins for the product. • Messages for all Synplicity products – Updates and critical bulletins for all Synopsys FPGA products. • Special promotions – Promotional packages and pricing, occasional surveys, and other related information.

Help Menu

There are four help systems accessible from the Help menu:

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

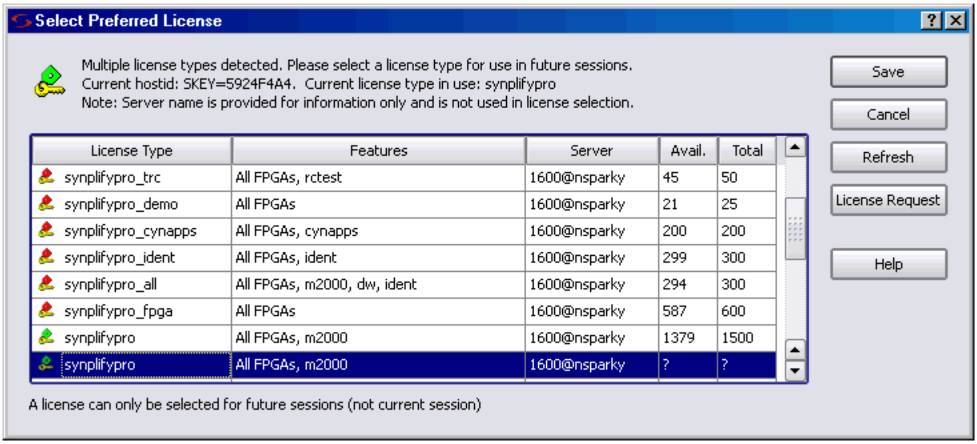
The following table describes the Help menu commands. Some commands are only available in certain views.

Command	Description
Help	Displays hyperlinked online help for the product.
Additional Products	Displays the Synopsys FPGA family of products and a brief description.
How to Use Help	Displays help on how to use Synopsys FPGA online help.
Online Documents	Displays an Open dialog box with hyperlinked PDF documentation on the product including release notes, user guide, reference manual, and licensing configuration and setup. You need Adobe Acrobat Reader® to view the PDF files.
TCL	Displays help for Tcl commands.
Mouse Stroke Tutor	Displays the Mouse Stroke Tutor dialog box which provides information on the available mouse strokes – see Using Mouse Strokes, on page 87 for details.
License Agreement	Displays the Synopsys FPGA software license agreement.
License Request	Displays a dialog box where you can request a trial license or license upgrade.
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 280.
Tip of the Day	Displays a daily tip on how to use the Synopsys FPGA synthesis tools better. See Tip of the Day Command, on page 281.
About this program	Displays the About dialog box, showing the synthesis tool product name, license expiration date, customer identification number, version number, and copyright. Clicking the Versions button in the About dialog box displays the Version Information dialog box, listing the installation directory and the versions of all the synthesis tool compiler and mapper programs.

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 Synopsys FPGA synthesis tool. The new session uses the preferred license you selected.



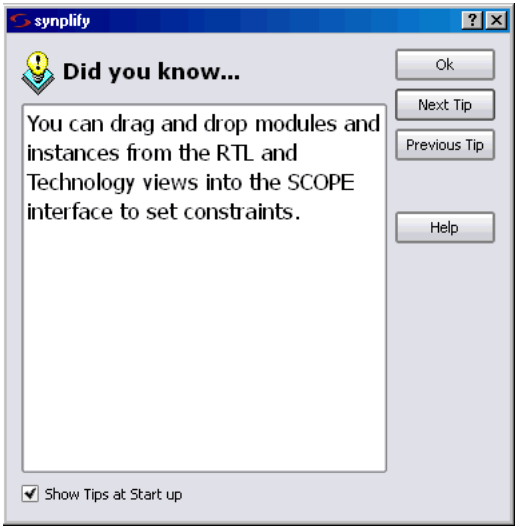
Multiple license types detected. Please select a license type for use in future sessions.
Current hostid: SKEY=5924F4A4. Current license type in use: synplifypro
Note: Server name is provided for information only and is not used in license selection.

License Type	Features	Server	Avail.	Total
synplifypro_trc	All FPGAs, rctest	1600@nsparky	45	50
synplifypro_demo	All FPGAs	1600@nsparky	21	25
synplifypro_cynapps	All FPGAs, cynapps	1600@nsparky	200	200
synplifypro_ident	All FPGAs, ident	1600@nsparky	299	300
synplifypro_all	All FPGAs, m2000, dw, ident	1600@nsparky	294	300
synplifypro_fpga	All FPGAs	1600@nsparky	587	600
synplifypro	All FPGAs, m2000	1600@nsparky	1379	1500
synplifypro	All FPGAs, m2000	1600@nsparky	?	?

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 Synopsys FPGA 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.



You can drag and drop modules and instances from the RTL and Technology views into the SCOPE interface to set constraints.

☑ Show Tips at Start up

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 Menus, on page 287 for details
SCOPE window	Contains commonly used commands from the Edit menu.
Watch Window	See Watch Window Popup Menu, on page 282 for details.

Popup Menu	Description
Tcl window	Contains commands from the Edit menu. For details, see Edit Menu Commands for the Text Editor, on page 125.
Text Editor window	See Text Editor Popup Menu, on page 282 for more information.
RTL and Technology views	See RTL and Technology Views Popup Menus, on page 301.
FSM viewer	See FSM Viewer Popup Menu, on page 285.

Watch Window Popup Menu

The Watch window popup menu contains the following commands:

Command	Description
Configure Watch	Displays the Log Watch Configuration dialog box, where you choose the implementations to watch.
Refresh	Refreshes (updates) the window display.
Clear Parameters	Empties the Watch window.

For more information on the Watch window and the Configure Watch dialog box, see Watch Window, on page 58.

Tcl Window Popup Menu

The Tcl window popup menu contains the Copy, Paste, and Find commands from the Edit menu, as well as the Clear command, which empties the Tcl window. For information on the Edit menu commands available in the Tcl window, see Edit Menu Commands for the Text Editor, on page 125.

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 125 and HDL Analyst Menu, on page 238.

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

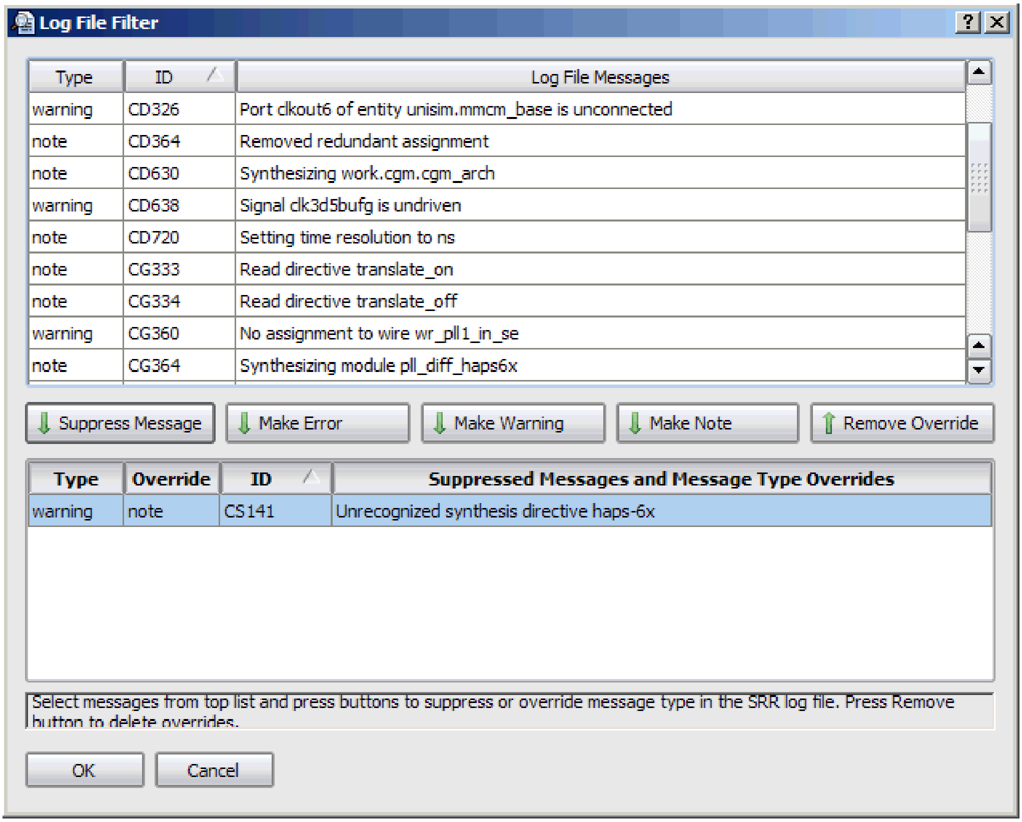
Command	Description
Filter Analyst	Filters your design to show only the currently selected objects in the HDL text file. This is the same as HDL Analyst->Filter Schematic.
Select in Analyst	Crossprobes from the Text Editor and selects the objects in the HDL Analyst view. To use this command, the Enhanced Text Crossprobing (option must be engaged.

Log File Popup Menu

The popup menu in the log file contains commands that control operations in the log file. The popup menu differs when the log file is opened in the HTML mode or in the ASCII text mode.

Log File Filter Dialog Box

The Log File Filter dialog box is available by selecting Log File Message Filter from the log file popup menu when the log file is opened in the HDML mode. The dialog box allows messages in the current session to be promoted or demoted in severity or suppressed from the log files for subsequent sessions. For additional information on using this dialog box, see Log File Message Controls, on page 252 of the *User Guide*.



Type	ID	Log File Messages
warning	CD326	Port clkout6 of entity unisim.mmcm_base is unconnected
note	CD364	Removed redundant assignment
note	CD630	Synthesizing work.cgm.cgm_arch
warning	CD638	Signal clk3d5bufg is undriven
note	CD720	Setting time resolution to ns
note	CG333	Read directive translate_on
note	CG334	Read directive translate_off
warning	CG360	No assignment to wire wr_pll1_in_se
note	CG364	Synthesizing module pll_diff_haps6x

Type	Override	ID	Suppressed Messages and Message Type Overrides
warning	note	CS141	Unrecognized synthesis directive haps-6x

Select messages from top list and press buttons to suppress or override message type in the SRR log file. Press Remove button to delete overrides.

The following table describes the dialog box functions.

Function	Description
Log File Messages window	Displays the message ID and text and the default message type of messages generated during the current session.
Suppress Message button	Suppresses the selected note, warning, or advisory message. The selected message is removed from the upper Log File Messages window and displayed in the lower window with the Override column indicating suppress status. Note that error messages cannot be suppressed.
Make Error button	Promotes the status of the selected warning (or note) to an error. The selected message is removed from the upper Log File Messages window and displayed in the lower window with the Override column indicating error status.

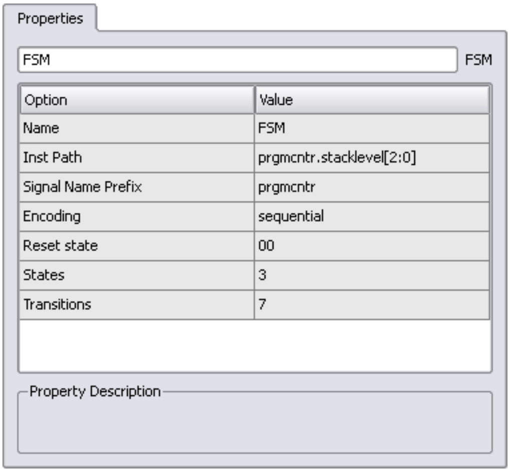
Function	Description
Make Warning button	Promotes the status of the selected note to a warning. The selected message is removed from the upper Log File Messages window and displayed in the lower window with the Override column indicating warning status.
Make Note button	Demotes the status of the selected warning to a note. The selected message is removed from the upper Log File Messages window and displayed in the lower window with the Override column indicating note status.
Remove Override button	Removes the override status on the selected message in the lower window and returns the message to the upper Log File Messages window.
lower window	Lists the status of all messages that have been promoted, demoted, or suppressed.
OK button	Updates the status of any changed messages in the .pfl file. Note that you must recompile/resynthesize the design before any message status changes become effective.

FSM Viewer Popup Menu

The popup menu in the FSM Viewer contains commands that determine what is shown in the FSM Viewer. The following table lists the popup commands in the FSM Viewer.

Command	Description
Properties	Displays the Object Properties dialog box and view properties of a selected state or transition. Information about a selected transition includes the conditions enabling the transition and the identities of its origin and destination states. Information about a selected state includes its name, RTL encoding, and mapped encoding.
Filter	See View Menu: FSM Viewer Commands, on page 139
Unfilter	See View Menu: FSM Viewer Commands, on page 139
FSM Properties	Displays the Object Properties dialog box indicating the FSM identity and location, encoding style, reset state, and the number of states and transitions.

FSM Properties

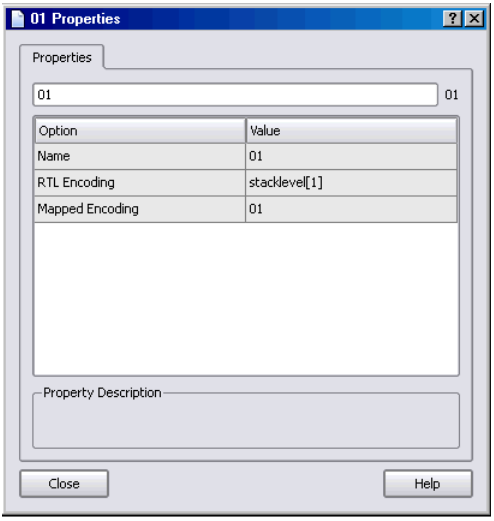


FSM FSM

Option	Value
Name	FSM
Inst Path	prgmcntr.stacklevel[2:0]
Signal Name Prefix	prgmcntr
Encoding	sequential
Reset state	00
States	3
Transitions	7

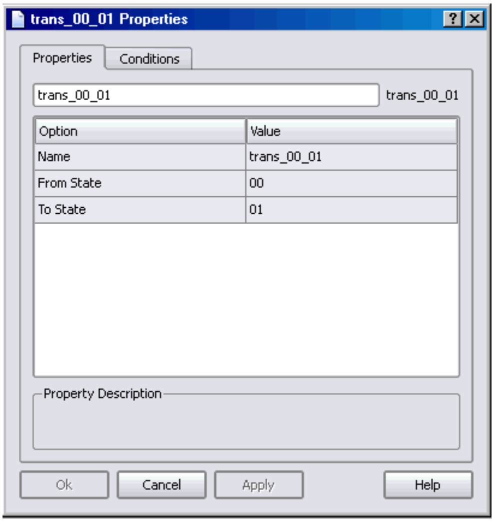
State properties (state selected)

Transition properties (transition selected)



01 01

Option	Value
Name	01
RTL Encoding	stacklevel[1]
Mapped Encoding	01



trans_00_01 trans_00_01

Option	Value
Name	trans_00_01
From State	00
To State	01

Project View Popup Menus

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
Project Window (Project Files tab in Synplify Pro)	
Open Project	Displays the Open Project Dialog. See Open Project Command, on page 122.
New Project	Creates a new empty project in the Project Window.
Build Workspace	Creates a project workspace. In the Project view, select existing projects that you want to include in the project workspace. See Build Workspace Popup Menu Command, on page 297.
Refresh	Refreshes the display.
Project View Options	Displays the Project View Options dialog. See Project View Options Command, on page 254.
Project Selected	
Project or Workspace Options	Displays project or workspace properties such as name and location. See Project or Workspace Options Popup Menu Command, on page 296.
Open as Text	Opens the selected file in the Text Editor.
Add File	Displays the Add Files to Project dialog. See Add Source File Command, on page 144.
New Implementation	Displays the Implementation Options dialog box. See Implementation Options Command, on page 158
Synthesize	Compiles and maps your design.
Compile Only	Compiles your design.

Command	Description
Write Output Netlist Only	Writes the mapped output netlist to structural Verilog (vm) or VHDL (vhm) format. Same as enabling: • Write Mapped Verilog Netlist • Write Mapped VDHL Netlist on the Implementation Results tab of the Implementation Options dialog box.
Arrange VHDL Files	Reorders the VHDL source files.
Save Project	Displays the Save Project As dialog box.
Close Project	Closes your project.
	Project Folder or File Selected
Add Folder	Creates a folder with the new name you specified and adds it to the Project view. See Add Folder Command, on page 291.
Rename Folder	Renames an existing folder with the new name you specified in the Project view. See Rename Folder Command, on page 291.
Delete Folder	Deletes the specified folder and all its contents as necessary. See Delete Folder Command, on page 291.
Remove from Folder	Removes the selected file from its corresponding folder.
Place in Folder	Places the selected file into the folder you specify.
	Constraint File Selected
File Options	Displays the File Options dialog box. See File Options Popup Menu Command, on page 292.
Open	Opens the SCOPE window.
Open as Text	Opens the selected file in the Text Editor.
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 Popup Menu Command, on page 295.

Command	Description
Change File	Opens the Source File dialog box where you choose a new file to replace the selected file. See Change File Command, on page 148
Remove File From Project	Removes the file from the project.
HDL File Selected	
File Options	Displays the File Options dialog box. See File Options Popup Menu Command, on page 292.
Open	Opens the file in the Text Editor.
Syntax Check	Runs a syntax check on your design code. Reports errors, warnings, or notes in the Tcl Window.
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 performed. Reports errors, warnings, or notes in the Tcl Window.
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 Popup Menu Command, on page 295.
Change File	Opens the Source File dialog box where you choose a new file to replace the selected file. See Change File Command, on page 148
Remove File From Project	Removes the file from the project.
Implementation Selected	
Implementation Options	Displays the Implementation Options dialog box. See Implementation Options Command, on page 158.
Change Implementation Name	Displays the Implementation Name dialog box, where you rename the selected implementation. (See Change Implementation Popup Menu Commands, on page 295.)
Show Compile Points	Displays the compile points of the selected implementation and lets you edit them. See Show Compile Points Popup Menu Command, on page 296.

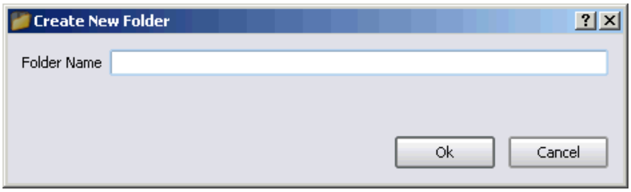
Command	Description
Copy Implementation	Copies the selected implementation and adds it to the current project with the name you specify in the dialog box. (See Change Implementation Popup Menu Commands, on page 295.)
Remove Implementation	Removes the selected implementation from the project.
RTL View	Creates an RTL View based on the properties of the selected implementation.
Tech View	Creates a Technology View based on the properties of the selected implementation.
Add P&R Implementation	Displays the Add New Place & Route Task dialog box where you set options to run place & route after synthesis. See Add P&R Implementation Popup Menu Command, on page 298
Run	Starts a synthesis run on your design.
	Place & Route Implementation Selected
P & R Options	Displays the Options for Place & Route on Implementation dialog box, so you can change options and rerun placement and routing. See Options for Place & Route Jobs Popup Menu Command, on page 299 for a description of the features.
Add Place & Route Job	Displays the Add New Place & Route Task dialog box, so you can set options and run placement and routing. See Add P&R Implementation Popup Menu Command, on page 298.
Remove Place & Route Job	Deletes the place-and-route implementation from the project.
Run Place & Route Job	Runs the place-and-route job for the design.

Project View Popup Folder Commands

The Project view popup menu includes commands for manipulating folders.

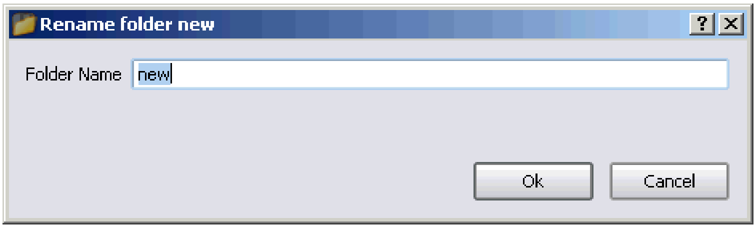
Add Folder Command

Use this option to add a folder to the Project view.



Rename Folder Command

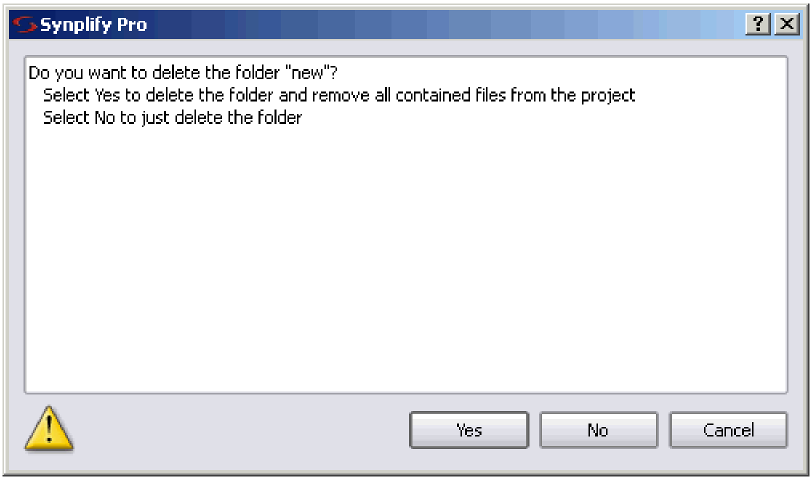
Use this option to rename an existing folder in the Project view.



Folder Name new

Delete Folder Command

Use this option to delete a folder from the Project view.



Do you want to delete the folder "new"?
Select Yes to delete the folder and remove all contained files from the project
Select No to just delete the folder

This dialog box includes the following options:

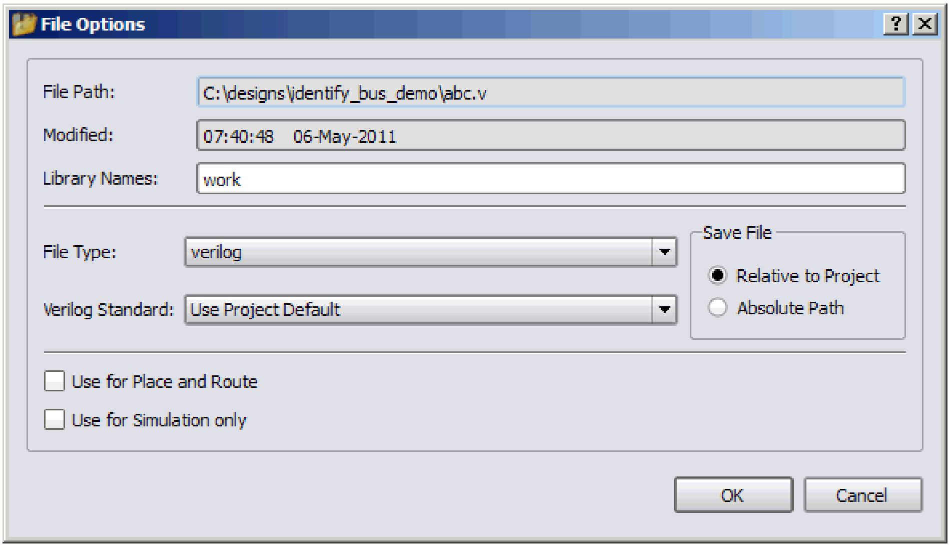
Feature	Description
Yes	Select Yes to delete the folder and all files contained in the folder from the Project view.
No	Select No to delete just the folder from the Project view.
Cancel	Select Cancel, to discontinue the operation.

File Options Popup Menu Command

To display the File Options dialog box, right-click on a project file and select File Options from the popup menu.

Field/Option	Description
File Path	Path to the selected file.
File Type	The folder type for the selected file. You can select the file folder type from a large list of file types. Changing the folder file type does *not* change the file contents or its extension; it simply places the file in the specified Project view folder. For example, if you change the file type of a VHDL file to Verilog, the file retains its Verilog extension, but is moved from the VHDL folder to the Verilog folder.
Library Names	Name of the library which must be compatible with the HDL simulator. For VHDL files, the dialog box is the same as that accessed by Project->Set VHDL Library – see Set VHDL Library Command, on page 148.
Last modified	Date the file was last modified.
Save file	The format for the path type: choose either Relative to Project (the default) or with an Absolute Path.
Verilog Standard (Verilog only)	Select the Verilog file type from the menu: Use Project Default, Verilog 95, Verilog 2001, or SystemVerilog. Use Project Default sets the type of the selected file to the default for the project (new projects default to SystemVerilog).

The following is the Verilog dialog box:



File Path: C:\designs\identify_bus_demo\abc.v

Modified: 07:40:48 06-May-2011

Library Names: work

File Type: verilog

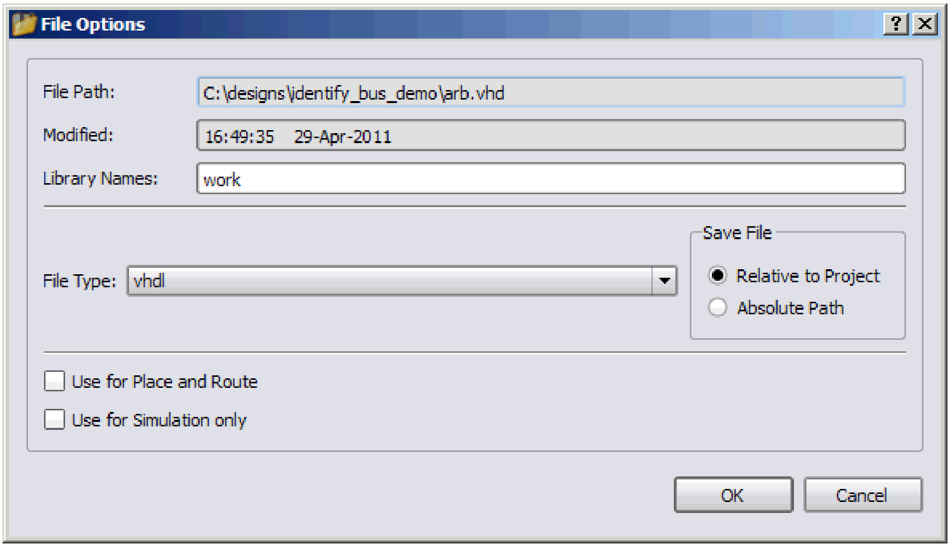
Verilog Standard: Use Project Default

Save File
● Relative to Project
○ Absolute Path

□ Use for Place and Route

□ Use for Simulation only

The following is the VHDL dialog box:



File Path: C:\designs\identify_bus_demo\arb.vhd

Modified: 16:49:35 29-Apr-2011

Library Names: work

File Type: vhdl

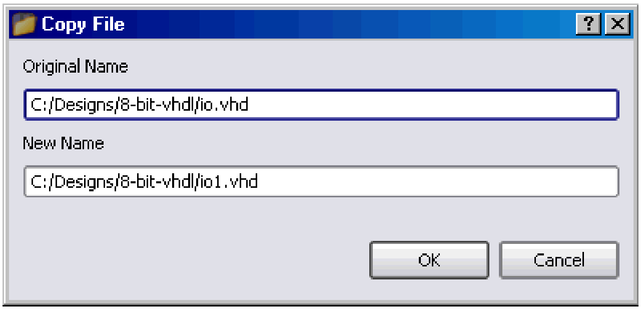
Save File
● Relative to Project
○ Absolute Path

□ Use for Place and Route

□ Use for Simulation only

Copy File Popup Menu 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.



Copy File

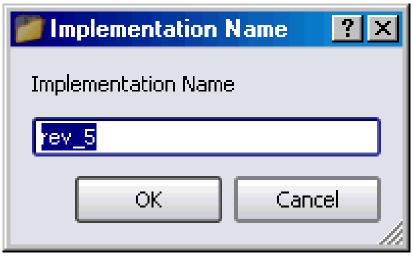
Original Name: C:/Designs/8-bit-vhdl/io.vhd

New Name: C:/Designs/8-bit-vhdl/io1.vhd

Change Implementation Popup Menu Commands

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.

Command	Description
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.



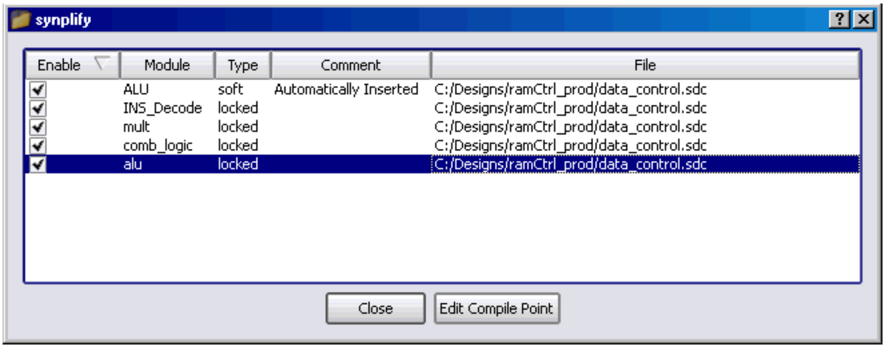
Implementation Name

Implementation Name: rev_5

Show Compile Points Popup Menu Command

With an implementation selected, select the Show Compile Points popup menu command to display the Compile Points dialog box and view or edit the compile points of the selected implementation.

Compile points are only available for certain technologies. For more information on compile points and the compile-point synthesis flow, see Compile Point Types, on page 419 and Synthesizing Compile Points, on page 433 of the *User Guide*.



Enable	Module	Type	Comment	File
☑	ALU	soft	Automatically Inserted	C:/Designs/ramCtrl_prod/data_control.sdc
☑	INS_Decode	locked		C:/Designs/ramCtrl_prod/data_control.sdc
☑	mult	locked		C:/Designs/ramCtrl_prod/data_control.sdc
☑	comb_logic	locked		C:/Designs/ramCtrl_prod/data_control.sdc
☑	alu	locked		C:/Designs/ramCtrl_prod/data_control.sdc

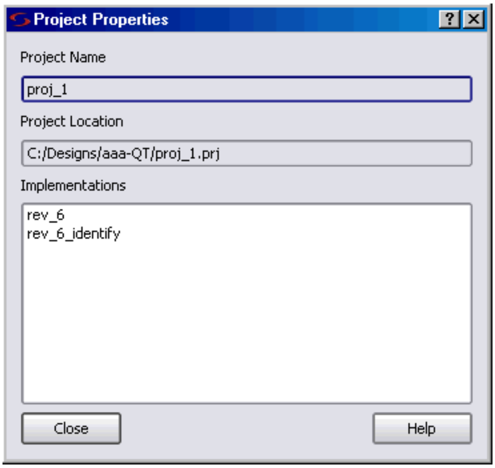
The columns Enb, Module, Type, and Comment in the dialog box correspond to the columns Enabled, Module, Type, and Comment in the SCOPE spreadsheet for the compile point. The File column lists the top-level constraint file where the compile point is defined.

To open and edit the SCOPE spreadsheet for a compile point, either double-click the row of the compile point or select it and click the Edit Compile Point button.

Project or Workspace Options Popup Menu Command

With a project or workspace selected, select the Project Options or Workspace 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

proj_1

Project Location

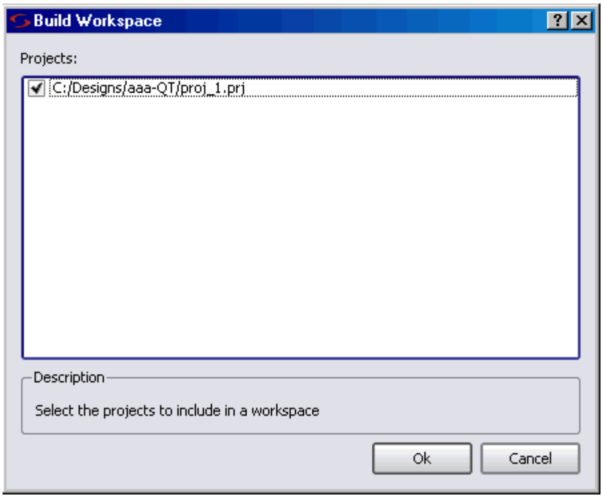
C:/Designs/aaa-QT/proj_1.prj

Implementations

rev_6
rev_6_identify

Build Workspace Popup Menu Command

To use the Workspace feature, right-click in the project view and select the Build Workspace popup menu command. The Build Workspace dialog box appears where you select projects to include in a workspace. If no projects exist, click OK to build an empty workspace.



Projects:

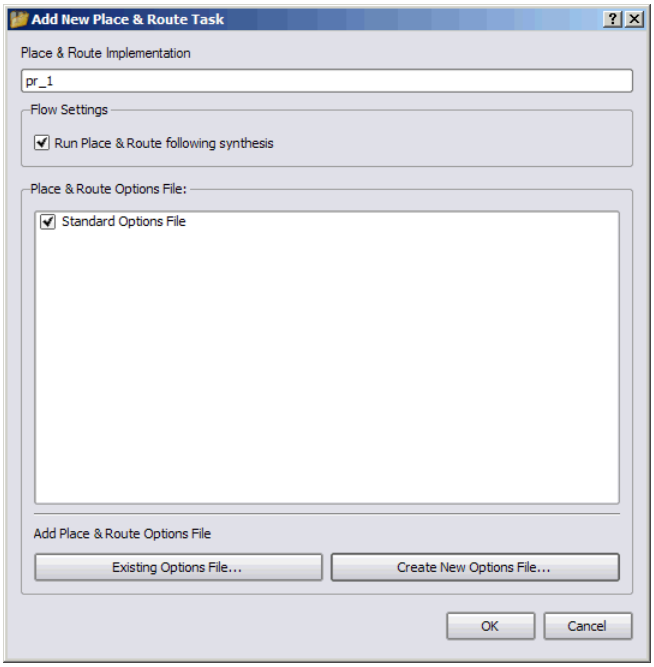
☑ C:/Designs/aaa-QT/proj_1.prj

Description

Select the projects to include in a workspace

Add P&R Implementation Popup Menu Command

Displays the Add New Place & Route Task dialog box. For information about using this command for place-and-route encapsulation, see Running Place-and-Route after Synthesis, on page 356 in the *User Guide*.

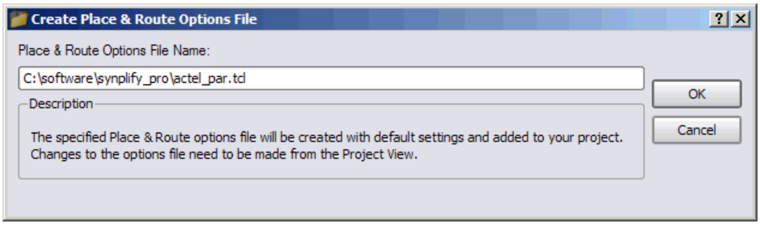

Add New Place & Route Task
? ✕
Flow Settings
Place & Route Options File:
Add Place & Route Options File
Existing Options File...
Create New Options File...
OK
Cancel

Place & Route Implementation: pr_1

☑ Run Place & Route following synthesis

☑ Standard Options File

Command	Description
Place & Route Implementation Name	Enter a name for the place & route implementation. Do not use spaces for the implementation name.
Flow Settings	
Run Place & Route following synthesis	Enable/disable the running of the place & route tool from the synthesis tool immediately following synthesis.
Place & Route Options File	This option lets you specify a place & route options file. You can select either the: • Standard Options File – use this option to run the standard synthesis place-and-route flows. • Specify another option file.

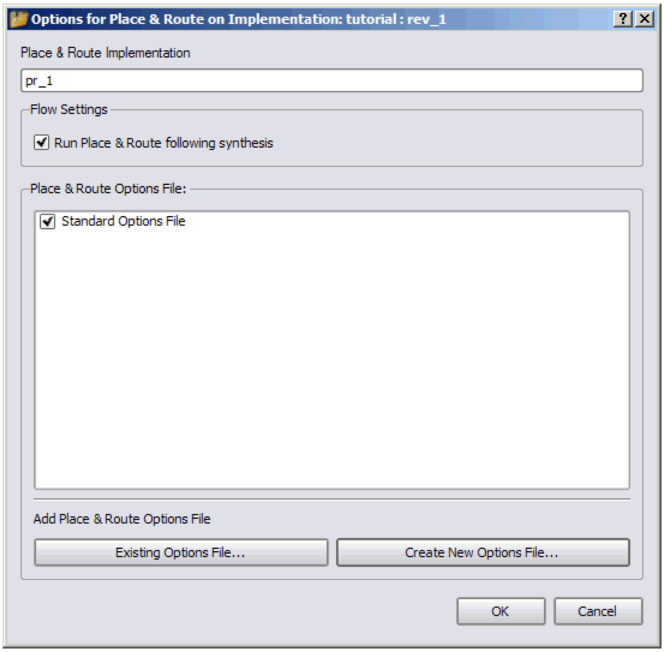
Command	Description
Add Place & Route Options File Existing Options File	This option opens the Select Place & Route option file dialog box where you browse for an existing place & route options file. See Running Place-and-Route after Synthesis, on page 356 for information about using this feature.
Add Place & Route Options File Create New Options File	This option opens the Create Place & Route Options File dialog box where you specify a new place & route options file. See Running Place-and-Route after Synthesis, on page 356 for information about creating a new options file.  Create Place & Route Options File Place & Route Options File Name: C:\software\synplify_pro\actel_par.tcl Description The specified Place & Route options file will be created with default settings and added to your project. Changes to the options file need to be made from the Project View.

Once the par implementation is created, then you can right-click and perform any of the following options:

- P&R Options—See Options for Place & Route Jobs Popup Menu Command, on page 299.
- Add Place & Route Job—See Add P&R Implementation Popup Menu Command, on page 298.
- Run Place & Route Job—Runs the place-and-route job.

Options for Place & Route Jobs Popup Menu Command

You can select a place-and-route job for a particular implementation, easily change options and then rerun the job. These options are the same found on the Options for Place & Route on Implementation dialog box. For a description of these options, see Add P&R Implementation Popup Menu Command, on page 298.



Options for Place & Route on Implementation: tutorial : rev_1

Place & Route Implementation: pr_1

Flow Settings

☑ Run Place & Route following synthesis

Place & Route Options File:

☑ Standard Options File

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. See the following:

- Hierarchy Browser Popup Menu Commands, on page 301
- RTL View and Technology View Popup Menu Commands, on page 301

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.
Filter	Highlights and filters objects such as ports, instances, and primitives in the HDL analyst window.
Reload	Refreshes the Hierarchy Browser. Use this if the Hierarchy Browser and schematic view do not match.
Hide/Unhide Instances	Hides or unhides selected instances in the HDL analyst window. For more information on hidden instances, see Hidden Hierarchical Instances, on page 316.

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 238). The following table lists the unique commands.

Common Commands

Command	See...
Show Critical Path	HDL Analyst Menu: Timing Commands, on page 245
Timing Analyst	HDL Analyst Menu: Timing Commands, on page 245
Find	Find Command (HDL Analyst), on page 129
Filter Schematic	HDL Analyst Menu: Filtering and Flattening Commands, on page 241
Push/Pop Hierarchy	HDL Analyst Menu: RTL and Technology View Submenus, on page 238
Select All Schematic	HDL Analyst Menu: Selection Commands, on page 249
Select All Sheet	HDL Analyst Menu: Selection Commands, on page 249
Unselect All	HDL Analyst Menu: Selection Commands, on page 249
Flatten Schematic	HDL Analyst Menu: Filtering and Flattening Commands, on page 241
Unflatten Current Schematic	HDL Analyst Menu: Filtering and Flattening Commands, on page 241
HDL Analyst Options	HDL Analyst Options Command, on page 263
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 (Edit Attributes Popup Menu Command, on page 305), where you select the constraint file to edit. If no constraint file exists, you are prompted to create one.

SCOPE->Edit Compile Point Constraints (module <*module name*>)	For technologies that support compile points, it opens a SCOPE window where you can enter constraints for the selected compile point. It displays the Select Compile Point Definition File dialog box and lets you create or edit a compile-point constraint file for the selected region or instance. See Edit Attributes Popup Menu Command, on page 305.
SCOPE->Edit Module Constraints (module <*module name*>)	Opens a SCOPE window so you can define module constraints for the selected module). If you do not have a constraint file, it prompts you to create one. The file created is a separate, module-level constraint file.

Instance Selected

Command	**See...**
Isolate Paths	Isolate Paths, on page 246
Expand Paths	Hierarchical->Expand Paths, on page 240
Current Level Expand Paths	Current Level->Expand Paths, on page 241.
Show Context	Show Context, on page 246
Hide Instance	Hide Instances, on page 246
Unhide Instance	Unhide Instances, on page 246
Show All Hier Pins	Show All Hier Pins, on page 247
Dissolve Instance	Dissolve Instances, on page 247
Dissolve to Gates	Dissolve to Gates, on page 247

Port Selected

Command	**See...**
Expand to Register/Port	Hierarchical->Expand to Register/Port, on page 240
Expand Inwards	Hierarchical->Expand Inwards, on page 240
Current Level->Expand	Current Level->Expand, on page 240
Current Level->Expand to Register/Port	Current Level->Expand to Register/Port, on page 241

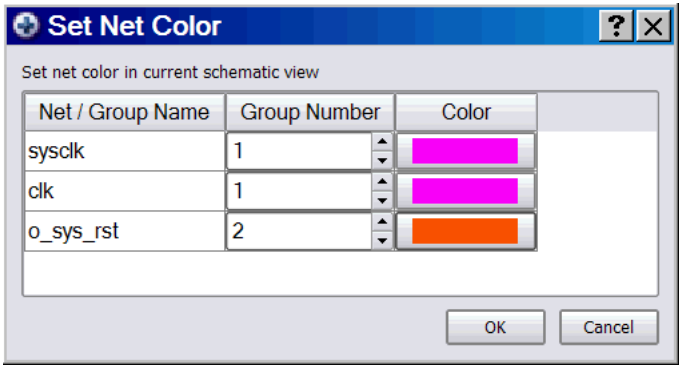
Current Level->Expand Paths	Current Level->Expand Paths, on page 241
Properties	Properties Popup Menu Command, on page 305

Net Selected

Command	**See...**
Goto Net Driver	Hierarchical->Goto Net Driver, on page 240
Select Net Driver	Hierarchical->Select Net Driver, on page 240
Select Net Instances	Hierarchical->Select Net Instances, on page 240
Current Level->Goto Net Driver	Current Level->Goto Net Driver, on page 241
Current Level->Select Net Driver	Current Level->Select Net Driver, on page 241
Current Level->Select Net Instances	Current Level->Select Net Instances, on page 241
Set Net Color	Sets the color of the selected net from a color pallet. For details, see Set Net Color Popup Menu Command, on page 304.

Set Net Color Popup Menu Command

The set net color command sets the color of the selected net in the HDL Analyst for the current session. To use the command, select the desired net or nets in the RTL view and select set net color from the popup menu to display the dialog box.



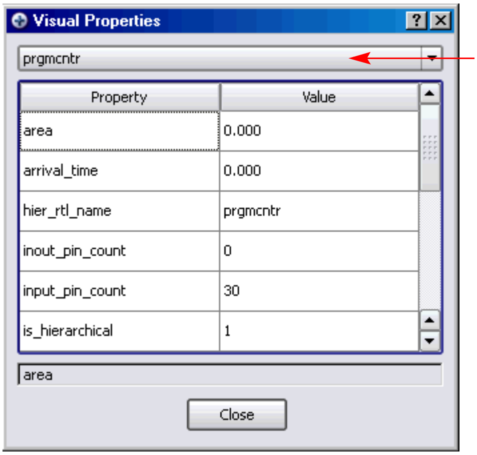
Set net color in current schematic view

Net / Group Name	Group Number	Color
sysclk	1	🟪
clk	1	🟪
o_sys_rst	2	🟧

Double click on the corresponding color in the Color column to display the color pallet and then double click the desired color and click OK. Nets can be grouped and assigned to the same color by selecting the same group number in the Group Number column.

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 Visual Properties Panel, on page 268 or Viewing Object Properties, on page 260 in the *User Guide* for more information about viewing object properties.

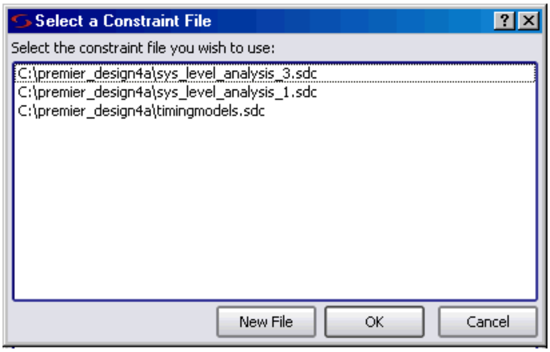


Property	Value
area	0.000
arrival_time	0.000
hier_rtl_name	prgmcntr
inout_pin_count	0
input_pin_count	30
is_hierarchical	1

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

Edit Attributes Popup Menu Command

You use the Select a Constraint File dialog box to choose or create a constraint file. You can open the constraint file and edit it. For technologies that support the compile points, it lets you create or edit a compile-point constraint file for the selected region or instance.



C:\premier_design4a\sys_level_analysis_3.sdc
C:\premier_design4a\sys_level_analysis_1.sdc
C:\premier_design4a\timingmodels.sdc

For more information about creating constraint files, see Specifying Timing Exceptions, on page 70 of the *User Guide*.

CHAPTER 4

HDL Analyst Tool

The HDL Analyst tool helps you examine your design and synthesis results, and analyze how you can improve design performance and area.

The following describe 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 16, *Process Optimization and Automation* in the *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 68 and Technology View, on page 69 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:

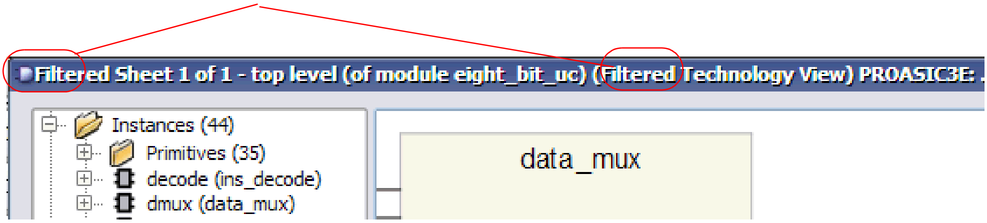
- Filtered and Unfiltered Schematic Views, on page 308
- Accessing HDL Analyst Commands, on page 309

Filtered and Unfiltered Schematic Views

HDL Analyst views (RTL View, on page 68 and Technology View, on page 69) 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) PROASIC3E: .
Instances (44)
Primitives (35)
decode (ins_decode)
dmux (data_mux)
data_mux

Filtering commands affect only the displayed schematic, not the underlying design. For a detailed description of filtering, see Filtering and Flattening Schematics, on page 336. For procedures on using filtering, see Filtering Schematics, on page 305 in the *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 menus appear when you 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 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 238
- View Menu, on page 137
- RTL and Technology Views Popup Menus, on page 301
- Analyst Toolbar, on page 96

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 310
- Sheet Connectors, on page 311
- Primitive and Hierarchical Instances, on page 312
- Hidden Hierarchical Instances, on page 316
- Transparent and Opaque Display of Hierarchical Instances, on page 314
- Schematic Display, on page 316

For most objects, you select them to perform an operation. 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

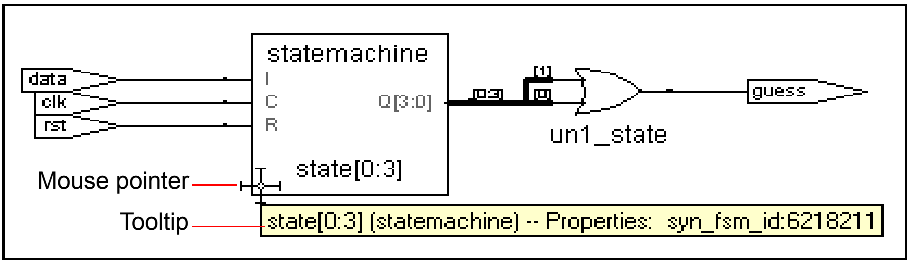
To obtain information about specific objects, you can view object properties with the Properties command from the right-click popup menu, or place the pointer over the object and view the object information displayed. With the latter method, information about the object displays in these two places until 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. If HDL Analyst->Show Timing Information is enabled, the status bar also displays timing information for the object. 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:



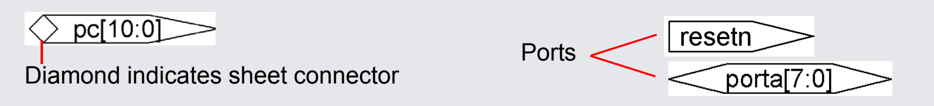
  To disable tooltip display, select View -> Toolbars and disable the Show Tooltips option. Do this if you want to reduce clutter.

See also

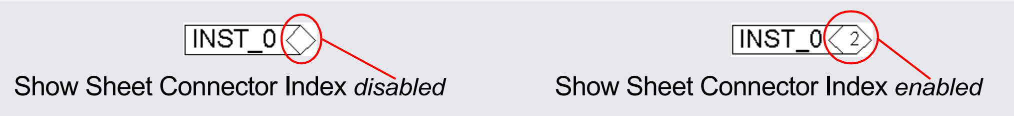
- Pin and Pin Name Display for Opaque Objects, on page 318
- HDL Analyst Options Command, on page 263

Sheet Connectors

When the HDL Analyst tool divides a schematic into multiple sheets, sheet connector symbols indicate how sheets are related. A sheet connector symbol is like a port symbol, but it has an empty diamond with sheet numbers at one end. Use the Options->HDL Analyst Options command (see Sheet Size Panel, on page 266) to control how the schematic is divided into multiple sheets.



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*

See also

- Multiple-sheet Schematics, on page 326
- HDL Analyst Options Command, on page 263
- RTL and Technology Views Popup Menus, on page 301

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:

RTL View	Technology View
High-level logic primitives, like XOR gates or priority-encoded multiplexers	Black boxes
Inferred ROMs, RAMs, and state machines	Technology-specific primitives, like LUTs or FPGA block RAMs
Black boxes	
Technology-specific primitives, like LUTs or FPGA block RAMs	

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 73). 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 to be primitives. Hierarchical instances correspond to Verilog modules and VHDL entities.

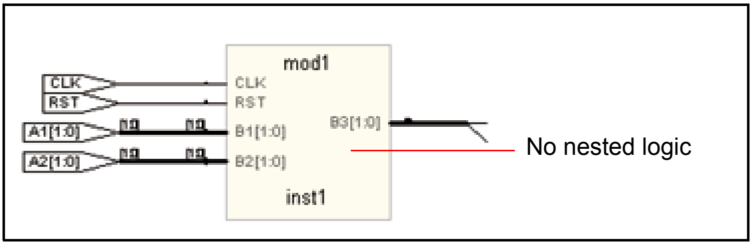
The Hierarchy Browser lists hierarchical instances under Instances, and uses this symbol: ▣ . In a schematic, the display of hierarchical instances depends on the combination of the following:

- 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 314 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 316 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.

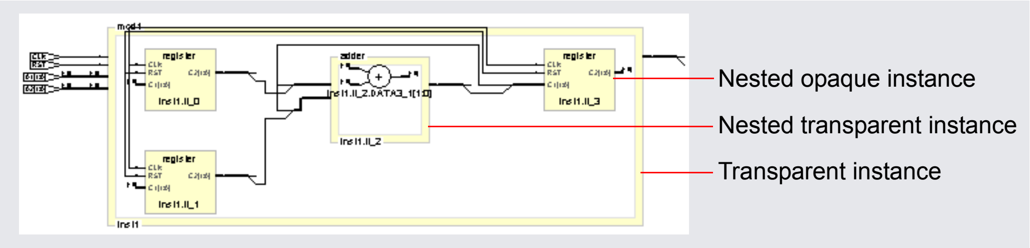
- Opaque instances are pale yellow boxes, and do not display their internal hierarchy. This is the default display.



No nested logic

- 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 334 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.



Nested opaque instance

Nested transparent instance

Transparent instance

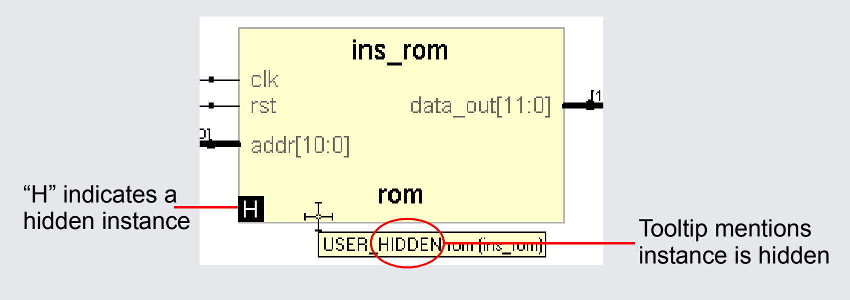
See also

- Looking Inside Hierarchical Instances, on page 334
- Multiple Sheets for Transparent Instance Details, on page 328
- Filtered and Unfiltered Schematic Views, on page 308

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 301). 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.



"H" indicates a hidden instance

Tooltip mentions instance is hidden

Schematic Display

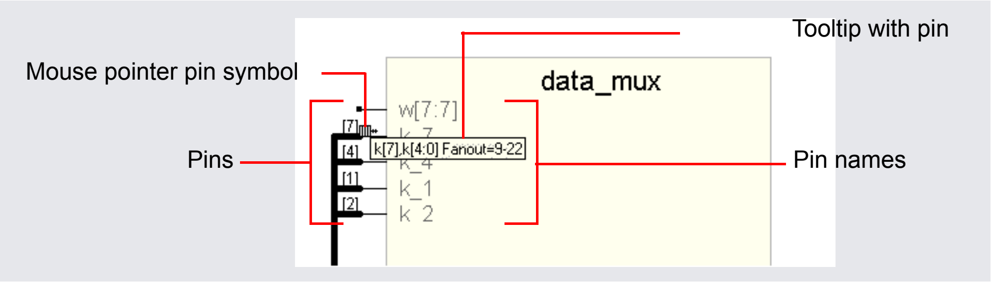
The HDL Analyst Options dialog box controls general properties for all HDL Analyst views, and can determine the display of schematic object information. Setting a display option affects all objects of the given type in all views. Some schematic options only take effect in schematic windows opened after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

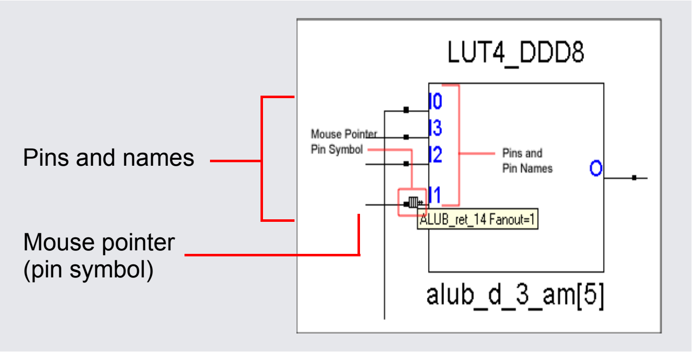
The following are some commonly used settings that affect the display of schematic objects. See HDL Analyst Options Command, on page 263 for a complete list of display options.

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 318 and Pin and Pin Name Display for Transparent Objects, on page 318 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. The first figure shows pins and pin names of an opaque hierarchical instance, and the second figure shows the pins of a technology-specific primitive with its 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.

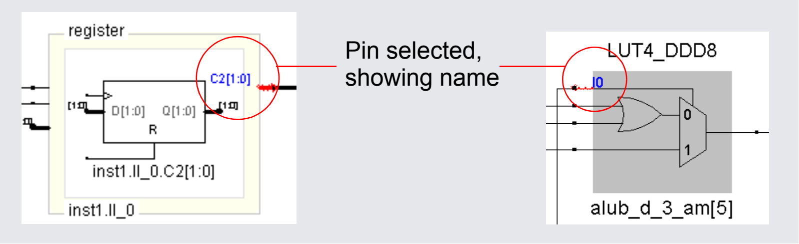
Transparent Hierarchical Instances

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.



See also:

- HDL Analyst Options Command, on page 263
- Controlling the Amount of Logic on a Sheet, on page 326
- Analyzing Timing in Schematic Views, on page 324 in the *User Guide*

Basic Operations on Schematic Objects

Basic operations on schematic objects include the following:

- Finding Schematic Objects, on page 320
- Selecting and Unselecting Schematic Objects, on page 322
- Crossprobing Objects, on page 323
- Dragging and Dropping Objects, on page 325

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

- Filtering and Flattening Schematics, on page 336
- Timing Information and Critical Paths, on page 342
- Multiple-sheet Schematics, on page 326
- Exploring Design Hierarchy, on page 329

Finding Schematic Objects

You can use the following techniques to find objects in the schematic. For step-by-step procedures using these techniques, see Finding Objects, on page 280 in the *User Guide*.

- Zooming and panning
- HDL Analyst Hierarchy Browser

  You can use the 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. See Browsing With the Hierarchy Browser, on page 280 in the *User Guide* for details.

- Edit -> Find command

  The Edit -> Find command is described in Find Command (HDL Analyst), on page 129. It 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 also fine-tune your search by setting a range for the search.

This command selects all found objects, 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. See Using Find for Hierarchical and Restricted Searches, on page 282 in the *User Guide*.

- Edit -> Find command combined with filtering

  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.

- Filtering before searching with Edit->Find

  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 336 for further information.

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 265 in the *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.
- Deselect All deselects all objects in *all* HDL Analyst views.

See also

- Finding Schematic Objects, on page 320
- HDL Analyst Menu, on page 238

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.

Crossprobing Between Different Views

You can crossprobe objects (including logic inside hidden instances) between RTL views, Technology views, the FSM Viewer, 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 293 in the *User Guide*.

From...	To...	Do this...
Text Editor: log file	Text Editor: HDL source file	Double-click a log file note, error, or warning. The corresponding HDL source code appears in the Text Editor.
Text Editor: HDL code	Analyst view FSM Viewer	The RTL view or Technology view must be open. Select the code in the Text Editor that corresponds to the object(s) you want to crossprobe. 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, right-click and choose the Select in Analyst command. To cross-probe from text other than source code, first select Options->HDL Analyst Options and then enable Enhanced Text Crossprobing.

From...	To...	Do this...
FSM Viewer	Analyst view	The target view must be open. The state machine must be encoded with the onehot style to crossprobe from the transition table. Select a state anywhere in the FSM Viewer (bubble diagram or transition table). The corresponding object is automatically selected in the HDL Analyst view.
Analyst view FSM Viewer	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. If the target view is the FSM Viewer, then the state machine must be encoded as onehot.
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.
Text Editor: any text containing instance names, like a timing report	Corresponding instance	Highlight the text, then right-click & choose Select or Filter. Use this to filter critical paths reported in a text file by the FPGA timing analysis tool.

Dragging and Dropping Objects

You can drag and drop objects like instances, nets, and pins from the HDL Analyst schematic views to other windows to help you analyze your design or set constraints. You can drag and drop objects from an RTL or Technology views to the following other windows:

- SCOPE editor
- Text editor window
- Tcl window

Multiple-sheet Schematics

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 311) mark logic connections from one sheet to the next.

For more information, see

- Controlling the Amount of Logic on a Sheet, on page 326
- Navigating Among Schematic Sheets, on page 326
- Multiple Sheets for Transparent Instance Details, on page 328

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 263
- Setting Schematic View Preferences, on page 269 of the *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.

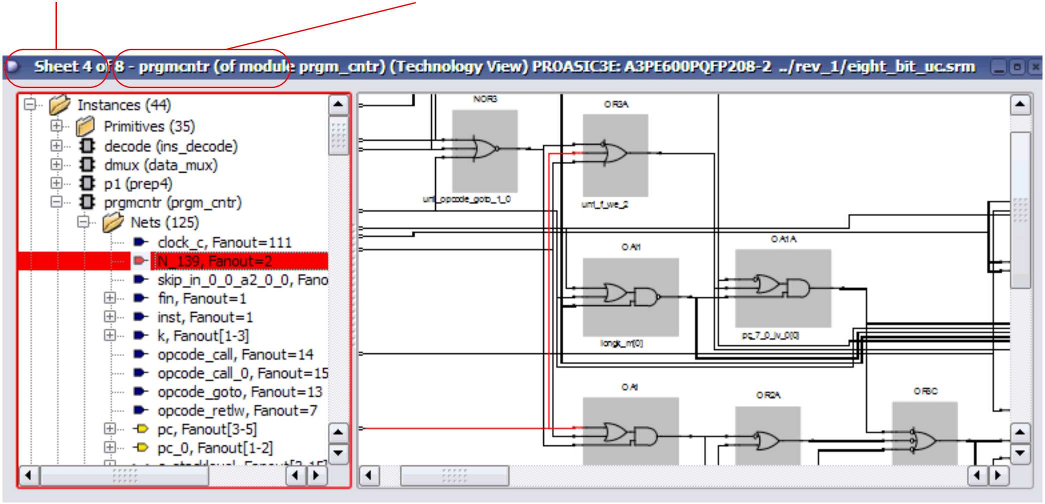
Multisheet Orientation in the Title Bar

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.

Sheet # of total #

Context (level) of current sheet: instance name and module



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 267 in the *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 View->Forward. After pushing down, you can either pop back up or use View->Back.

See also:

- Filtering and Flattening Schematics, on page 336
- View Menu: RTL and Technology Views Commands, on page 138
- Pushing and Popping Hierarchical Levels, on page 329

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 326.
- 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 326
- View Menu: RTL and Technology Views Commands, on page 138

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 329
- Navigating With a Hierarchy Browser, on page 332
- Looking Inside Hierarchical Instances, on page 334

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 332 and Looking Inside Hierarchical Instances, on page 334 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:

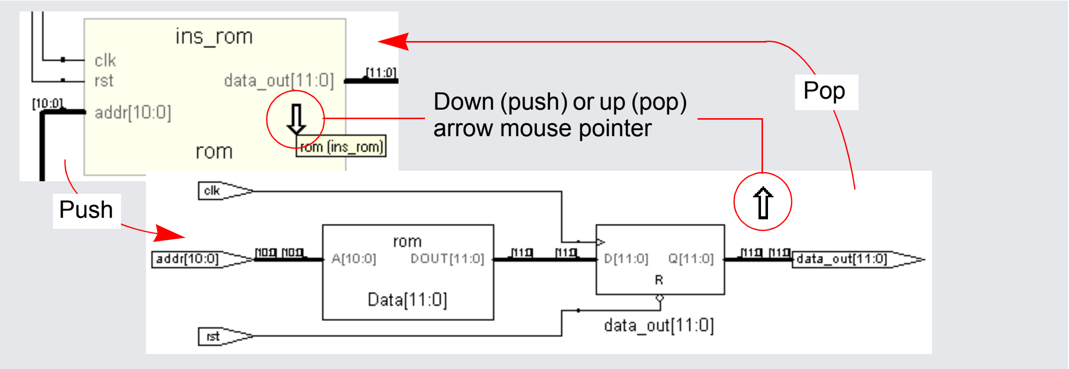
- When you push into an inferred state machine in an RTL view, the FSM Viewer opens, with graphical information about the FSM. See the FSM Viewer Window, on page 74, for more information.
- When you push into an inferred ROM in an RTL view, the Text Editor window opens and displays the ROM data table (`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 330 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.

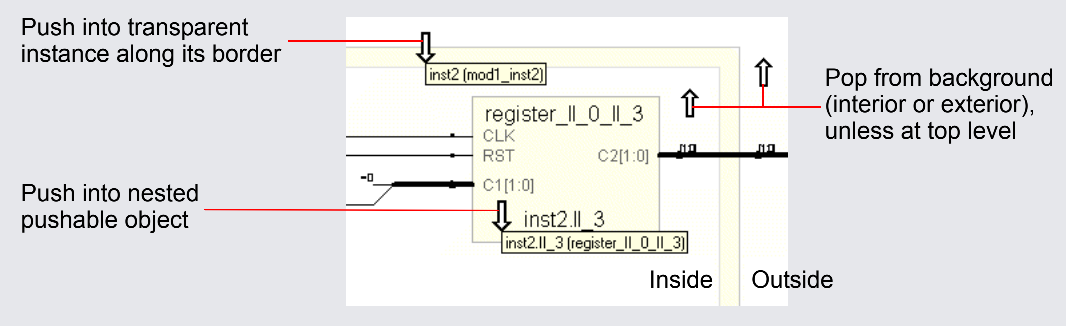
How to Push and Pop Hierarchical Levels

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.



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

The following arrow mouse pointers indicate status in Push/Pop mode. For other indicators, see Pushable Schematic Objects, on page 329.

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 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:

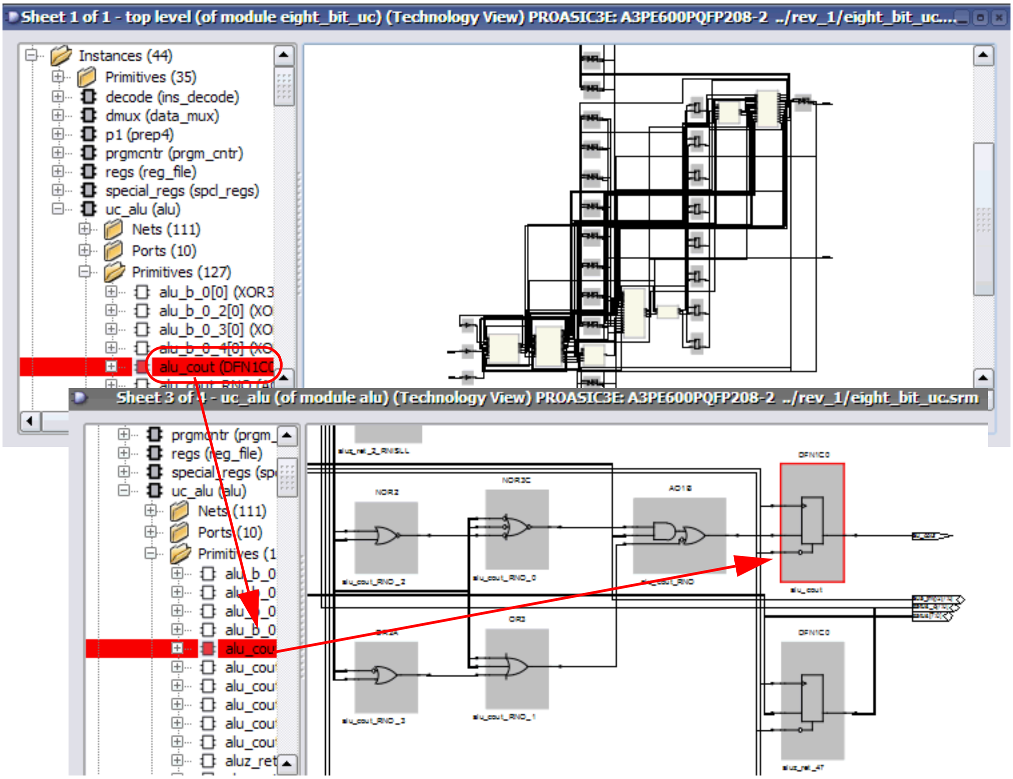
- Hidden Hierarchical Instances, on page 316
- Transparent and Opaque Display of Hierarchical Instances, on page 314
- Using Mouse Strokes, on page 87
- Navigating With a Hierarchy Browser, on page 332

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 329 and Looking Inside Hierarchical Instances, on page 334 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. The following figure shows crossprobing between a schematic and the 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.

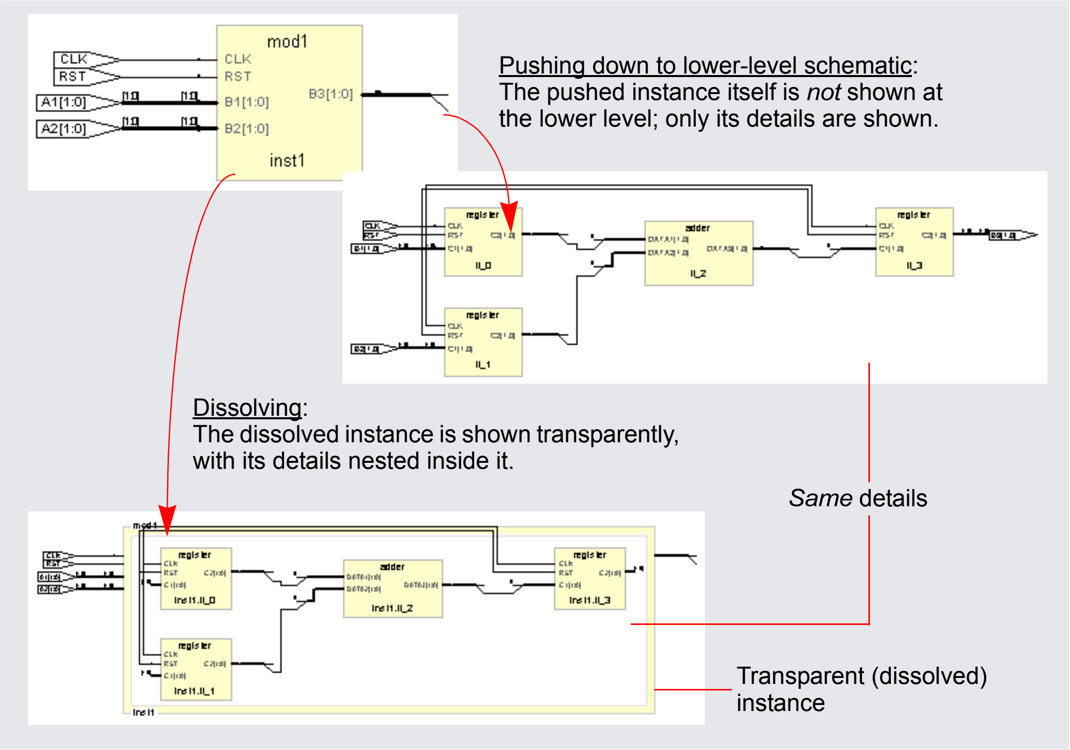
See also:

- Crossprobing Objects, on page 323
- Pushing and Popping Hierarchical Levels, on page 329
- Hierarchy Browser Popup Menu Commands, on page 301

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 332 and Navigating With a Hierarchy Browser, on page 332 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:



Pushing down to lower-level schematic: The pushed instance itself is *not* shown at the lower level; only its details are shown.

Dissolving: The dissolved instance is shown transparently, with its details nested inside it.

Same details

Transparent (dissolved) instance

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 336 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 329
- Navigating With a Hierarchy Browser, on page 332
- HDL Analyst Command, on page 239
- Transparent and Opaque Display of Hierarchical Instances, on page 314
- Hidden Hierarchical Instances, on page 316

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 336
- Combined Filtering Operations, on page 337
- Returning to The Unfiltered Schematic, on page 337
- Commands That Flatten Schematics, on page 338
- Selective Flattening, on page 339
- Filtering Compared to Flattening, on page 341

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 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 305* in the *User Guide*.

See also:

- Filtered and Unfiltered Schematic Views, on page 308
- HDL Analyst Menu, on page 238 and RTL and Technology Views Popup Menus, on page 301

Combined Filtering Operations

Filtering operations are designed to be used in combination, successively. You can perform a sequence of operations like the following:

1. Use Filter Schematic to filter your design to examine a particular instance. See HDL Analyst Menu: Filtering and Flattening Commands, on page 241 for a description of the command.
2. Select Expand to expand from one of the output pins of the instance to add its immediate successor cells to the display. See HDL Analyst Menu: Hierarchical and Current Level Submenus, on page 239 for a description of the command.
3. Use Select Net Driver to add the net driver of a net connected to one of the successors. See HDL Analyst Menu: Hierarchical and Current Level Submenus, on page 239 for a description of the command.
4. Use Isolate Paths to isolate the net driver instance, along with any of its connecting paths that were already displayed. See HDL Analyst Menu: Analysis Commands, on page 245 for a description of the command.

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 320 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 301
- RTL and Technology Views Popup Menus, on page 301
- View Menu: RTL and Technology Views Commands, on page 138

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 339).

See also:

- Filtering Compared to Flattening, on page 341
- Selective Flattening, on page 339

Selective Flattening

By default, flattening operations are not very selective. However, you can selectively flatten particular instances with these commands (see RTL and Technology Views Popup Menus, on page 301 for descriptions):

- 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 as outlined in the following table:

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.

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 301
- Selective Flattening, on page 339

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.

- Timing Reports, on page 342
- Critical Paths and the Slack Margin Parameter, on page 343
- Examining Critical Path Schematics, on page 344

See the following for more information about timing and result analysis:

- Watch Window, on page 58
- Log File, on page 435
- Chapter 16, *Process Optimization and Automation* in the *User Guide*

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 Analysis->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 for 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. When analyzing timing, any latches in the path are treated as level-sensitive registers.

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 163). 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:

- Analysis Menu, on page 226
- Timing Reports, on page 441
- Log File, on page 435

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 238
- HDL Analyst Command, on page 239
- Handling Negative Slack, on page 330 of the *User Guide*
- Analyzing Timing in Schematic Views, on page 324 of the *User Guide*

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.

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 337
- Filtering and Flattening Schematics, on page 336

CHAPTER 5

Constraints

Constraints are used in the FPGA synthesis environment to achieve optimal design results. Timing constraints set performance goals, non-timing constraints (design constraints) guide the tool through optimizations that further enhance performance and physical constraints define regions and locations for placement-aware synthesis.

This chapter provides an overview of how constraints are handled in the FPGA synthesis environment.

Constraint Types

One way to ensure the FPGA synthesis tools achieve the best quality of results for your design is to define proper constraints. In the FPGA environment, constraints can be categorized by the following types:

Type	Description
Timing	Performance constraints that guide the synthesis tools to achieve optimal results. Examples: clocks (create_clock), clock groups (set_clock_groups), and timing exceptions like multicycle and false paths (set_multicycle_path...) See Timing Constraints, on page 351 for information on defining these constraints.
Design	Additional design goals that enhance or guide tool optimizations. Examples: Attributes and directives (define_attribute, define_global_attribute), I/O standards (define_io_standard), and compile points (define_compile_point).

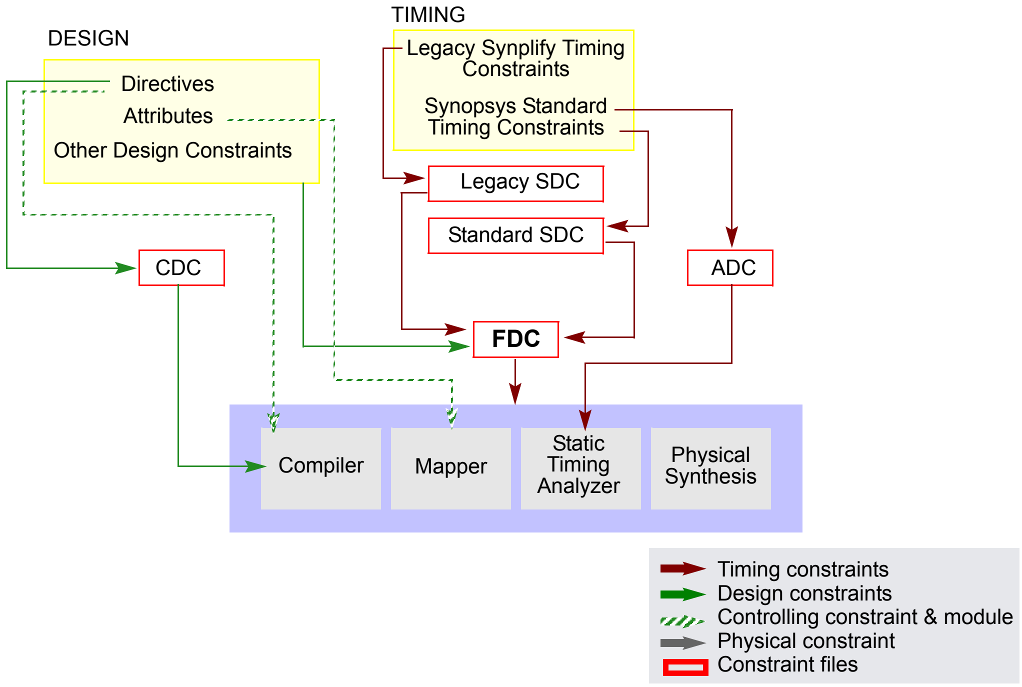
The easiest way to specify constraints is through the SCOPE interface. The tool saves timing and design constraints to an FDC file that you add to your project.

See Also

- Constraint Files, on page 349 for an overview of the types of constraint files that are passed to the tool.
- Timing Constraints, on page 351 for an overview of defining timing constraints and generating FDC files.
- SCOPE Constraints Editor, on page 361 for details on how to automatically create timing and design constraints.
- Chapter 9, *Timing Constraint Syntax* for timing constraint syntax.
- Chapter 10, *FPGA Design Constraint Syntax* for design constraint syntax.
- Chapter 12, *Batch Commands and Scripts* for physical constraint syntax.

Constraint Files

The figure below shows the files used for specifying various types of constraints. The FDC file is the most important one and is the primary file for both timing and non-timing design constraints. The other constraint files are used for specific features or as input files to generate the FDC file, as described in Timing Constraints, on page 351. The figure also indicates the specific processes controlled by attributes and directives.



The table summarizes constraint files.

File	Type	Common Commands	Comments
FDC	Timing constraints	create_clock, set_multicycle_delay …	Used for synthesis. Includes timing constraints that follow the Synopsys standard format as well as design constraints.
	Design constraints	define_attribute, define_io_standard …	
ADC	Timing constraints for timing analysis	create_clock, set_multicycle_delay …	Used with the stand-alone timing analyzer.
SDC (Synopsys Standard)	FPGA timing constraints	create_clock, set_clock_latency, set_false_path …	Use sdc2fdc to convert constraints to an FDC file so that they can be passed to the synthesis tools.
SDC (Legacy)	Legacy timing constraints and non-timing (or design) constraints	define_clock, define_false_path define_attribute, define_collection …	Use sdc2fdc to convert the constraints to an FDC file so that they can be passed to the synthesis tools.

Timing Constraints

Releases prior to G-2012.09M had two types of constraint files that could be used in a design project:

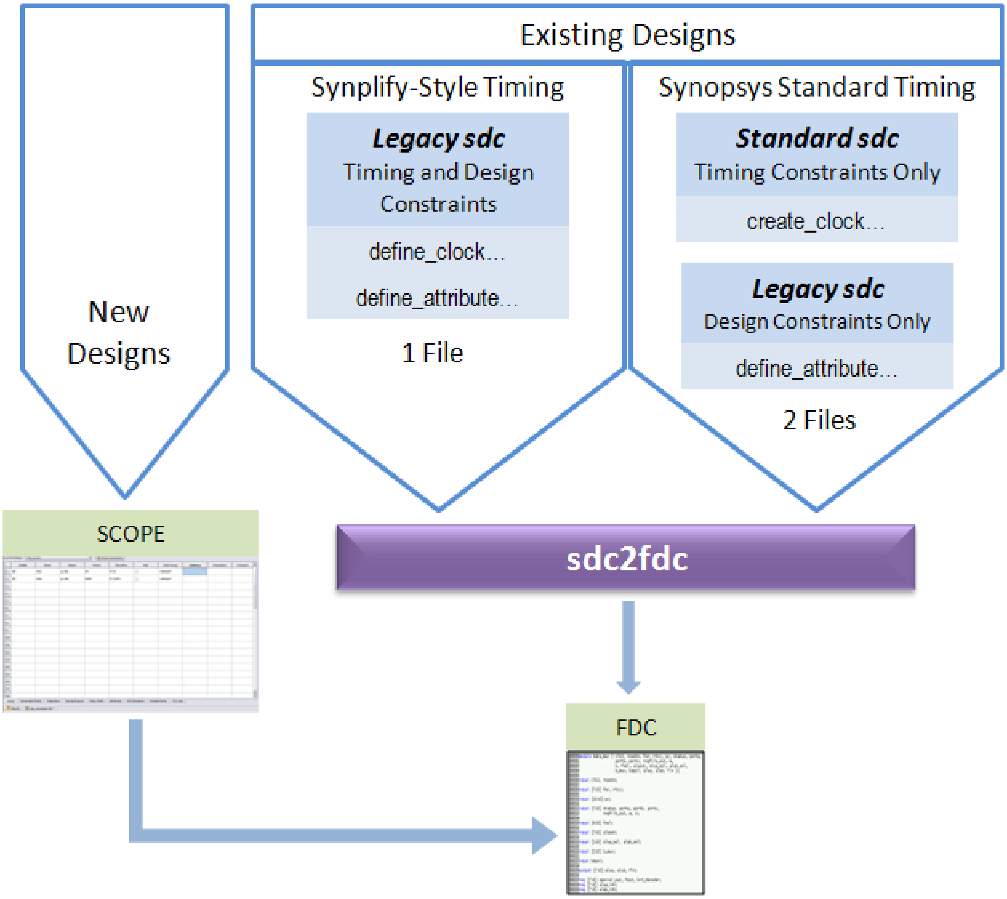
- Legacy “Synplify-style” timing constraints (define_clock, define_false_path...) saved to an sdc file. The same file also included non-timing design constraints, like attributes and compile points.
- Synopsys standard timing constraints (create_clock, set_false_path…). These constraints were also saved to an sdc file, however, contained only timing constraints and no design constraints. Any non-timing constraints were placed in a separate sdc file. The tool used the two files together, drawing timing constraints from one and non-timing constraints from the other.

Starting with the G-2012.09M release, the legacy-style timing constraints have been replaced by Synopsys standard timing constraints; and SDC constraint files have been replaced by FDC (FPGA design constraint) files.

As a result of these updates, there are some changes in the use model:

- Instead of a constraint file in the legacy format, the tool now supports an FDC file, that includes both timing and non-timing constraints. This file uses the Synopsys standard syntax for timing constraints (create_clock, set_multicyle_path...). The syntax for non-timing design constraints remains unchanged (define_attribute, define_io_standard…).
- The SCOPE editor has been enhanced to support the timing constraint changes, so that new constraints can be entered correctly.
- For older designs, use the sdc2fdc command to do a one-time conversion of the constraints.

The following figure summarizes constraint-file handling:



FDC Constraints

The FPGA design constraints (FDC) file contains constraints that the tool uses during synthesis. This FDC file includes both timing constraints and non-timing constraints in a single file.

- Timing constraints define performance targets to achieve optimal results. The constraints follow the Synopsys standard format, such as create_clock, set_input_delay, and set_false_path.
- Non-timing (or design constraints) define additional goals that help the tool optimize results. These constraints are unique to the FPGA synthesis tools and include constraints such as define_attribute, define_io_standard, and define_compile_point.

The recommended method to define constraints is to enter them in the SCOPE editor, and the tool automatically generates the appropriate syntax. If you define constraints manually, use the appropriate syntax for each type of constraint (timing or non-timing), as described above. See Methods for Creating Constraints, on page 354 for details on generating constraint files.

Prior to release G-2012.09M, designs used timing constraints in either legacy Synplify-style format or Synopsys standard format. You must do a one-time conversion on any existing SDC files to convert them to FDC files using the following command:

```
% sdc2fdc
```

sdc2fdc converts constraints as follows:

For legacy Synplify-style timing constraints	Converts timing constraints to Synopsys standard format and saves them to an FDC file.
For Synopsys standard timing constraints	Preserves Synopsys standard format timing constraints and saves them to an FDC file.
For non-timing or design constraints	Preserves the syntax for these constraints and saves them to an FDC file.

Once defined, the FDC file can be added to your project. Double-click this file from the Project view to launch the SCOPE editor to view and/or modify your constraints. See Converting SDC to FDC, on page 92 for details on how to run sdc2fdc.

Methods for Creating Constraints

Constraints are passed to the synthesis environment in FDC files using Tcl command syntax.

New Designs

For new designs, you can specify constraints using any of the following methods:

Definition Method	Description
SCOPE Editor (fdc file)–Recommended	Use this method to specify constraints wherever possible. The SCOPE editor automatically generates fdc constraints with the right syntax. You can use it for most constraints. See Chapter 6, *SCOPE Constraints Editor*, for information how to use SCOPE to automatically generate constraint syntax. Access: File->New->FPGA Design Constraints …
Manually-Entered Text Editor (fdc File, all other constraint files)	You can manually enter constraints in a text file. Make sure to use the correct syntax for the timing and design commands. The SCOPE GUI includes a TCL View with an advanced text editor, where you can manually generate the constraint syntax. For a description of this view, see TCL View, on page 385. You can also open any constraint file in a text editor to modify it.
Source Code Attributes/Directives (HDL files)	Directives must be entered in the source code because they affect the compiler. Do not include any other constraints in the source code, as this makes the source code less portable. In addition, you must recompile the design for the constraints to take effect. Attributes can be entered through the SCOPE interface, as they affect the mapper, not the compiler
Automatic— First Pass	Enable the Auto Constrain button in the Project view to have the tool automatically generate constraints based on inferred clocks. See set_false_path -from {{$en_regs}} -to {{i:dataout[31:0]}}, on page 340 in the *User Guide* for details. Use this method as a quick first pass to get an idea of what constraints can be set.

If there are multiple timing exception constraints on the same object, the software uses the guidelines described in Conflict Resolution for Timing Exceptions, on page 403 to determine the constraint that takes precedence.

See Also

To specify the correct syntax for the timing and design commands, see:

- Chapter 9, *Timing Constraint Syntax*
- Chapter 10, *FPGA Design Constraint Syntax*
- Chapter 11, *Synthesis Attributes and Directives*

Existing Designs

The SCOPE editor in this release does not save constraints to SDC files. For designs prior to G-2012.09M, it is recommended that you migrate your timing constraints to FDC format to take advantage of the tool's enhanced handling of these types of constraints. To migrate constraints, use the sdc2fdc command (see Converting SDC to FDC, on page 921) on your sdc files.

Note: If you need to edit an SDC file, either use a text editor, or double-click the file to open the legacy SCOPE editor. For information on editing older SDC files, see SCOPE User Interface (Legacy), on page 407 or Synplify-Style Timing Constraints (Legacy), on page 865.

See Also

To use the current SCOPE editor, see:

- Chapter 6, *SCOPE Constraints Editor*
- Chapter 4, *Specifying Constraints*

Constraint Translation

The tool includes a number of scripts for converting constraints to the correct format. The sdc2fdc script translates sdc files to fdc files. For constraints from vendor tools, you must first use the appropriate utility to translate the constraints to sdc, and then migrate them to fdc with the sdc2fdc script.

This table lists the translation scripts:

Vendor	Command/Utility	For details, see
Synopsys	sdc2fdc	sdc2fdc Tcl Shell Command, on page 770

For information about using these utilities, see:

- Converting SDC to FDC, on page 92

Constraint Checking

The synthesis tools include several features to help you debug and analyze design constraints. Use the constraint checker to check the syntax and applicability of the timing constraints in the project. The synthesis log file includes a timing report as well as detailed reports on the compiler, mapper, and resource usage information for the design. A stand-alone timing analyzer (STA) generates a customized timing report when you need more details about specific paths or want to modify constraints and analyze, without resynthesizing the design. The following sections provide more information about these features.

Constraint Checker

Check syntax and other pertinent information on your constraint files using Run->Constraint Check or the Check Constraints button in the SCOPE editor. This command generates a report that checks the syntax and applicability of the timing constraints that includes the following information:

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

See Constraint Checking Report, on page 450 for details.

Timing Constraint Report Files

The results of running constraint checking, synthesis, and stand-alone timing analysis are provided in reports that help you analyze constraints.

Use these files for additional timing constraint analysis:

File	Description
`_cck.rpt`	Lists the results of running the constraint checker (see Constraint Checking Report, on page 450).
`.ta`	Reports timing analysis results (see Generating Custom Timing Reports with STA, on page 331).
`.srr` or `.htm`	Reports post-synthesis timing results as part of the text or HTML log file (see Timing Reports, on page 441 and Log File, on page 435).

Database Object Search

To apply constraints, you have to search the database to find the appropriate objects. Sometimes you might want to search for and apply the same constraint to multiple objects. The FPGA tools provide some Tcl commands to facilitate the search for database objects:

Commands	Common Commands	Description
Find	Tcl Find, open_design…	Lets you search for design objects to form collections that can apply constraints to the group. See Using Collections, on page 76 and find Command (Batch), on page 1098.
Collections	define_collection, c_union…	Create, copy, evaluate, traverse, and filter collections. See Using Collections, on page 76 and Collection Commands, on page 1087 for more information.

Forward Annotation

The tool can automatically generate vendor-specific constraint files for forward annotation to the place-and-route tools when you enable the Write Vendor Constraints switch (on the Implementation Results tab) or use the -write_apr_constraint option of the set_option command.

Vendor	File Extension
Microsemi	_SDC.SDC

For information about how forward annotation is handled for your target technology, refer to the appropriate vendor chapter of the *FPGA Synthesis Reference Manual*.

Auto Constraints

Auto constraints are automatically generated by the synthesis tool, however, these do not replace regular timing constraints in the normal synthesis flow. Auto constraints are intended as a quick first pass to evaluate the kind of timing constraints you need to set in your design.

To enable this feature and automatically generate register-to-register constraints, use the Auto Constrain option on the left panel of the Project view. For details, see Using Auto Constraints, on page 341 in the *User Guide*.

CHAPTER 6

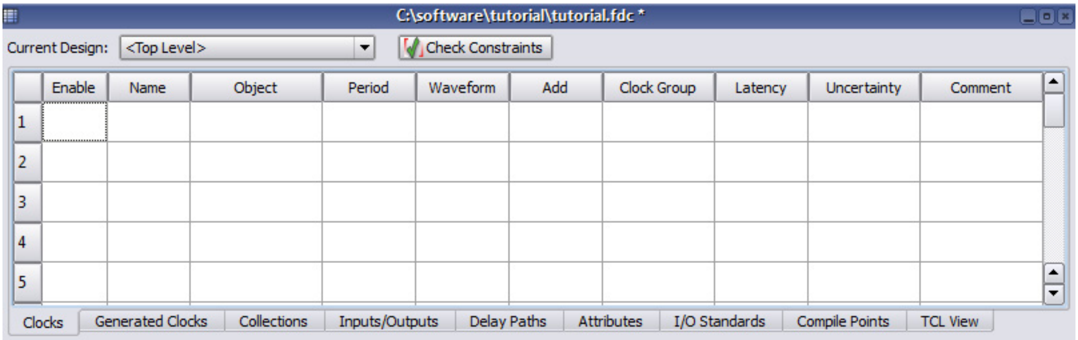
SCOPE Constraints Editor

The SCOPE (Synthesis Constraints Optimization Environment®) editor automatically generates syntax for synthesis constraints. Enter information in the SCOPE tabs, panels, columns, and pulldowns to define constraints and parameter values. You can also drag-and-drop objects from the HDL Analyst UI to populate values in the constraint fields.

This interface creates Tcl-format *Synopsys Standard timing constraints* and *Synplify-style design constraints* and saves the syntax to an FPGA design constraints (FDC) file that can automatically be added to your synthesis project. See Constraint Types, on page 348 for definitions of synthesis constraints.

SCOPE User Interface

The SCOPE editor contains a number of panels for creating and managing timing constraints and design attributes. This GUI offers the easiest way to create constraint files for your project. The syntax is saved to a file using an FDC extension and can be included in your design project.



C:\software\tutorial\tutorial.fdc *

Current Design: <Top Level>

	Enable	Name	Object	Period	Waveform	Add	Clock Group	Latency	Uncertainty	Comment
1										
2										
3										
4										
5										

From this editor, you specify timing constraints for clocks, ports, and nets as well as design constraints such as attributes, collections, and compile points. However, you cannot set black-box constraints from the SCOPE window.

To bring up the editor, use one of the following methods from the Project view:

- For a new file (the project file is open and the design is compiled):
  - Choose File->New-> FPGA Design Constraints; select FPGA Constraint File (SCOPE).
  - Click the SCOPE icon in the toolbar; select FPGA Constraint File (SCOPE).
- You can also open the editor using an existing constraint file. Double-click on the constraint file (FDC), or use File->Open, specifying the file type as FPGA Design Constraints File (`*.fdc`).

See Also:

- Using the SCOPE Editor, on page 52 in the *User Guide*.
- SCOPE User Interface (Legacy), on page 407

SCOPE Tabs

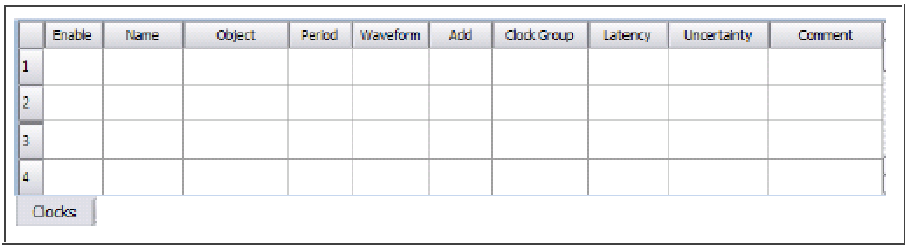
Here is a summary of the constraints created through the SCOPE editor:

SCOPE Panel	See...
Clocks	Clocks, on page 363
Generated Clocks	Generated Clocks, on page 369
Collections	Collections, on page 371
Inputs/Outputs	Inputs/Outputs, on page 373
Delay Paths	Delay Paths, on page 376
Attributes	Attributes, on page 379
I/O Standards	I/O Standards, on page 380
Compile Points	Compile Points, on page 382
TCL View	TCL View, on page 385

If you choose an object from a SCOPE pull-down 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 Naming Syntax, on page 886 and VHDL Naming Syntax, on page 887 for details.

Clocks

You use the Clocks panel of the SCOPE spreadsheet to define a signal as a clock.



	Enable	Name	Object	Period	Waveform	Add	Clock Group	Latency	Uncertainty	Comment
1										
2										
3										
4										

The Clocks panel includes the following options:

Field	Description
Name	Specifies the clock object name. Clocks can be defined on the following objects: • Pins • Ports • Nets For virtual clocks, the field must contain a unique name not associated with any port, pin, or net in the design.
Period	Specifies the clock period in nanoseconds. This is the minimum time over which the clock waveform repeats. The period must be greater than zero.
Waveform	Specifies the rise and fall edge times for the clock waveforms of the clock in nanoseconds, over an entire clock period. The first time in the list is a rising transition, typically the first rising transition after time zero. There must be two edges, and they are assumed to be rise and then fall. The edges must be monotonically increasing. If you do not specify this option, a default waveform is assumed, which has a rise edge of 0.0 and a fall edge of period/2.
Add	Specifies whether to add this clock to the existing clock or to overwrite it. Use this option when multiple clocks must be specified on the same source for simultaneous analysis with different clock waveforms. When you use this option, you must also specify the clock and the clocks with the same source must have different names.

Field	Description
Clock Group	Assigns clocks to asynchronous clock groupings. The clock grouping is inclusionary (for example, clk2 and clk3 can each be related to clk1 without being related to each other). For details, see Clock Groups, on page 365.
Latency	Specifies the clock latency applied to clock ports and clock aliases. Applying the latency constraint on a port can be used to model the off-chip clock delays in a multichip environment. Clock latency can only: • Apply to clocks defined on input ports. • Be used for source latency. • Apply to port clock objects.
Uncertainty	Specifies the clock uncertainty (skew characteristics) of the specified clock networks. You can only apply latency to clock objects.

Clock Groups

With this version of the software, clock grouping is associative-based; two clocks can be asynchronous to each other but both can be synchronous with a third clock.

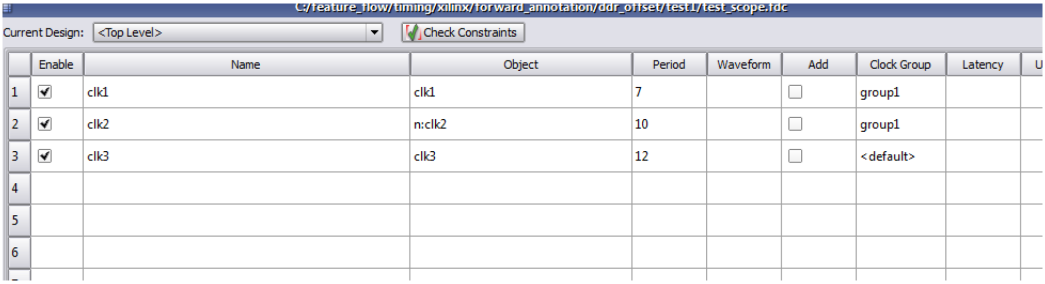
The SCOPE GUI prompts you for a clock group for each clock that you define, and, by default, assigns all clocks to the default clock group. When you add a name to the clock group that differs from the default clock group name, the clock is assigned its own clock group and is asynchronous to the default clock group as well as all other named clock groups.

This section presents scenarios for defining clocks and includes the following examples:

- *Example 1 – SCOPE Definition*
- *Example 2 – Equivalent Tcl Syntax*
- *Example 3 – Establish Clock Relationships*
- *Example 4 – Using a Single Group Option*
- *Example 6 – Legacy Clock Grouping*

Example 1 – SCOPE Definition

A design has three clocks: clk1, clk2, clk3 and you want clk1 and clk2 to be in the same clock groups—synchronous to each other but asynchronous to clk3. You can do this by adding a name in the Clock Group column in the Clocks tab of as shown below.



C:/feature_flow/timing/xilinx/forward_annotation/ddr_offset/test1/test_scope.fdc

Current Design: <Top Level> ▾ Check Constraints

	Enable	Name	Object	Period	Waveform	Add	Clock Group	Latency	U
1	☑	clk1	clk1	7		□	group1		
2	☑	clk2	n:clk2	10		□	group1		
3	☑	clk3	clk3	12		□	<default>		
4									
5									
6									

This specification assigns clk1 and clk2 to clock group group1, synchronous to each other and asynchronous to clk3. The equivalent command appears in the text editor window as:

```
set_clock_groups -derive -asynchronous -name {group1}
              -group { {c:clk1} {c:clk2} }
```

Example 2 – Equivalent Tcl Syntax

A design has three clocks: clk1, clk2, clk3. You can use the following commands to set clk2 synchronous to clk3, but asynchronous to clk1:

```
set_clock_groups –asynchronous –group [get_clocks {clk3 clk2}]

set_clock_groups –asynchronous –group [get_clocks {clk1}]
```

Example 3 – Establish Clock Relationships

A design has the following clocks defined:

```
create_clock -name {clka} {p:clka} -period 10 -waveform {0 5.0}
create_clock -name {clkb} {p:clkb} -period 20 -waveform {0 10.0}
create_clock -name {my_sys} {p:sys_clk} -period 200 -waveform {0
100.0}
```

The desired clock relationships are:

1. clka and clkb are asynchronous to each other
2. clka and clkb are synchronous to my_sys

For the tool to establish these relationships, multiple -group options are needed in a single set_clock_groups command. Clocks defined by the first –group option are asynchronous to clocks in the subsequent –group option. Therefore, you would use the following syntax to establish the relationships described above:

```
set_clock_groups -asynchronous -group [get_clocks {clka}]
            -group [get_clocks {clkb}]
```

Example 4 – Using a Single Group Option

set_clock_groups has a unique behavior when a single –group option is specified in the command. For example, in the following constraint specification:

```
set_clock_groups  -asynchronous -name {default_clkgroup_0} -group
[get_clocks {clka my_sys}]

set_clock_groups  -asynchronous -name {default_clkgroup_1} -group
[get_clocks {clkb my_sys}]
```

The first statement assigns clka AND my_sys as asynchronous to clkb, and the second statement assigns clkb AND my_sys as asynchronous to clka. Therefore, with this specification, all three clocks are established as asynchronous to each other.

Example 6 – Legacy Clock Grouping

This section shows how the legacy clock group definitions (Synplify-style timing constraints) are converted to the Synopsys standard timing syntax (FDC). Legacy clock grouping can be represented through Synopsys standard constraints, but the multi-grouping in the Synopsys standard constraints cannot be represented in legacy constraints. For example, the following legacy clock definitions:

```
define_clock -name{clka}{p:clka}-period 10
      -clockgroup default_clkgroup_0

define_clock -name {clkb}{p:clkb} -freq 150
      -clockgroup default_clkgroup_1

define_clock -name {clkc} {p:clkc} -freq 200
      -clockgroup default_clkgroup_1
```

... become these FDC clock definitions:

```
###==== BEGIN Clocks - (Populated from SCOPE tab, do not edit)

create_clock -name {clka} {p:clka} -period 10 -waveform {0 5.0}
create_clock -name {clkb} {p:clkb} -period 6.667
      -waveform {0 3.3335}
create_clock -name {clkc} {p:clkc} -period 5.0 -waveform {0 2.5}

set_clock_groups -derive -name default_clkgroup_0 -asynchronous
      -group  {c:clka}

set_clock_groups -derive -name default_clkgroup_1 -asynchronous
      -group  {c:clkb c:clkc}

###==== END Clocks
```

The create_generated_clock constraints used in legacy SDC are preserved in FDC. The -derive option directs the create_generated_clock command to inherit the -source clock group. This behavior is unique to FDC and is an extension of the Synopsys SDC standard functionality.

See Also

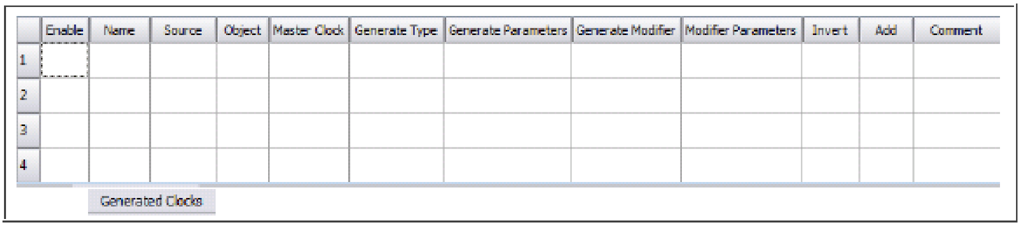
For equivalent Tcl syntax, see:

- create_clock, on page 835
- set_clock_groups, on page 841
- set_clock_latency, on page 843
- set_clock_uncertainty, on page 846

For information about all SCOPE panels, see SCOPE Tabs, on page 363.

Generated Clocks

Use the Generated Clocks panel of the SCOPE spreadsheet to define a signal as a generated clock. The equivalent Tcl constraint is create_generated_clock; its syntax is described in create_generated_clock, on page 837.



	Enable	Name	Source	Object	Master Clock	Generate Type	Generate Parameters	Generate Modifier	Modifier Parameters	Invert	Add	Comment
1												
2												
3												
4												

The Generated Clocks panel includes the following options:

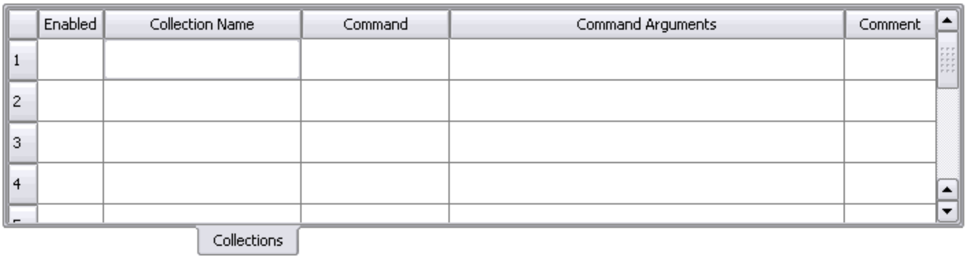
Field	Description
Name	Specifies the name of the generated clock. If this option is not used, the clock gets the name of the first clock source specified in the source.
Source	Specifies the master clock source (a clock source pin in the design), from which the clock waveform is derived. The actual delays (latency) for the generated clock are computed using its own source pins and not the master pin.
Object	Generated clocks can be defined on the following objects: • Pins • Ports • Nets • Instances—where instances have only one output
Master Clock	Specifies the master clock to be used for this generated clock, when multiple clocks fan into the master pin.
Generate Type	Specifies any of the following: edges – Specifies a list of integers that represents edges from the source clock that are to form the edges of the generated clock. The edges are interpreted as alternating rising and falling edges and each edge must not be less than its previous edge. The number of edges must be an odd number and not less than 3 to make one full clock cycle of the generated clock waveform. For example, 1 represents the first source edge, 2 represents the second source edge, and so on.
	divide_by – Specifies the frequency division factor. If the divide factor value is *2*, the generated clock period is twice as long as the master clock period.
	multiply_by – Specifies the frequency multiplication factor. If the multiply factor value is *3*, the generated clock period is one-third as long as the master clock period.
Generate Parameters	Specifies integers that define the type of generated clock.
Generate Modifier	Defines the secondary characteristics of the generated clock.

Field	Description
Modify Parameters	Defines modifier values of the generated clock.
Invert	Specifies whether to use invert – Inverts the generated clock signal (in the case of frequency multiplication and division).
Add	Either add this clock to the existing clock or overwrite it. Use this option when multiple generated clocks must be specified on the same source, because multiple clocks fan into the master pin. Ideally, one generated clock must be specified for each clock that fans into the master pin. If you specify this option, you must also specify the clock and master clock. The clocks with the same source must have different names.

For more information about other SCOPE options, see SCOPE Tabs, on page 363.

Collections

The Collections tab allows you to set constraints for a group of objects you have defined as a collection with the Tcl command. For details, see Creating and Using Collections (SCOPE Window), on page 77 of the *User Guide*.



	Enabled	Collection Name	Command	Command Arguments	Comment
1					
2					
3					
4					

Field	Description
Collection Name	Enter the collection name.

Field	Description
Command	Select a collection creation command from the drop-down menu. See Collection Commands, on page 1087 for descriptions of the commands.
Command Arguments	Specify the Tcl syntax for the constraint you want to apply to the collection.
Comment	Enter comments that are included in the constraints file.

You can crossprobe the collection results to an HDL Analyst view. To do this, right-click in the SCOPE cell and select the option Select in Analyst.

Collection Commands

You can use the collection commands on collections or Tcl lists. Tcl lists can be just a single element long.

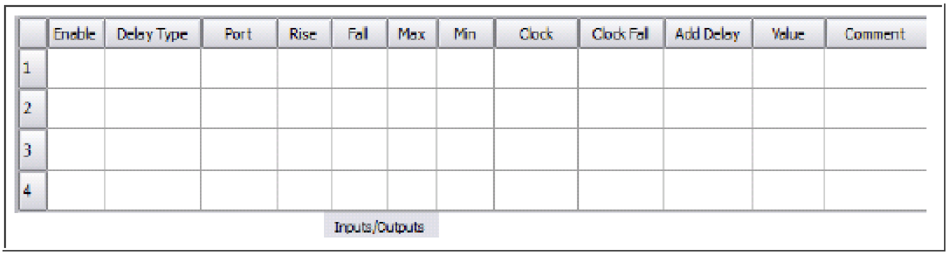
To...	Use this command...
Create a collection	set modules To create and save a collection, assign it to a variable. You can also use this command to create a collection from any combination of single elements, TCL lists and collections: set modules [define_collection {v:top} {v:cpu} $mycoll $mylist] Once you have created a collection, you can assign constraints to it in the SCOPE interface.
Copy a collection	set modules_copy $modules This copies the collection, so that any change to $modules does not affect $modules_copy.
Evaluate a collection	c_print This command returns all objects in a column format. Use this for visual inspection. c_list This command returns a Tcl list of objects. Use this to convert a collection to a list. You can manipulate a Tcl list with standard Tcl list commands.
Concatenate a list to a collection	c_union

To...	Use this command...
Identify differences between lists or collections	c_diff Identifies differences between a list and a collection or between two or more collections. Use the -print option to display the results.
Identify objects common to a list and a collection	c_intersect Use the -print option to display the results.
Identify objects common to two or more collections	c_sub Use the -print option to display the results.
Identify objects that belong exclusively to only one list or collection	c_symdiff Use this to identify unique objects in a list and a collection, or two or more collections. Use the -print option to display the results.

For information about all SCOPE panels, see SCOPE Tabs, on page 363.

Inputs/Outputs

The Inputs/Outputs panel models the interface of the FPGA with the outside environment. You use it to specify delays outside the device.



	Enable	Delay Type	Port	Rise	Fall	Max	Min	Clock	Clock Fall	Add Delay	Value	Comment
1												
2												
3												
4												

The Inputs/Outputs panel includes the following options:

Field	Description
Delay Type	Specifies whether the delay is an input or output delay.
Port	Specifies the name of the port.
Rise	Specifies that the delay is relative to the rising transition on specified port.
Fall	Specifies that the delay is relative to the falling transition on specified port
Max	Specifies that the delay value is relative to the longest path.
Min	Specifies that the delay value is relative to the shortest path.
Clock	Specifies the name of a clock for which the specified delay is applied. If you specify the clock fall, you must also specify the name of the clock.
Clock Fall	Specifies that the delay relative to the falling edge of the clock. For examples, see Input Delays, on page 374 and Output Delays, on page 375.
Add Delay	Specifies whether to add delay information to the existing input delay or overwrite the input delay. For examples, see Input Delays, on page 374 and Output Delays, on page 375.
Value	Specifies the delay path value.

Input Delays

Here is how this constraint applies for input delays:

- Clock Fall – The default is the rising edge or rising transition of a reference pin. If you specify clock fall, you must also specify the name of the clock.
- Add Delay – Use this option to capture information about multiple paths leading to an input port relative to different clocks or clock edges.

  For example, set_input_delay 5.0 -max -rise -clock phi1 {A} removes all maximum rise input delay from A, because the -add_delay option is not specified. Other input delays with different clocks or with -clock_fall are removed.

  In this example, the -add_delay option is specified as set_input_delay 5.0 -max -rise -clock phi1 -add_delay {A}. If there is an input maximum rise delay for A relative to clock phi1 rising edge, the larger value is used. The smaller value does not result in critical timing for maximum delay. For minimum delay, the smaller value is used. If there is maximum rise input delay relative to a different clock or different edge of the same clock, it remains with the new delay.

Output Delays

Here is how this constraint applies for output delays:

- Clock Fall – If you specify clock fall, you must also specify the name of the clock.
- Add Delay – By using this option, you can capture information about multiple paths leading from an output port relative to different clocks or clock edges.

  For example, the set_output_delay 5.0 -max -rise -clock phi1 {OUT1} command removes all maximum rise output delays from OUT1, because the -add_delay option is not specified. Other output delays with a different clock or with the -clock_fall option are removed.

  In this example, the -add_delay option is specified: set_output_delay 5.0 -max -rise -clock phi1 -add_delay {Z}. If there is an output maximum rise delay for Z relative to the clock phi1 rising edge, the larger value is used. The smaller value does not result in critical timing for maximum delay. For minimum delay, the smaller value is used. If there is a maximum rise output delay relative to a different clock or different edge of the same clock, it remains with the new delay.

See Also

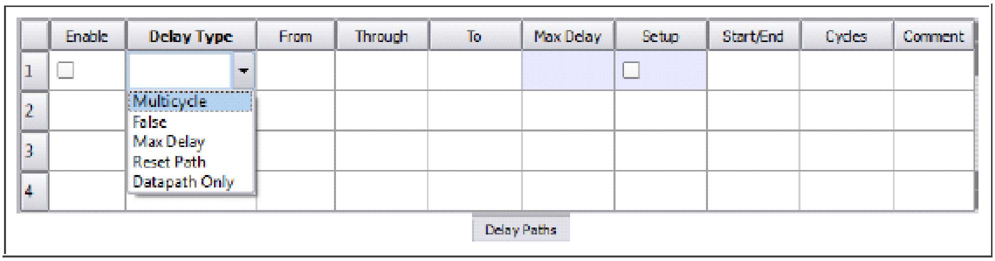
For equivalent Tcl syntax, see:

- set_input_delay, on page 850
- set_output_delay, on page 858

For information about all SCOPE panels, see SCOPE Tabs, on page 363.

Delay Paths

Use the Delay Paths panel to define the timing exceptions.



	Enable	Delay Type	From	Through	To	Max Delay	Setup	Start/End	Cycles	Comment
1	□						□			
2										
3										
4										

The Path Delay panel includes the following options:

Field	Description
Delay Type	Specifies the type of delay path you want the synthesis tool to analyze. Choose one of the following types: • Multicycle • False • Max Delay • Reset Path • Datapath Only
From	Starting point for the path. From points define timing start points and can be defined for clocks (c:), registers (i:), top-level input or bi-directional ports (p:), or black box output pins (i:). For details, see the following: • Defining From/To/Through Points for Timing Exceptions • Object Naming Syntax, on page 885
Through	Specifies the intermediate points for the timing exception. Intermediate points can be combinational nets (n:), hierarchical ports (t:), or instantiated cell pins (t:). If you click the arrow in a column cell, you open the Product of Sums (POS) interface where you can set through constraints. For details, see the following: • Product of Sums Interface • Defining From/To/Through Points for Timing Exceptions • Object Naming Syntax, on page 885
To	Ending point of the path. To points must be timing end points and can be defined for clocks (c:), registers (i:), top-level output or bi-directional ports (p:), or black box input pins (i:). For details, see the following: • Defining From/To/Through Points for Timing Exceptions • Object Naming Syntax, on page 885
Max Delay	Specifies the maximum delay value for the specified path in nanoseconds.

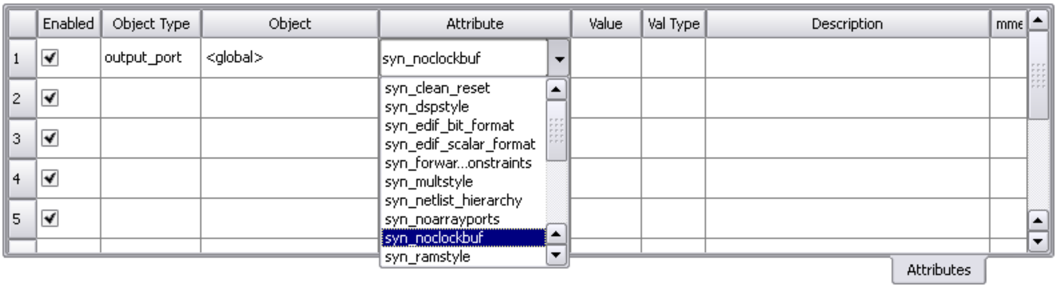
Field	Description
Setup	Specifies the setup (maximum delay) calculations used for specified path.
Start/End	Used for multicycle paths with different start and end clocks. This option determines the clock period to use for the multiplicand in the calculation for clock distance. If you do not specify a start or end clock, the end clock is the default.
Cycles	Specifies the number of cycles required for the multicycle path.

See Also

- For equivalent Tcl syntax, see:
  - set_multicycle_path, on page 855
  - set_false_path, on page 848
  - set_max_delay, on page 852
  - reset_path, on page 839
- For more information on timing exception constraints and how the tool resolves conflicts, see:
  - Delay Path Timing Exceptions, on page 390
  - Conflict Resolution for Timing Exceptions, on page 403
- For information about all SCOPE panels, see SCOPE Tabs, on page 363.

Attributes

You can assign attributes directly in the editor.



	Enabled	Object Type	Object	Attribute	Value	Val Type	Description	mme
1	☑	output_port	<global>	syn_noclockbuf				
2	☑							
3	☑							
4	☑							
5	☑							

Here are descriptions for the Attributes columns:

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 pull-down 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.
Attribute	(Required) Specifies the attribute name. You can choose from a pull-down 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 Synopsys FPGA synthesis tool filters the available choices in the Object field. You must select an attribute before entering a value. If a valid attribute does not appear in the pull-down list, simply type it in this field and then apply appropriate values.
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, by clicking in a cell and choosing from the pull-down menu.

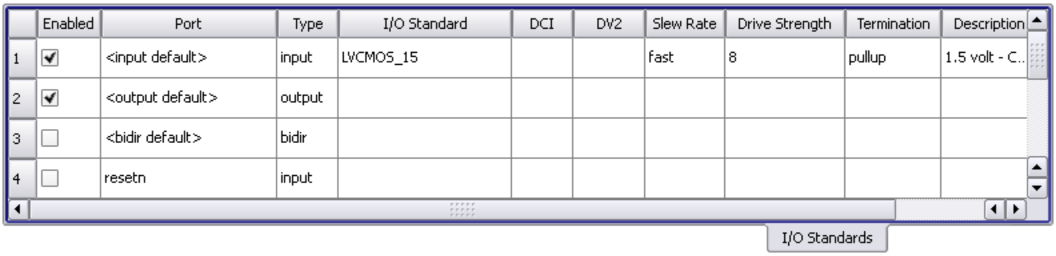
To specify an object to which you want to assign an attribute, you may also drag-and-drop it from the RTL or Technology view into a cell in the Object column. After you have entered the attributes, save the constraint file and add it to your project.

See Also

- For more information on specifying attributes, see How Attributes and Directives are Specified, on page 894.
- For information about all SCOPE panels, see SCOPE Tabs, on page 363.

I/O Standards

You can specify a standard I/O pad type to use in the design. Define an I/O standard for any port appearing in the I/O Standards panel.



	Enabled	Port	Type	I/O Standard	DCI	DV2	Slew Rate	Drive Strength	Termination	Description
1	☑	<input default>	input	LVCMOS_15			fast	8	pullup	1.5 volt - C..
2	☑	<output default>	output							
3	□	<bidir default>	bidir							
4	□	resetn	input							

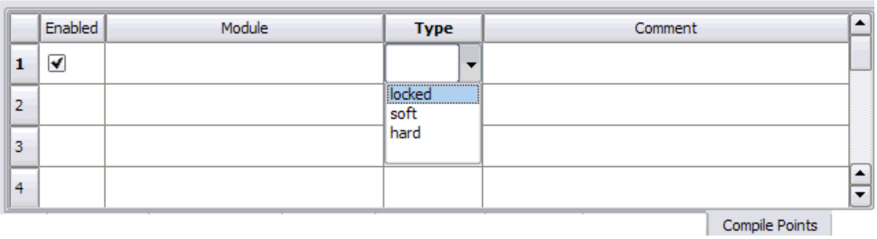
Field	Description
Enabled	(Required) Turn this on to enable the constraint, or off to disable a previous constraint.
Port	(Required) Specifies the name of the port. If you have initialized a compiled design, you can select a port name from the pull-down list. The first two entries let you specify global 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.
I/O Standard	Supported I/O standards by Synopsys FPGA products. See Industry I/O Standards, on page 387 for a description of the standards.
Slew Rate Drive Strength Termination Power Schmitt	The values for these parameters are based on the selected I/O standard.
Description	Describes the selected I/O Standard.
Comment	Enter comments about an I/O standard.

See Also

- For information about all SCOPE panels, see SCOPE Tabs, on page 363.

Compile Points

Use the Compile Points panel to specify compile points in your design, and to enable/disable them. This panel, available only if the device technology supports compile points, is used to define a top-level constraint file.



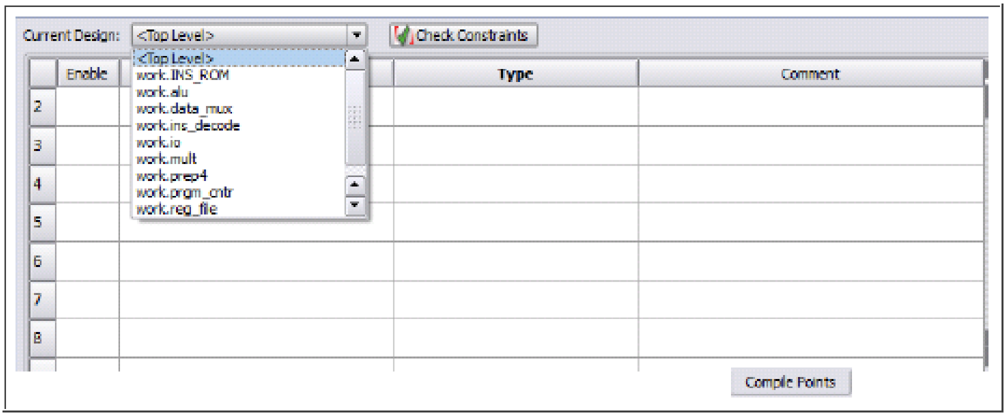
	Enabled	Module	Type	Comment
1	☑			
2				
3				
4				

Here are the descriptions of the fields in the Compile Points panel.

Field	Description
Enabled	(Required) Turn this on to enable the constraint.
Module	(Required) Specifies the name of the compile-point module. You can also drag-and-drop the compile-point module from the RTL Analyst view into this SCOPE field. The v: prefix must be specified to identify the module as a view. For example, v:alu.
Type	(Required) Specifies the type of compile point: • locked (default) – no timing reoptimization is done on the compile point. The hierarchical interface is unchanged and an interface logic model is constructed for the compile point. • soft – compile point is included in the top-level synthesis, boundary optimizations can occur. • hard – compile point is included in the top-level synthesis, boundary optimizations can occur, however, the boundary remains unchanged. Although, the boundary is not modified, instances on both sides of the boundary can be modified using top-level constraints. For details, see Compile Point Types, on page 419 in the *User Guide*.
Comment	Lets you enter a comment about the compile point.

Constraints for Compile Points

You can set constraints at the top-level or for modules to be used as the compile points from the Current Design pull-down menu shown below. Use the Compile Points tab to select compile points and specify their types.

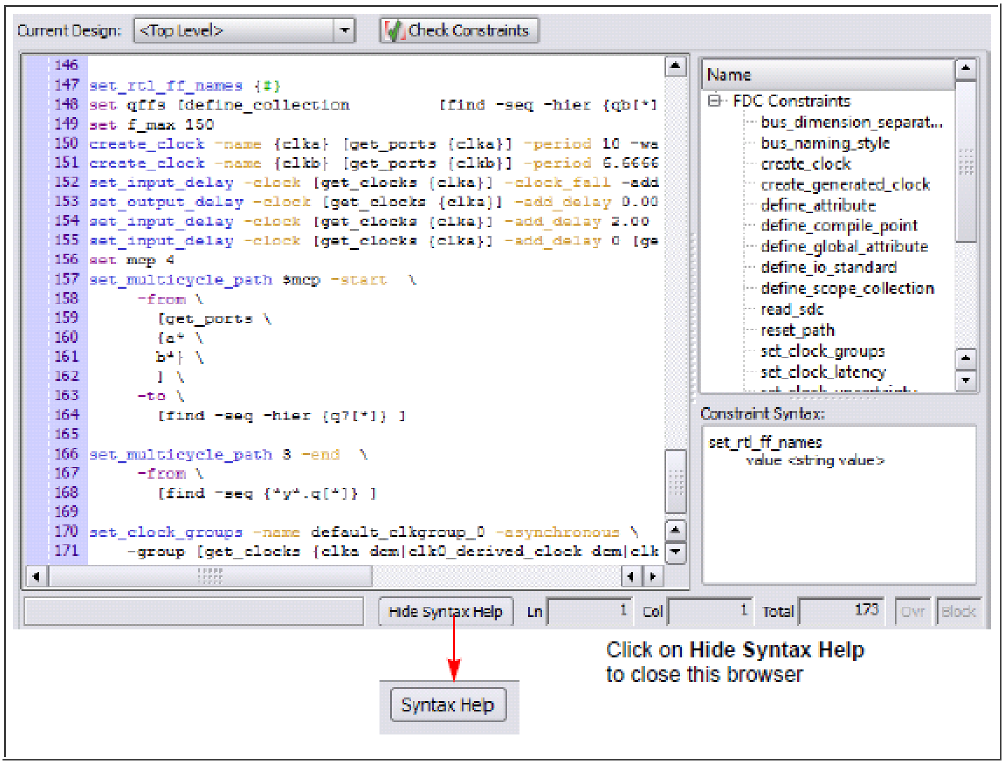


See Also

- The Tcl equivalent is define_compile_point.
- For more information on compile points and using the Compile Points panel, see Synthesizing Compile Points, on page 433 in the *User Guide*.
- For information about all SCOPE panels, see SCOPE Tabs, on page 363.

TCL View

The TCL View is an advanced text file editor for defining FPGA timing and design constraints.



```

set_rtl_ff_names {#}
set qffs [define_collection          [find -seq -hier {qb[*]
set f_max 150
create_clock -name {clka} [get_ports {clka}] -period 10 -wa
create_clock -name {clkb} [get_ports {clkb}] -period 6.6666
set_input_delay -clock [get_clocks {clka}] -clock_fall -add
set_output_delay -clock [get_clocks {clka}] -add_delay 0.00
set_input_delay -clock [get_clocks {clka}] -add_delay 2.00
set_input_delay -clock [get_clocks {clka}] -add_delay 0  [ge
set mcp 4
set_multicycle_path $mcp -start  \
     -from \
       [get_ports \
       {a* \
       b*} \
       ] \
     -to \
       [find -seq -hier {q?[*]} ]

set_multicycle_path 3 -end  \
     -from \
       [find -seq {*y*.q[*]} ]

set_clock_groups -name default_clkgroup_0 -asynchronous \
    -group [get_clocks {clka dcm|clk0_derived_clock dcm|clk
```

```
set_rtl_ff_names
    value <string value>
```

Click on **Hide Syntax Help** to close this browser

This text editor provides the following capabilities:

- Uses dynamic keyword expansion and tool tips for commands that
  - Automatically completes the command from a popup list
  - Displays complete command syntax as a tool tip
  - Displays parameter options for the command from a popup list
  - Includes a keyword command syntax help
- Checks command syntax and uses color indicators that
  - Validate commands and command syntax
  - Identifies FPGA design constraints and SCOPE legacy constraints

- Allows for standard editor commands, such as copy, paste, comment/un-comment a group of lines, and highlighting of keywords

For information on how to use this Tcl text editor, see Using the TCL View of SCOPE GUI, on page 63.

See Also

- For Tcl timing constraint syntax, see FPGA Timing Constraints, on page 834.
- For Tcl design constraint syntax, see Chapter 10, *FPGA Design Constraint Syntax*.
- You can also use the SCOPE editor to set attributes. See How Attributes and Directives are Specified, on page 894 for details.

Industry I/O Standards

The synthesis tool lets you specify a standard I/O pad type to use in your design. You can define an I/O standard for any port supported from the industry standard and proprietary I/O standards.

For industry I/O standards, see Industry I/O Standards, on page 387.

For vendor-specific I/O standards, see Microsemi I/O Standards, on page 1121.

Industry I/O Standards

The following table lists industry I/O standards.

I/O Standard	Description
AGP1X	Intel Corporation Accelerated Graphics Port
AGP2X	Intel Corporation Accelerated Graphics Port
BLVDS_25	Bus Differential Transceiver
CTT	Center Tap Terminated - EIA/JEDEC Standard JESD8-4
DIFF_HSTL_15_Class_I	1.5 volt - Differential High Speed Transceiver Logic - EIA/JEDEC Standard JESD8-6
DIFF_HSTL_15_Class_II	1.5 volt - Differential High Speed Transceiver Logic - EIA/JEDEC Standard JESD8-6
DIFF_HSTL_18_Class_I	1.8 volt - Differential High Speed Transceiver Logic - EIA/JEDEC Standard JESD8-9A
DIFF_HSTL_18_Class_II	1.8 volt - Differential High Speed Transceiver Logic - EIA/JEDEC Standard JESD8-9A
DIFF_SSTL_18_Class_II	1.8 volt - Differential Stub Series Terminated Logic - EIA/JEDEC Standard JESD8-6
DIFF_SSTL_2_Class_I	2.5 volt - Pseudo Differential Stub Series Terminated Logic - EIA/JEDEC Standard JESD8-9A
DIFF_SSTL_2_Class_II	2.5 volt - Pseudo Differential Stub Series Terminated Logic - EIA/JEDEC Standard JESD8-9A
GTL	Gunning Transceiver Logic - EIA/JEDEC Standard JESD8-3
GTL+	Gunning Transceiver Logic Plus
GTL25	Gunning Transceiver Logic - EIA/JEDEC Standard JESD8-3
GTL+25	Gunning Transceiver Logic Plus
GTL33	Gunning Transceiver Logic - EIA/JEDEC Standard JESD8-3
GTL+33	Gunning Transceiver Logic Plus

I/O Standard	Description
HSTL_12	1.2 volt - High Speed Transceiver Logic - EIA/JEDEC Standard JESD8-6
HSTL_15_Class_II	1.5 volt - High Speed Transceiver Logic - EIA/JEDEC Standard JESD8-6
HSTL_18_Class_I	1.8 volt - High Speed Transceiver Logic - EIA/JEDEC Standard JESD8-6
HSTL_18_Class_II	1.8 volt - High Speed Transceiver Logic - EIA/JEDEC Standard JESD8-6
HSTL_18_Class_III	1.8 volt - High Speed Transceiver Logic - EIA/JEDEC Standard JESD8-6
HSTL_18_Class_IV	1.8 volt - High Speed Transceiver Logic - EIA/JEDEC Standard JESD8-6
HSTL_Class_I	1.5 volt - High Speed Transceiver Logic - EIA/JEDEC Standard JESD8-6
HSTL_Class_II	1.5 volt - High Speed Transceiver Logic - EIA/JEDEC Standard JESD8-6
HSTL_Class_III	1.5 volt - High Speed Transceiver Logic - EIA/JEDEC Standard JESD8-6
HSTL_Class_IV	1.5 volt - High Speed Transceiver Logic - EIA/JEDEC Standard JESD8-6
HyperTransport	2.5 volt - Hypertransport - HyperTransport Consortium

I/O Standard	Description
LVCMOS_12	1.2 volt - EIA/JEDEC Standard JESD8-16
LVCMOS_15	1.5 volt - EIA/JEDEC Standard JESD8-7
LVCMOS_18	1.8 volt - EIA/JEDEC Standard JESD8-7
LVCMOS_25	2.5 volt - EIA/JEDEC Standard JESD8-5
LVCMOS_33	3.3 volt CMOS - EIA/JEDEC Standard JESD8-B
LVCMOS_5	5.0 volt CMOS
LVDS	Differential Transceiver - ANSI/TIA/EIA-644-95
LVDSEXT_25	Differential Transceiver
LVPECL	Differential Transceiver - EIA/JEDEC Standard JESD8-2
LVTTL	3.3 volt TTL - EIA/JEDEC Standard JESD8-B
MINI_LVDS	Mini Differential Transceiver
PCI33	3.3 volt PCI 33MHz - PCI Local Bus Spec. Rev. 3.0 (PCI Special Interest Group)
PCI66	3.3 volt PCI 66MHz - PCI Local Bus Spec. Rev. 3.0 (PCI Special Interest Group)
PCI-X_133	3.3 volt PCI-X - PCI Local Bus Spec. Rev. 3.0 (PCI Special Interest Group)
PCML	3.3 volt - PCML
PCML_12	1.2 volt - PCML
PCML_14	1.4 volt - PCML
PCML_15	1.5 volt - PCML
PCML_25	2.5 volt - PCML
RSDS	Reduced Swing Differential Signalling
SSTL_18_Class_I	1.8 volt - Stub Series Terminated Logic - EIA/JEDEC Standard JESD8-15
SSTL_18_Class_II	1.8 volt - Stub Series Terminated Logic - EIA/JEDEC Standard JESD8-15
SSTL_2_Class_I	2.5 volt - Stub Series Terminated Logic - EIA/JEDEC Standard JESD8-9B
SSTL_2_Class_II	2.5 volt - Stub Series Terminated Logic - EIA/JEDEC Standard JESD8-9B
SSTL_3_Class_I	3.3 volt - Stub Series Terminated Logic - EIA/JEDEC Standard JESD8-8
SSTL_3_Class_II	3.3 volt - Stub Series Terminated Logic - EIA/JEDEC Standard JESD8-8
ULVDS_25	Differential Transceiver

Delay Path Timing Exceptions

For details about the following path types, see:

- Multicycle Paths, on page 390
- False Paths, on page 393

Multicycle Paths

Multicycle paths 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 875. This section describes the following:

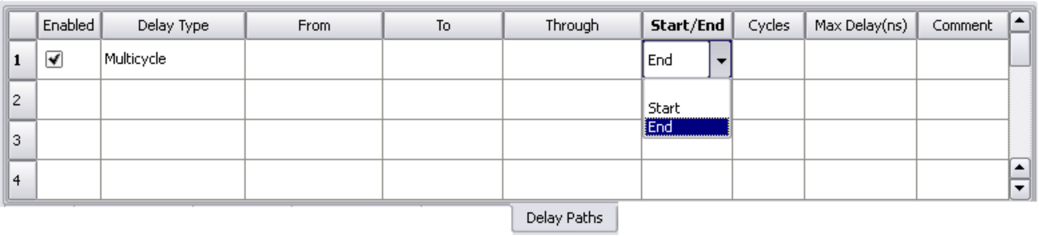
- Multi-cycle Path with Different Start and End Clocks, on page 390
- Multicycle Path Examples, on page 391

Multi-cycle Path with Different Start and End Clocks

The start/end option determines the clock period to use for the multiplicand in the calculation for required time. The following table describes the behavior of the multi-cycle path constraint using different start and end clocks. In all equations, *n* is number of clock cycles, and *clock_distance* is the default, single-cycle relationship between clocks that is calculated by the tool.

Basic required time for a multi-cycle path	clock_distance + [(n-1) * end_clock_period]
Required time with no end clock defined	clock_distance + [(n-1) * global_period]
Required time with -start option defined	clock_distance + [(n-1) * start_clock_period]
Required time with no start clock defined	clock_distance + [(n-1) * global_period]

If you do not specify a start or end option, by default the end clock is used for the constraint. Here is an example:

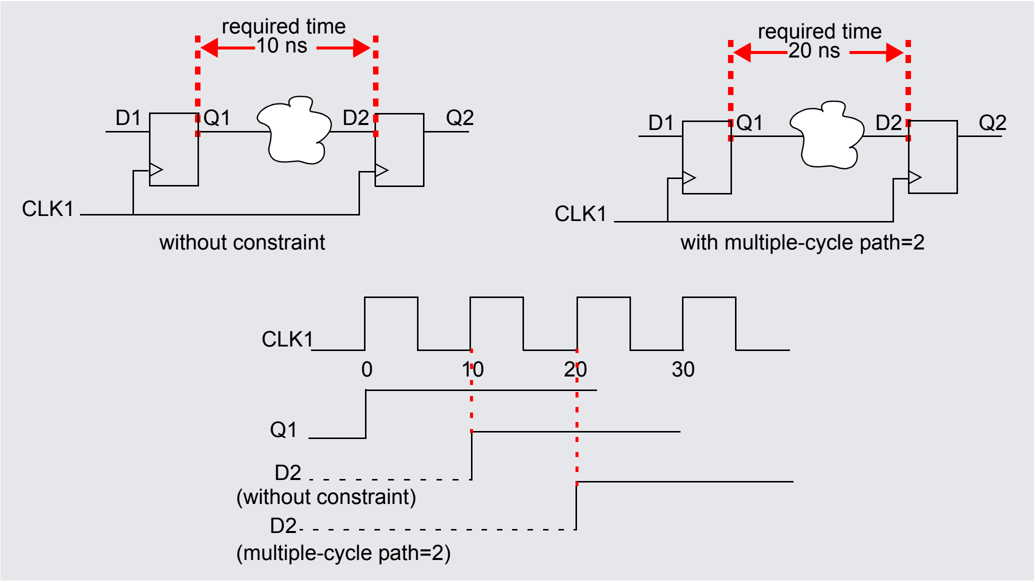


	Enabled	Delay Type	From	To	Through	Start/End	Cycles	Max Delay(ns)	Comment
1	☑	Multicycle				End			
2						Start			
3						End			
4									

Multicycle Path Examples

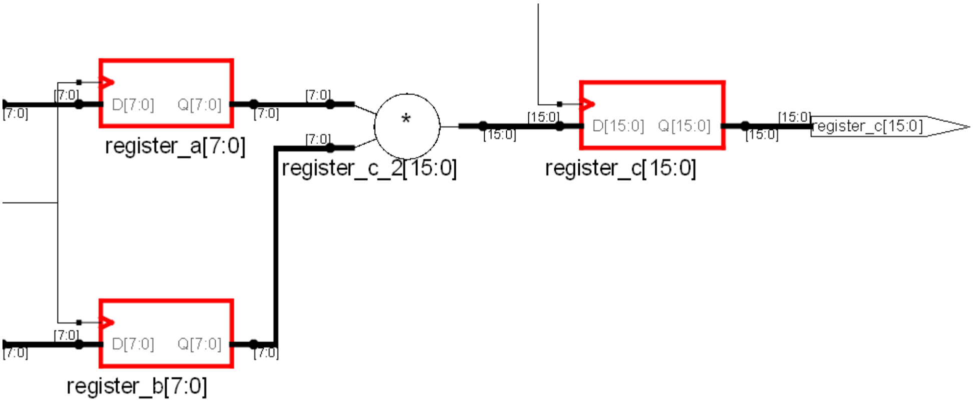
Multicycle Path Example 1

If you apply a multi-cycle path constraint from D1 to D2, the allowed time is #cycles x normal time between D1 and D2. 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.



Multicycle Path 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. The RTL view for this example is shown below. 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
set_multicycle_path -from register_a 2

# Paths from register_b use 2 clock cycles
set_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
set_multicycle_path -to register_c 2
```

False Paths

You use the Delay Paths constraint to specify clock paths that you want the 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 870.

This section describes the following:

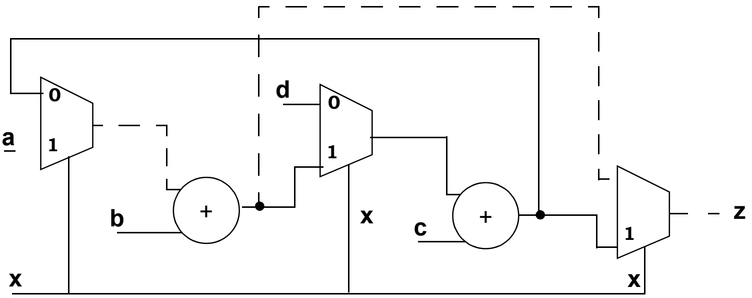
- Types of False Paths, on page 393
- Priority of False Path Constraints, on page 394

Types of 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.

Priority of False Path Constraints

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 Delay Paths pane of the SCOPE UI, or the following Tcl syntax:

  set_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 actions:
  - You assign clocks to different groups (Clocks pane of SCOPE UI).
  - You assign an implicit false path (by selecting the false option in the Delay (ns) column of the SCOPE Clock to Clock panel). (This condition applies for legacy timing constraints.)
  - You disable the Use clock period for unconstrained IO option (Project -> Implementation Options->Constraints).

  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.

False Path Constraint Examples

In this example, 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. You can do this in many ways:

- 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
  set_false_path -from {i: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
  set_false_path -to {i: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
  set_false_path -from {i:register_a} -to {i:register_c}
  ```

Specifying From, To, and Through Points

The following section describes from, to, and through points for timing exceptions specified by the multi-cycle paths, false paths, and max delay paths constraints.

- Timing Exceptions Object Types, on page 396
- From/To Points, on page 396
- Through Points, on page 398
- Product of Sums Interface, on page 399
- Clocks as From/To Points, on page 401

Timing Exceptions Object Types

Timing exceptions must contain the type of object in the constraint specification. You must explicitly specify an object type, n: for a net, or i: for an instance, in the instance name parameter of all timing exceptions. For example:

```
set_multicycle_path -from {i:inst2.lowreg_output[7]}
   -to {i:inst1.DATA0[7]} 2
```

If you use the SCOPE UI to specify timing exceptions, it automatically attaches the object type qualifier to the object name.

From/To Points

From specifies the starting point for the timing exception. To specifies the ending point for the timing exception. When you specify an object, use the appropriate prefix (see Object Naming Syntax, on page 885) to avoid confusion. The following table lists the objects that can serve as starting and ending points:

From Points	To Points
Clocks. See Clocks as From/To Points, on page 401 for more information.	Clocks. See Clocks as From/To Points, on page 401 for more information.
Registers	Registers
Top-level input or bi-directional ports	Top-level output or bi-directional ports
Instantiated library primitive cells (gate cells)	Instantiated library primitive cells (gate cells)
Black box outputs	Black box inputs

You can specify multiple from points in a single exception. This is most common when specifying exceptions that apply to all the bits of a bus. For example, you can specify constraints From A[0:15] to B – in this case, there is an exception, starting at any of the bits of A and ending on B.

Similarly, you can specify multiple to points in a single exception. If you specify both multiple starting points and multiple ending points such as From A[0:15] to B[0:15], there is actually an exception from any start point to any end point. In this case, the exception applies to all 16 * 16 = 256 combinations of start/end points.

Through Points

Through points are *limited to nets*; however, there are many ways to specify these constraints.

- Single Point
- Single List of Points
- Multiple Through Points
- Multiple Through Lists

You define these constraints in the appropriate SCOPE panels, or in the POS UI (see Product of Sums Interface, on page 399). When a port and net have the same name, preface the name of the through point with n: for nets, t: for hierarchical ports, and p: for top-level ports. For example n:regs_mem[2] or t:dmux.bdpol. The n: prefix must be specified to identify nets; otherwise, the associated timing constraint will not be applied for valid nets.

Single Point

You can specify a single through point. In this case, the constraint is applied to any path that passes through regs_mem[2]:

```
set_false_path -through regs_mem[2]
```

Single List of Points

If you specify a list of through points, the through option behaves as an OR function and applies to any path that passes through any of the points in the list. In the following example, the constraint is applied to any path through regs_mem[2] OR prgcntr.pc[7] OR dmux.alub[0] with a maximum delay value of 5 ns (-max 5):

```
set_path_delay
-through {regs_mem[2], prgcntr.pc[7], dmux.alub[0]}
-max 5
```

Multiple Through Points

You can specify multiple points for the same constraint by preceding each point with the -through option. In the following example, the constraint operates as an AND function and applies to paths through regs_mem[2] AND prgcntr.pc[7] AND dmux.alub[0]:

```
set_path_delay
-through regs_mem[2]
-through prgcntr.pc[7]
-through dmux.alub[0]
-max 5
```

Multiple Through Lists

If you specify multiple -through lists, the constraint is applied as an AND/OR function and is applied to the paths through all points in the lists. The following constraint applies to all paths that pass through {A_1 or A_2 or...A_n} AND {B_1 or B_2 or B_3}:

set_false_path –through {A_1 A_2...A_n} –through {B_1 B_2 B_3}

In this example,

```
set_multicycle_path
-through {net1, net2}
-through {net3, net4}
2
```

all paths that pass through the following nets are constrained at 2 clock cycles:

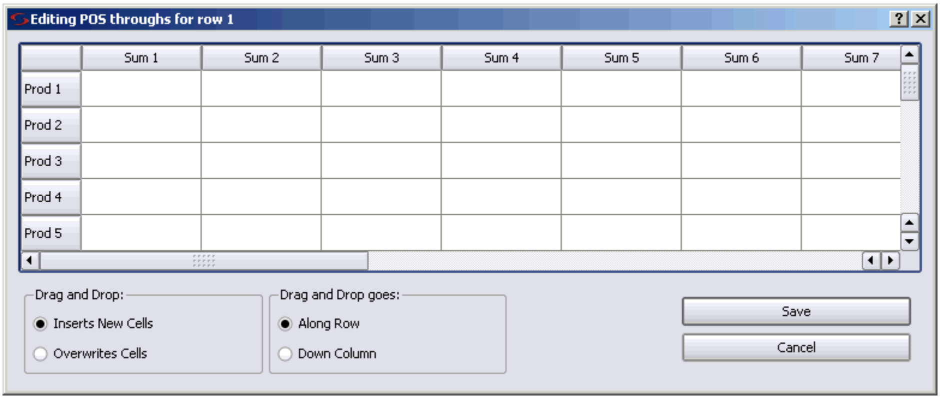
```
net1 AND net3
OR net1 AND net4
OR net2 AND net3
OR net2 AND net4
```

Product of Sums Interface

You can use the SCOPE UI to format -through points for nets with multi-cycle path, false path, and max delay path constraints in the Product of Sums (POS) interface of the SCOPE window. You can also manually specify constraints that use the -through option. For more information, see Defining From/To/Through Points for Timing Exceptions, on page 70 in the *User Guide*.

The POS interface is accessible by clicking the arrow in a Through column cell in the following SCOPE panels:

- Multi-Cycle Paths
- False Paths
- Delay Paths



	Sum 1	Sum 2	Sum 3	Sum 4	Sum 5	Sum 6	Sum 7
Prod 1							
Prod 2							
Prod 3							
Prod 4							
Prod 5							

Drag and Drop:
● Inserts New Cells
○ Overwrites Cells

Drag and Drop goes:
● Along Row
○ Down Column

Field	Description
Prod 1, 2, etc.	Type the first net name in a cell in a Prod row, or drag the net from a HDL Analyst view into the cell. Repeat this step along the same row, adding other nets in the Sum columns. The nets in each row form an OR list.
Sum 1, 2, etc.	Type the first net name in the first cell in a Sum column, or drag the net from a HDL Analyst view into the cell. Repeat this step down the same Sum column. The nets in each column form an AND list.
Drag and Drop Goes	Along Row - places objects in multiple Sum columns, utilizing only one Prod row. Down Column - places objects in multiple Prod rows, utilizing only one Sum column.
Drag and Drop	Inserts New Cells - New cells are created when dragging and dropping nets. Overwrites Cells - Existing cells are overwritten when dragging and dropping nets.
Save/Cancel	Saves or cancels your session.

Clocks as From/To Points

You can specify clocks as from/to points in your timing exception constraints. Here is the syntax:

define_*timing_exception* **-from | -to { c:***clock_name* [**:***edge*] **}**

where

- *timing_exception* is one of the following constraint types: multicycle_path, false_path, or path_delay
- **c:***clock_name***:***edge* is the name of the clock and clock edge (r or f). If you do not specify a clock edge, by default both edges are used.

See the following sections for details and examples on each timing exception.

Multicycle Path Clock Points

When you specify a clock as a from or to point, the multicycle path constraint applies to all registers clocked by the specified clock.

The following constraint allows two clock periods for all paths from the rising edge of the flip-flops clocked by clk1:

```
set_multicycle_path -from {c:clk1:r} 2
```

You cannot specify a clock as a through point. However, you can set a constraint from or to a clock and through an object (net, pin, or hierarchical port). The following constraint allows two clock periods for all paths to the falling edge of the flip-flops clocked by clk1 and through bit 9 of the hierarchical net:

```
set_multicycle_path -to {c:clk1:f} -through (n:MYINST.mybus2[9]} 2
```

False Path Clock Points

When you specify a clock as a from or to point, the false path constraint is set on all registers clocked by the specified clock. False paths are ignored by the timing analyzer. The following constraint disables all paths from the rising edge of the flip-flops clocked by clk1:

```
set_false_path -from {c:clk1:r}
```

You cannot specify a clock as a through point. However, you can set a constraint from or to a clock and through an object (net, pin, or hierarchical port). The following constraint disables all paths to the falling edge of the flip-flops clocked by clk1 and through bit 9 of the hierarchical net.

```
set_false_path -to {c:clk1:f} -through (n:MYINST.mybus2[9]}
```

Path Delay Clock Points

When you specify a clock as a from or to point for the path delay constraint, the constraint is set on all paths of the registers clocked by the specified clock. This constraint sets a max delay of 2 ns on all paths to the falling edge of the flip-flops clocked by clk1:

```
set_path_delay -to {c:clk1:f} -max 2
```

You cannot specify a clock as a through point, but you can set a constraint from or to a clock and through an object (net, pin, or hierarchical port). The next constraint sets a max delay of 0.2 ns on all paths from the rising edge of the flip-flops clocked by clk1 and through bit 9 of the hierarchical net:

```
set_path_delay -from {c:clk1:r} -through (n:MYINST.mybus2[9]} -max .2
```

Conflict Resolution for Timing Exceptions

The term *timing exceptions* refers to the false path, path delay, and multicycle path timing constraints. When the tool encounters conflicts in the way timing exceptions are specified through the constraint file, the software uses a set priority to resolve these conflicts. Conflict resolution is categorized into four levels, meaning that there are four different tiers at which conflicting constraints can occur, with one being the highest. The table below summarizes conflict resolution for constraints. The sections following the table provide more details on how conflicts can occur and examples of how they are resolved.

Conflict Level	Constraint Conflict	Priority	For Details, see ...
1	Different timing exceptions set on the same object.	1 – False Path 2 – Path Delay 3 – Multi-cycle Path	Conflicting Timing Exceptions, on page 404.
2	Timing exceptions of the same constraint type, using different semantics (from/to/through).	1 – From 2 – To 3 – Through	Same Constraint Type with Different Semantics, on page 405.
3	Timing exceptions of the same constraint type using the same semantic, but set on different objects.	1 – Ports/Instances/Pins 2 – Clocks	Same Constraint and Semantics with Different Objects, on page 406.
4	Identical timing constraints, except constraint values differ.	Tightest, or most constricting constraint.	Identical Constraints with Different Values, on page 406.

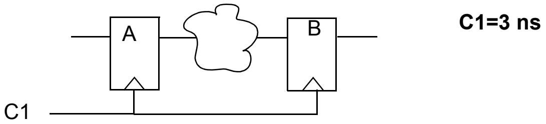
In addition to the four levels of conflict resolution for timing exceptions, there are priorities for the way the tool handles multiple I/O delays set on the same port and implicit and explicit false path constraints. For information on resolving these types of conflicts, see Priority of Multiple I/O Constraints, on page 420 and Priority of False Path Constraints, on page 394.

Conflicting Timing Exceptions

The first (and highest) level of resolution occurs when timing exceptions—false paths, path delay, or multi-cycle path constraints—conflict with each other. The tool follows this priority for applying timing exceptions:

1. False Path
2. Path Delay
3. Multicycle Path

For example:



```
set_false_path -from {c:C1:r}
set_path_delay -from {i:A} -to {i:B} -max 10
set_multicycle_path -from {i:A} -to {i:B} 2
```

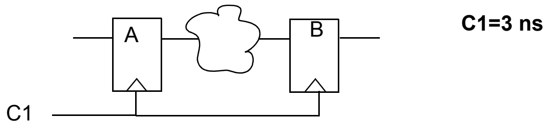
These constraints are conflicting because the path from A to B has three different constraints set on it. When the tool encounters this type of conflict, the false path constraint is honored. Because it has the highest priority of all timing exceptions, set_false_path is applied and the other timing exceptions are ignored.

Same Constraint Type with Different Semantics

The second level of resolution occurs when conflicts between timing exceptions that are of the same constraint type, use different semantics (from/to/through). The priority for these constraints is as follows:

1. From
2. To
3. Through

If there are two multicycle constraints set on the same path, one specifying a from point and the other specifying a to point, the constraint using -from takes precedence, as in the following example.



```
set_multicycle_path -from {i:A} 3
set_multicycle_path -to {i:B} 2
```

In this case, the tool uses:

```
set_multicycle_path -from {i:A} 3
```

The other constraint is ignored even though it sets a tighter constraint.

Same Constraint and Semantics with Different Objects

The third level resolves timing exceptions of the same constraint type that use the same semantic, but are set on different objects. The priority for design objects is as follows:

1. Ports/Instances/Pins
2. Clocks

If the same constraints are set on different objects, the tool ignores the constraint set on the clock for that path.

```
set_multicycle_path -from {i:mac1.datax[0]} -start 4
set_multicycle_path -from {c:clk1:r} 2
```

In the example above, the tool uses the first constraint set on the instance and ignores the constraint set on the clock from i:mac1.datax[0], even though the clock constraint is tighter.

For details on how the tool prioritizes multiple I/O delays set on the same port or implicit and explicit false path constraints, see Priority of False Path Constraints, on page 394 and Priority of Multiple I/O Constraints, on page 420.

Identical Constraints with Different Values

Where timing constraints are identical except for the constraint value, the tightest or most constricting constraint takes precedence. In the following example, the tool uses the constraint specifying two clock cycles:

```
set_multicycle_path -from {i:special_regs.trisa[7:0]} 2
set_multicycle_path -from {i:special_regs.trisa[7:0]} 3
```

SCOPE User Interface (Legacy)

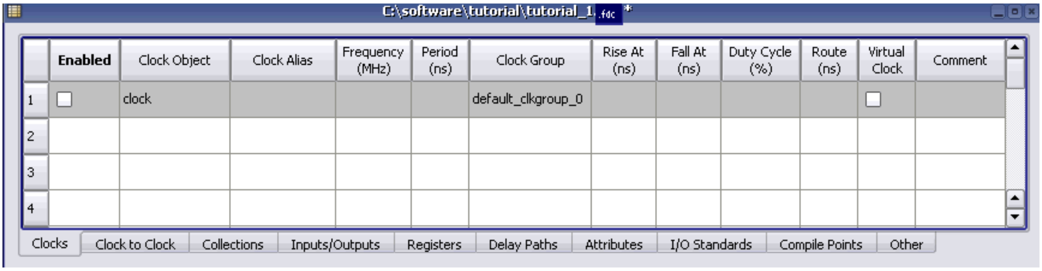
You can use the Legacy SCOPE editor for the SDC constraint files created before release version G-2012.09. However, it is recommended that you translate your SDC files to FDC files to enable the latest version of the SCOPE editor and to utilize the enhanced timing constraint handling in the tool. The latest version of the SCOPE editor automatically formats timing constraints using Synopsys Standard syntax (such as create_clock, and set_multicyle_path).

To do this, add your SDC constraint files to your project and run the following at the command line:

```
% sdc2fdc
```

This feature translates all SDC files in your project.

If you want to edit your existing SDC file, to open the legacy SCOPE editor, double-click on your constraint file in the Project view.



	Enabled	Clock Object	Clock Alias	Frequency (MHz)	Period (ns)	Clock Group	Rise At (ns)	Fall At (ns)	Duty Cycle (%)	Route (ns)	Virtual Clock	Comment
1	□	clock				default_clkgroup_0					□	
2												
3												
4												

For the Tcl command syntax created by the Legacy SCOPE editor, see:

- Synplify-Style Timing Constraints (Legacy), on page 865
- FPGA Design Constraint Syntax, on page 889
- How Attributes and Directives are Specified, on page 894

SCOPE Tabs (Legacy)

This section contains a description of the constraint panels in the SCOPE interface, and details about the constraints you can enter that are saved to an SDC file. SCOPE timing and design constraints include:

SCOPE Panel	**See...**
Clocks	Clocks (Legacy), on page 409
Clock to Clock	Clock to Clock (Legacy), on page 415
Collections	Collections, on page 371
Inputs/Outputs	Inputs/Outputs (Legacy), on page 417
Registers	Registers (Legacy), on page 421
Delay Paths	Delay Paths (Legacy), on page 422
Attributes	Attributes, on page 379
I/O Standards	I/O Standards, on page 380
Compile Points	Compile Points, on page 382
Other	Other (Legacy), on page 424

See Also:

- For the Tcl constraint syntax, see Timing Constraint Summary (Legacy), on page 409
- For design constraints, see Chapter 10, *FPGA Design Constraint Syntax*.

Timing Constraint Summary (Legacy)

The following table shows the correlation between each SCOPE panel and the Tcl constraint syntax it creates:

SCOPE Panel/GUI	Tcl .sdc File
Clocks (Legacy)	define_clock
Clock to Clock (Legacy)	define_clock_delay
Delay Paths (Legacy)	define_false_path
Delay Paths (Legacy)	define_multicycle_path
Delay Paths (Legacy)	define_path_delay
Inputs/Outputs (Legacy)	define_input_delay
Inputs/Outputs (Legacy)	define_output_delay
Registers (Legacy)	define_reg_input_delay
Registers (Legacy)	define_reg_output_delay
Frequency in the Project view Implementation Options -> Constraints Panel	set_option -frequency

Clocks (Legacy)

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 866.

For more information on using the Clocks panel, see Defining Clocks, on page 101 of the *User Guide*. For information about setting clock constraints automatically, see Using Auto Constraints, on page 341 in the *User Guide*.

Clocks Panel Description

The Clocks panel includes the following fields:

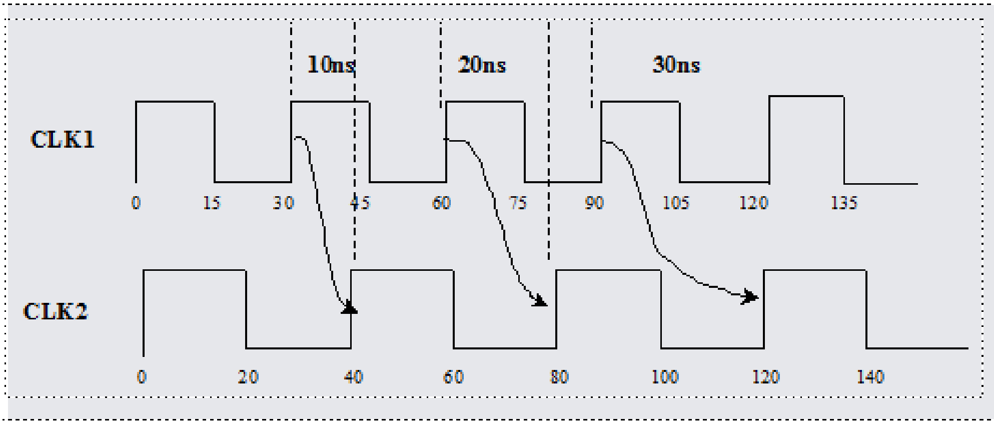
SCOPE Cell	Description
Enabled	Turn this on to enable the constraint. Turn it off to disable an existing constraint.
Clock Object	(Required) Specifies the clock object name. Clocks *can* be defined on the following objects: • Top-level input ports • Nets • Output pins of instantiated cells Clocks *cannot* be defined on the following objects: • Top-level output ports • Input pins of instantiated gates • Pins of inferred instances You can also assign a clock alias name to be used in the timing reports. See the Clock Alias option, below. For virtual clocks, the field must contain a unique name not associated with any port or instance in the design.
Clock Alias	Specifies a name for the clock if you want to use a name other than the object name. This alias name is used in the timing reports for the clock.
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 Attribute, on page 1001.
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 assigns each clock to a group called default_clkgroup_*n*, where *n* is a sequential number. • 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). See Clock Groups, on page 412 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 413 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 414 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.

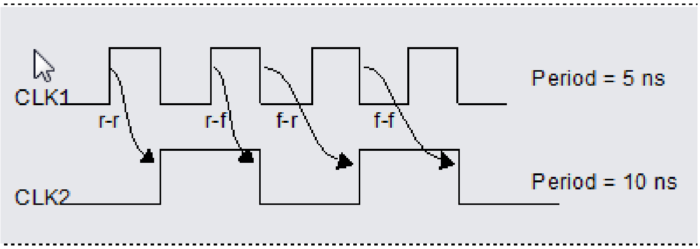
Clock Groups

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 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.



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.



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 separate clock groups. For most accurate results, re-assign clocks explicitly so that related clocks are in the same group, using the Clock Group field in the SCOPE spreadsheet.

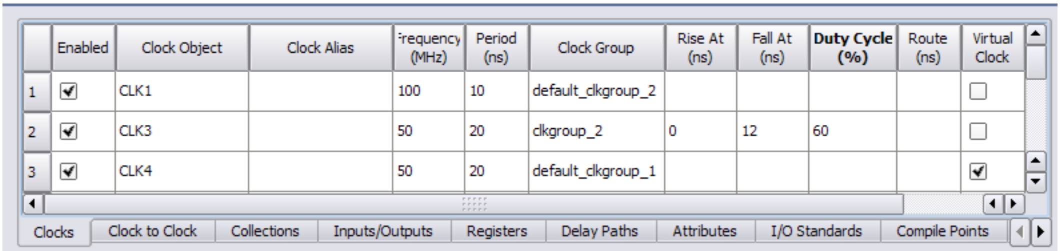
For example, if your design has three clocks (clk1, clk2, and clk3), by default they are each assigned to different clock groups. If clk1 and clk3 are related, assign them to the same clock group so that paths between the two clocks are analyzed and optimized. For timing optimization, the software treats them as related clocks. Keep clk2 in a separate group, as it is unrelated to the other two clocks. 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 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.

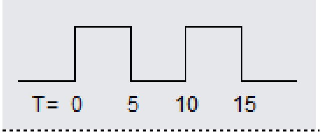
To understand how the relationships between source and destination clocks are computed using the Rise At and Fall At values, consider the following example.



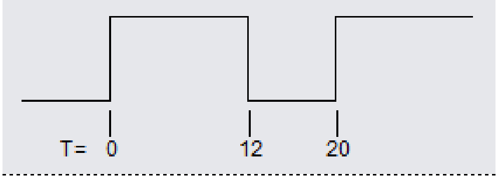
	Enabled	Clock Object	Clock Alias	Frequency (MHz)	Period (ns)	Clock Group	Rise At (ns)	Fall At (ns)	**Duty Cycle (%)**	Route (ns)	Virtual Clock
1	☑	CLK1		100	10	default_clkgroup_2					□
2	☑	CLK3		50	20	clkgroup_2	0	12	60		□
3	☑	CLK4		50	20	default_clkgroup_1					☑

In this example:

- CLK1 is a clock with a period of 10 ns (100 MHz) in clock group default_clkgroup_2. 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 clkgroup_2. 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.

Check the Performance Summary for warnings about inferred clocks; an inferred clock in the summary may mean a declared clock is missing. The Performance Summary can also identify the alias names of derived clocks, for which timing exceptions (false path, multicycle path, or path delay) that target the derived clocks are needed.

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 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.

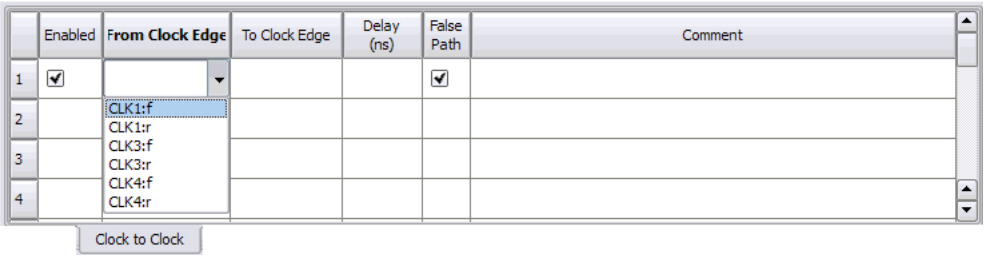
```
# In this example, input_a has a 10.0 ns input delay.
# Rerun synthesis with an added 1.8 ns constraint, to
# improve accuracy and match post-place-and-route results.

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 (Legacy)

Defines the time available from data signals on all paths between the specified clock edges; only specified clock edge combinations are affected. For example, if you specify a new relationship for the rise-to-rise clock edge, other relationships are not affected. By default, the software automatically calculates clock delay based on the clock parameters defined in the Clock 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 869.



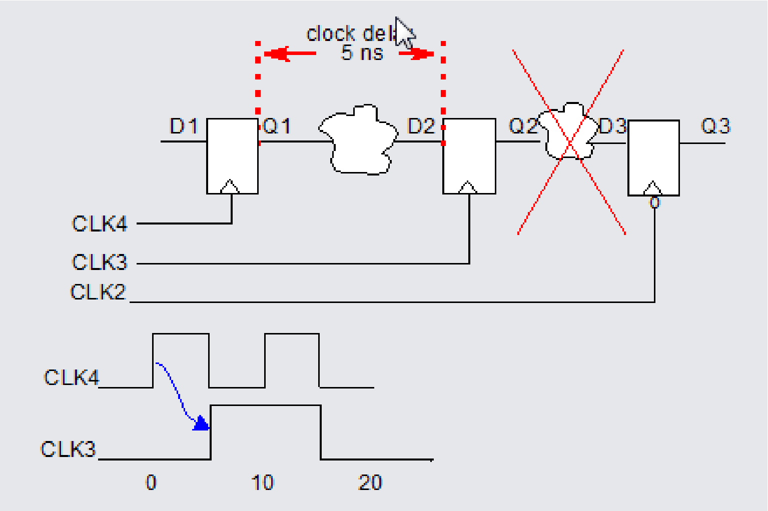
	Enabled	From Clock Edge	To Clock Edge	Delay (ns)	False Path	Comment
1	☑				☑	
2						
3						
4						

Note: Maintenance for the define_clock_delay command is limited. It is recommended that you use the define_false_path and define_path_delay commands instead, since these commands support clock aliases.

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). For example, CLK1:r.
To Clock Edge	Clock edge that triggers the event of the ending point: clock_name:r or f (rise or fall). For example, CLK3:f
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 75 in the *User Guide* for details.
False Path	You can enable the False Path checkbox 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 where a 5 ns delay constraint is placed on CLK4 rising to CLK3 rising, and an implicit false path constraint is placed on CLK3 rising to CLK2 falling.

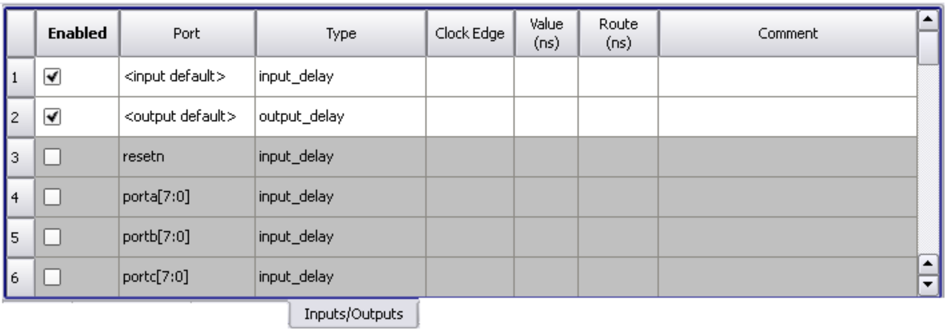


Inputs/Outputs (Legacy)

This panel models the interface of the FPGA with the outside environment. You use it to specify delays outside the device. The equivalent Tcl constraints are define_input_delay, on page 873 and define_output_delay, on page 879. The default delay outside an FPGA is 0.0 ns.

You can specify multiple constraints on the same I/O port. See Priority of Multiple I/O Constraints, on page 420 for details.

For information about auto-constraining I/O paths, see Using Auto Constraints, on page 341 in the *User Guide*.



	Enabled	Port	Type	Clock Edge	Value (ns)	Route (ns)	Comment
1	☑	<input default>	input_delay				
2	☑	<output default>	output_delay				
3	□	resetn	input_delay				
4	□	porta[7:0]	input_delay				
5	□	portb[7:0]	input_delay				
6	□	portc[7:0]	input_delay				

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 pull-down 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.
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 419 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 Synopsys FPGA 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.

Priority of 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

  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:

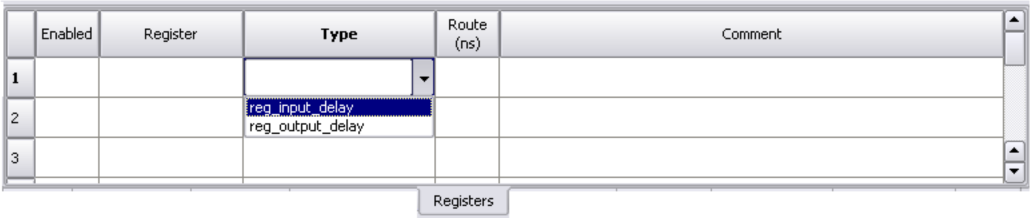
  ```
  a[3:0]3 clk1:r
  a[3:0]3 clk2:r
  a[0]2 clk1:r
  ```

  In this case, port a[0] only gets one constraint of 2 ns. Ports a[1], a[2], and a[3] get two constraints of 3 ns each.
- If at any given level (bit, 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 port a[0] is ignored.

  ```
  a[0]2 clk1:r
  a[0]1
  ```

Registers (Legacy)

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 884, and define_reg_output_delay, on page 885 for the equivalent Tcl commands.



The Registers SCOPE panel includes the following fields:

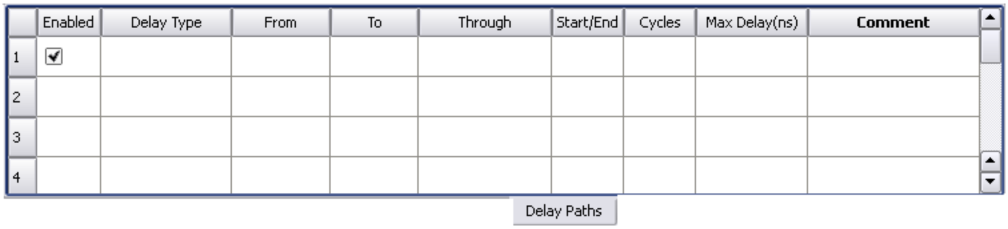
Field	Description
Enabled	(Required) Turn this on to enable the constraint.
Register	(Required) Specifies the name of the register. If you have initialized a compiled design, you can choose from the pull-down list.
Type	(Required) Specifies whether the delay is an input or output delay.
Route	(Required) 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 419.
Comment	Lets you enter comments that are included in the constraints file.

Delay Paths (Legacy)

Use the Delay Paths panel to specify the following timing exceptions:

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

For more details on specifying these constraints, see Delay Path Timing Exceptions, on page 390.



	Enabled	Delay Type	From	To	Through	Start/End	Cycles	Max Delay(ns)	**Comment**
1	☑								
2									
3									
4									

This table provides brief descriptions of the fields in the 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
Enabled	(Required) Turn this on to enable the constraint.
Delay Type	Specify the type of delay path you want the synthesis tool to analyze. Choose one of the following types: • Multicycle • False • Max Delay
From	Starting point for the path. From points define timing start points and can be clocks (c:), registers (i:), top-level input or bi-directional ports (p:), or black box outputs (i:). For details, see the following: • False Paths, on page 393 • syn_black_box Directive, on page 921 • Defining From/To/Through Points for Timing Exceptions, on page 70 in the *User Guide*

SCOPE Cell	Description
To	Ending point of the path. To points must be timing end points and can be: clocks (c:), registers (i:), top-level output or bi-directional ports (p:), or black box inputs (i:). For details, see the following: • syn_black_box Directive, on page 921 • False Paths, on page 393 • Defining From/To/Through Points for Timing Exceptions, on page 70 in the *User Guide*
Through	Specifies the intermediate points for the timing exception. Intermediate points can be combinational nets (n:), hierarchical ports (t:), or instantiated cell pins (t:). If you click the arrow in a column cell, you open the Product of Sums (POS) interface where you can set through constraints. For details, see the following: • syn_black_box Directive, on page 921 • False Paths, on page 393 • Product of Sums Interface, on page 399 • Defining From/To/Through Points for Timing Exceptions, on page 70 in the *User Guide*
Start/End	Used for multi-cycle paths with different start and end clocks. This option determines the clock period to use for the multiplicand in the calculation for clock distance. If you do not specify a start or end clock, the end clock is the default. For further details, see Multi-cycle Path with Different Start and End Clocks, on page 390 and Defining Multi-cycle Paths, on page 74 in the *User Guide.*
Cycles	Number of cycles for the required time calculated for the multi-cycle path.
Max Delay (ns)	Maximum delay value in nanoseconds.
Comment	Lets you enter comments that are included in the constraints file.

Other (Legacy)

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.

See Specifying Timing Exceptions, on page 70 in the *User Guide* for information.

CHAPTER 7

Input and Result 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.

Extension	File	Description
`.adc`	Analysis Design Constraint	Contains timing constraints to use for stand-alone timing analysis. Constraints in this file are used only for timing analysis and do not change the result files from synthesis. Constraints in the .adc file are applied in addition to .fdc constraints used during synthesis. Therefore, .adc constraints affect timing results only if there are no conflicts with .fdc constraints. You can forward annotate adc constraints to your vendor constraint file without rerunning synthesis. See Using Analysis Design Constraints, on page 334 of the *User Guide* for details.
`.fdc`	Synopsys FPGA Design Constraint	Create FPGA timing and design constraints with SCOPE. You can run the sdc2fdc utility to translate legacy FPGA timing constraints (SDC) to Synopsys FPGA timing constraints (FDC). For details, see the sdc2fdc Tcl Shell Command, on page 770.
`.ini`	Configuration and Initialization	Governs the behavior of the synthesis tool. You normally do *not* need to edit this file. For example, use the HDL Analyst Options dialog box, instead, to customize behavior. See HDL Analyst Options Command, on page 263. On the Windows 7 platforms, the .ini file is in the C:\Users\userName\AppData\Roaming\Synplicity directory, and on the Windows XP platforms, the .ini file is in the C:\Documents and Settings\userName\Application Data\Synplicity directory. On Linux workstations, the .ini file is in the following directory: (~/.synplicity, where ~ is your home directory, which can be set with the environment variable $HOME).
`.prj`	Project	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.

Extension	File	Description
`.sdc`	Constraint	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 creating the .sdc file, see SCOPE Tabs, on page 363.
`.vhd`	Source files (VHDL)	Design source files in VHDL format. See VHDL, on page 428 and Chapter 10, *VHDL Language Support* for details. For information about using VHDL and Verilog files together in a design, see Using Mixed Language Source Files, on page 40 of the *User Guide.*
`.v`	Source files (Verilog)	Design source files in Verilog format. For more information about the Verilog language, and the synthesis commands and attributes you can include, see Verilog, on page 429, Chapter 8, *Verilog Language Support*, and Chapter 9, *SystemVerilog Language Support*. For information about using VHDL and Verilog files together in a design, see Using Mixed Language Source Files, on page 40 of the *User Guide.*
`.sv`	Source files (Verilog)	Design source files in SystemVerilog format. The `sv` source file is added to the Verilog directory in the Project view. For more information about the Verilog and SystemVerilog languages, and the synthesis commands and attributes you can include, see Verilog, on page 429, Chapter 8, *Verilog Language Support*, and Chapter 9, *SystemVerilog Language Support*. For information about using VHDL and Verilog files together in a design, see Using Mixed Language Source Files, on page 40 of the *User Guide.*

HDL Source Files

The HDL source files for a project can be in either VHDL (`vhd`), Verilog (`v`), or SystemVerilog (`sv`) format.

The Synopsys FPGA 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 Synopsys FPGA 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 Synopsys FPGA synthesis tool library contains an attributes package (*installDirectory*/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*.

Verilog

The Synopsys FPGA synthesis tool supports a synthesizable subset of Verilog 2001 and Verilog 95 (IEEE 1364) and SystemVerilog extensions. For more information about the Verilog language, and the synthesis commands and attributes you can include, see Chapter 8, *Verilog Language Support* and Chapter 9, *SystemVerilog Language Support*.

The Synopsys FPGA 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.

Libraries

You can instantiate components from a library, which can be either in Verilog or VHDL. For example, you might have technology-specific or custom IP components in a library, or you might have generic library components. The *installDirectory*/lib directory included with the software contains some component libraries you can use for instantiation.

There are two kinds of libraries you can use:

- Technology-specific libraries that contain I/O pad, macro, or other component descriptions. The lib directory lists these kinds of libraries under vendor sub-directories. The libraries are named for the technology family, and in some cases also include a version number for the version of the place-and-route tool with which they are intended to be used.

  For information about using vendor-specific libraries to instantiate LPMs, PLLs, macros, I/O pads, and other components, refer to the appropriate sections in Chapter 11, *Optimizing for Microsemi Designs* in the *User Guide*.

- Technology-independent libraries that contain common components. You can have your own library or use the one Synopsys provides. The Synplicity library is a Verilog library of common logic elements, much like the Synopsys® GTECH component library. See The Synplicity Generic Technology Library, on page 430 for a description of this library.

The Synplicity Generic Technology Library

The synthesis software includes this Verilog library for generic components under the *installDirectory*/lib/generic_technology directory. Currently, the library is only available in Verilog format. The library consists of technology-independent common logic elements, which help the designer to develop technology-independent parts. The library models extract the functionality of the component, but not its implementation. During synthesis, the mappers implement these generic components in implementations that are appropriate to the technology being used.

To use components from this directory, add the library to the project by doing either of the following:

- Add add_file -verilog "$LIB/generic_technology/gtech.v to your prj file or type it in the Tcl window.
- In the tool window, click the Add file button, navigate to the *installDirectory*/lib/generic_technology directory and select the gtech.v file.

When you synthesize the design, the tool uses components from this library.

You cannot use the Synplicity generic technology library together with other generic libraries, as this could result in a conflict. If you have your own GTECH library that you intend to use, do not use the Synplicity generic technology library.

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.

Extension	File	Description
`_cck.rpt`	Constraint Checker Report	Checks the syntax and applicability of the timing constraints in the .fdc file for your project and generates a report (*projectName*_cck.rpt). See Constraint Checking Report, on page 450 for more information.
`.info`	Design component files	Design-dependent. Contains detailed information about design components like state machines or ROMs.
`.fse`	FSM information file	Design-dependent. Contains information about encoding types and transition states for all state machines in the design.
.pfl	Message Filter criteria	Output file created after filtering messages in the Messages window. See Updating the projectName.pfl file, on page 255 in the *User Guide*.
Results file: • `.edf` • `.edn`	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 tool you are using. Generally, the format is EDIF, but there could be vendor-specific formats, like the Microsemi `.edf` format. Specify this file on the Implementation Results panel of the Implementation Options dialog box (Implementation Results Panel, on page 163).

Extension	File	Description
`run_options.txt`	Project settings for implementations	This file is created when a design is synthesized and contains the project settings and options used with the implementations. These settings and options are also processed for displaying the Project Status view after synthesis is run. For details, see Project Status View, on page 47.
`.sap`	Synplify Annotated Properties	This file is generated after the Annotated Properties for Analyst option is selected in the Device panel of the Implementation Options dialog box. After the compile stage, the tool annotates the design with properties like clock pins. You can find objects based on these annotated properties using Tcl Find. For more information, see find Command (Batch), on page 1098Using the Tcl Find Command to Define Collections, on page 83 in the *User Guide*.
`.sar`	Archive file	Output of the Synopsys FPGA Archive utility in which design project files are stored into a single archive file. Archive files use Synplicity Proprietary Format. See Archive Project Command, on page 150 for details on archiving, unarchiving and copying projects.
`_scck.rpt`	Constraint Checker Report	Generates a report that contains an overview of the design information, such as, the top-level view, name of the constraints file, if there were any constraint syntax issues, and a summary of clock specifications. For details about any constraint issues, see the companion cck.rpt file that is also created at this time.
`.srd`	Intermediate mapping files	Used to save mapping information between synthesis runs. You do not need to use these files.

Extension	File	Description
.srm	Mapping output files	Output file after mapping. It contains the actual technology-specific mapped design. This is the representation that appears graphically in a Technology view.
.srr	Synthesis log file	Provides information on the synthesis run, as well as area and timing reports. See Log File, on page 435, for more information.
.srs	Compiler output file	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.
synlog folder	Intermediate technology mapping files	This folder contains intermediate netlists and log files after technology mapping has been run. Timestamp information is contained in these netlist files to manage jobs with up-to-date checks. For more information, see Using Up-to-date Checking for Job Management, on page 234.
synwork folder	Intermediate pre-mapping files	This folder contains intermediate netlists and log files after pre-mapping has been run. Timestamp information is contained in these netlist files to manage jobs with up-to-date checks. For more information, see Using Up-to-date Checking for Job Management, on page 234.
.ta	Customized Timing Report	Contains the custom timing information that you specify through Analysis->Timing Analyst. See Analysis Menu, on page 226, for more information.

Extension	File	Description
`_ta.srm`	Customized mapping output file	Creates a customized output netlist when you generate a custom timing report with HDL Analyst->Timing Analyst. It contains the representation that appears graphically in a Technology view. See Analysis Menu, on page 226 for more information.
`.tap`	Timing Annotated Properties	This file is generated after the Annotated Properties for Analyst option is selected in the Device panel of the Implementation Options dialog box. After the compile stage, the tool annotates the design with timing properties and the information can be analyzed in the RTL view. You can also find objects based on these annotated properties using Tcl Find. For more information, see Using the Tcl Find Command to Define Collections, on page 83 in the *User Guide*.
`.tlg`	Log file	This log file contains a list of all the modules compiled in the design.
vendor constraint file	Constraints file for forward annotation	Contains synthesis constraints to be forward-annotated to the place-and-route tool. The constraint file type varies with the vendor and the technology. Refer to the vendor chapters for specific information about the constraints you can forward-annotate. Check the Implementation Results dialog (Implementation Options) for supported files. See Implementation Results Panel, on page 163.

Extension	File	Description

`.vm`
`.vhm`
	Mapped Verilog or VHDL netlist	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 used as input for synthesis. Specify these files on the Implementation Results dialog box (Implementation Options). See Implementation Results Panel, on page 163.
		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 report, located in the implementation directory, is written out in two file formats: text (*projectName*.srr), and HTML with an interactive table of contents (*projectName*.htm and *projectName*_srr.htm) where *projectName* is the name of your project. Select View Log File in HTML in the Options->Project View Options dialog box to enable viewing the log file in HTML. Select the View Log button in the Project view (Buttons and Options, on page 110) to see the log file report.

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 compiler information. As a precaution, a backup copy of the log file (srr) is written to the backup sub-directory in the Implementation Results directory. Only one backup log file is updated for subsequent synthesis runs.

The log file contains detailed reports on the compiler, mapper, timing, and resource usage information for your design. Errors, notes, warnings, and messages appear in both the log file and the warning tab in the Tcl window.

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.	Compiler Report, on page 437
Buffers added to clocks in certain supported technologies.	Timing Reports, on page 441
Buffers added to nets.	Net Buffering Report, on page 438
Timing results. This section of the log file begins with “START TIMING REPORT” section. If you use the Timing Analyst to generate a custom timing report, its format is the same as the timing report in the log file, but the customized timing report is in a .ta file.	Timing Reports, on page 441
Compile point remapping.	Compile Point Information, on page 438
Resources used by synthesis mapping.	Resource Usage Report, on page 439
Design changes made as a result of retiming.	Retiming Report, on page 439

Compiler Report

This report 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, FSM extractions and inferred RAMs/ROMs. It also includes informational or warning messages about unused ports, removal of redundant logic, and latch inference. See Errors, Warnings, Notes, and Messages, on page 439 for details about the kinds of messages.

Premap Report

This report begins with the pre-mapper version and date, and reports the following:

- File loading times and memory usage
- Clock summary

Mapper Report

This report 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 such as flattened instances, extraction of counters, FSM implementations, clock nets, buffered nets, replicated logic, flip-flop optimizations, and informational or warning messages. See Errors, Warnings, Notes, and Messages, on page 439 for details about the kinds of messages.

Clock Buffering Report

This section of the log file 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
```

Compile Point Information

The Summary of Compile Points section of the log file (*projectName*.srr) lists each compile point, together with an indication of whether it was remapped, and, if so, why. Also, a timing report is generated for each compile point located in its respective results directories in the Implementation Directory. The compile point is the top-level design for this report file.

For more information on compile points and the compile-point synthesis flow, see Synthesizing Compile Points, on page 433of the *User Guide.*

Timing Section

A default timing report is written to the log file (*projectName*.srr) in the “START OF TIMING REPORT” section. See Timing Reports, on page 441, for details.

For certain device technologies, you can use the Timing Analyst to generate additional timing reports for point-to-point analysis (see Analysis Menu, on page 226). Their format is the same as the timing report.

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

Retiming Report

Whenever retiming is enabled, a retiming report is added to the log file (*projectName*.srr). It includes information about the design changes made as a result of retiming, such as the following:

- The number of flip-flops added, removed, or modified because of retiming. Flip-flops modified by retiming have a _ret suffix added to their names.
- Names of the flip-flops that were *moved* by retiming and no longer exist in the Technology view.
- Names of the flip-flops *created* as result of the retiming moves, that did not exist in the RTL view.
- Names of the flip-flops *modified* by retiming; for example, flip-flops that are in the RTL and Technology views, but have different fanouts because of retiming.

Errors, Warnings, Notes, and Messages

Throughout the log file, interactive error, note, warning, and informational messages appear.

- Error messages begin with “`@E:`”

- Warning messages begin with "`@W:`"
- Notes begin with "`@N:`"
- Advisories begin with "`@A:`"
- Informational messages begin with "`@I:`"

Colors distinguish different types of messages:

Color	Message Type	Example
Blue	Information (@I) Notes (@N)	@I: :"C:\designs\Designs6\module1\mychip.v" @N: CL201 |Trying to extract state machine for ...
Brown	Warnings (@W)	@W: CG146 |Creating black_box for empty module ...
Red	Errors(@E)	@E: CS106 |Reference to undefined module ...

The errors, warnings, and notes are also displayed in the Messages tab of the Output window. To get help on a message, you can single click on the numeric ID at the beginning of the message in the log file or Messages window. To crossprobe to the corresponding HDL source code, single click on the source file name.

Timing Reports

Timing results can be written to one or more of the following files:

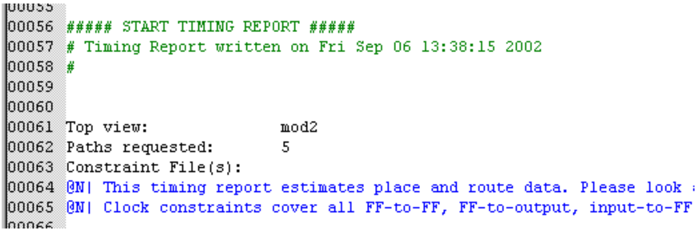
`.srr` or `.htm`	Log file that contains a default timing report. To find this information, after synthesis completes, open the log file (View -> Log File), and search for START TIMING REPORT.
`.ta`	Timing analysis file that contains timing information based on the parameters you specify in the stand-alone Timing Analyst (Analysis->Timing Analyst).
`designName``_async_clk .rpt.scv`	Asynchronous clock report file that is generated when you enable the related option in the stand-alone Timing Analyzer (Analysis->Timing Analyst). This report can be displayed in a spreadsheet tool and contains information for paths that cross between multiple clock groups. See Asynchronous Clock Report, on page 448 for details on this report.

The timing reports in the .srr/.htm and .ta files have the following sections:

- Timing Report Header, on page 442
- Performance Summary, on page 442
- Clock Relationships, on page 444
- Interface Information, on page 445
- Detailed Clock Report, on page 446
- Asynchronous Clock Report, on page 448

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
```

You can control the size of the timing report by choosing Project -> Implementation Options, clicking the Timing Report tab of the panel, and specifying the number of start/end points and the number of critical paths to report. See Timing Report Panel, on page 165, for details.

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 446). The Performance Summary lists the following information for each clock in the design:

Performance Summary Column	Description
Starting Clock	Clock at the start point of the path. 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/Estimated Frequency	Target frequency goal /estimated value after synthesis. See Cross-Clock Path Timing Analysis, on page 444 for information on how cross-clock path slack is reported.
Requested/Estimated Period	Target clock period/estimated value after synthesis.

Performance Summary Column	Description
Slack	Difference between estimated and requested period. See Cross-Clock Path Timing Analysis, on page 444 for information on how cross-clock path slack is reported.
Clock Type	The type of clock: inferred, declared, derived or system. The system clock is the delay for the combinatorial path.
Clock Group	Name of the clock group that a clock belongs.

The 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 Types

The synthesis timing reports include the following types of clocks:

- Declared Clocks

  User-defined clocks specified in the constraint file.

- Inferred Clocks

  These are clocks that the synthesis timing engine finds during synthesis, but which have not been constrained by the user. The tool assigns the default global frequency specified for the project to these clocks.

- Derived Clocks

  These are clocks that the synthesis tool identifies from a clock divider or multiplier. The tool reports these clocks for timing purposes only.

- System Clock

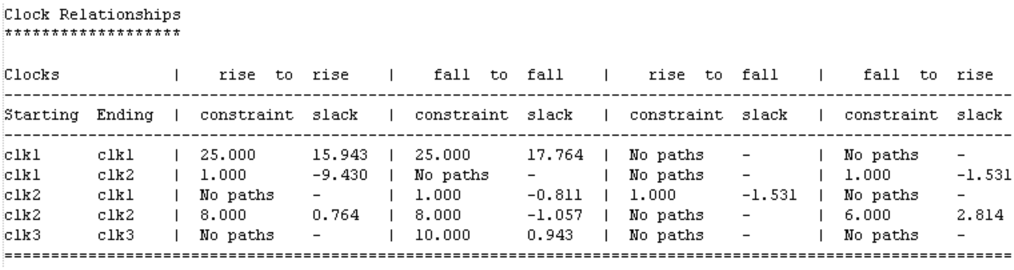
  The system clock is the delay for the combinatorial path. Additionally, a system clock can be reported if there are sequential elements in the design for a clock network that cannot be traced back to a clock. Also, the system clock can occur for unconstrained I/O ports. You must investigate these conditions.

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. See Cross-Clock Path Timing Analysis, on page 444 for details about cross-clock paths.

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 412. For tips on using clock groups, see Defining Other Clock Requirements, on page 106 in the *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    -
=============================================================================================================
```

Cross-Clock Path Timing Analysis

The following describe how the timing analyst calculates cross-clock path frequency and slack.

Cross-Clock Path Frequency

For each data path, the tool estimates the highest frequency that can be set for the clock(s) without a setup violation. It finds the largest scaling factor that can be applied to the clock(s) without causing a setup violation. If the start clock is not the same as the end clock, it scales both by the same factor.

scale = (*minimum time period* -(-*current slack*))/*minimum time period*

It assumes all other delays in the setup calculation (e.g., uncertainty) are fixed.

It applies relevant multicycle constraints to the setup calculation.

The estimated frequency for a clock is the minimum frequency over all paths that start or end on that clock, with the following exceptions:

- The tool does not consider paths between the system clock and another clock to estimate frequency.
- It considers paths with a path delay constraint to be asynchronous, and does not use them to estimate frequency.
- It considers paths between clocks in different domains to be asynchronous, and does not use them to estimate frequency.

Slack for Cross-Clock Paths

The slack reported for a cross-clock path is the worst slack for any path that starts on that clock. Note that this differs from the estimated frequency calculation, which is based on the worst slack for any path starting or ending on that 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.
Arrival Time	Input ports: set_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 – set_output_delay. Default value of set_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.

Clock Information	Description
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.
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 Implementation Options 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 delay, while the requested period in the Performance Summary (Performance Summary, on page 442) 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:

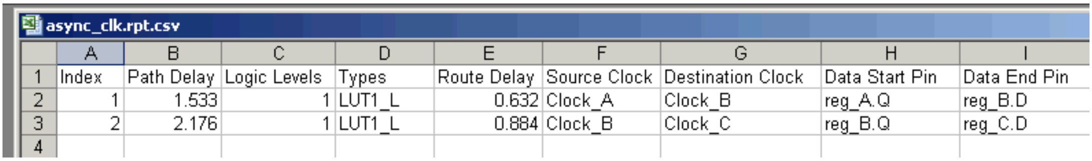
Critical path information	Description
Instance/Net Name	Technology view names for the instances and nets in the critical path
Type	Type of cell
Pin Name	Name of the pin
Pin Dir	Pin direction
Delay	The delay value.
Arrival Time	Clock delay at the source + the propagation delay through the path
Fan Out	Number of fanouts for the point in the path

Asynchronous Clock Report

You can generate a report for paths that cross between clock groups using the stand-alone Timing Analyst (Analysis->Timing Analyst, Generate Asynchronous Clock Report check box). Generally, paths in different clock groups are automatically handled as false paths. This option provides a file that contains information on each of the paths and can be viewed in a spreadsheet tool. To display the CSV-format report:

1. Locate the file in your results directory `projectName_async_clk.rpt.csv.`
2. Open the file in your spreadsheet tool.

Column	Description
Index	Path number.
Path Delay	Delay value as reported in standard timing (ta) file.
Logic Levels	Number of logic levels in the path (such as LUTs, cells, and so on) that are between the start and end points.
Types	Cell types, such as LUT, logic cell, and so on.
Route Delay	As reported for each path in ta.
Source Clock	Start clock.
Destination Clock	End clock.
Data Start Pin	Sequential device output pin at start of path.
Data End Pin	Setup check pin at destination.



async_clk.rpt.csv

	A	B	C	D	E	F	G	H	I
1	Index	Path Delay	Logic Levels	Types	Route Delay	Source Clock	Destination Clock	Data Start Pin	Data End Pin
2	1	1.533	1	LUT1_L	0.632	Clock_A	Clock_B	reg_A.Q	reg_B.D
3	2	2.176	1	LUT1_L	0.884	Clock_B	Clock_C	reg_B.Q	reg_C.D
4									

Constraint Checking Report

Use the Run->Constraint Check command to generate a report on the constraint files in your project. The *projectName*_cck.rpt file provides information such as invalid constraint syntax, constraint applicability, and any warnings or errors. For details about running Constraint Check, see Checking Constraint Files, on page 48 in the *User Guide*.

This section describes the following topics:

- Reporting Details, on page 450
- Inapplicable Constraints, on page 451
- Applicable Constraints With Warnings, on page 452
- Sample Constraint Check Report, on page 453

Reporting Details

This constraint checking file reports the following:

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

It contains the following sections:

Summary	Statement which summarizes the total number of issues defined as an error or warning (x) out of the total number of constraints with issues (y) for the total number of constraints (z) in the .fdc file. `Found <x> issues in <y> out of <z> constraints`
Clock Relationship	Standard timing report clock table, without slack.
Unconstrained Start/End Points	Lists I/O ports that are missing input/output delays.

Unapplied constraints	Constraints that cannot be applied because objects do not exist or the object type check is not valid. See Inapplicable Constraints, on page 451 for more information.
Applicable constraints with issues	Constraints will be applied either fully or partially, but there might be issues that generate warnings which should be investigated, such as some objects/collections not existing. Also, whenever at least one object in a list of objects is not specified with a valid object type a warning is displayed. See Applicable Constraints With Warnings, on page 452 for more information.
Constraints with matching wildcard expressions	Lists constraints or collections using wildcard expressions up to the first 1000, respectively.

Inapplicable Constraints

Refer to the following table for constraints that were not applied because objects do not exist or the object type check was not valid:

For these constraints...	Objects must be...
Attributes	Valid definitions
create_clock	• Ports • Nets • Pins • Registers • Instantiated buffers
set_clock_delay	Clocks
define_compile_point	• Region • View
define_current_design	v:*view*

For these constraints...	Objects must be...
set_false_path set_multicycle_path set_path_delay	For -to or -from objects: • i:sequential instances • p:ports • i:black boxes For -through objects • n:nets • t:hierarchical ports • t:pins
set_multicycle_path	Specified as a positive integer
set_input_delay	• Input ports • bidir ports
set_output_delay	• Output ports • Bidir ports
define_reg_input_delay define_reg_output_delay	Sequential instances

Applicable Constraints With Warnings

The following table lists reasons for warnings in the report file:

For these constraints...	Objects must be...
create_clock	• Ports • Nets • Pins • Registers • Instantiated buffers
set_clock_delay	A single object. Multiple objects are not supported.
define_compile_point	A single object. Multiple objects are not supported.
define_current_design	v:*view*

For these constraints...	Objects must be...
set_false_path set_multicycle_path set_path_delay	For -to or -from objects: • i:sequential instances • p:ports • i:black boxes For -through objects: • n:nets • t:hierarchical ports • t:pins
set_input_delay	A single object. Multiple objects are not supported.
set_output_delay	A single object. Multiple objects are not supported.
define_reg_input_delay define_reg_output_delay	A single object. Multiple objects are not supported.

Sample Constraint Check Report

The following is a sample report generated by constraint checking:

```
# Synopsys Constraint Checker, version maprc, Build 658R, built Aug 25 2011
# Copyright (C) 1994-2011, Synopsys, Inc.

# Written on Thu Oct 20 09:42:22 2011
##### DESIGN INFO #######################################################

Top View:                 "decode_top"
Constraint File(s):       "C:\timing_88\FPGA_decode_top.fdc"
##### SUMMARY ############################################################

Found 3 issues in 2 out of 27 constraints
```

```
##### DETAILS ############################################################

Clock Relationships
*******************

Starting  Ending  |  rise to rise   |  fall to fall   |  rise to fall   |  fall to rise
------------------------------------------------------------------------------------
clk2x     clk2x   |  24.000         |  24.000         |  12.000         |  12.000
------------------------------------------------------------------------------------
clk2x     clk     |  24.000         |  No paths       |  No paths       |  12.000
clk       clk2x   |  24.000         |  No paths       |  12.000         |  No paths
clk       clk     |  48.000         |  No paths       |  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 groups

Unconstrained Start/End Points
******************************
p:test_mode

Inapplicable constraints
************************

set_false_path -from p:next_synd -through i:core.tab1.ram_loader
@E:|object "i:core.tab1.ram_loader" does not exist
@E:|object "i:core.tab1.ram_loader" is incorrect type; "-through" objects must be of
type net (n:), or pin (t:)

Applicable constraints with issues
**********************************

set_false_path -from {core.decoder.root_mult*.root_prod_pre[*]} -to
{i:core.decoder.omega_inst.omega_tmp_d_lch[7:0]}
@W:|object "core.decoder.root_mult*.root_prod_pre[*]" is missing qualifier which may
result in undesired results; "-from" objects must be of type clock (c:), inst (i:), port
(p:), or pin (t:)

Constraints with matching wildcard expressions
**********************************************

set_false_path -from {core.decoder.root_mult*.root_prod_pre[*]} -to
{i:core.decoder.omega_inst.omega_tmp_d_lch[7:0]}
@N:|expression "core.decoder.root_mult*.root_prod_pre[*]" applies to objects:
core.decoder.root_mult1.root_prod_pre[14:0]
core.decoder.root_mult.root_prod_pre[14:0]
```

```
set_false_path -from {i:core.decoder.*.root_prod_pre[*]} -to {i:core.decoder.t_*_[*]}
@N:|expression "core.decoder.*.root_prod_pre[*]" applies to objects:
core.decoder.root_mult1.root_prod_pre[14:0]
core.decoder.root_mult.root_prod_pre[14:0]
@N:|expression "core.decoder.t_*_[*]" applies to objects:
core.decoder.t_20_[7:0]
core.decoder.t_19_[7:0]
core.decoder.t_18_[7:0]
core.decoder.t_17_[7:0]
core.decoder.t_16_[7:0]
core.decoder.t_15_[7:0]
core.decoder.t_14_[7:0]
core.decoder.t_13_[7:0]
core.decoder.t_12_[7:0]
core.decoder.t_11_[7:0]
core.decoder.t_10_[7:0]
core.decoder.t_9_[7:0]
core.decoder.t_8_[7:0]
core.decoder.t_7_[7:0]
core.decoder.t_6_[7:0]
core.decoder.t_5_[7:0]
core.decoder.t_4_[7:0]
core.decoder.t_3_[7:0]
core.decoder.t_2_[7:0]
core.decoder.t_1_[7:0]
core.decoder.t_0_[7:0]

set_false_path -from {i:core.decoder.root_mult*.root_prod_pre[*]} -to
{i:core.decoder.err[7:0]}
N:|expression "core.decoder.root_mult*.root_prod_pre[*]" applies to objects:
core.decoder.root_mult1.root_prod_pre[14:0]
core.decoder.root_mult.root_prod_pre[14:0]

set_false_path -from {i:core.decoder.root_mult*.root_prod_pre[*]} -to
{i:core.decoder.omega_inst.deg_omega[4:0]}
@N:|expression "core.decoder.root_mult*.root_prod_pre[*]" applies to objects:
core.decoder.root_mult1.root_prod_pre[14:0]
core.decoder.root_mult.root_prod_pre[14:0]

set_false_path -from {i:core.decoder.root_mult*.root_prod_pre[*]} -to
{i:core.decoder.omega_inst.omega_tmp[0:7]}
@N:|expression "core.decoder.root_mult*.root_prod_pre[*]" applies to objects:
core.decoder.root_mult1.root_prod_pre[14:0]
core.decoder.root_mult.root_prod_pre[14:0]

set_false_path -from {i:core.decoder.root_mult*.root_prod_pre[*]} -to
{i:core.decoder.root[7:0]}
@N:|expression "core.decoder.root_mult*.root_prod_pre[*]" applies to objects:
core.decoder.root_mult1.root_prod_pre[14:0]
core.decoder.root_mult.root_prod_pre[14:0]
```

```
set_false_path -from {i:core.decoder.root_mult*.root_prod_pre[*]} -to
{i:core.decoder.root_inst.count[3:0]}
N:|expression "core.decoder.root_mult*.root_prod_pre[*]" applies to objects:
core.decoder.root_mult1.root_prod_pre[14:0]
core.decoder.root_mult.root_prod_pre[14:0]

set_false_path -from {i:core.decoder.root_mult*.root_prod_pre[*]} -to
{i:core.decoder.root_inst.q_reg[7:0]}
@N:|expression "core.decoder.root_mult*.root_prod_pre[*]" applies to objects:
core.decoder.root_mult1.root_prod_pre[14:0]
core.decoder.root_mult.root_prod_pre[14:0]

set_false_path -from {i:core.decoder.root_mult*.root_prod_pre[*]} -to
{i:core.decoder.root_inst.q_reg_d_lch[7:0]}
@N:|expression "core.decoder.root_mult*.root_prod_pre[*]" applies to objects:
core.decoder.root_mult1.root_prod_pre[14:0]
core.decoder.root_mult.root_prod_pre[14:0]

set_false_path -from {i:core.decoder.root_mult.root_prod_pre[*]} -to
{i:core.decoder.error_inst.den[7:0]}
@N:|expression "core.decoder.root_mult.root_prod_pre[*]" applies to objects:
core.decoder.root_mult.root_prod_pre[14:0]

set_false_path -from {i:core.decoder.root_mult1.root_prod_pre[*]} -to
{i:core.decoder.error_inst.num1[7:0]}
@N:|expression "core.decoder.root_mult1.root_prod_pre[*]" applies to objects:
core.decoder.root_mult1.root_prod_pre[14:0]

set_false_path -from {i:core.decoder.synd_reg_*_[7:0]} -to {i:core.decoder.b_*_[7:0]}
@N:|expression "core.decoder.synd_reg_*_[7:0]" applies to objects:
core.decoder.un1_synd_reg_0_[7:0]
core.decoder.synd_reg_20_[7:0]
core.decoder.synd_reg_19_[7:0]
core.decoder.synd_reg_18_[7:0]
core.decoder.synd_reg_17_[7:0]
core.decoder.synd_reg_16_[7:0]
core.decoder.synd_reg_15_[7:0]
core.decoder.synd_reg_14_[7:0]
core.decoder.synd_reg_13_[7:0]
core.decoder.synd_reg_12_[7:0]
core.decoder.synd_reg_11_[7:0]
core.decoder.synd_reg_10_[7:0]
core.decoder.synd_reg_9_[7:0]
core.decoder.synd_reg_8_[7:0]
core.decoder.synd_reg_7_[7:0]
core.decoder.synd_reg_6_[7:0]
core.decoder.synd_reg_5_[7:0]
core.decoder.synd_reg_4_[7:0]
core.decoder.synd_reg_3_[7:0]
core.decoder.synd_reg_2_[7:0]
core.decoder.synd_reg_1_[7:0]
```

```
@N:|expression "core.decoder.b_*_[7:0]" applies to objects:
core.decoder.un1_b_0_[7:0]
core.decoder.b_calc.un1_lambda_0_[7:0]
core.decoder.b_20_[7:0]
core.decoder.b_19_[7:0]
core.decoder.b_18_[7:0]
core.decoder.b_17_[7:0]
core.decoder.b_16_[7:0]
core.decoder.b_15_[7:0]
core.decoder.b_14_[7:0]
core.decoder.b_13_[7:0]
core.decoder.b_12_[7:0]
core.decoder.b_11_[7:0]
core.decoder.b_10_[7:0]
core.decoder.b_9_[7:0]
core.decoder.b_8_[7:0]
core.decoder.b_7_[7:0]
core.decoder.b_6_[7:0]
core.decoder.b_5_[7:0]
core.decoder.b_4_[7:0]
core.decoder.b_3_[7:0]
core.decoder.b_2_[7:0]
core.decoder.b_1_[7:0]
core.decoder.b_0_[7:0

Library Report
**************

# End of Constraint Checker Report
```

CHAPTER 8

Verilog Language Support

This chapter discusses Verilog support in the synthesis tool. SystemVerilog support is described separately, in Chapter 9, *SystemVerilog Language Support*. This chapter includes the following topics:

Support for Verilog Language Constructs

This section describes support for various Verilog language constructs:

- Supported and Unsupported Verilog Constructs, on page 460
- Ignored Verilog Language Constructs, on page 462

Supported and Unsupported Verilog Constructs

The following table lists the supported and unsupported Verilog constructs. If the tool encounters an unsupported construct, it generates an error message and stops.

Supported Verilog Constructs	Unsupported Verilog Constructs
Net types wire, tri, supply1, supply0	Net types: trireg, wor, trior, wand, triand, tri0, tri1, charge strength
Register types: • reg, integer, time (64-bit reg) • arrays of reg	Register types: real
Gate primitive and module instantiations	Built-in unidirectional and bidirectional switches, and pull-up/pull-down
always blocks, user tasks, user functions	Named events and event triggers
inputs, outputs, and inouts to a module	UDPs and specify blocks
All operators -, -, *, /, %, <, >, <=, >=, ==, !=, ===, !==, &&, ||, !, ~, &, ~&, |, ~|, ^~, ~^, ^, <<, >>, ?:, { }, {{ }} (See Verilog Operator Support, on page 461 for additional details.)	Net names: force, release, and hierarchical net names (for simulation only)
Procedural statements: assign, if-else-if, case, casex, casez, for, repeat, while, forever, begin, end, fork, join	Procedural statements: deassign, wait

Procedural assignments: • Blocking assignments = • Non-blocking assignments <= Do not use = with <= for the same register. Use parameter override: # and defparam (down one level of hierarchy only).
Continuous assignments
Compiler directives: `define, `ifdef, ‘ifndef, `else, `elsif, `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

Verilog Operator Support

Note the following:

- The / and % operators are supported for compile-time constants and constant powers of two.
- When an equality (==) or inequality (!=) operator includes unknown bits (e.g., A==4'b10x1 or A!=4b111z), the Synopsys FPGA Verilog compiler assumes that the output is always False. This assumption contradicts the LRM which states that the output should be x (unknown) and can result in a possible simulation mismatch.

Ignored Verilog Language Constructs

When it encounters certain Verilog constructs, the tool ignores them and continues the synthesis run. The following constructs are ignored:

- delay, delay control, and drive strength
- scalared, vectored
- initial block
- Compiler directives (except for `define, `ifdef, `ifndef, `else, `elsif, `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 Popup Menu Command, on page 292 and Setting Verilog and VHDL Options, on page 139 of the *User Guide*. The synthesis tool supports the following Verilog 2001 features:

Feature	Description
Combined Data, Port Types (ANSI C-style Modules)	Module data and port type declarations can be combined for conciseness.
Comma-separated Sensitivity List	Commas are allowed as separators in sensitivity lists (as in other Verilog lists).
Wildcards (*) in Sensitivity List	Use @* or @(*) to include all signals in a procedural block to eliminate mismatches between RTL and post-synthesis simulation.
Signed Signals	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 Assignment by Name	Assigns values to parameters by name, inline.
Constant Function	Builds complex values at elaboration time.
Configuration Blocks	Specifies a set of rules that defines the source description applied to an instance or module.
Localparams	A constant which cannot be redefined or modified.
$signed and $unsigned Built-in Functions	Built-in Verilog 2001 function that converts types between signed and unsigned.
$clog2 Constant Math Function	Returns the value of the log base-2 for the argument passed.
Generate Statement	Creates multiple instances of an object in a module. You can use generate with loops and conditional statements.
Automatic Task Declaration	Dynamic allocation and release of storage for tasks.

Feature	Description
Multidimensional Arrays	Groups elements of the declared element type into multi-dimensional objects.
Variable Partial Select	Supports indexed part select expressions (**+**: and **-**:), which use a variable range to provide access to a word or part of a word.
Cross-Module Referencing	Accesses elements across modules.
ifndef and elsif Compiler Directives	`ifndef and `ilsif compiler directive support.

Combined Data, Port Types (ANSI C-style Modules)

In Verilog 2001, you can combine module data and port type declarations to be concise, as shown below:

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);
```

Comma-separated Sensitivity List

In Verilog 2001, you can use commas as separators in sensitivity lists (as in other Verilog lists).

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;
```

Wildcards (*) in Sensitivity List

In Verilog 2001, you can use @* or @(*) to include all signals in a procedural block, eliminating mismatches between RTL and post-synthesis simulation.

Verilog ‘95

```
always @(a or b or cin)
   sum = a - b - cin;
```

Verilog 2001

```
// Style 1:
always @(*)
   sum = a - b - cin;
// Style 2:
always @*
   sum = a - b - cin;
```

Signed Signals

In Verilog 2001, data types net and reg, module ports, integers of different bases and signals can all be signed. You can assign and compare signed signals, and perform signed operations for vectors of any length.

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
```

Inline Parameter Assignment by Name

In Verilog 2001, you can assign values to parameters by name, inline:

```
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
```

Constant Function

In Verilog 2001, you can use constant functions to build complex values at elaboration time.

Example – Constant function

```
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
```

Configuration Blocks

Verilog configuration blocks define a set of rules that explicitly specify the exact source description to be used for each instance in a design. A configuration block is defined outside the module. Currently, support is limited to single configuration blocks.

Syntax

config *configName*;
    design *libraryIdentifier***.***moduleName*;
    default liblist *listofLibraries*;
    configurationRule;
endconfig

Design Statement

The design statement specifies the library and module for which the configuration rule is to defined.

design *libraryIdentifier***.***moduleName*;
    libraryIdentifier **:-** Library Name
    moduleName **:-** Module Name

Default Statement

The default liblist statement lists the library from which the definition of the module and sub-modules can be selected. A use clause cannot be used in this statement.

default liblist *listof_Libraries*;
    listofLibraries **:-** List of Libraries

Configuration Rule Statement

In this section, rules are defined for different instances or cells in the design. The rules are defined using instance or cell clauses.

- Instance Clause – specifies the particular source description for a given instance in the design.
- Cell Clause – specifies the source description to be picked for a particular cell/module in a given design.

A configuration rule can be defined as any of the following:

- instance clause with liblist

  `instance` *moduleName*`.`*instance* `liblist` *listofLibraries*`;`

- instance clause with use clause

  `instance` *moduleName*`.`*instance* `use` *libraryIdentifier*`.`*cellName*`;`

- cell clause with liblist

  `cell` *cellName* `liblist` *listofLibraries*`;`

- cell clause with use clause

  `cell` *cellName* `use` *libraryIdentifier*`.`*cellName*`;`

Configuration Block Examples

The following examples illustrate Verilog 2001 configuration blocks.

Example – Configuration with instance clause

The following example has different definitions for the leaf module compiled into the multlib and xorlib libraries; configuration rules are defined specifically for instance u2 in the top module to have the definition of leaf module as XOR (by default the leaf definition is multiplier). This example uses an instance clause with liblist to define the configuration rule.

```
//********Leaf module with the Multiplication definition

// Multiplication definition is compiled to the library "multlib"
// Command to be added in the synplify project file to compile a
// specific HDL to a specific library is
// add_file -verilog -lib multlib "leaf_mult.v"
```

```
module leaf
(
//Input Port
   input [7:0] d1,
   input [7:0] d2,
//Output Port
   output reg [15:0] dout
);

always@*
   dout = d1 * d2;
endmodule //EndModule

//********Leaf module with the XOR definition

// XOR definition is compiled to the library "xorlib"
// Command to be added in the synplify project file to compile a
// specific HDL to a specific library is
// add_file -verilog -lib xorlib "leaf_xor.v"

module leaf
(
//Input Port
   input [7:0] d1,
   input [7:0] d2,
//Output Port
   output reg[15:0] dout
);

always@(*)
   dout = d1 ^ d2;
endmodule //EndModule

//********Top module definition

// Top module definition is compiled to the library "TOPLIB"
// Command to be added in the synplify project file to compile a
// specific HDL to a specific library is
// add_file -verilog -lib TOPLIB "top.v"

module top
(
//Input Port
   input [7:0] d1,
   input [7:0] d2,
   input [7:0] d3,
   input [7:0] d4,
//Output Port
   output [15:0] dout1,
   output [15:0] dout2
);

leaf
u1
(
   .d1(d1),
   .d2(d2),
   .dout(dout1)
);

leaf
u2
(
   .d1(d3),
   .d2(d4),
   .dout(dout2)
);

endmodule //End Module

//********Configuration Definition

// Configuration definition is compiled to the library "TOPLIB"
// Command to be added in the synplify project file to compile a
// specific HDL to a specific library is
// add_file -verilog -lib TOPLIB "cfg.v"

config cfg;
   design TOPLIB.top;
   default liblist multlib xorlib TOPLIB; //By default the leaf
      // definition is Multiplication definition
   instance top.u2 liblist xorlib; //For instance u2 the default
      // definition is overridden and the "leaf" definition is
      // picked from "xorlib" which is XOR.
endconfig //EndConfiguration
```

Example – Configuration with cell clause

In the following example, different definitions of the leaf module are compiled into the multlib and xorlib libraries; a configuration rule is defined for cell leaf that picks the definition of the cell from the multlib library. This example uses a cell clause with a use clause to define the configuration rule.

```
//********Leaf module with the Multiplication definition

// Multiplication definition is compiled to the library "multlib"
// Command to be added in the synplify project file to compile a
// specific HDL to a specific library is
// add_file -verilog -lib multlib "leaf_mult.v"

module leaf
(
//Input Port
   input [7:0] d1,
   input [7:0] d2,
//Output Port
   output reg [15:0] dout
);

always@*
   dout = d1 * d2;
endmodule //EndModule

//********Leaf module with the XOR definition

// XOR definition is compiled to the library "xorlib"
// Command to be added in the synplify project file to compile a
// specific HDL to a specific library is
//add_file -verilog -lib xorlib "leaf_xor.v"

module leaf
(
//Input Port
   input [7:0] d1,
   input [7:0] d2,
//Output Port
   output reg[15:0] dout
);

always@(*)
   dout = d1 ^ d2;
endmodule //EndModule

//********Top module definition
```

```
// Top module definition is compiled to the library "TOPLIB"
// Command to be added in the synplify project file to compile a
// specific HDL to a specific library is
// add_file -verilog -lib TOPLIB "top.v"

module top
(
//Input Port
   input [7:0] d1,
   input [7:0] d2,
   input [7:0] d3,
   input [7:0] d4,
//Output Port
   output [15:0] dout1,
   output [15:0] dout2
);

leaf
u1
(
   .d1(d1),
   .d2(d2),
   .dout(dout1)
);

leaf
u2
(
   .d1(d3),
   .d2(d4),
   .dout(dout2)
);
endmodule //End Module

//********Configuration Definition

// Configuration definition is compiled to the library "TOPLIB"
// Command to be added in the synplify project file to compile a
// specific HDL to a specific library is
// add_file -verilog -lib TOPLIB "cfg.v"
```

```
config cfg;
   design TOPLIB.top;
   default liblist xorlib multlib TOPLIB; //By default the leaf
      // definition uses the XOR definition
   cell leaf use multlib.leaf; //Definition of the instances u1 and
u2
      // will be Multiplier which is picked from "multlib"
endconfig //EndConfiguration
```

Example – Hierarchical reference of the module inside the configuration

In the following example, different definitions of leaf are compiled into the multlib, addlib, and xorlib libraries; the configuration rule is defined for instance u2 that is referenced in the hierarchy as the lowest instance module using an instance clause.

```
//********Leaf module with the Multiplication definition

// Multiplication definition is compiled to the library "multlib"
// Command to be added in the synplify project file to compile a
// specific HDL to a specific library is
// add_file -verilog -lib multlib "leaf_mult.v"

module leaf
(
//Input Port
   input [7:0] d1,
   input [7:0] d2,
//Output Port
   output reg [15:0] dout
);

always@*
   dout = d1 * d2;
endmodule //EndModule

//********Leaf module with the XOR definition

// XOR definition is compiled to the library "xorlib"
// Command to be added in the synplify project file to compile a
// specific HDL to a specific library is
// add_file -verilog -lib xorlib "leaf_xor.v"
```

```
module leaf
(
//Input Port
   input [7:0] d1,
   input [7:0] d2,
//Output Port
   output reg[15:0] dout
);

always@(*)
   dout = d1 ^ d2;
endmodule //EndModule

//********Leaf module with the ADDER definition

// ADDER definition is compiled to the library "addlib"
// Command to be added in the synplify project file to compile a
// specific HDL to a specific library is
// add_file -verilog -lib xorlib "leaf_add.v"

module leaf
(
//Input Port
   input [7:0] d1,
   input [7:0] d2,
//Output Port
   output [15:0] dout
);

assign dout = d1 + d2;
endmodule

//********Sub module definition

// Sub module definition is compiled to the library "SUBLIB"
// Command to be added in the synplify project file to compile a
// specific HDL to a specific library is
// add_file -verilog -lib SUBLIB "sub.v"

module sub
(
//Input Port
   input [7:0] d1,
   input [7:0] d2,
   input [7:0] d3,
   input [7:0] d4,
//Output Port
   output [15:0] dout1,
   output [15:0] dout2
);

leaf
u1
(
   .d1(d1),
   .d2(d2),
   .dout(dout1)
);

leaf
u2
(
   .d1(d3),
   .d2(d4),
   .dout(dout2)
);
endmodule //End Module

//********Top module definition

// Top module definition is compiled to the library "TOPLIB"
// Command to be added in the synplify project file to compile a
// specific HDL to a specific library is
// add_file -verilog -lib TOPLIB "top.v"

module top
(
//Input Port
   input [7:0] d1,
   input [7:0] d2,
   input [7:0] d3,
   input [7:0] d4,
//Output Port
   output [15:0] dout1,
   output [15:0] dout2
);

sub
u1
(
   .d1(d1),
   .d2(d2),
   .d3(d3),
   .d4(d4),
   .dout1(dout1),
   .dout2(dout2)
);
endmodule //End Module

//********Configuration Definition

// Configuration definition is compiled to the library "TOPLIB"
// Command to be added in the synplify project file to compile a
// specific HDL to a specific library is
// add_file -verilog -lib TOPLIB "cfg.v"

config cfg;
   design TOPLIB.top;
   default liblist addlib multlib xorlib TOPLIB SUBLIB; //By
default,
      //the leaf definition uses the ADDER definition
   instance top.u1.u2 liblist xorlib multlib; //For instances u2 is
      //referred hierarchy to lowest instances and the default
definition
      //is overridden by XOR definition for this instanceendconfig
//EndConfiguration
```

Limitations

- Verilog configuration support is limited to single configuration.
- Nested configuration is not supported.
- Top-level design name in the project file must match the top-level design name in the design clause of the configuration construct.
- A use clause with the cell name or library name omitted is not supported.
- The case where the configuration name and the module name are the same is not supported.
- Mixed HDL configuration is not supported.
- Multiple top levels in the design clause are not supported.
- Compiling the same configuration file to multiple libraries is not supported.

Localparams

In Verilog 2001, localparams (constants that cannot be redefined or modified) follow the same parameter rules in regard to size and sign.

Example:

```
parameter ONE = 1
localparam TWO=2*ONE
localparam [3:0] THREE=TWO+1;
localparam signed [31:0] FOUR=2*TWO;
```

$signed and $unsigned Built-in Functions

In Verilog 2001, the built-in Verilog 2001 functions can be used to convert types between signed and unsigned.

```
c = $signed (s); /* Assign signed valued of s to c. */
d = $unsigned (s); /* Assign unsigned valued of s to d. */
```

$clog2 Constant Math Function

Verilog-2005 includes the $clog2 constant math function which returns the value of the log base-2 for the argument passed. This system function can be used to compute the minimum address width necessary to address a memory of a given size or the minimum vector width necessary to represent a given number of states.

Syntax

$clog2(*argument***)**

In the above syntax, argument is an integer or vector.

Example 1 – Constant Math Function Counter

```
module top
#( parameter COUNT = 256 )
//Input
(  input clk,
   input rst,
//Output
//Function used to compute width based on COUNT value of counter:
   output [$clog2(COUNT)-1:0] dout );
reg[$clog2(COUNT)-1:0]count;
```

```
always@(posedge clk)
begin
   if(rst)
      count = 'b0;
   else
      count = count + 1'b1;
   end
assign dout = count;
endmodule
```

Example 2 – Constant Math Function RAM

```
module top
#
(  parameter DEPTH = 256,
   parameter WIDTH = 16 )
(
//Input
   input clk,
   input we,
   input rst,
//Function used to compute width of address based on depth of RAM:
   input [$clog2(DEPTH)-1:0] addr,
   input [WIDTH-1:0] din,
//Output
   output reg[WIDTH-1:0] dout );
reg[WIDTH-1:0] mem[(DEPTH-1):0];

always @ (posedge clk)
   if (rst == 1)
      dout = 0;
   else
      dout = mem[addr];

always @(posedge clk)
   if (we) mem[addr] = din;

endmodule
```

Generate Statement

The newer Verilog 2005 generate statement is now supported in Verilog 2001. Defparams, parameters, and function and task declarations within generate statements are supported. In addition, the naming scheme for registers and instances is enhanced to include closer correlation to specified generate symbolic hierarchies. Generated data types have unique identifier names and can be referenced hierarchically. Generate statements are created using one of the following three methods: generate-loop, generate-conditional, or generate-case.

```
// 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
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
```

Automatic Task Declaration

In Verilog 2001, tasks can be declared as automatic to dynamically allocate new storage each time the task is called and then automatically release the storage when the task exits. Because there is no retention of tasks from one call to another as in the case of static tasks, the potential conflict of two concurrent calls to the same task interfering with each other is avoided. Automatic tasks make it possible to use recursive tasks.

This is the syntax for declaring an automatic task:

task automatic *taskName* **(***argument* **[,** *argument* **, ...]) ;**

Arguments to automatic tasks can include any language-defined data type (reg, wire, integer, logic, bit, int, longint, or shortint) or a user-defined datatype (typedef, struct, or enum). Multidimensional array arguments are not supported.

Automatic tasks can be synthesized but, like loop constructs, the synthesis tool must be able to statically determine how many levels of recursive calls are to be made. Automatic (recursive) tasks are used to calculate the factorial of a given number.

Example

```
module automatic_task (input byte in1,
   output bit [8:0] dout);
parameter FACT_OP = 3;
bit [8:0] dout_tmp;

task automatic factorial(input byte operand,
   output bit [8:0] out1);
integer nFuncCall = 0;
begin
   if (operand == 0)
   begin
      out1 = 1;
   end
   else
   begin
      nFuncCall++;
      factorial((operand-1), out1);
      out1 = out1 * operand;
   end
end
endtask

always_comb
factorial(FACT_OP,dout_tmp);
assign dout = dout_tmp + in1 ;
endmodule
```

Multidimensional Arrays

In Verilog 2001, 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. Refer to the following examples.

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];`
3-dimensional wire object	my_wire is an 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];`
3-dimensional wire object	my_wire is an 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];`

These examples apply to register types too:

```
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.
```

There is a Verilog restriction which prohibits bit access into memory words. Verilog 2001 removes all such restrictions. This applies equally to wires types. For example:

```
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.
```

Variable Partial Select

In Verilog 2001, indexed partial select expressions (+: and -:), which use a variable range to provide access to a word or part of a word, are supported. The software extracts the size of the operators at compile time, but the index expression range can remain dynamic. You can use the partial select operators to index any non-scalar variable.

The syntax to use these operators is described below.

vectorName [*baseExpression* **+:** *widthExpression*]
vectorName [*baseExpression* **-:** *widthExpression*]

vectorName	Name of vector. Direction in the declaration affects the selection of bits
baseExpression	Indicates the starting point for the array. Can be any legal Verilog expression.
+:	The +: expression selects bits starting at the *baseExpression* while adding the *widthExpression*. Indicates an upward slicing.
-:	The -: expression selects bits starting at the *baseExpression* while subtracting the *widthExpression*. Indicates a downward slicing.
widthExpression	Indicates the width of the slice. It must evaluate to a constant at compile time. If it does not, you get a syntax error.

This is an example using partial select expressions:

```
module part_select_support (down_vect, up_vect, out1, out2, out3);
output [7:0] out1;
output [1:0] out2;
output [7:0] out3;
input [31:0] down_vect;
input [0:31] up_vect;
wire [31:0] down_vect;
wire [0:31] up_vect;
wire [7:0] out1;
wire [1:0] out2;
wire [7:0] out3;
wire [5:0] index1;
assign index1 = 1;
assign out1 = down_vect[index1+:8]; // resolves to [8:1]
assign out2 = down_vect[index1-:8]; // should resolve to [1:0],
   // but resolves to constant 2'b00 instead
assign out3 = up_vect[index1+:8]; // resolves to [1:8]
endmodule
```

For the Verilog code above, the following description explains how to validate partial select assignments to out2:

- The compiler first determines how to slice down_vect.
  - down_vect is an array of [31:0]

  - assign out2 = down_vect [1 -: 8] will slice down_vect starting at value 1 down to -6 as [1 : -6], which includes [1, 0, -1, -2, -3, -4, -5, -6]
- Then, the compiler assigns the respective bits to the outputs.
  - out2 [0] = down_vect [-6]
    out2 [1] = down_vect [-5]
  - Negative ranges cannot be specified, so out2 is tied to “00”.
  - Therefore, change the following expression in the code to:
    assign out2 = down_vect [1 -: 2], which resolves to down_vect [1,0]

Cross-Module Referencing

Cross-module referencing is a method of accessing an element across modules in Verilog and SystemVerilog. Verilog supports accessing elements across different scopes using the hierarchical reference (.) operator. Cross-module referencing can also be done on the variable of any of the data types available in SystemVerilog.

Cross-module referencing can be downward or upward, starting with the top module.

Downward Cross-Module Referencing

In downward cross-module referencing, you reference elements of lower-level modules in the higher-level modules through instantiated names. This is the syntax for a downward cross-module reference:

port/variable **=** *inst1***.***inst2***.***value*; // XMR Read

*inst1***.***inst2***.***port/variable* **=** *value*; // XMR Write

In this syntax, *inst1* is the name of an instance instantiated in the top module and *inst2* is the name of an instance instantiated in *inst1*. *Value* can be a constant, parameter, or variable. *Port/variable* is defined/declared once in the current module.

Example – Downward Read Cross-Module Reference

```
module top (
   input a,
   input b,
   output c,
   output d );

sub inst1 (.a(a), .b(b), .c(c) );
assign d = inst1.a;
endmodule

module sub (
   input a,
   input b,
   output c );

assign c = a  & b;
endmodule
```

Example – Downward Write Cross-Module Reference

```
module top
(  input a,
   input b,
   output c,
   output d
);

sub inst1 (.a(a), .b(b), .c(c), .d(d) );
assign top.inst1.d = a;
endmodule

module sub
(  input a,
   input b,
   output c,
   output d
);

assign c = a & b;
endmodule
```

Upward Cross-Module Referencing

In upward cross-module referencing, a lower-level module references items in a higher-level module in the hierarchy through the name of the top module.

This is the syntax for an upward reference from a lower module:

> *port/variable* **=** *top***.***inst1***.***inst2***.***value***;** // XMR Read
>
> *top***.***inst1***.***inst2***.***port/variable* **=** *value***;** // XMR Write

The starting reference is the top-level module. In this syntax, *top* is the name of the top-level module, *inst1* is the name of an instance instantiated in top module and *inst2* is the name of an instance instantiated in *inst1*. Value can be a constant, parameter, or variable. *Port/variable* is the one defined/declared in the current module.

Example – Upward Read Cross-Module Reference

```
module top (
   input a,
   input b,
   output c,
   output d );

sub inst1 (.a(a), .b(b), .c(c), .d(d) );
endmodule

module sub (
   input a,
   input b,
   output c,
   output d );

assign c = a & b;
assign d = top.a;
endmodule
```

Limitations

Cross-module referencing currently has the following limitations:

- The tools do not support cross-module referencing through an array of instances.
- The tools do not support cross-module referencing into generate blocks.
- In upward cross-module referencing, the reference must be an absolute path. An absolute path is always from the top-level module.
- You cannot access functions and tasks through cross-module reference notation.
- You can only use cross-module referencing with Verilog/SystemVerilog elements. You cannot access VHDL elements with hierarchical references.

ifndef and elsif Compiler Directives

Verilog 2001 supports the `ifndef and `elsif compiler directives. Note that the `ifndef directive is the opposite of `ifdef.

```
module top(output out);
   `ifndef a
      assign out = 1'b01;
   `elsif b
      assign out = 1'b10;
   `else
      assign out = 1'b00;
   `endif
endmodule
```

Verilog Synthesis Guidelines

This section provides guidelines for synthesis using Verilog and covers the following topics:

General Synthesis Guidelines

Some general guidelines are presented here to help you synthesize your Verilog design. See Verilog Module Template, on page 507 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 Implementation Options 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.

Library Support in Verilog

Verilog libraries are used to compile design units; this is similar to VHDL libraries. Use the libraries in Verilog to support mixed-HDL designs, where the VHDL design includes instances of a Verilog module that is compiled into a specific library. Library support in Verilog can be used with Verilog 2001 and SystemVerilog designs.

Compiling Design Units into Libraries

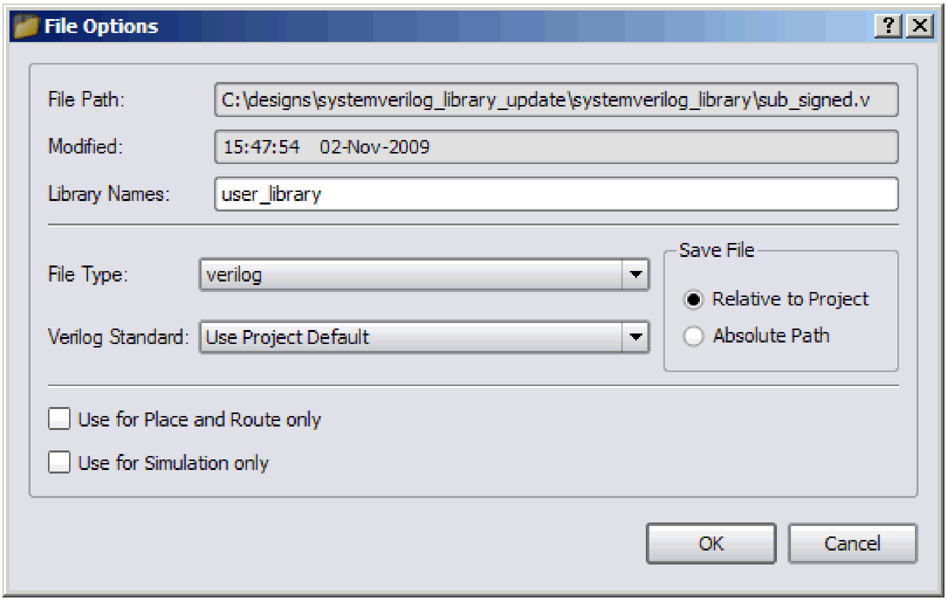
By default, the Verilog source files are compiled into the work library. You can compile these Verilog source files into any user-defined library.

To compile a Verilog file into a user-defined library:

1. Select the file in the Project view.

   The library name appears next to the filename; it directly follows the filename.

2. Right-click and select File Options from the popup menu. Specify the name for your library in the Library Names field. You can:
   – Compile multiple files into the same library.
   – Also compile the same file into multiple libraries.



File Path: C:\designs\systemverilog_library_update\systemverilog_library\sub_signed.v

Modified: 15:47:54 02-Nov-2009

Library Names: user_library

File Type: verilog

Verilog Standard: Use Project Default

Save File

● Relative to Project

○ Absolute Path

□ Use for Place and Route only

□ Use for Simulation only

Searching for Verilog Design Units in Mixed-HDL Designs

When a VHDL file references a Verilog design unit, the compiler first searches the corresponding library for which the VHDL file was compiled. If the Verilog design unit is not found in the user-defined library for which the VHDL file was compiled, the compiler searches the work library and then all the other Verilog libraries.

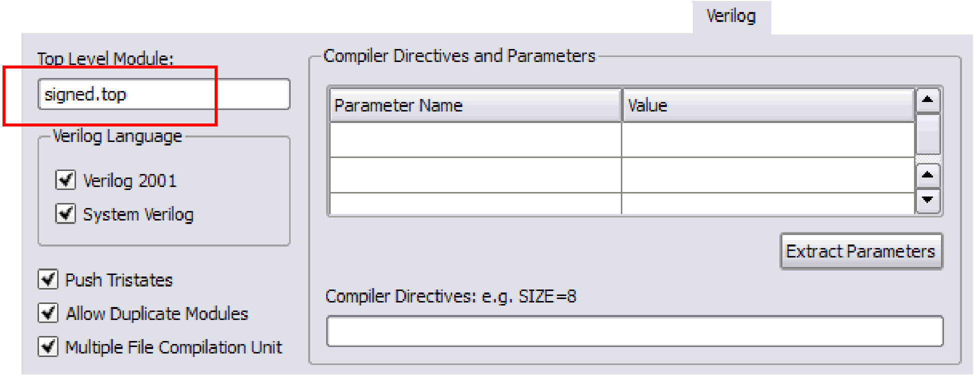
Therefore, to use a specific Verilog design unit in the VHDL file, compile the Verilog file into the same user-defined library for which the corresponding VHDL file was compiled.

You cannot use the VHDL library clause for Verilog libraries.

Specifying the Verilog Top-level Module

To set the Verilog top-level module for a user-defined library, use *libraryName.moduleName* in the Top Level Module field on the Verilog tab of the Implementation Options dialog box. You can also specify the following equivalent Tcl command:

```
set_option -top_module "signed.top"
```



Top Level Module: signed.top

Verilog Language

☑ Verilog 2001

☑ System Verilog

☑ Push Tristates

☑ Allow Duplicate Modules

☑ Multiple File Compilation Unit

Compiler Directives: e.g. SIZE=8

Limitations

The following functions are not supported:

- Direct Entity Instantiation
- Configuration for Verilog Instances

Example 1: Specifying Verilog Top-level Module—Compiled to the Non-work Library

```
//top_unsigned.v compiled into a user defined library - "unsigned"
//add_file -verilog -lib unsigned "./top_unsigned.v"
module top ( input unsigned [7:0] a, b,
output unsigned [15:0] result );
assign result = a * b;
endmodule

//top_signed.v compiled into a user defined library - "signed"
//add_file -verilog -lib signed "./top_signed.v"
module top ( input signed [7:0] a, b,
output signed [15:0] result );
assign result = a * b;
endmodule
```

To set the top-level module from the signed library:

- Specify the prefix library name for the module in the Top Level Module option in the Verilog panel of the Implementation Options dialog box.
- `set_option -top_module "signed.top"`

Example 2: Referencing Verilog Module from VHDL

This example includes two versions of the Verilog sub module that are compiled into the signed_lib and unsigned_lib libraries. The compiler uses the sub module from unsigned_lib when the top.vhd is compiled into unsigned_lib.

```
//Sub module sub in sub_unsigned is compiled into unsigned_lib
//add_file -verilog -lib unsigned_lib "./sub_unsigned.v"
module sub ( input unsigned [7:0] a, b,
output unsigned [15:0] result );
assign result = a * b;
endmodule

//Sub module sub in sub_signed is compiled into signed_lib
//add_file -verilog -lib signed_lib "./sub_signed.v"
module sub ( input signed [7:0] a, b,
output signed [15:0] result );
assign result = a * b;
endmodule

//VHDL Top module top is compiled into unsigned_lib library
// add_file -vhdl -lib unsigned_lib "./top.vhd"
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY top IS
GENERIC(
size_t : integer := 8
);
   PORT( a_top : IN std_logic_vector(size_t-1 DOWNTO 0);
   b_top : IN std_logic_vector(size_t-1 DOWNTO 0);
   result_top : OUT std_logic_vector(2*size_t-1 DOWNTO 0)
);
END top;

ARCHITECTURE RTL OF top IS
component sub
   PORT(a : IN std_logic_vector(7 DOWNTO 0);
   b : IN std_logic_vector(7 DOWNTO 0);
   result : OUT std_logic_vector(15 DOWNTO 0));
END component;
BEGIN
U1 : sub
   PORT MAP (
      a => a_top,
      b => b_top,
   result => result_top
);
END ARCHITECTURE RTL;
```

Constant Function Syntax Restrictions

For Verilog 2001, the syntax for constant functions is identical to the existing function definitions in Verilog. Restrictions on constant 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 functions are 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.
```

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 */;
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 Verilog 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;
wire [3:0] inputa_signbits, inputb_signbits;
reg [3:0]inputa;
reg [3:0]inputb;
reg [7:0]out, result;
assign inputa_signbits = {4{inputa[3]}};
assign inputb_signbits = {4{inputb[3]}};

always @(inputa or inputb or inputa_signbits or inputb_signbits)
begin
   out = {inputa_signbits,inputa} * {inputb_signbits,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.

Initial Values in Verilog

In Verilog, you can now store and pass initial values that the synthesis software previously ignored. Initial values specified in Verilog only affect the compiler output. This ensures that the synthesis results match the simulation results. You declare the initial values differently for registers and RAMs.

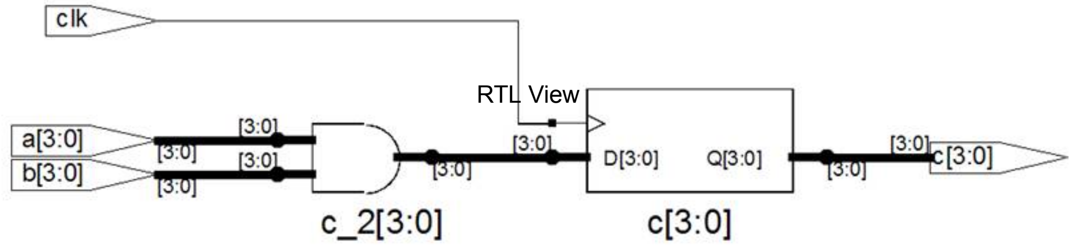
Initial Values for Registers

The synthesis compiler reads the procedural assign statements with initial values. It then stores the values, propagates them to inferred logic, and passes them down stream. The initial values only affect the output of the compiler; initial value properties are not forward-annotated to the final netlist.

If synthesis removes an unassigned register that has an initial value, the initialization values are still propagated forward. If bits of a register are unassigned, the compiler removes the unassigned bits and propagates the initial value.

In the following example (without an initial value specified), register one does not get any input. If it is not initialized, it is removed during the optimization process:

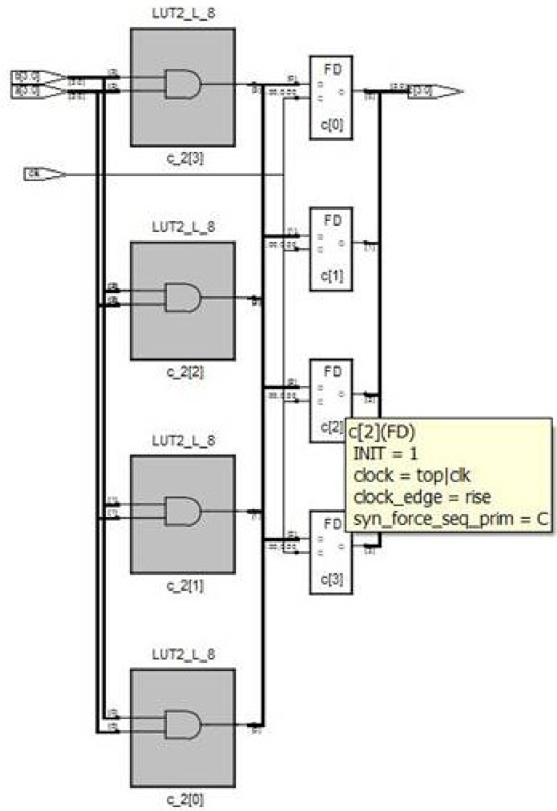
RTL View



However, if the register is initialized to a value of 1, the compiler ensures that the initial value is used in synthesis.

```
module top( a, b, c, clk);

   input clk;
   input [3:0]a, b;
   output [3:0]c;

reg [3:0]c =3'b1100;

always@(posedge clk)

   begin
      c<= a & b;
   end

endmodule
```

Technology View



c[2](FD)
INIT = 1
clock = top|clk
clock_edge = rise
syn_force_seq_prim = C

Initial Values for RAMs

You can specify initial values for a RAM in a data file and then include the appropriate task enable statement, $readmemb or $readmemh, in the initial statement of the RTL code for the module. The inferred logic can be different due to the initial statement. The syntax for these two statements is as follows:

$readmemh ("*fileName***",** *memoryName* [**,** *startAddress* [**,** *stopAddress*]]**);**

$readmemb ("*fileName***",** *memoryName* [**,** *startAddress* [**,** *stopAddress*]]**);**

$readmemb	Use this with a binary data file.
$readmemh	Use this with a hexadecimal data file.
fileName	Name of the data file that contains initial values. See Initialization Data File, on page 502 for format examples.
memoryName	The name of the memory.
startAddress	Optional starting address for RAM initialization; if omitted, defaults to first available memory location.
stopAddress	Optional stopping address for RAM initialization; *startAddress* must be specified

RAM Initialization Example

This example shows a single-port RAM that is initialized using the $readmemb binary task enable statement which reads the values specified in the binary mem.ini file. See Initialization Data File, on page 502 for details of the binary and hexadecimal file formats.

```
module ram_inference (data, clk, addr, we, data_out);
input [27:0] data;
input clk, we;
input [10:0] addr;
output [27:0] data_out;
reg [27:0] mem [0:2000] /* synthesis syn_ramstyle = "no_rw_check" */;
reg [10:0] addr_reg;

initial
begin
   $readmemb ("mem.ini", mem, 2, 1900) /* Initialize RAM with contents */
      /* from locations 2 thru 1900*/;
end

always @(posedge clk)
begin
   addr_reg <= addr;
end
```

```
always @(posedge clk)
begin
   if(we)
   begin
      mem[addr] <= data;
   end
end

assign data_out = mem[addr_reg];
endmodule
```

Initialization Data File

The initialization data file, read by the $readmemb and $readmemh system tasks, contains the initial values to be loaded into the memory array. This initialization file can reside in the project directory or can be referenced by an include path relative to the project directory. The system $readmemb or $readmemh task first looks in the project directory for the named file and, if not found, searches for the file in the list of directories on the Verilog tab in include-path order.

If the initialization data file does not contain initial values for every memory address, the unaddressed memory locations are initialized to 0. Also, if a width mismatch exists between an initialization value and the memory width, loading of the memory array is terminated; any values initialized before the mismatch is encountered are retained.

Unless an internal address is specified (see Internal Address Format, on page 504), each value encountered is assigned to a successive word element of the memory. If no addressing information is specified either with the $readmem task statement or within the initialization file itself, the default starting address is the lowest available address in the memory. Consecutive words are loaded until either the highest address in the memory is reached or the data file is completely read.

If a start address is specified without a finish address, loading starts at the specified start address and continues upward toward the highest address in the memory. In either case, loading continues upward. If both a start address and a finish address are specified, loading begins at the start address and continues until the finish address is reached (or until all initialization data is read).

For example:

```
initial
begin
//$readmemh ("mem.ini", ram_bank1)
   /* Initialize RAM with contents from locations 0 thru 31*/;

//$readmemh ("mem.ini", ram_bank1,0)
   /* Initialize RAM with contents from locations 0 thru 31*/;

$readmemh ("mem.ini", ram_bank1, 0, 31)
   /* Initialize RAM with contents from locations 0 thru 31*/;

$readmemh ("mem.ini", ram_bank2, 31, 0)
   /* Initialize RAM with contents from locations 31 thru 0*/;
```

The data initialization file can contain the following:

- White space (spaces, new lines, tabs, and form-feeds)
- Comments (both comment formats are allowed)
- Binary values for the $readmemb task, or hexadecimal values for the $readmemh tasks

In addition, the data initialization file can include any number of hexadecimal addresses (see Internal Address Format, on page 504).

Binary File Format

The binary data file mem.ini that corresponds to the example in RAM Initialization Example, on page 501 looks like this:

```
1111111111111111111100110111   /* data for address 0 */
1111111111111111111101100111   /* data for address 1 */
1111111111111111111111000010
1111111111111111111100100001
1111111111111111111101110000
1111111111111111111011100110
... /* continues until Address 1999 */
```

Hex File Format

If you use $readmemh instead of $readmemb, the hexadecimal data file for the example in RAM Initialization Example, on page 501 looks like this:

```
FFFFF37   /* data for address 0 */
FFFFF63   /* data for address 1 */
FFFFFC2
FFFFF21
.../* continues until Address 1999 */
```

Internal Address Format

In addition to the binary and hex formats described above, the initialization file can include embedded hexadecimal addresses. These hexadecimal addresses must be prefaced with an at sign (@) as shown in the example below.

```
FFFFF37  /* data for address 0 */
FFFFF63  /* data for address 1 */
@0EA     /* memory address 234
FFFFFC2  /* data for address 234*/
FFFFF21  /* data for address 235*/
...
@0A7     /* memory address 137
FFFFF77  /* data for address 137*/
FFFFF7A  /* data for address 138*/
...
```

Either uppercase or lowercase characters can be used in the address. No white space is allowed between the @ and the hex address. Any number of address specifications can be included in the file, and in any order. When the $readmemb or $readmemh system task encounters an embedded address specification, it begins loading subsequent data at that memory location.

When addressing information is specified both in the system task and in the data file, the addresses in the data file must be within the address range specified by the system task arguments; otherwise, an error message is issued, and the load operation is terminated.

Forward Annotation of Initial Values

Initial values for RAMs and sequential shift components are forward annotated to the netlist. The compiler currently generates netlist (`srs)` files with seqshift, ram1, ram2, and nram components. If initial values are specified in the HDL code, the synthesis tool attaches an attribute to the component in the `srs` file.

Cross-language Parameter Passing in Mixed HDL

The compiler supports the passing of parameters for integers, natural numbers, real numbers, and strings from Verilog to VHDL. The compiler also supports the passing of these same generics from VHDL to Verilog.

Library Directory Specification for the Verilog Compiler

Currently, if a module is instantiated in a module top without a module definition, the Verilog compiler errors out. Verilog simulators provide a command line switch (-y *libraryDirectory*) to specify a set of library directories which the compiler searches.

Library directories are specified in the Library Directories section in the Verilog panel of the Implementations Options dialog box.

Example:

If the project has one Verilog file specified

```
module foo(input a, b, output z);

foobar u1 (a, b, z);

endmodule
```

and the project directories D:/libdir and D:/lib2dir are specified as the library directories, the following is passed

c_ver *some options* **-y "D:/libdir" -y "D:/lib2dir"** *more options* **foo.v**

to the Verilog compiler. Then, if foobar.v exists in one of the specified directories, it is loaded into the compiler.

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 comment can span any number of lines but cannot be nested inside another block comment.

Scalable Modules

This section describes creating and using scalable Verilog modules. The topics include:

- Creating a Scalable Module, on page 508
- Using Scalable Modules, on page 509

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 *parameterName* **=** *value* **;**

You can define more than one parameter per declaration by using comma-separated *parameterName* **=** *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 *instanceName***.***parameterName* **=** *newValue* **;**

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

moduleName **# (***newValuesList***)** *instanceName* **(***portList***) ;**

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 a 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 must explicitly pass the parameter value down through the hierarchy.

```
parameter my_size = 16;    // I want a 16-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 16-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

gateTypeKeyword [*instanceName*] **(***portList***) ;**

The gate type keywords for simple and tristate gates are listed in the following tables. The *instanceName* is a unique instance name, and is optional. The signal names in the *portList* 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.

Keyword (Simple Gates)	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

Keyword (Tristate Gates)	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 *installDirectory*/examples/verilog/common_rtl/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 automatically 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 Comma-separated Sensitivity List, on page 465.

```
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 *installDirectory*/examples/verilog/common_rtl/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 518
- Level-sensitive Latches, on page 519
- Sets and Resets, on page 522
- SRL Inference, on page 526

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 @ (*edgeKeyword clockName***)**

where *edgeKeyword* 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 @ (*clockName* **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 @ (*edgeKeyword clockSignal* **or** *edgeKeyword resetSignal* **or**
edgeKeyword setSignal **)**

EdgeKeyword 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.

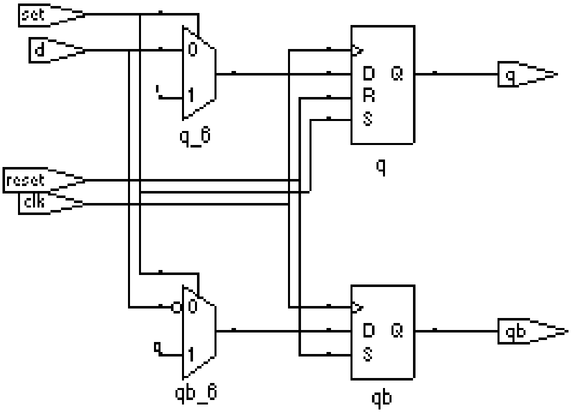
Do not list the set and reset signal names in the event expression of an always block so they do not trigger the always block to execute upon changing. Instead, trigger the always block on the active clock edge, and check the reset and set inside the always block first.

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. The following figure shows flip-flops with synchronous sets and resets.



Event Control Syntax

always @ (*edgeKeyword clockName* **)**

In the syntax line, *edgeKeyword* 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
```

SRL Inference

Sequential elements can be mapped into SRLs using an initialization assignment in the Verilog code. You can now infer SRLs with initialization values. Enable the System Verilog option on the Verilog tab of the Implementation Options dialog box before you run synthesis.

This is an example of a SRL with no resets. It has four 4-bit wide registers and a 4-bit wide read address. Registers shift when the write enable is 1.

```
module test_srl(clk, enable, dataIn, result, addr);
input clk, enable;
input [3:0] dataIn;
input [3:0] addr;
output [3:0] result;
reg [3:0] regBank[3:0]='{4'h0,4'h1,4'h2,4'h3};
integer i;
```

```
always @(posedge clk) begin
   if (enable == 1) begin
      for (i=3; i>0; i=i-1) begin
         regBank[i] <= regBank[i-1];
      end
   regBank[0] <= dataIn;
   end
end

assign result = regBank[addr];
endmodule
```

Verilog State Machines

This section describes Verilog state machines: guidelines for using them, defining state values, and dealing with asynchrony. The topics include:

- State Machine Guidelines, on page 528
- State Values, on page 530
- Asynchronous State Machines, on page 531
- Limited RAM Resources, on page 532
- Additional Glue Logic, on page 533
- Synchronous READ RAMs, on page 533
- Multi-Port RAM Extraction, on page 533

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 531, 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 529.

- 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 529.
- 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 where a non-default encoding is implemented if it produces better results. See Values for syn_encoding, on page 927 for a list of default and other encodings. When you specify a particular encoding style with syn_encoding, that value is used during the mapping stage to determine encoding style.

  object **/*synthesis syn_encoding="sequential"*/;**

  See syn_encoding Attribute, on page 927, 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

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 parameter or `define statements. It is recommended that you use parameter, for the following reasons:

- The `define is applied globally whereas parameter definitions are local. With global `define definitions, you cannot reuse common state names that you might want to use in multiple designs, like RESET, IDLE, READY, READ, WRITE, ERROR and DONE. Local definitions make it easier to reuse certain state names in multiple FSM designs. If you work around this restriction by using `undef and then redefining them with `define in the new FSM modules, it makes it difficult to probe the internal values of FSM state buses from a testbench and compare them to state names.
- The tool only displays state names in the FSM Viewer if they are defined using parameter.

Example 1: Using Parameter*s* 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 – Asynchronous FSM with Continuous Assignment

Example – Asynchronous FSM with an always Block

Verilog Guidelines for RAM Inference

This section provides Verilog guidelines and examples for coding RAMs for synthesis. It covers the following topics:

RAM Inference from Verilog Code

The synthesis tool can infer synchronous and synchronously resettable RAMs from your Verilog source code and, where appropriate, generate technology-specific single-port or dual-port RAMs. You do not need any special input in your source code like attributes or directives.

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 533, for information on coding your Verilog description to ensure that technology-specific RAM primitives with synchronous READs are used.

Single-port and dual-port RAM are supported for Microsemi technologies.

Example – RAM Inferencing

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 Glue Logic

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 inferred by the compiler (RTL view) have asynchronous READs.

Multi-Port RAM Extraction

The compiler extracts multi-write port RAMs, creating an nram primitive which can have n write ports. The nrams are implemented differently depending on your technology. For information about the implementations, see the appropriate vendor chapter.

The following aspects of multi-port RAM extraction are described here:

- Prerequisites for Multi-Port RAM Extraction, on page 534
- Write Processes and Multi-Port RAM Inference, on page 534
- Multi-Port RAM Recommended Coding Style, on page 535
- Multi-Port RAM Examples, on page 536

Prerequisites for Multi-Port RAM Extraction

In order to infer the nram, you must do the following:

- The compiler will infer the nram even if you do not register the read address, as subsequent examples show. However, for many technology families, you must register the read address for the nram to be implemented as vendor RAMs.
- If the writes are in different processes, you no longer need to specify no_rw_check for the syn_ramstyle attribute in order to infer the RAM. Note that the simulator produces x's when the two writes occur at the same time while the synthesis tool assumes that one write gets priority over the other based on the code.

Write Processes and Multi-Port RAM Inference

The compiler infers the multi-port RAMs from the write processes as follows:

- When all the writes are made in one process, there are no address conflicts, and the compiler generates an nram. The following example results in an nram with two write ports, one with write address waddr0 and the other with write address waddr1:

```
always @(posedge clk)
begin
   if(we1) mem[waddr0] <= data1;
   if(we2) mem[waddr1] <= data2;
end
```

waddr0:	we1 && (!we2 || (we2 && waddr0 != waddr1))
waddr1	we2

- When writes are made in multiple processes, the software does not infer a multi-port RAM unless there is an explicit syn_ramstyle attribute. If there is no attribute, the software does not infer a RAM. If there is an attribute, the software infers a RAM with multiple write ports. You might get a warning about simulation mismatches when the two addresses are the same.

  In the following case, the compiler infers an nram with two write ports because of the syn_ramstyle attribute. The write enable associated with waddr0 is we1 and the write enable associated with waddr1 is we2.

```
reg [1:0] mem [7:0] /* synthesis syn_ramstyle="no_rw_check" */ ;

always @(posedge clk1)
begin
   if(we1) mem[waddr0] <= data1;
end

always @(posedge clk2)
begin
   if(we2) mem[waddr1] <= data2;

end
```

Multi-Port RAM Recommended Coding Style

Currently, you must use a registered read address when you code the RAM. If you do not, you could get error messages. The following is an example of how to code the RAM:

```
module ram(data0, data1, waddr0, waddr1, we0,we1,
   clk0, clk1, q0, q1);
parameter d_width = 8;
parameter addr_width = 8;
parameter mem_depth = 256;

input [d_width-1:0] data0, data1;
input [addr_width-1:0] waddr0, waddr1;
input we0, we1, clk0, clk1;
output [d_width-1:0] q0, q1;
reg [addr_width-1:0] reg_addr0, reg_addr1;
reg [d_width-1:0] mem [mem_depth-1:0] /* synthesis syn_ramstyle="no_rw_check" */;

assign q0 = mem[reg_addr0];
assign q1 = mem[reg_addr1];
```

```
always @(posedge clk0)
begin
   reg_addr0 <= waddr0;
   if (we0)
      mem[waddr0] <= data0;
end

always @(posedge clk1)
begin
   reg_addr1 <= waddr1;
   if (we1)
      mem[waddr1] <= data1;
end

endmodule
```

There are two situations which can result in the error message: "@E:MF216 : ram.v(29) | Found NRAM mem_1[7:0] with multiple processes":

- An nram with two clocks and two write addresses has syn_ramstyle set to a value of registers. The software cannot implement this, because there is a physical FPGA limitation that does not allow registers with multiple writes.
- You have a registered output for an nram with two clocks and two write addresses.

Multi-Port RAM Examples

This section consists of the following examples of multi-port RAM extraction:

- A RAM with Two Write Ports, on page 537
- Another RAM with Two Write Ports, on page 537
- A RAM with Three Write Ports, on page 537
- A Two-Write Port RAM, on page 538
- Two Write Processes with Two Different Clocks, on page 538
- Two Write Processes with One Clock, on page 539

A RAM with Two Write Ports

```
module ram0(input [1:0] data1, data2, input [2:0] waddr0,
   waddr1, addr, input we1, we2, clk, output reg [1:0] q);
reg [1:0] mem [7:0]/* synthesis syn_ramstyle="no_rw_check" */;

always @(posedge clk)
begin
   if(we1) mem[waddr0] <= data1;
   if(we2) mem[waddr1] <= data2;
      q = mem[addr];
end
endmodule
```

Another RAM with Two Write Ports

```
module ram0(input [1:0] data0, data1, data2, data3,
   input [1:0] waddr0, waddr1, waddr2, waddr3, addr,
   input sel, sel2, we1, we2, clk, output reg [1:0] q);
reg [1:0] mem [3:0]/* synthesis syn_ramstyle="no_rw_check" */;

always @(posedge clk)
begin
   if(sel)
      mem[waddr0] <= data0;
   else if(sel2)
   begin
      if(we1) mem[waddr1] <= data1;
      if(we2) mem[waddr2] <= data2;
   end
   else
      mem[waddr3] <= data3;
      q = mem[addr];
end
endmodule
```

A RAM with Three Write Ports

```
module ram0(input [1:0] data0, data1, data2, data3, data4,
   input [1:0] waddr0, waddr1, waddr2, waddr3, waddr4, addr,
   input sel, sel2, we1, we2, clk, output reg [1:0] q);
reg [1:0] mem [3:0]/* synthesis syn_ramstyle="no_rw_check" */;
```

```
always @(posedge clk)
begin
   if(sel)
      mem[waddr0] <= data0;
   else if(sel2)
   begin
      if(we1) mem[waddr1] <= data1;
      if(we2) mem[waddr2] <= data2;
   end
   else
      mem[waddr3] <= data3;
      mem[waddr4] <= data4;
      q = mem[addr];
end
endmodule
```

A Two-Write Port RAM

```
module ram0(input [1:0] data0, data1, data2, data3,
   input [1:0] waddr0, waddr1, waddr2, waddr3, addr, input sel,
   sel2, clk, output [1:0] q);
reg [1:0] mem [3:0]; /* synthesis syn_ramstyle="no_rw_check" */
always @(posedge clk)
begin
   if(sel)
      mem[waddr0] <= data0;
   else
      mem[waddr1] <= data1;
   if(sel2)
      mem[waddr2] <= data2;
   else
      mem[waddr3] <= data3;
end
assign q = mem[addr];
endmodule
```

Two Write Processes with Two Different Clocks

This example has the writes in two processes, and generates a two-write port RAM with two different clocks.

```
module ram0(input [1:0] data1, data2, input [2:0] waddr1, addr,
input we1, we2, clk1, clk2, output reg [1:0] q);

reg [1:0] mem [7:0] /* synthesis syn_ramstyle="no_rw_check" */ ;
```

```
always @(posedge clk1)
begin
   if(we1)
      mem[addr] <= data1;
      q = mem[addr];
end

always @(posedge clk2)
begin
   if(we2) mem[waddr1] <= data2;
end

endmodule
```

Two Write Processes with One Clock

This example has the writes in two processes, and generates a two-write port RAM that uses only one clock.

```
module ram0(input [1:0] data1, data2, input [2:0] waddr0, waddr1,
   addr, input we1, we2, clk, output reg [1:0] q);
reg [1:0] mem [7:0] /* synthesis syn_ramstyle="no_rw_check" */ ;

always @(posedge clk)
begin
   if(we1) mem[waddr0] <= data1;
   q = mem[addr];
end

always @(posedge clk)
begin
   if(we2) mem[waddr1] <= data2;
end

endmodule
```

RAM Instantiation with SYNCORE

The SYNCORE Memory Compiler in the IP Wizard helps you generate HDL code for your specific RAM implementation requirements. For information on using the SYNCORE Memory Compiler, see Specifying RAMs with SYNCore, on page 376 in the *User Guide*.

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 Synopsys FPGA 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.

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 directive. If the macro is an I/O, use black_box_pad_pin=1 on the external pad pin.

For most of the technology-vendor architectures, macro libraries are provided (in *installDirectory*/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 Directive, on page 1030 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 directive (see black_box_tri_pins Directive, on page 909).

The input, output, and delay through a black box is specified with special black box timing directives (see Black Box Timing Constraints, on page 1125).

For information on how to instantiate black boxes and technology-vendor I/Os, see *Defining Black Boxes for Synthesis, on page 168* of the *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 *installDirectory*/examples/verilog/common_rtl/prep:

- PREP Benchmark 1, Data Path (prep1.v)
- PREP Benchmark 2, Timer/Counter (prep2.v)
- PREP Benchmark 3, Small State Machine (prep3.v)
- PREP Benchmark 4, Large State Machine (prep4.v)
- PREP Benchmark 5, Arithmetic Circuit (prep5.v)
- PREP Benchmark 6, 16-Bit Accumulator (prep6.v)
- PREP Benchmark 7, 16-Bit Counter (prep7.v)
- PREP Benchmark 8, 16-Bit Pre-scaled Counter (prep8.v)
- PREP Benchmark 9, Memory Map (prep9.v)

The source code for the benchmarks can be used for design examples for synthesis or for doing your own FPGA vendor comparisons.

Hierarchical or Structural Verilog Designs

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 *macroName*. 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.

textMacroDefinition **::= define** *textMacroName macroText*

textMacroName **::=** *textMacroIdentifier*[**(***formalArgumentList***)**]

formalArgumentList **::=** formalArgumentIdentifier **{,** formalArgumentIdentifier**}**

When formal arguments are used to define a text macro, the scope of the formal argument is extended 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
```

Verilog Attribute and Directive Syntax

Verilog 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.
- *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 like attributes.

Verilog does not have predefined attributes or directives for synthesis. To define directives or attributes in Verilog, attach them to the appropriate objects in the source code as comments. You can use either of the following comment styles:

- Regular line comments
- Block 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.

Attribute Syntax and Examples using Verilog Line Comments

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.

Attribute Syntax and Examples Using Verilog C-Style Comments

Here is the syntax for specifying attributes and directives with the C-style block comment:

> **/* synthesis** *directive* **|** *attribute* [**= "***value***"**] ***/**

This example shows the syn_hier attribute specified with a C-style comment:

```
/* synthesis syn_hier = "firm" */
```

The following are some other rules for using C-style comments to define attributes:

- If you use C-style comments, you must place the comment *before* the semicolon of the statement. For example:

  ```
  module bl_box(out, in) /* synthesis syn_black_box */ ;
  ```

- To specify more than one directive or attribute for a given design object, place them within the same comment, separated by a space. Do *not* use commas as separators. Here is an example where the syn_preserve and syn_state_machine directives are specified in a single comment:

  ```
  module radhard_dffrs(q,d,c,s,r)
     /* synthesis syn_preserve=1 syn_state_machine=0 */;
  ```

- To make source code more readable, you can split long block 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, note these exceptions:
  - 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 (=).

  Take this block comment specification for example:

```
/* synthesis syn_probe=1 xc_loc="P20,P21,P22,P23,P24,P25,P26,P27" */;
```

  You cannot split the line before you specify the first attribute, `syn_probe`. You cannot split the line before either of the equal signs (`syn_probe=` or `xc_loc=`). You can split it anywhere within the string value `"P20,P21,P22,P23,P24,P25,P26,P27"`.

Attribute Examples Using Verilog 2001 Parenthetical Comments

Here is the syntax for specifying attributes and directives as Verilog 2001 parenthetical comments:

(* *directive* **|** *attribute* [**= "***value***"**] ***)**

Verilog 2001 parenthetical comments can be applied to:

- individual objects
- multiple objects
- individual objects within a module definition

The following example shows two syn_keep attributes specified as parenthetical comments:

```
module example2(out1, out2, clk, in1, in2);
output out1, out2;
input clk;
input in1, in2;
wire and_out;
(* syn_keep=1 *) wire keep1;
(* syn_keep=1 *) wire keep2;
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
```

For the above example, a single parenthetical comment could be added directly to the reg statement to apply the syn_keep attribute to both out1 and out2:

```
(* syn_keep=1 *) reg out1, out2;
```

The following rules apply when using parenthetical comments to define attributes:

- Always place the comment *before* the design object (and terminating semicolon). For example:

```
(* syn_black_box *) module bl_box(out, in);
```

- To specify more than one directive or attribute for a given object, place the attributes within the same parenthetical comment, separated by a space (do *not* use commas as separators). The following example shows the syn_preserve and syn_state_machine directives applied in a single parenthetical comment:

```
(* syn_preserve=1 syn_state_machine=0 *)
   module radhard_dffrs(q,d,c,s,r);
```

- Parenthetical comments can be applied to individual objects within a module definition. For example,

```
module example2 (out1, (*syn_preserve=1*) out2, clk, in1, in2);
```

applies a syn_preserve attribute to out2, and

```
module example2 ( (*syn_preserve=1*) out1,
   (*syn_preserve=1*) out2, clk, in1, in2);
```

applies a syn_preserve attribute to both out1 and out2

CHAPTER 9

SystemVerilog Language Support

This chapter describes support for the SystemVerilog standard in the Synopsys FPGA synthesis tools. For information on the Verilog standard, see Chapter 8, *Verilog Language Support*. The following describe SystemVerilog-specific support:

Feature Summary

SystemVerilog is an IEEE (P1800) standard with extensions to the IEEE Std.1364-2001 Verilog standard. The extensions integrate features from C, C++, VHDL, OVA, and PSL. The following table summarizes the SystemVerilog features currently supported in the Synopsys FPGA Verilog compilers. See SystemVerilog Limitations, on page 559 for a list of limitations.

Feature	Brief Description
Unsized Literals	Specification of unsized literals as single-bit values without a base specifier.
Data Types • Typedefs • Enumerated Types • Struct Construct • Union Construct • Static Casting	Data types that are a hybrid of both Verilog and C including: • User-defined types that allow you to create new type definitions from existing types • Variables and nets defined with a specific set of named values • Structure data type to represent collections of variables referenced as a single name • Data type collections sharing the same memory location • Conversion of one data type to another data type.
Arrays • Arrays • Arrays of Structures	Packed, unpacked, and multi-dimensional arrays of structures.
Data Declarations • Constants • Variables • Nets • Data Types in Parameters • Type Parameters	Data declarations including constant, variable, net, and parameter data types.

Feature	Brief Description
Operators and Expressions • Operators • Aggregate Expressions • Streaming Operator • Set Membership Operator • Type Operator	C assignment operators and special bit-wise assignment operators.
Procedural Statements and Control Flow • Do-While Loops • For Loops • Unnamed Blocks • Block Name on end Keyword • Unique and Priority Modifiers	Procedural statements including variable declarations and block functions.
Processes • always_comb • always_latch • always_ff	Specialized procedural blocks that reduce ambiguity and indicate the intent.
Tasks and Functions • Implicit Statement Group • Formal Arguments • endtask /endfunction Names	Information on implicit grouping for multiple statements, passing formal arguments, and naming end statements for functions and tasks.
Hierarchy • Compilation Units • Packages • Port Connection Constructs • Extern Module	Permits sharing of language-defined data types, user-defined types, parameters, constants, function definitions, and task definitions among one or more compilation units, modules, or interfaces (pkgs)
Interface • Interface Construct • Modports	Interface data type to represent port lists and port connection lists as single name.
System Tasks and System Functions • $bits System Function • Array Querying Functions	Queries to returns number of bits required to hold an expression as a bit stream or array.

Feature	Brief Description
Generate Statement • Conditional Generate Constructs	Generate-loop, generate-conditional, or generate-case statements with defparams, parameters, and function and task declarations. Conditional if-generate and case-generate constructs
Assertions • SVA System Functions	SystemVerilog assertion support.
Keyword Support	Supported and unsupported keywords.

SystemVerilog Limitations

The following SystemVerilog limitations are present in the current release.

Interface

- An array of interfaces cannot be used as a module port.
- An interface cannot have a multi-dimensional port.
- Access of array type elements outside of the interface are not supported. For example:

```
interface ff_if (input logic din, input [7:0] DHAin1,
   input [7:0] DHAin2, output logic dout);
logic [1:0] [1:0] [1:0] DHAout_intf;

always_comb
DHAout_intf = DHAin1 + DHAin2;

modport write (input din, output dout);
endinterface: ff_if
```

- `ff_if ff_if_top(.*);`
  `DHAout = ff_if_top.DHAout_intf;` Modport definitions within a Generate block are not supported. For example:

```
interface myintf_if (input logic [7:0] a , input logic [7:0]  b,
   output logic [7:0] out1, output logic [7:0] out2);
generate
   begin: x
   genvar i;
      for (i = 0;i <= 7;i=i+1)
      begin : u
         modport myinst(input .ma(a[i]), input .mb(b[i]),
            output .mout1(out1[i]) , output .mout2(out2[i]));
      end
   end
endgenerate
endinterface
```

Compilation Unit and Package

- Write access to the variable defined in package/compilation unit is not supported. For example:

```
package MyPack;
typedef struct packed {
   int r;
   longint g;
   byte b;
}  MyStruct ;

MyStruct StructMyStruct;
endpackage: MyPack

import MyPack::*;
module top ( ...
...

always@(posedge clk)
StructMyStruct <= '{default:254};
```

Unsized Literals

SystemVerilog allows you to specify unsized literals without a base specifier (auto-fill literals) as single-bit values with a preceding apostrophe ('). All bits of the unsized value are set to the value of the specified bit.

```
'0, '1, 'X, 'x, 'Z, 'z // sets all bits to this value
```

In other words, this feature allows you to fill a register, wire, or any other data types with 0, 1, X, or Z in simple format.

Verilog Example	SystemVerilog equivalent
a = 4'b1111;	a = '1;

Data Types

SystemVerilog makes a clear distinction between an *object* and its *data type*. A data type is a set of values, or a set of operations that can be performed on those values. Data types can be used to declare data objects.

SystemVerilog offers the following data types, representing a hybrid of both Verilog and C:

Data Type	Description
shortint	2-state, SystemVerilog data type, 16-bit signed integer
int	2-state, SystemVerilog data type, 32-bit signed integer
longint	2-state, SystemVerilog data type, 64-bit signed integer
byte	2-state, SystemVerilog data type, 8-bit signed integer or ASCII character
bit	2-state, SystemVerilog data type, user-defined vector size
logic	4-state, SystemVerilog data type, user-defined vector size

Data types are characterized as either:

- 4-state (4-valued) data types that can hold 1, 0, X, and Z values
- 2-state (2-valued) data types that can hold 1 and 0 values

The following apply when using data types:

- The data types byte, shortint, int, integer and longint default to signed; data types bit, reg, and logic default to unsigned, as do arrays of these types.
- The signed keyword is part of Verilog. The unsigned keyword can be used to change the default behavior of signed data types.
- The Verilog compiler does not generate an error even if a 2-state data type is assigned X or Z. It treats it as a “don't care” and issues a warning.
- Do not use the syn_keep directive on nets with SystemVerilog data types. When you use data types such as bit, logic, longint, or shortint, the synthesis software might not be aware of the bit sizes on the LHS and RHS for the net. For example:

  ```
  bit x;
     shortint y;
     assign y =x;
  ```

  In this case, bit defaults to a 1-bit width and includes a shortint of 16-bit width. If syn_keep is applied on y, the software does not use the other 15 bits.

Typedefs

You can create your own names for type definitions that you use frequently in your code. SystemVerilog adds the ability to define new net and variable user-defined names for existing types using the typedef keyword.

Example – Simple typedef Variable Assignment

Example – Using Multiple typedef Assignments

Enumerated Types

The synthesis tools support SystemVerilog enumerated types in accordance with SV LRM section: 4.10; enumerated methods are not supported.

The enumerated types feature allows variables and nets to be defined with a specific set of named values. This capability is particularly useful in state-machine implementation where the states of the state machine can be verbally represented

Data Types

Enumerated types have a base data type which, by default, is int (a 2-state, 32-bit value). By default, the first label in the enumerated list has a logic value of 0, and each subsequent label is incremented by one.

For example, a variable that has three legal states:

```
enum {WAITE, LOAD, READY} State ;
```

The first label in the enumerated list has a logic value of 0 and each subsequent label is incremented by one. In the example above, State is an int type and WAITE, LOAD And READY have 32-bit int values. WAITE is 0, LOAD is 1, and READY is 2.

You can specify an explicit base type to allow enumerated types to more specifically model hardware. For example, two enumerated variables with one-hot values:

```
enum logic [2:0] {WAITE=3'b001, LOAD=3'b010,READY=3'b100} State;
```

Specifying Ranges

SystemVerilog enumerated types also allow you to specify ranges that are automatically elaborated. Types can be specified as outlined in the following table.

Syntax	Description
name	Associates the next consecutive number with the specified name.
name = C	Associates the constant C to the specified name.
name[*N*]	Generates N named constants in this sequence: *name0, name1,..., nameN-1*. N must be a positive integral number.

Syntax	Description
name[*N*] = C	Optionally assigns a constant to the generated named constants to associate that constant with the first generated named constant. Subsequent generated named constants are associated with consecutive values. N must be a positive integral number.
name[*N:M*]	Creates a sequence of named constants, starting with *nameN* and incrementing or decrementing until it reaches named constant *nameM*. N and M are non-negative integral numbers.
name[*N:M*] = C	Optionally assigns a constant to the generated named constants to associate that constant with the first generated named constants. Subsequent generated named constants are associated consecutive values. N and M must be positive integral numbers.

The following example declares enumerated variable vr, which creates the enumerated named constants register0 and register1, which are assigned the values 1 and 2, respectively. Next, it creates the enumerated named constants register2, register3, and register4 and assigns them the values 10, 11, and 12.

```
enum { register[2] = 1, register[2:4] = 10 } vr;
```

State-Machine Example

The following is an example state-machine design in SystemVerilog.

Example – State-machine Design

Type Casting Using Enumerated Types

By using enumerated types, you can define a type. For example:

```
typedef enum { red,green,blue,yellow,white,black } Colors;
```

The above definition assigns a unique number to each of the color identifiers and creates the new data type Colors. This new type can then be used to create variables of that type.

Valid assignment would be:

```
Colors c;

C = green;
```

Enumerated Types in Expressions

Elements of enumerated types can be used in numerical expressions. The value used in the expression is the value specified with the numerical value. For example:

```
typedef enum { red,green,blue,yellow,white,black } Colors;
integer a,b;
a = blue *3 // 6 is assigned to a
b = yellow + green; // 4 is assigned to b
```

Struct Construct

SystemVerilog adds several enhancements to Verilog for representing large amounts of data. In SystemVerilog, the Verilog array constructs are extended both in how data can be represented and for operations on arrays. A structure data type has been defined as a means to represent collections of data types. These data types can be either standard data types (such as int, logic, or bit) or, they can be user-defined types (using SystemVerilog typedef). Structures allow multiple signals, of various data types, to be bundled together and referenced by a single name.

Structures are defined under section 4.11 of IEEE Std 1800-2005 (IEEE Standard for SystemVerilog).

In the example structure floating_pt_num below, both characteristic and mantissa are 32-bit values of type bit.

```
struct {
   bit [31:0] characteristic;
   bit [31:0] mantissa;
}  floating_pt_num;
```

Alternately, the structure could be written as:

```
typedef struct {
   bit [31:0] characteristic;
   bit [31:0] mantissa;
}  flpt;
flpt floating_pt_num;
```

In the above sequence, a type flpt is defined using typedef which is then used to declare the variable floating_pt_num.

Assigning a value to one or more fields of a structure is straight-forward.

```
floating_pt_num.characteristic = 32'h1234_5678;

floating_pt_num.mantissa       = 32'h0000_0010;
```

As mentioned, a structure can be defined with fields that are themselves other structures.

```
typedef struct {
   flpt x;
   flpt y;
}  coordinate;
```

Packed Struct

Various other unique features of SystemVerilog data types can also be applied to structures. By default, the members of a structure are *unpacked*, which allows the Synopsys FPGA tools to store structure members as independent objects. It is also possible to *pack* a structure in memory without gaps between its bit fields. This capability can be useful for fast access of data during simulation and possibly result in a smaller footprint of your simulation binary.

To pack a structure in memory, use the packed keyword in the definition of the structure:

```
typedef struct packed {
   bit [31:0] characteristic;
   bit [31:0] mantissa;
} flpt;
```

An advantage of using packed structures is that one or more bits from such a structure can be selected as if the structure was a packed array. For instance, flpt[47:32] in the above declaration is the same as characteristic[15:0].

Struct members are selected using the .name syntax as shown in the following two code segments.

```
// segment 1
typedef struct {
   bit [7:0] opcode;
   bit [23:0] addr;
} instruction; // named structure type
instruction IR; // define variable
IR.opcode = 1; //set field in IR.

// segment 2
struct {
   int x,y;
} p;
p.x = 1;
```

Union Construct

A union is a collection of different data types similar to structure with the exception that members of the union share the same memory location. At any given time, you can write to any one member of the union which can then be read by the same member or a different member of that union.

Union is broadly classified as:

- Packed Union
- Unpacked Union

Currently, only packed unions are supported.

Packed Union

A packed union can only have members that are of the packed type (packed structure, packed array of logic, bit, int, etc.). All members of a packed union must be of equal size.

Syntax

Union packed
{
    *member1***;**
    *member2***;**
} *unionName***;**

Unpacked Union

The members of an unpacked union can include both packed and unpacked types (packed/unpacked structures, arrays of packed/unpacked logic, bit, int, etc.) with no restrictions as to the size of the union members.

Syntax

Union
{
    *member1***;**
    *member2***;**
} *unionName***;**

Example 1 – Basic Packed Union (logical operation)

Example 2 – Basic Packed Union (arithmetic operation)

Example 3 – Nested Packed Union

Example 4 – Array of packed Union

Limitations

The SystemVerilog compiler does not support the following union constructs:

- unpacked union
- tagged packed union
- tagged unpacked union

Currently, support is limited to packed unions, arrays of packed unions, and nested packed unions.

Static Casting

Static casting allows one data type to be converted to another data type. The static casting operator is used to change the data type, the size, or the sign:

- Type casting – a predefined data type is used as a *castingType* to change the data type.
- Size casting – a positive decimal number is used as a *castingType* to change the number of data bits.
- Sign casting – signed/unsigned are used to change the sign of data type.
- Bit-stream casting – type casting that is applied to unpacked arrays and structs. During bit-stream casting, both the left and right sides of the equation must be the same size. Arithmetic operations cannot be combined with static casting operations as is in the case of singular data types.

Syntax

castingType **'** **(***castingExpression***)**

Example – Type Casting of Singular Data Types

Example – Type Casting of Aggregate Data Types

Example – Bit-stream Casting

Example – Size Casting

Example – Sign Casting

Arrays

Topics in this section include:

- Arrays, below
- Arrays of Structures, on page 573

Arrays

SystemVerilog uses the term *packed array* to refer to the dimensions declared before the object name (same as Verilog *vector width*). The term *unpacked array* refers to the dimensions declared after the object name (same as Verilog *array dimensions*). For example:

```
reg [7:0] foo1; //packed array
reg foo2 [7:0]; //unpacked array
```

A packed array is guaranteed to be represented as a contiguous set of bits and, therefore, can be conveniently accessed as array elements. While unpacked is not guaranteed to work so, but in terms of hardware, both would be treated or bit-blasted into a single dimension.

```
module test1 (input [3:0] data, output [3:0] dout);
   //example on packed array four-bit wide.

assign dout = data;
endmodule

module test2 (input data [3:0], output dout [3:0]);
//unpacked array of 1 bit by 4 depth;

assign dout = data;
endmodule
```

Multi-dimensional packed arrays unify and extend Verilog's notion of *registers* and *memories*:

```
reg [1:0][2:0] my_var[32];
```

Classical Verilog permitted only one dimension to be declared to the left of the variable name. SystemVerilog permits any number of such *packed* dimensions. A variable of packed array type maps 1:1 onto an integer arithmetic quantity. In the example above, each element of my_var can be used in expressions as a six-bit integer. The dimensions to the right of the name (32 in this case) are referred to as *unpacked* dimensions. As in Verilog-2001, any number of unpacked dimensions is permitted.

The general rule for multi-dimensional packed array is as follows:

```
reg/wire [matrix$_n$:0] … [matrix$_1$:0][depth:0][width:0] temp;
```

The general rule for multi-dimensional unpacked array is as follows:

```
reg/wire temp1 [matrix$_n$:0]… [matrix$_1$:0][depth:0]; //single bit wide
reg/wire [width$_m$:0] temp2 [matrix$_n$:0]… [matrix$_1$:0][depth:0]; //
width$_m$ bit wide
```

The general rule for multi-dimensional array, mix of packed/unpacked, is as follows:

```
reg/wire [width$_m$:0] temp3 [matrix:0]… [depth:0];

reg/wire [depth:0][width:0] temp4 [matrix$_m$:0]… [matrix$_1$:0]
```

For example, in a multi-dimensional declaration, the dimensions declared following the type and before the name vary more rapidly than the dimensions following the name.

Multi-dimensional arrays can be used as ports of the module.

The following items are now supported for multi-dimensional arrays:

1. The assignment of a whole multi-dimensional array to another.
2. The access (reading) of an entire multi-dimensional array.
3. The assignment of an index (representing a complete dimension) of a multi-dimensional array to another.
4. The access (reading) of 3. above.
5. The assignment of a slice of a multi-dimensional array.
6. The access of a slice of a multi-dimensional array.
7. The access of a variable part-select of a multi-dimensional array.

In addition, packed arrays are supported with the access/store mechanisms listed above. Packed arrays can also be used as ports and arguments to functions and tasks. The standard multi-dimensional access of packed arrays is supported.

Unpacked array support is the same as packed array supported stated in items one through seven above.

Example – Multi-dimensional Packed Array with Whole Assignment

Example – Multi-dimensional Packed Array with Partial Assignment

Example – Multi-dimensional Packed Array with Arithmetic Ops

Example – Packed/Unpacked Array with Partial Assignment

Arrays of Structures

SystemVerilog supports multi-dimensional arrays of structures which can be used in many applications to manipulate complex data structures. A multi-dimensional array of structure is a structured array of more than one dimension. The structure can be either packed or unpacked and the array of this structure can be either packed or unpacked or a combination of packed and unpacked. As a result, there are many combinations that define a multi-dimensional array of structure.

A multi-dimensional array of structure can be declared as either anonymous type (inline) or by using a typedef (user-defined data type).

Some applications where multi-dimensional arrays of structures can be used are where multi-channeled interfaces are required such as packet processing, dot-product of floating point numbers, or image processing.

Example – Multi-dimensional Array of Packed and Unpacked Structures Using typedef

Example – Multi-dimensional Array of UnPacked Structures Using typedef

Example – Multi-dimensional Array of Packed Structures Using Anonymous Type

Example – Multi-dimensional Array of Packed Structures Using typedef

Array Querying Functions

SystemVerilog provides system functions that return information about a particular dimension of an array. For information on this function, see Array Querying Functions, on page 619.

Data Declarations

There are several data declarations in SystemVerilog: *literals*, *parameters*, *constants*, *variables*, *nets*, and *attributes*. The following are described here:

- Constants, below
- Variables, on page 575
- Nets, on page 575
- Data Types in Parameters, on page 576
- Type Parameters, on page 576

Constants

Constants are named data variables, which never change. A typical example for declaring a constant is as follows:

```
const a = 10;

const logic  [3:0] load = 4'b1111;

const reg  [7:0] load1 = 8'h0f, dataone = '1;
```

The Verilog compiler generates an error if constant is assigned a value.

```
const shortint a = 10;
assign a = '1;     // This is illegal
```

Variables

Variables can be declared two ways:

Method 1	Method 2

shortint a, b;
logic [1:0] c, d;
	var logic [15:0] a;
	var a,b; // equivalent var logic a, b
	var [1:0] c, d; // equivalent var logic [1:0] c, d
	input var shortint datain1,datain2;
	output var logic [15:0] dataout1,dataout2;

Method 2 uses the keyword var to preface the variable. In this type of declaration, a data type is optional. If the data type is not specified, logic is inferred.

Typical module declaration:

```
module test01 (input var shortint datain1,datain2,
   output var logic [15:0] dataout1,dataout2 );
```

A variable can be initialized as follows:

```
var  a = 1'b1;
```

Nets

Nets are typically declared using the wire keyword. Any 4-state data type can be used to declare a net.

```
wire logic tmp;
```

However, a lexical restriction applies to a net or port declaration. The Verilog net type keyword wire cannot be followed by reg.

```
input wire reg a; // Illegal
```

Data Types in Parameters

In SystemVerilog with different data types being introduced, the *parameter* can be of any data type (i.e., language-defined data type, user-defined data type, and packed/unpacked arrays and structures). By default, parameter is the int data type.

Syntax

parameter *dataType varaibleName* **=** *value*

In the above syntax, *dataType* is a language-defined data type, user-defined data type, or a packed/unpacked structure or array.

Example – Parameter is of Type longint

Example – Parameter is of Type enum

Example – Parameter is of Type structure

Example – Parameter is of Type longint Unpacked Array

Type Parameters

SystemVerilog includes the ability for a parameter to also specify a data type. This capability allows modules or instances to have data whose type is set for each instance – these *type* parameters can have different values for each of their instances.

Note: Overriding a type parameter with a defparam statement is illegal.

Syntax

parameter type *typeIdentifierName* **=** *dataType***;**

localparam type *typeIdentifierName* **=** *dataType***;**

In the above syntax, *dataType* is either a language-defined data type or a user-defined data type.

Example – Type Parameter of Language-Defined Data Type

```
//Compilation Unit
module top
#(
   parameter type PTYPE = shortint,
   parameter type PTYPE1 = logic[3:2][4:1] //parameter is of
      //2D logic type
)
(
//Input Ports
   input PTYPE din1_def,
   input PTYPE1 din1_oride,

//Output Ports
   output PTYPE dout1_def,
   output PTYPE1 dout1_oride
);

sub u1_def //Default data type
(
   .din1(din1_def),
   .dout1(dout1_def)
);

sub #
(
   .PTYPE(PTYPE1) //Parameter type is override by 2D Logic
)
u2_oride
(
   .din1(din1_oride),
   .dout1(dout1_oride)
);

endmodule
```

```
//Sub Module
module sub
#(
   parameter type PTYPE = shortint //parameter is of shortint type
)
(
//Input Ports
   input PTYPE din1,
//Output Ports
   output PTYPE dout1
);

always_comb
begin
   dout1 = din1 ;
end
endmodule
```

Example – Type Parameter of User-Defined Data Type

```
//Compilation Unit
typedef logic [0:7]Logic_1DUnpack[2:1];
typedef struct {
   byte R;
   int B;
   logic[0:7]G;
}  Struct_dt;

module top
#(
   parameter type PTYPE = Logic_1DUnpack,
   parameter type PTYPE1 = Struct_dt
)
(
//Input Ports
   input PTYPE1    din1_oride,
//Output Ports
   output PTYPE1    dout1_oride
);
```

```
sub #
(
   .PTYPE(PTYPE1) //Parameter type is override by a structure type
)
u2_oride
(
   .din1(din1_oride),
   .dout1(dout1_oride)
);

endmodule

//Sub Module
module sub
#(
   parameter type PTYPE = Logic_1DUnpack // Parameter 1D
      // logic Unpacked data type
)
(
//Input Ports
   input PTYPE din1,
//Output Ports
   output PTYPE dout1
);

always_comb
begin
   dout1.R = din1.R;
   dout1.B = din1.B ;
   dout1.G = din1.G ;
end
endmodule
```

Example – Type Local Parameter

```
//Compilation Unit
module sub
#(
parameter type PTYPE1 = shortint, //Parameter is of shortint type
parameter type PTYPE2 = longint //Parameter is of longint type
)
```

```
(
//Input Ports
   input PTYPE1 din1,
//Output Ports
   output PTYPE2 dout1
);

//Localparam type definitation
localparam type SHORTINT_LPARAM = PTYPE1;
SHORTINT_LPARAM sig1;
assign sig1 = din1;
assign dout1 = din1 * sig1;
endmodule
```

Operators and Expressions

Topics in this section include:

- Operators, below
- Aggregate Expressions, on page 583
- Streaming Operator, on page 584
- Set Membership Operator, on page 585
- Type Operator, on page 586

Operators

SystemVerilog includes the C assignment operators and special bit-wise assignment operators:

+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, <<<=, >>>=

An assignment operator is semantically equivalent to a blocking assignment with the exception that the expression is only evaluated once.

Operator Example	Same as
`A += 2;`	`A = A + 2;`
`B -= B;`	`B = B - A;`
`C *= B;`	`C = C * B;`
`D /= C;`	`D = D / C;`
`E %= D;`	`E = E % D;`
`F &= E;`	`F = F & E;`
`G |= F;`	`G = G | F;`
`H ^= G;`	`H = H ^ G;`
`I <<= H;`	`I = I << H;`

Operator Example	Same as
`J >>= I;`	`J = J  >> I;`
`K <<<=J;`	`K = K  <<< J;`
`L >>>=K;`	`L = L  >>> K;`

In addition, SystemVerilog also has the increment/decrement operators *i*++, *i*--, ++*i*, and --*i*.

Operator Example	Same as
`A++;`	A = A + 1;
`A--;`	A = A - 1;
`++A;`	Increment first and then use A
`--A;`	Decrement first and then use A

In the following code segment, out1 gets r1 and out2 gets the twice-decremented value of out1:

```
always @(*)
   begin
      out1 = r1--;
      out2 = --r1;
   end
```

Aggregate Expressions

Aggregate expressions (aggregate pattern assignments) are primarily used to initialize and assign default values to unpacked arrays and structures.

Syntax

SystemVerilog aggregate expressions are constructed from braces; an apostrophe prefixes the opening (left) brace.

'{ *listofValues* **}**

In the syntax, *listofValues* is a comma-separated list. SystemVerilog also provides a mechanism to initialize all of the elements of an unpacked array by specifying a default value within the braces using the following syntax:

'{ default: *value* **}**
'{ data type:*value* **}**
'{ index:*value* **}**

The aggregate (pattern) assignment can be used to initialize any of the following.

- a 2-dimensional unpacked array under a reset condition (see Initializing Unpacked Array Under Reset Condition example).
- all the elements of a 2-dimensional unpacked array to a default value using the default keyword under a reset condition (see Initializing Unpacked Array to Default Value example).
- a specific data type using the keyword for *type* instead of default (see Initializing Specific Data Type example).

Aggregate (pattern) assignment can also be specified in a package (see Aggregate Assignment in Package example) and in a compilation unit (see Aggregate Assignment in Compilation Unit example).

Example – Initializing Unpacked Array Under Reset Condition

Example – Initializing Unpacked Array to Default Value

Example – Initializing Specific Data Type

Example – Aggregate Assignment in Package

Example – Aggregate Assignment in Compilation Unit

Streaming Operator

The streaming operator (>> or <<) packs the bit-stream type to a particular sequence of bits in a user-specified order. Bit-stream types can be any integral, packed or unpacked type or structure. The streaming operator can be used on either the left or right side of the expression.

The streaming operator determines the order of bits in the output data stream:

- The left-to-right operator (>>) arranges the output data bits in the same order as the input bit stream
- The right-to-left operator (<<) arranges the output data bits in reverse order from the input bit stream

Syntax

streamingExpression **::= {** *streamOperator* [*sliceSize*] *streamConcatenation* **}**

streamOperator **::= >> | <<**

sliceSize **::=** *dataType* | *constantExpression*

streamConcatenation **::= {***streamExpression* {, *streamExpression*} **}**

streamExpression **::=** *arrayRangeExpression*

When an optional *sliceSize* value is included, the stream is broken up into the slice-size segments prior to performing the specified streaming operation. By default, the *sliceSize* value is 1.

Usage

The streaming operator is used to:

- Reverse the entire data stream
- Bit-stream from one data type to other

When the slice size is larger than the data stream, the stream is left-justified and zero-filled on the right. If the data stream is larger than the left side variable, an error is reported.

Example – Packed type inputs/outputs with RHS operator

Example – Unpacked type inputs/outputs with RHS operator

Example – Packed type inputs/outputs with LHS operator

Example – Slice-size streaming with RHS operator

Example – Slice-size streaming with LHS slice operation

Set Membership Operator

The set membership operator, also referred to as the *inside* operator, returns the value TRUE when the expression value (i.e., the LHS of the operator) is present in the value list of the RHS operator. If the expression value is not present in the RHS operator, returns FALSE.

Syntax

(*expressionValue***) inside {***listofValues***}**

expressionValue **::=** *singularExpression*

listofValues **::=** *rangeofValues*, *expressions*, *arrayofAggregateTypes*

Example – Inside operator with dynamically changing input at LHS operator

Example – Inside operator with expression at LHS operator

Example – Inside operator with dynamically changing input at LHS and RHS operators

Example – Inside operator with array of parameter at LHS operator

Type Operator

SystemVerilog provides a type operator as a way of referencing the data type of a variable or an expression.

Syntax

type(*dataType* | *expression***)**

dataType – a user-defined data type or language-defined data type

expression – any expression, variable, or port

An e*xpression* inside the type operator results in a self-determined type of expression; the expression is not evaluated. Also the *expression* cannot contain any hierarchical references.

Data Declaration

The type operator can be used while declaring signals, variables, or ports of a module/interface or a member of that interface.

Example – Using Type Operator to Declare Input/Output Ports

```
typedef logic signed[4:1]logicdt;
// Module top
module top(
   input type(logicdt) d1,
   output type(logicdt) dout1 );
type(logicdt) sig;
```

```
var type(logicdt) sig1;
assign sig  = d1;
assign sig1= d1+1'b1;
assign dout1= sig + sig1;
endmodule
```

Data Type Declaration

Defining of the user-defined data type can have the type operator, wherein a variable or another user-defined data type can be directly referenced while defining a data type using the type operator. The data type can be defined in the compilation unit, package, or inside the module or interface.

Example – Using Type Operator to Declare Unpacked Data Type

```
typedef logic[4:1] logicdt;
typedef type(logicdt)Unpackdt[2:1];

module top(
   input Unpackdt d1,
   output Unpackdt dout1 );
assign dout1[2] = d1[2];
assign dout1[1] = d1[1];
endmodule
```

Type Casting

The type operator can be used to directly reference the data type of a variable or port, or can be user-defined and used in type casting to convert either signed to unsigned or unsigned to signed.

Example – Using Type Operator to Reference Data Type

```
typedef logic [20:0]dt;
//Module top
module top (
   input byte d1,d2,
   output int unsigned dout1 );
assign dout1 = type(dt)'(d1 * d2);
endmodule
```

Defining Type Parameter/Local Parameter

The type operator can be used when defining a Type parameter to define the data type. The definition can be overridden based on user requirements.

Example – Using Type Operator to Declare Parameter Type Value

```
// Module top
module top(
   input byte a1,
   input byte a2,
   output shortint dout1 );
parameter type dtype = type(a1);
dtype sig1;
assign sig1 = a1;
assign dout1 = ~sig1;
endmodule
```

Comparison and Case Comparison

The type operator can be used to compare two types when evaluating a condition or a case statement.

Example – Using Type Operator in a Comparison

```
// Module top
module top (
   input byte d1,
   input shortint d2,
   output shortint dout1 );

always_comb begin
   if(type(d1) == type(d2))
      dout1 = d1;
   else
      dout1 = d2;
end
endmodule
```

Limitations

The type operator is not supported on complex expressions (for example type(d1*d2)).

Procedural Statements and Control Flow

Topics in this section include

Do-While Loops

The while statement executes a loop for as long as the loop-control test is true. The control value is tested at the *beginning* of each pass through the loop. However, a while loop does not execute at all if the test on the control value is false the first time the loop is encountered. This top-testing behavior can require extra coding prior to beginning the while loop, to ensure that any output variables of the loop are consistent.

SystemVerilog enhances the for loop and adds a do-while loop, the same as in C. The control on the do-while loop is tested at the *end* of each pass through the loop (instead of at the beginning). This implies that each time the loop is encountered in the execution flow, the loop statements are executed at least once.

Because the statements within a do-while loop are going to execute at least once, all the logic for setting the outputs of the loop can be placed inside the loop. This bottom-testing behavior can simplify the coding of while loops, making the code more concise and more intuitive.

Example – Simple Do-while Loop

Example – Do-while with If Else Statement

Example – Do-while with Case Statement

For Loops

SystemVerilog simplifies declaring local variables for use in for loops. The declaration of the for loop variable can be made within the for loop. This eliminates the need to define several variables at the module level, or to define local variables within named begin...end blocks as shown in the following example.

Example – Simple for Loop

A variable defined as in the example above, is local to the loop. References to the variable name within the loop see the local variable, however, reference to the same variable outside the loop encounters an error. This type of variable is created and initialized when the for loop is invoked, and destroyed when the loop exits.

SystemVerilog also enhances for loops by allowing more than one initial assignment statement. Multiple initial or step assignments are separated by commas as shown in the following example.

Example – For Loop with Two Variables

Unnamed Blocks

SystemVerilog allows local variables to be declared in unnamed blocks.

Example – Local Variable in Unnamed Block

Block Name on end Keyword

SystemVerilog allows a block name to be defined after the end keyword when the name matches the one defined on the corresponding begin keyword. This means, you can name the start and end of a begin statement for a block. The additional name does not affect the block semantics, but does serve to enhance code readability by documenting the statement group that is being completed.

Example – Including Block Name with end Keyword

Unique and Priority Modifiers

SystemVerilog adds unique and priority modifiers to use in case statements. The Verilog full_case and parallel_case statements are located inside of comments and are ignored by the Verilog simulator. For synthesis, full_case and parallel_case directives instruct the tool to take certain actions or perform certain optimizations that are unknown to the simulator.

To prevent discrepancies when using full_case and parallel_case directives and to ensure that the simulator has the same understanding of them as the synthesis tool, use the priority or unique modifier in the case statement. The priority and unique keywords are recognized by all tools, including the Verilog simulators, allowing all tools to have the same information about the design.

The following table shows how to substitute the SystemVerilog unique and priority modifiers for Verilog full_case and parallel_case directives for synthesis.

Verilog using full_case, parallel_case	SystemVerilog using unique/priority case modifiers

`case (...)`
`...`
`endcase`
	`case (...)`
	`...`
	`endcase`

`case (...) //full_case`
`...`
`endcase`
	`priority case (...)`
	`...`
	`endcase`

`case (...) //parallel_case`
`...`
`endcase`
	`unique case (...)`
	`...`
	`default : ...`
	`endcase`

`case (...) //full_case parallel_case`
`...`
`endcase`
	`unique case (...)`
	`...`
	`endcase`

Example – Unique Case

Example – Priority Case

Processes

In Verilog, an “if” statement with a missing “else” condition infers an unintentional latch element, for which the Synopsys FPGA compiler currently generates a warning. Many commercially available compilers do not generate any warning, causing a serious mismatch between intention and inference. SystemVerilog adds three specialized procedural blocks that reduce ambiguity and clearly indicate the intent:

- always_comb, on page 593
- always_latch, on page 595
- always_ff, on page 596

Use them instead of the Verilog general purpose always procedural block to indicate design intent and aid in the inference of identical logic across synthesis, simulation, and formal verification tools.

always_comb

The SystemVerilog always_comb process block models combinational logic, and the logic inferred from the always_comb process must be combinational logic. The Synopsys FPGA compiler warns you if the behavior does not represent combinational logic.

The semantics of an always_comb block are different from a normal always block in these ways:

- It is illegal to declare a sensitivity list in tandem with an always_comb block.
- An always_comb statement cannot contain any block, timing, or event controls and fork, join, or wait statements.

Note the following about the always_comb block:

- There is an inferred sensitivity list that includes all the variables from the RHS of all assignments within the always_comb block and variables used to control or select assignments See Examples of Sensitivity to LHS and RHS of Assignments, on page 595.
- The variables on the LHS of the expression should not be written by any other processes.

- The always_comb block is guaranteed to be triggered once at time zero after the initial block is executed.
- always_comb is sensitive to changes within the contents of a function and not just the function arguments, unlike the always@(*) construct of Verilog 2001.

Example – always_comb Block

Invalid Use of always_comb Block

The following code segments show use of the construct that are *NOT VALID*.

```
always_comb @(a or b) //Wrong. Sensitivity list is inferred not
   //declared
begin
   foo;
end

always_comb
begin
   @clk out <=in; //Wrong to use trigger within this always block
end

always_comb
begin
   fork //Wrong to use fork-join within this always block
   out <=in;
   join
end

always_comb
begin
   if(en)mem[waddr]<=data; //Wrong to use trigger conditions
      //within this block
end
```

Examples of Sensitivity to LHS and RHS of Assignments

In the following code segment, sensitivity only to the LHS of assignments causes problems.

```
always @(y)
   if (sel)
      y= a1;
   else
      y= a0;
```

In the following code segment, sensitivity only to the RHS of assignments causes problems.

```
always @(a0, a1)
   if (sel)
      y= a1;
   else
      y= a0;
```

In the following code segment, sensitivity to the RHS of assignments and variables used in control logic for assignments produces correct results.

```
always @(a0, a1, sel)
   if (sel)
      y= a1;
   else
      y= a0;
```

always_latch

The SystemVerilog always_latch process models latched logic, and the logic inferred from the always_latch process must only be latches (of any kind). The Synopsys FPGA compiler warns you if the behavior does not follow the intent.

Note the following:

- It is illegal for always_latch statements to contain a sensitivity list, any block, timing, or event controls, and fork, join, or wait statements.
- The sensitivity list of an always_latch process is automatically inferred by the compiler and the inferring rules are similar to the always_comb process (see always_comb, on page 593).

Example – always_latch Block

Invalid Use of always_latch Block

The following code segments show use of the construct that are *NOT VALID*.

```
always_latch
begin
   if(en)
      treg<=1;
   else
      treg<=0; //Wrong to use fully specified if statement
end

always_latch
begin
   @(clk)out <=in; //Wrong to use trigger events within this
      //always block
end
```

always_ff

The SystemVerilog always_ff process block models sequential logic that is triggered by clocks. The compiler warns you if the behavior does not represent the intent. The always_ff process has the following restrictions:

- An always_ff block must contain only one event control and no blocking timing controls.
- Variables on the left side of assignments within an always_ff block must not be written to by any other process.

Example – always_ff Block

Invalid Use of always_ff Block

The following code segments show use of the construct that are *NOT VALID*.

```
always_ff @(posedge clk or negedge rst)
begin
   if(rst)
      treg<=in; //Illegal; wrong polarity for rst in the
         //sensitivity list and the if statement
end
```

```
always_ff
begin
   @(posedgerst)treg<=0;
   @(posedgeclk)treg<=in; //Illegal; two event controls
end

always_ff @(posedge clk or posedge rst)
begin
   treg<=0; //Illegal; not clear which trigger is to be
               // considered clk or rst
end
```

Tasks and Functions

Support for task and function calls includes the following:

- Implicit Statement Group
- Formal Arguments, on page 598
- endtask /endfunction Names, on page 601

Implicit Statement Group

Multiple statements in the task or function definition do not need to be placed within a begin…end block. Multiple statements are implicitly grouped, executed sequentially as if they are enclosed in a begin…end block.

```
/* Statement grouping */
function int incr2(int a);
   incr2 = a + 1;
   incr2 = incr2 + 1;
endfunction
```

Formal Arguments

This section includes information on passing formal arguments when calling functions or tasks. Topics include:

- Passing Arguments by Name
- Default Direction and Type
- Default Values

Passing Arguments by Name

When a task or function is called, SystemVerilog allows for argument values to be passed to the task/function using formal argument names; order of the formal arguments is not important. As in instantiations in Verilog, named argument values can be passed in any order, and are explicitly passed through to the specified formal argument. The syntax for the named argument passing is the same as Verilog's syntax for named port connections to a module instance. For example:

```
/* General functions */
function [1:0] inc(input [1:0] a);
   inc = a + 1;
endfunction
function [1:0] sel(input [1:0] a, b, input s);
   sel = s ? a : b;
endfunction

/* Tests named connections on function calls */
assign z0 = inc(.a(a));
assign z2 = sel(.b(b), .s(s), .a(a));
```

Default Direction and Type

In SystemVerilog, input is the default direction for the task/function declaration. Until a formal argument direction is declared, all arguments are assumed to be inputs. Once a direction is declared, subsequent arguments will be the declared direction, the same as in Verilog.

The default data type for task/function arguments is logic, unless explicitly declared as another variable type. (In Verilog, each formal argument of a task/function is assumed to be reg). For example:

```
/* Tests default direction of argument */
function int incr1(int a);
   incr1 = a + 1;
endfunction
```

In this case, the direction for a is input even though this is not explicitly defined.

Default Values

SystemVerilog allows an optional default value to be defined for each formal argument of a task or function. The default value is specified using a syntax similar to setting the initial value of a variable. For example:

```
function int testa(int a = 0, int b, int c = 1);
   testa = a + b + c;
endfunction

task testb(int a = 0, int b, int c = 1, output int d);
   d = a + b + c;
endtask
```

When a task/function is called, it is not necessary to pass a value to the arguments that have default argument values. If nothing is passed to the task/function for that argument position, the default value is used. Specifying default argument values allows a task/function definition to be used in multiple ways. Verilog requires that a task/function call have the exact same number of argument expressions as the number of formal arguments. SystemVerilog allows the task/function call to have fewer argument expressions than the number of formal arguments. A task/function call must pass a value to an argument, if the formal definition of the argument does not have a default value. Consider the following examples:

```
/* functions With positional associations and missing arguments */
assign a = testa(,5); /* Same as testa(0,5,1) */
assign b = testa(2,5); /* Same as testa(2,5,1) */
assign c = testa(,5,); /* Same as testa(0,5,1) */
assign d = testa(,5,7); /* Same as testa(0,5,7) */
assign e = testa(1,5,2); /* Same as testa(1,5,2) */

/* functions With named associations and missing arguments */
assign k = testa(.b(5)); /* Same as testa(0,5,1) */
assign l = testa(.a(2),.b(5)); /* Same as testa(2,5,1) */
assign m = testa(.b(5)); /* Same as testa(0,5,1) */
assign n = testa(.b(5),.c(7)); /* Same as testa(0,5,7) */
assign o = testa(.a(1),.b(5),.c(2)); /* Same as testa(1,5,2) */
```

In general, tasks are not supported outside the scope of a procedural block (even in previous versions). This is primarily due to the difference between tasks and function.

Here are some task examples using default values:

```
always @(*)
begin
/* tasks With named associations and missing arguments */
testb(.b(5),.d(f)); /* Same as testb(0,5,1) */
testb(.a(2),.b(5),.d(g)); /* Same as testb(2,5,1) */
testb(.b(5),.d(h)); /* Same as testb(0,5,1) */
testb(.b(5),.c(7),.d(i)); /* Same as testb(0,5,7) */
testb(.a(1),.b(5),.c(2),.d(j)); /* Same as testb(1,5,2) */

/* tasks With positional associations and missing arguments */
testb(,5,,p); /* Same as testb(0,5,1) */
testb(2,5,,q); /* Same as testb(2,5,1) */
testb(,5,,r); /* Same as testb(0,5,1) */
testb(,5,7,s); /* Same as testb(0,5,7) */
testb(1,5,2,t); /* Same as testb(1,5,2) */
```

endtask /endfunction Names

SystemVerilog allows a name to be specified with the endtask or endfunction keyword. The syntax is:

endtask : *taskName*

endfunction : *functionName*

The space before and after the colon is optional. The name specified must be the same as the name of the corresponding task or function as shown in the following example.

```
/* Function w/ statement grouping, also has an endfunction label */

function int incr3(int a);
   incr3 = a + 1;
   incr3 = incr3 + 1;
   incr3 = incr3 + 1;
endfunction : incr3

/* Test with a task - also has an endtask label */
task task1;
input [1:0] in1,in2,in3,in4;
output [1:0] out1,out2;
   out1 = in1 | in2;
   out2 = in3 & in4;
endtask : task1
```

```
/* Test with a task - some default values */
task task2(
input [1:0] in1=2'b01,in2= 2'b10,in3 = 2'b11,in4 = 2'b11,
output [1:0] out1 = 2'b10,out2);

   out2 = in3 & in4;
endtask : task2

/* Tests default values for arguments */
function int dflt0(input int a = 0, b = 1);
   dflt0 = a + b;
endfunction

/* Call to function with default direction */
assign z1 = incr1(3);
assign z3 = incr2(3);
assign z4 = incr3(3);
assign z9 = dflt0();
assign z10 = dflt0(.a(7), .b());
always @(*)
begin
   task1(.in1(in1), .out2(z6), .in2(in2), .out1(z5),
      .in3(in3), .in4(in4));
   task1(in5, in6, in7, in8, z7, z8);
   task2(in5, in6, in7, in8, z11, z12);
   task2(in5, in6, , , z13, z14);
   task2(.out1(z15), .in1(in5), .in2(in6), .out2(z16),
      .in3(in7), .in4(in8));
   task2(.out2(z18), .in2(in6), .in1(in5), .in3(),
      .out1(z17), .in4());
end
```

Hierarchy

Topics in this section include:

- Compilation Units, below
- Packages, on page 605
- Port Connection Constructs, on page 606
- Extern Module, on page 609

Compilation Units

Compilation units allow declarations to be made outside of a package, module, or interface boundary. These units are visible to all modules that are compiled at the same time.

A compilation unit's scope exists only for the source files that are compiled at the same time; each time a source file is compiled, a compilation unit scope is created that is unique to only that compilation.

Syntax

//$unit definitions

*declarations***;**

//End of $unit

module ();
. . .
. . .
. . .
endmodule

In the above syntax, declarations can be variables, nets, constants, user-defined data types, tasks, or functions

Usage

Compilation units can be used to declare variables and nets, constants, user-defined data types, tasks, and functions as noted in the following examples.

A variable can be defined within a module as well as within a compilation unit. To reference the variable from the compilation unit, use the **$unit::***variableName* syntax. To resolve the scope of a declaration, local declarations must be searched first followed by the declarations in the compilation unit scope.

Example – Compilation Unit Variable Declaration

Example – Compilation Unit Net Declaration

Example – Compilation Unit Constant Declaration

Example – Compilation Unit User-defined Datatype Declaration

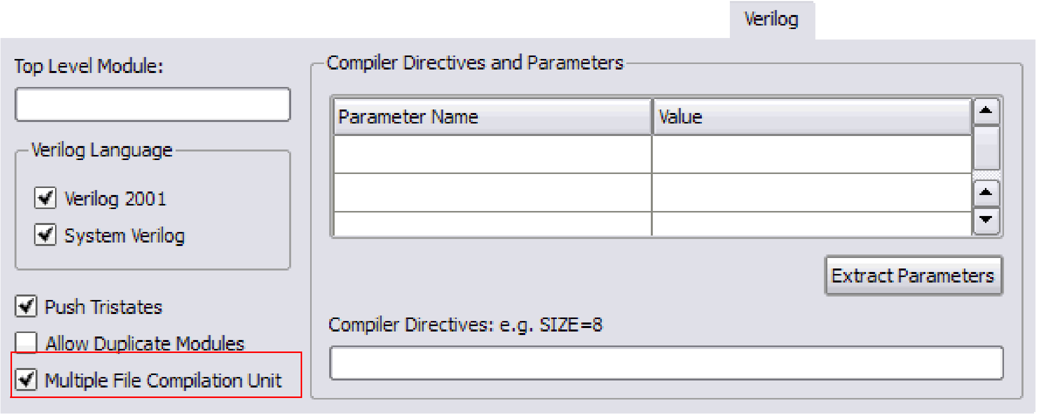
Example – Compilation Unit Task Declaration

Example – Compilation Unit Function Declaration

Example – Compilation Unit Access

Example – Compilation Unit Scope Resolution

To use the compilation unit for modules defined in multiple files, enable the Multiple File Compilation Unit check box on the Verilog tab of the Implementation Options dialog box as shown below.



☑ Verilog 2001
☑ System Verilog
☑ Push Tristates
□ Allow Duplicate Modules
☑ Multiple File Compilation Unit

You can also enable this compiler directive by including the following Tcl command in your project (prj) file:

```
set_option -multi_file_compilation_unit 1
```

Limitations

Compilation unit elements can only be accessed or read, and cannot appear between module and endmodule statements.

Packages

Packages permit the sharing of language-defined data types, typedef user-defined types, parameters, constants, function definitions, and task definitions among one or more compilation units, modules, or interfaces. The concept of packages is leveraged from the VHDL language.

Syntax

SystemVerilog packages are defined between the keywords package and endpackage.

package packageIdentifier;

packageItems

endpackage : packageIdentifier

PackageItems includes user-defined data types, parameter declarations, constant declarations, task declarations, function declarations, and import statements from other packages. To resolve the scope of any declaration, the local declarations are always searched before declarations in packages.

Referencing Package Items

As noted in the following examples, package items can be referenced by:

- Direct reference using a scope resolution operator (::). The scope resolution operator allows referencing a package by the package name and then selecting a specific package item.
- Importing specific package items using an import statement to import specific package items into a module.
- Importing package items using a wildcard (*) instead of naming a specific package item.

Example – Direct Reference Using Scope Resolution Operator (::)

Example – Importing Specific Package Items

Example – Wildcard (*) Import Package Items

Example – User-defined Data Types (typedef)

Example – Parameter Declarations

Example – Constant Declarations

Example – Task Declarations

Example – Function Declarations

Example – import Statements from Other Packages

Example – Scope Resolution

Limitations

The variables declared in packages can only be accessed or read; package variables cannot be written between a module statement and its end module statement.

Port Connection Constructs

Instantiating modules with a large number of ports is unnecessarily verbose and error-prone in Verilog. The SystemVerilog *.name* and “.*” constructs extend the 1364 Verilog feature of allowing named port connections on instantiations, to implicitly instantiate ports.

.name Connection

The SystemVerilog *.name* connection is semantically equivalent to a Verilog named port connection of type *.port_identifier(name)*. Use the *.name* construct when the name and size of an instance port are the same as those on the module. This construct eliminates the requirement to list a port name twice when both the port name and signal name are the same and their sizes are the same as shown below:

```
module myand(input [2:0] in1, in2, output [2:0] out);
...
endmodule

module foo (….ports….)
wire [2:0] in1, out;
wire [7:0] tmp;
wire [7:0] in2 = tmp;
myand mand1(.in1, .out, .in2(tmp[2:0]));  // valid
```

Note: SystemVerilog *.name* connection is currently not supported for mixed-language designs.

Restrictions to the *.name* feature are the same as the restrictions for named associations in Verilog. In addition, the following restrictions apply:

- Named associations and positional associations cannot be mixed:

  ```
  myand mand2(.in1, out, tmp[2:0]);
  ```

  The syntax above is not valid because ...
- Sizes must match in mixed named and positional associations. The example below is not valid because of the size mismatch on in2.

  ```
  myand mand3(.in1, .out, .in2);
  ```
- The identifier referred by the *.name* must not create an implicit declaration, regardless of the compiler directive ‘*default_nettype*.
- You cannot use the *.name* connection to create an implicit cast.
- Currently, the *.name* port connection is not supported for mixed HDL source code.

.* Connection

The SystemVerilog “.*” connection is semantically identical to the default *.name* connection for every port in the instantiated module. Use this connection to implicitly instantiate ports when the instance port names and sizes match the connecting module’s variable port names and sizes. The implicit .*

port connection syntax connects all other ports on the instantiated module.Using the .* connection facilitates the easy instantiation of modules with a large number of ports and wrappers around IP blocks.

The ".*" connection can be freely mixed with *.name* and *.port_identifier*(*name*) type connections. However, it is illegal to have more than one ".*" expression per instantiation.

The use of ".*" facilitates easy instantiation of modules with a large number of ports and wrappers around IP blocks as shown in the code segment below:

```
module myand(input [2:0] in1, in2, output [2:0] out);
...
endmodule

module foo (….ports….)
wire [2:0] in1, in2, out;
wire [7:0] tmp;

myand and1(.*); // Correct usage, connect in1, in2, out
myand and2(.in1, .*) // Correct usage, connect in2 and out
myand and3(.in1(tmp[2:0]), .*); // Correct Usage, connect
   // in2 and out
myand and5(.in1, .in2, .out, .*); //Correct Usage, ignore the .*
```

Note: SystemVerilog “.*” connection is currently not supported for mixed-language designs.

Restrictions to the .* feature are the same as the restrictions for the *.name* feature. See .name Connection, on page 606. In addition, the following restrictions apply:

- Named associations and positional associations cannot be mixed. For example

  ```
  myand and4(in1, .*);
  ```

  is illegal (named and positional connections cannot be mixed)

- Named associations where there is a mismatch of variable sizes or names generate an error.
- You can only use the .* once per instantiation, although you can mix the .* connection with *.name* and *.port_identifier(name)* type connections.
- If you use a .* construction but all remaining ports are explicitly connected, the compiler ignores the .* construct.
- Currently, the .* port connection is not supported for mixed HDL source code.

Extern Module

SystemVerilog simplifies the compilation process by allowing you to specify a prototype of the module being instantiated. The prototype is defined using the extern keyword, followed by the declaration of the module and its ports. Either the Verilog-1995 or the Verilog-2001 style of module declaration can be used for the prototype.

The extern module declaration can be made in any module, at any level of the design hierarchy. The declaration is only visible within the scope in which it is defined. Support is limited to declaring extern module outside the module.

Syntax

extern module *moduleName* **(***direction port1***,** *direction portVector port2***,**
*direction port3***);**

Example 1 – Extern Module Instantiation

Example 2 – Extern Module Reference

Limitations

An extern module declaration is not supported within a module.

Interface

Topics in this section include:

- Interface Construct, below
- Modports, on page 616
- Limitations and Non-Supported Features, on page 617

Interface Construct

SystemVerilog includes enhancements to Verilog for representing port lists and port connection lists characterized by name repetition with a single name to reduce code size and simplify maintenance. The interface and modport structures in SystemVerilog perform this function. The interface construct includes all of the characteristics of a module with the exception of module instantiation; support for interface definitions is the same as the current support for module definitions. Interfaces can be instantiated and connected to client modules using generates.

Interface Definition: Internal Logic and Hierarchical Structure

Per the SystemVerilog standard, an interface definition can contain any logic that a module can contain with the exception that interfaces cannot contain module instantiations. An interface definition can contain instantiations of other interfaces. Like modules, interface port declaration lists can include interface-type ports. Synthesis support for interface logic is the same as the current support for modules.

Port Declarations and Port Connections for Interfaces

Per the SystemVerilog standard, interface port declaration and port connection syntax/semantics are identical to those of modules.

Interface Member Types

The following interface member types are visible to interface clients:

- 4-State var types: reg, logic, integer
- 2-State var types: bit, byte, shortint, int, longint
- Net types: wire, wire-OR, and wire-AND
- Scalars and 1-dimensional packed arrays of above types
- Multi-dimensional packed and unpacked arrays of above types
- SystemVerilog struct types

Interface Member Access

The members of an interface instance can be accessed using the syntax:

*interfaceRef***.***interfaceMemberName*

In the above syntax, *interfaceRef* is either:

- the name of an interface-type port of the module/interface containing the member access
- the name of an interface instance that is instantiated directly within the module/interface containing the member access.

Note that reference to interface members using full hierarchical naming is not supported and that only the limited form described above for instances at the current level of hierarchy is supported.

Access to an interface instance by clients at lower levels of the design hierarchy is achieved by connecting the interface instance to a compatible interface-type port of a client instance and connecting this port to other compatible interface-type ports down the hierarchy as required. This chaining of interface ports can be done to an arbitrary depth. Note that interface instances can be accessed only by clients residing at the same or lower levels of the design hierarchy.

Interface-Type Ports

Interface-type ports are supported as described in the SystemVerilog standard, and generic interface ports are supported. A modport qualifier can appear in either a port declaration or a port connection as described in the SystemVerilog standard. Interface-type ports:

- can appear in either module or interface port declarations
- can be used to access individual interface items using "." syntax:

  *interfacePortname***.***interfaceMemberName*

- can be connected directly to compatible interface ports of module/interface instances

Interface/Module Hierarchy

Interfaces can be instantiated within either module or interface definitions. See Interface Member Access, on page 611 for additional details on hierarchical interface port connections.

Interface Functions and Tasks

Import-only functions and tasks (using import keyword in modport) are supported.

Element access outside the interface

Interface can have a collection of variables or nets, and this collection can be of a language-defined data type, user-defined data type, or array of language and user-defined data type. All of these variables can be accessed outside the interface.

The following example illustrates accessing a 2-dimensional structure type defined within the interface that is being accessed from another module.

Example – Accessing a 2-dimensional structure

```
typedef struct
{
   byte st1;
}Struct1D_Dt[1:0][1:0];
```

```
//Interface Definition
interface intf(
   input bit clk,
   input bit rst
);

   Struct1D_Dt i1; //2D - Structure type
   modport MP( input i1,input clk,input rst); //Modport Definition
endinterface

//Sub1 Module definition
module sub1(
   intf INTF1, //Interface
   input int d1
);

   assign INTF1.i1[1][1].st1 = d1[7:0];
   assign INTF1.i1[1][0].st1 = d1[15:8];
   assign INTF1.i1[0][1].st1 = d1[23:16];
   assign INTF1.i1[0][0].st1 = d1[31:24];
endmodule

//Sub2 Module definition
module sub2(
   intf.MP IntfMp, //Modport Interface
   output byte dout[3:0]
);

always_ff@(posedge IntfMp.clk)
begin
   if(IntfMp.rst)
   begin
      dout <= '{default:'1};
   end
   else begin
      dout[3] <= IntfMp.i1[1][1].st1;
      dout[2] <= IntfMp.i1[1][0].st1;
      dout[1] <= IntfMp.i1[0][1].st1;
      dout[0] <= IntfMp.i1[0][0].st1;
   end
end
endmodule

//Top Module definition
module top(
   input bit clk,
   input bit rst,
   input int d1,
   output byte dout[3:0]
);
intf intu1(.clk(clk),.rst(rst));
sub1 sub1u1(.INTF1(intu1),.d1(d1));
sub2 sub2u1(.IntfMp(intu1.MP),.dout(dout));
endmodule
```

Nested Interface

With the nested interface feature, nesting of interface is possible by either instantiating one interface in another or by using one interface as a port in another interface. Generic interface is not supported for nested interface; array of interface when using interface as a port also is not supported.

The following example illustrates the use of a nested interface. In the example, one interface is instantiated within another interface and this top-level interface is used in the modules.

Example – Nested Interface

```
//intf1 Interface definition
interface intf1;
   byte i11;
   byte i12;
endinterface

//IntfTop Top Interface definition
interface IntfTop;
   intf1 intf1_u1(); //Interface instantiated
   shortint i21;
endinterface

//Sub1 Module definition
module sub1(
   input byte d1,
   input byte d2,
   IntfTop intfN1
);
assign intfN1.intf1_u1.i11 = d1; //Nested interface being accessed
assign intfN1.intf1_u1.i12 = d2; //Nested interface being accessed
endmodule
```

```
//Sub2 Module definition
module sub2(
   IntfTop intfN2
);
assign intfN2.i21 = intfN2.intf1_u1.i11 + intfN2.intf1_u1.i12;
//Nested
   //interface being accessed
endmodule

//Sub3 Module definition
module sub3(
   IntfTop intfN3,
   output shortint dout
);
assign dout = intfN3.i21;
endmodule

//Top Module definition
module top(
   input byte d1,
   input byte d2,
   output shortint dout
);
IntfTop IntfTopU1();
   sub1 sub1U1(.d1(d1),.d2(d2),.intfN1(IntfTopU1));
   sub2 sub2U1(.intfN2(IntfTopU1));
   sub3 sub3U1(.intfN3(IntfTopU1), .dout(dout));
endmodule
```

Arrays of Interface Instances

In Verilog, multiple instances of the same module can be created using the array of instances concept. This same concept is extended for the interface construct in SystemVerilog to allow multiple instances of the same interface to be created during component instantiation or during port declaration. These arrays of interface instances and slices of interface instance arrays can be passed as connections to arrays of module instances across modules.

The following example illustrates the use of array of interface instance both during component instantiation and during port declaration.

Example – Array of interface during port declaration

```
//intf Interface Definition
interface intf;
   byte i1;
endinterface

//Sub1 Module definition
module sub1(
   intf IntfArr1 [3:0], //Array of interface during port
declaration
   input byte d1[3:0]
);
assign IntfArr1[0].i1 = d1[0];
assign IntfArr1[1].i1 = d1[1];
assign IntfArr1[2].i1 = d1[2];
assign IntfArr1[3].i1 = d1[3];
endmodule

//Sub2 Module definition
module sub2(
   intf IntfArr2[3:0], //Array of interface during port
declaration
   output byte dout[3:0]
);
assign dout[0] = IntfArr2[0].i1;
assign dout[1] = IntfArr2[1].i1;
assign dout[2] = IntfArr2[2].i1;
assign dout[3] = IntfArr2[3].i1;
endmodule

//Top module definition
module top(
   input byte d1[3:0],
   output byte dout[3:0]
);
intf intfu1[3:0](); //Array of interface instances
   sub1 sub1u1(intfu1,d1);
   sub2 sub2u1(intfu1,dout);
endmodule
```

Modports

Modport expressions are supported, and modport selection can be done in either the port declaration of a client module or in the port connection of a client module instance.

If a modport is associated with an interface port or instance through a client module, the module can only access the interface members enumerated in the modport. However, per the SystemVerilog standard, a client module is not constrained to use a modport, in which case it can access any interface members.

Modport Keywords

The input, output, inout, and import access modes are parsed without errors. The signal direction for input, output, and inout is ignored during synthesis, and the correct signal polarity is inferred from how the interface signal is used within the client module. The signal polarity keywords are ignored because the precise semantics are currently not well-defined in the SystemVerilog standard, and simulator support has yet to be standardized.

Example – Instantiating an interface Construct

Limitations and Non-Supported Features

The following restrictions apply when using interface/modport structures:

- Declaring interface within another interface is not supported
- Direction information in modports has no effect on synthesis.
- Exported (export keyword) interface functions and tasks are not supported.
- Virtual interfaces are not supported.
- Full hierarchical naming of interface members is not supported.
- Modports defined within generate statements are not supported.

System Tasks and System Functions

Topics in this section include:

- $bits System Function, below
- Array Querying Functions, on page 619

$bits System Function

SystemVerilog supports a $bits system function which returns the number of bits required to hold an expression as a bit stream. The syntax is:

$bits(*datatype***)**

$bits(*expression***)**

In the above syntax, *datatype* can be any language-defined data type (reg, wire, integer, logic, bit, int, longint, or shortint) or user-defined datatype (typedef, struct, or enum) and *expression* can be any value including packed and unpacked arrays.

The $bits system function is synthesizable and can be used with any of the following applications:

- Port Declaration
- Variable Declaration
- Constant Definition
- Function Definition

System tasks and system functions are described in Section 22 of IEEE Std 1800-2005 (IEEE Standard for SystemVerilog); $bits is described in Section 22.3.

Example – $bits System Function

Example – $bits System Function within a Function

Array Querying Functions

SystemVerilog provides system functions that return information about a particular dimension of an array.

Syntax

arrayQuery **(***arrayIdentifier*[,*dimensionExpression*]**);**
arrayQuery **(***dataTypeName*[,*dimensionExpression*]**);**
$dimensions | $unpacked_dimensions (*arrayIdentifier* **|** *dataTypeName***)**

In the above syntax, *arrayQuery* is one of the following array querying functions:

- **$left** - Returns the left bound (MSB) of the dimension.
- **$right** - Returns the right bound (LSB) of the dimension.
- **$low** – Returns the lowest value of the left and right bound dimension.
- **$high** – Returns the highest value of the left and right bound dimension.
- **$size** – Returns the number of elements in a given dimension.
- **$increment** – Returns a value "1" when the left bound is greater than or equal to the right bound, else it returns a value "-1".

In the third syntax example, $dimensions returns the total number of packed and unpacked dimensions in a given array, and $unpacked_dimensions returns the total number of unpacked dimensions in a given array. The variable *dimensionExpression*, by default, is "1". The order of dimension expression increases from left to right for both unpacked and packed dimensions, starting with the unpacked dimension for a given array.

Example 1 – Array Querying Function $left and $right Used on Packed 2D-data Type

Example 2 – Array Querying Function $low and $high Used on Unpacked 3D-data Type

Example 3 – Array Querying Function $size and $increment Used on a Mixed Array

Example 4 – Array Querying Function $dimensions and $unpacked_dimensions Used on a Mixed Array

Example 5 – Array Querying Function with Data Type as Input

Generate Statement

The synthesis tools support the Verilog 2005 generate statement, which conforms to the Verilog 2005 LRM. The tools also support defparam, parameter, and function and task declarations within generate statements. The naming scheme for registers and instances is also enhanced to include closer correlation to specified generate symbolic hierarchies. Generated data types have unique identifier names and can be referenced hierarchically. Generate statements are created using one of the following three methods: generate-loop, generate-conditional, or generate-case.

Note: The generate statement is a Verilog 2005 feature; to use this statement with the FPGA synthesis tools, you must enable SystemVerilog for your project.

Example 1 – Shift Register Using generate-for

Example 2 – Accessing Variables Declared in a generate-if

Example 3 – Accessing Variables Declared in a generate-case

Limitations

The following generate statement functions are not currently supported:

- Defparam support for generate instances
- Hierarchical access for interface
- Hierarchical access of function/task defined within a generate block

Note: Whenever the generate statement contains symbolic hierarchies separated by a hierarchy separator (.), the instance name includes the (\) character before this hierarchy separator (.).

Conditional Generate Constructs

The if-generate and case-generate conditional generate constructs allow the selection of, at most, one generate block from a set of alternative generate blocks based on constant expressions evaluated during elaboration. The generate and endgenerate keywords are optional.

Generate blocks in conditional generate constructs can be either named or unnamed and can consist of only a single item. It is not necessary to enclose the blocks with begin and end keywords; the block is still a generate block and, like all generate blocks, comprises a separate scope and a new level of hierarchy when it is instantiated. The if-generate and case-generate constructs can be combined to form a complex generate scheme.

Note: Conditional generate constructs are a Verilog 2005 feature; to use these constructs with the FPGA synthesis tools, you must enable SystemVerilog for your project.

Example 1 – Conditional Generate: if-generate

```
// test.v
module test
#  (parameter width = 8,
    sel = 2 )

   (input clk,
    input [width-1:0] din,
    output [width-1:0] dout1,
    output [width-1:0] dout2 );

if(sel == 1)
   begin:sh
   reg [width-1:0] sh_r;

   always_ff @ (posedge clk)
      sh_r <= din;
   end
else
   begin:sh
      reg [width-1:0] sh_r1;
      reg [width-1:0] sh_r2;
   always_ff @ (posedge clk)
   begin
      sh_r1 <= din;
      sh_r2 <= sh_r1;
   end
end

assign dout1 = sh.sh_r1;
assign dout2 = sh.sh_r2;

endmodule
```

Example 2 – Conditional Generate: case-generate

```
   // top.v
   module top
   #  (parameter mod_sel = 3,
       mod_sel2 = 3,
       width1 = 8,
       width2 = 16 )

      (input [width1-1:0] a1,
       input [width1-1:0] b1,
       output [width1-1:0] c1,
       input [width2-1:0] a2,
       input [width2-1:0] b2,
       output [width2-1:0] c2 );

   case(mod_sel)
      0:
         begin:u1
            my_or u1(.a(a1),.b(b1),.c(c1) );
         end
      1:
         begin:u1
            my_and u2(.a(a2),.b(b2),.c(c2) );
         end
      default:
         begin:u1
            my_or u1(.a(a1),.b(b1),.c(c1) );
         end
   endcase

   case(mod_sel2)
      0:
         begin:u3
            my_or u3(.a(a1),.b(b1),.c(c1) );
         end
      1:
         begin:u4
            my_and u4(.a(a2),.b(b2),.c(c2) );
         end
      default:
         begin:def
            my_and u2(.a(a2),.b(b2),.c(c2) );
         end
   endcase
   endmodule

   // my_and.v
   module my_and
   # (parameter width2 = 16 )

      (input [width2-1:0] a,
       input  [width2-1:0] b,
       output [width2-1:0] c
      );

   assign c = a & b;
   endmodule

   // my_or.v
   module  my_or
   #  (parameter width = 8)

      (input [width-1:0] a,
       input [width-1:0] b,
       output [width-1:0] c );

   assign c = a | b;
   endmodule
```

Assertions

The parsing of SystemVerilog Assertions (SVA) is supported as outlined in the following table.

Assertion Construct	Support Level	Comment
Immediate assertions	Supported	
Concurrent assertions	Partially Supported, Ignored	Multiclock properties are not supported
Boolean expressions	Partially Supported, Ignored	In the boolean expressions, $rose function having a clocking event is not supported.
Sequence	Supported, ignored	
Declaring sequences	Partially Supported, Ignored	Sequence with ports declared in global space is not supported
Sequence operations	Partially Supported, Ignored	All variations of first_match, within and intersect in a sequence is not supported.
Manipulating data in a sequence	Partially Supported, Ignored	More than one assignment in the parenthesis is not supported.
Calling subroutines on sequence match	Partially Supported, Ignored	Calling of more than one tasks is not supported
System functions	Partially Supported	System functions $onehot, $onehot1, and $countones supported; $isunknown not supported
Declaring properties	Partially Supported, Ignored	Declaring of properties in a package and properties with ports declared in global space are not supported
Multiclock support	Not Supported	
Clock resolutions	Partially Supported, Ignored	Default clocking is not supported

Assertion Construct	Support Level	Comment
Binding properties to scopes or instances	Not Supported	
Expect statement	Not Supported	
Clocking blocks and concurrent assertions	Not Supported	

SVA System Functions

SystemVerilog assertion support includes the $onehot, $onehot0, and $countones system functions. These functions check for specific characteristics on a particular signal and return a single-bit value.

- $onehot returns true when only one bit of the expression is true.
- $onehot0 returns true when no more than one bit of the expression is high (either one bit high or no bits are high).
- $countones returns true when the number of ones in a given expression matches a predefined value.

Syntax

$onehot (*expression***)**

$onehot0 (*expression***)**

$countones (*expression***)**

Example 1 – System Function within if Statement

The following example shows a $onehot/$onehot0 function used inside an if statement and ternary operator.

```
module top
   (
   //Input
   input byte d1,
   input byte d2,
   input shortint d3,
   //Output
   output byte dout1,
   output byte dout2
   );
byte sig1;
assign sig1 = d1 + d2;

//Use of $onehot
always_comb begin
   if($onehot(sig1))
      dout1 = d3[7:0];
   else
      dout1 = d3[15:8];
end

byte sig2;
assign sig2 = d1 ^ d2;
//Use of $onehot0
assign dout2 = $onehot0(sig2)? d3[7:0] : d3[15:8];

endmodule
```

Example 2 – System Function with Expression

The following example includes an expression, which is evaluated to a single-bit value, as an argument to a system function.

```
module top
   (
   //Input
   input byte d1,
   input byte d2,
   input shortint d3,
   //Output
   output byte dout1,
   output byte dout2
   );

//Use of $onehot with Expression inside onehot function
always@*
begin
   if($onehot((d1 == d2) ? d1[3:0] : d1[7:4]))
      dout1 = d3[7:0];
   else
      dout1 = d3[15:8];
end
```

```
//Use of $onehot0 with AND operation inside onehot function
assign dout2 = $onehot0(d1 & d2 )? d3[7:0] : d3[15:8];

endmodule
```

Example 3 – Ones Count

In the following example, a 4-bit count is checked for two and only two bits set to 1 which, when present, returns true.

```
module top(
   input clk,
   input rst,
   input byte d1,
   output byte dout
);
logic[3:0] count;

always_ff@(posedge clk)begin
   if(rst)
      count <= '0;
   else
      count <= count + 1'b1;
end

assign dout = $countones(count) == 3'd2 ? d1 : ~d1;
endmodule
```

Keyword Support

This table lists supported SystemVerilog keywords in the Synopsys FPGA synthesis tools:

always_comb	always_ff	always_latch	assert*
assume*	automatic	bind*	bit
break	byte	checker*	clocking*
const	continue	cover*	do
endchecker*	endclocking*	endinterface	endproperty*
endsequence*	enum	expect*	extern
final*	function	global*	import
inside	int	interface	intersect*
let*	logic	longint	modport
packed	package	parameter	priority
property*	restrict*	return	sequence*
shortint	struct	task	throughout*
timeprecision*	timeunit*	typedef	union
unique	void	within*	

* Reserved keywords for SystemVerilog assertion parsing; cannot be used as identifiers or object names

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. The topics include:

- Supported VHDL Language Constructs, on page 632
- Unsupported VHDL Language Constructs, on page 633
- Partially-supported VHDL Language Constructs, on page 634
- Ignored VHDL Language Constructs, on page 634

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
- File types
- All operators (-, -, *, /, **, mod, rem, abs, not, =, /=, <, <=, >, >=, and, or, nand, nor, xor, xnor, sll, srl, sla, sra, rol, ror, &)

Note: With the ** operator, arguments are compiler constants. When the left operand is 2, the right operand can be a variable.

- 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.

- 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

Partially-supported VHDL Language Constructs

When one of the following constructs in encountered, compilation continues, but will subsequently error out if logic must be generated for the construct.

- real data types (real data expressions are supported in VHDL-2008 IEEE float_pkg.vhd) – real data types are supported as constant declarations or as constants used in expressions as long as no floating point logic must be generated
- access types
- null arrays – null arrays are allowed as operands in concatenation expressions

Ignored VHDL Language Constructs

The synthesis tool ignores the following constructs in your design. If found, the tool parses and ignores the construct (provided that no logic is required to be synthesized) and continues with the synthesis run.

- disconnect
- assert and report
- initial values on inout ports

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
- Declaring and Assigning Objects in VHDL
- VHDL Dynamic Range Assignments
- Signals and Ports

- Variables
- VHDL Constants
- Libraries and Packages
- Operators
- VHDL Process
- Common Sequential Statements
- Concurrent Signal Assignments
- Resource Sharing
- Combinational Logic
- Sequential Logic
- Component Instantiation in VHDL
- VHDL Selected Name Support
- User-defined Function Support
- Demand Loading

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.

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

std_logic Values	Treated by the synthesis tool as...
'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

bit Values	Treated by the synthesis tool a...
'0'	logic 0
'1'	logic 1

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 synthesis tool uses a two's-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 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.

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 a 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.

VHDL Dynamic Range Assignments

The tools support VHDL assignments with dynamic ranges, which are defined as follows:

A(b downto c) <= D(e downto f);

A and D are constrained variables or signals, and b, c, e, and f are constants (generics) or variables. Dynamic range assignments can be used for RHS, LHS, or both.

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

```
entity test is
   port (data_out: out  std_logic_vector(63 downto 0);
         data_in: in std_logic_vector(63 downto 0);
         selector: in NATURAL range 0 to 7 );
end test;

architecture rtl of test is
begin
   data_out( (selector*8)+7 downto (selector*8))
      <= data_in((selector*8)+7 downto (selector*8));
end rtl;
```

Currently, the following limitations apply to dynamic range assignments:

- There is no support for procedures.
- There is no support for selected signal assignment; i.e., with *expression* Select.
- There is no support for use with concatenation operators.

Null Ranges

A null range is a range that specifies an empty subset of values. A range specified as *m* to *n* is a null range when *m* is greater than *n*, and a range specified as *n* downto *m* is a null range when *n* is less than *m*.

Support for null ranges allows ports with negative ranges to be compiled successfully. During compilation, any port declared with a null range and its related logic are removed by the compiler.

In the following example, port a_in1 (-1 to 0) is a null range and is subsequently removed by the compiler.

```
-- top.vhd
library ieee;
use ieee.std_logic_1164.all;

entity top is
generic (width : integer := 0);
   port (a_in1 : in std_logic_vector(width -1 downto 0);
         b_in1 : in std_logic_vector(3 downto 0);
         c_out1 : out std_logic_vector(3 downto 0) );
end top;
```

```
architecture struct of top is
component sub is
   port (a_in1 : in std_logic_vector(width -1 downto 0);
         b_in1 : in std_logic_vector(3 downto 0);
         c_out1 : out std_logic_vector(3 downto 0) );
end component;

begin
   UUT : sub port map ( a_in1 => a_in1, b_in1 => b_in1,
      c_out1 => c_out1);
end struct;

-- sub.vhd
library ieee;
use ieee.std_logic_1164.all;

entity sub is
generic (width : integer := 0);
   port (a_in1 : in std_logic_vector(width -1 downto 0);
         b_in1 : in std_logic_vector(3 downto 0);
         c_out1 : out std_logic_vector(3 downto 0) );
end sub;

architecture rtl of sub is
begin
   c_out1 <= not (b_in1 & a_in1);
end rtl;
```

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;
```

```
-- Declare an array type, a signal of that type, and
-- then make a valid assignment to the signal

   type array_type is array (1 downto 0) of
      std_logic_vector (7 downto 0);
   signal my_sig: array_type;
begin -- An architecture statement area
my_sig <= ("10101010","01010101");
```

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 predefined packages are described in the following table.

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
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 synthesis tools also have vendor-specific built-in macro libraries for components like gates, counters, flip-flops, and I/Os. The libraries are located in *installDirectory*/lib/*vendorName*. Use the built-in macro libraries to instantiate vendor macros directly into the VHDL designs and forward-annotate them to the output netlist. Refer to the appropriate vendor support chapter for more information.

Additionally, the synthesis tool library contains an attributes package of built-in attributes and timing constraints (*installDirectory*/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;
```

If you want the addition operator (+) to take two std_ulogic or std_ulogic_vector as inputs, you need the function defined in the std_logic_arith package (the cdn_arith.vhd file in *installDirectory*/lib/vhd/). Add this file to the project. To successfully compile, the VHDL file that uses the addition operator on these input types must have include the following statement:

```
use work.std_logic_arith.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:

Arithmetic Operator	Description
-	addition
-	subtraction
*	multiplication
/	division

Arithmetic Operator	Description
**	exponentiation (supported for compile-time constants and when left operand is 2; right operand can be a variable)
mod	modulus
rem	remainder

Relational 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')

Logical Operator	Description
and	and
or	or
nand	nand
nor	nor
xor	xor
xnor	xnor
not	not (takes only one operand)

Shift 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

Misc. 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.

Large Time Resolution

The support of predefined physical time types includes the expanded range from –2147483647 to +2147483647 with units ranging from femtoseconds, and secondary units ranging up to an hour. Predefined physical time types allow selection of a wide number range representative of time type.

Example 1 – Using Large Time Values in Comparisons

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

entity test is
   generic (INTERVAL1 : time := 1000 fs ;
            INTERVAL2 : time := 1 ps;
            INTERVAL3 : time := 1000 ps;
            INTERVAL4 : time := 1 ns
   );

   port (a : in std_logic_vector(3 downto 0);
         b : in std_logic_vector(3 downto 0);
         c : out std_logic_vector(3 downto 0);
         d : out std_logic_vector(3 downto 0)
   );
end test;

architecture RTL of test is
begin
   c <= (a and b) when (INTERVAL1 = INTERVAL2) else
      (a or b);
   d <= (a xor b) when (INTERVAL3 /= INTERVAL4) else
      (a nand b);
end RTL;
```

Example 2 – Using Large Time Values in Constant Calculations

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

entity test is
   generic (Interval : time := 20 ns;
            CLK_PERIOD : time := 8 ns );
   port (en : in std_logic;
         a : in std_logic_vector(10 downto 0);
         b : in std_logic_vector(10 downto 0);
         a_in : in std_logic_vector(7 downto 0);
         b_in : in std_logic_vector(7 downto 0);
         dummyOut : out std_logic_vector(7 downto 0);
         out1 : out std_logic_vector(10 downto 0) );
end entity;
```

```
architecture behv of test is
   constant my_time : positive := (Interval / 2 ns);
   constant CLK_PERIOD_PS : real := real(CLK_PERIOD / 1 ns);
   constant RESULT : positive := integer(CLK_PERIOD_PS);
   signal dummy : std_logic_vector (RESULT-1 downto 0);
   signal temp : std_logic_vector (my_time downto 0);
begin
   process (a, b)
   begin
      temp <= a and b;
      out1 <= temp;
   end process;
dummy <= (others => '0') when en = '1' else
   (a_in or b_in);
dummyOut <= dummy;
end behv;
```

Example 3 – Using Large Time Values in Generic Calculations

```
library IEEE;
use IEEE.std_logic_1164.all;
library unisim;
use unisim.vcomponents.all;

entity test is
   generic (clk_period : time := 6 ns);
   port (rst_in : in std_logic;
         in1 : in std_logic;
         CLK_PAD : in std_logic;
         RST_DLL : in std_logic;
         dout : out std_logic;
         CLK_out : out std_logic;
         LOCKED : out std_logic );
end test;

architecture STRUCT of test is
   signal CLK, CLK_int, CLK_dcm : std_logic;
   signal clk_dv : std_logic;
   constant clk_prd : real := real(clk_period / 2.0 ns);
begin
U1 : IBUFG port map (I => CLK_PAD, O => CLK_int);
U2 : DCM generic map
(CLKDV_DIVIDE  => clk_prd)
   port map (CLKFB => CLK,
            CLKIN => CLK_int,
            CLKDV => clk_dv,
            DSSEN => '0',
            PSCLK => '0',
                PSEN => '0',
                PSINCDEC => '0',
                RST => RST_DLL,
                CLK0 => CLK_dcm,
                LOCKED => LOCKED );
U3 : BUFG port map (I => CLK_dcm, O => CLK);
CLK_out <= CLK;
   process (clk_dv , rst_in, in1)
   begin
      if (rst_in = '1') then
         dout <= '0';
      elsif (clk_dv'event and clk_dv = '1') then
         dout <= in1;
      end if;
   end process;
end architecture STRUCT;
```

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 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 703

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 C1: 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

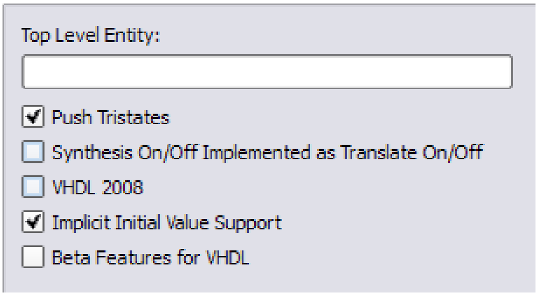
In the previous section, the user-defined function example successfully uses SNS to determine the func function to implement. However, neither PACKAGE1 nor PACKAGE2 was declared as a use package clause (for example, work.PACKAGE1.all;). How could func have been executed without a use package declaration? 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 Implicit Data-type Defaults

Type default propagation avoids potential simulation mismatches that are the result of differences in behavior with how initial values for registers are treated in the synthesis tools and how they are treated in the simulation tools.

With implicit data-type defaults, when there is no explicit initial-value declaration for a signal being registered, the VHDL compiler passes an init value through a syn_init property to the mapper, and the mapper then propagates the value to the respective register. Compiler requirements are based on specific data types. These requirements can be broadly grouped based on the different data types available in the VHDL language.

Implicit data-type defaults are enabled on the VHDL panel of the Implementation Options dialog box or through a -supporttypedflt argument to a set_option command.



Top Level Entity:

☑ Push Tristates

□ Synthesis On/Off Implemented as Translate On/Off

□ VHDL 2008

☑ Implicit Initial Value Support

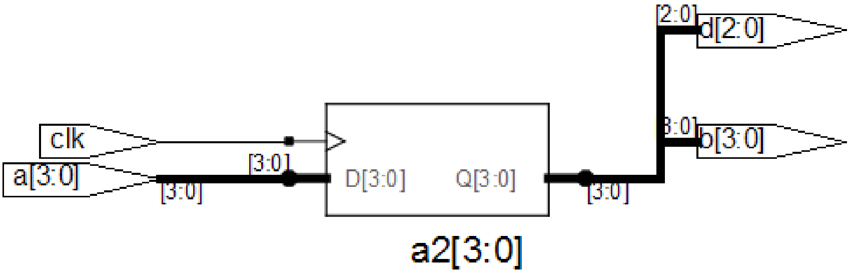
□ Beta Features for VHDL

To illustrate the use of implicit data-type defaults, consider the following example.

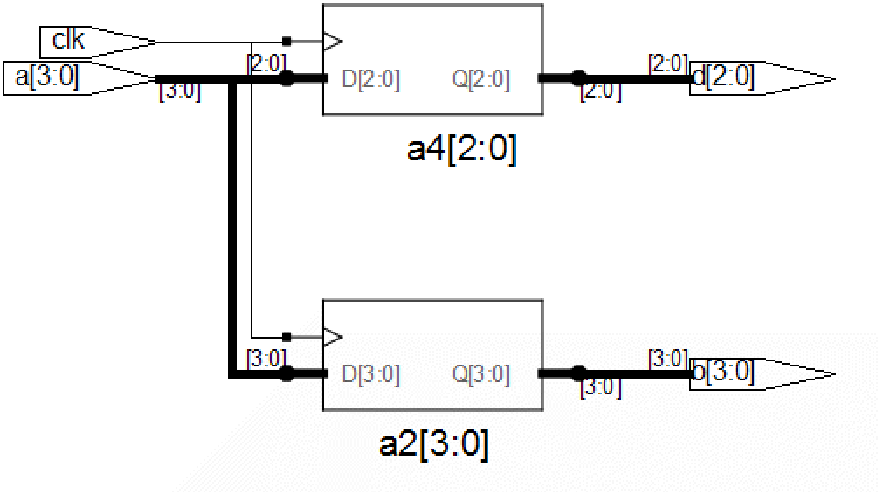
```
library ieee;
use ieee.std_logic_1164.all;

entity top is
   port (clk:in std_logic;
         a : in integer range 1 to 8;
         b : out integer range 1 to 8;
         d : out positive range 1 to 7 );
end entity top;
```

```
architecture rtl of top is
signal a1,a2 : integer range 1 to 8 ;
signal a3,a4 : positive range 1 to 7;
begin
a1 <= a ;
a3 <= a ;
b <= a2 ;
d <= a4 ;
   process(clk)
   begin
      if (rising_edge(clk))then
         a2 <= a1 ;
         a4 <= a3 ;
      end if;
   end process;
end rtl;
```

In the above example, two signals (a2 and a4) with different type default values are registered. Without implicit data-type defaults, if the values of the signals being registered are not the same, the compiler merges the redundant logic into a single register as shown in the figure below.



Enabling implicit data-type defaults prevents certain compiler and mapper optimizations to preserve both registers as shown in the following figure.



Example – Impact on Integer Ranges

The default value for the integer type when a range is specified is the minimum value of the range specified, and size is the upper limit of that range. With implicit data-type defaults, the compiler is required to propagate the minimum value of the range as the init value to the mapper. Consider the following example:

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

entity top is
   port (clk,set:in std_logic;
         a : in integer range -6 to 8;
         b : out integer range -6 to 8 );
end entity top;

architecture rtl of top is
signal a1,a2: integer range -6 to 8;
begin
a1 <= a ;
   process(clk,set)
   begin
      if (rising_edge(clk))then
```

```
            if set = '1' then
               a2 <= a;
            else
               a2 <= a1 ;
            end if;
         end if;
      end process;
   b <= a2;
   end rtl;
```

In the example,

```
signal a1, a2: integer range -6 to 8;
```

the default value is -6 (FA in 2's complement) and the range is -6 to 8. With a total of 15 values, the size of the range can be represented in four bits.

Example – Impact on RAM Inferencing

When inferencing a RAM with implicit data-type defaults, the compiler propagates the type default values as init values for each RAM location. The mapper must check if the block RAMs of the selected technology support initial values and then determine if the compiler-propagated init values are to be considered. If the mapper chooses to ignore the init values, a warning is issued stating that the init values are being ignored. Consider the following VHDL design:

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

entity top is
   port (clk : in std_logic;
         addr : in std_logic_vector (6 downto 0);
         din : in positive;
         wen : in std_logic;
         dout : out positive );
end top;

architecture behavioral of top is
-- RAM
type tram is array(0 to 127) of  positive ;
signal ram : tram ;
begin
-- Contents of RAM has initial value = 1
   process (clk)
   begin
      if clk'event and clk = '1' then
         if wen = '1' then
            ram(conv_integer(addr)) <= din_sig;
         end if;
      dout <= ram(conv_integer(addr));
      end if;
   end process;
end behavioral;
```

In the above example:

- The type of signal a1 is bit_vector
- The default value for type integer is 1 when no range is specified

Accordingly, a value of x00000001 is propagated by the compiler to the mapper with a syn_init property.

VHDL Synthesis Guidelines

This section provides guidelines for synthesis using VHDL. The following topics are covered:

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 Implementation Options 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 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 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 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;
```

Signed mod Support for Constant Operands

The synthesis tool supports signed mod for constant operands. Additionally, division operators (/, rem, mod), where the operands are compile-time constants and greater than 32 bits, are supported.

Example of using signed mod operator with constant operands

```
LIBRARY ieee; USE ieee.std_logic_1164.ALL;
LIBRARY ieee; USE ieee.numeric_std.all;

ENTITY divmod IS
   PORT (tstvec: out  signed(7 DOWNTO 0) );
END divmod;

ARCHITECTURE structure OF divmod IS
   CONSTANT NOMINATOR   : signed(7 DOWNTO 0) := "10000001";
   CONSTANT DENOMINATOR : signed(7 DOWNTO 0) := "00000011";
   CONSTANT RESULT      : signed(7 DOWNTO 0) := NOMINATOR mod
      DENOMINATOR;
BEGIN
   tstvec <= result;
```

```
END ARCHITECTURE structure;
```

Example of a signed division with a constant right operand.

```
LIBRARY ieee ; USE ieee.std_logic_1164.ALL;
LIBRARY ieee ; USE ieee.numeric_std.all;

ENTITY divmod IS
   PORT (tstvec: out  signed(7 DOWNTO 0) );
END divmod;

ARCHITECTURE structure OF divmod IS
   CONSTANT NOMINATOR   : signed(7 DOWNTO 0) := "11111001";
   CONSTANT DENOMINATOR : signed(7 DOWNTO 0) := "00000011";
   CONSTANT RESULT      : signed(7 DOWNTO 0) := NOMINATOR /
      DENOMINATOR;
BEGIN
   tstvec <= result;

END ARCHITECTURE structure;
```

An example where the operands are greater than 32 bits

```
LIBRARY ieee; USE ieee.std_logic_1164.ALL;
LIBRARY ieee; USE ieee.numeric_std.all;

ENTITY divmod IS
   PORT (tstvec: out  unsigned(33 DOWNTO 0) );
END divmod;

ARCHITECTURE structure OF divmod IS
   CONSTANT NOMINATOR   : unsigned(33 DOWNTO 0) :=
   "100000000000000000000000000000000";
   CONSTANT DENOMINATOR : unsigned(32 DOWNTO 0) :=
   "00000000000000000000000000000011";
   CONSTANT RESULT      : unsigned(33 DOWNTO 0) := NOMINATOR /
      DENOMINATOR;
BEGIN
   tstvec <= result;
END ARCHITECTURE structure;
```

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.

The topics include:

- Asynchronous Sets and Resets, on page 687
- Synchronous Sets and Resets, on page 688

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.

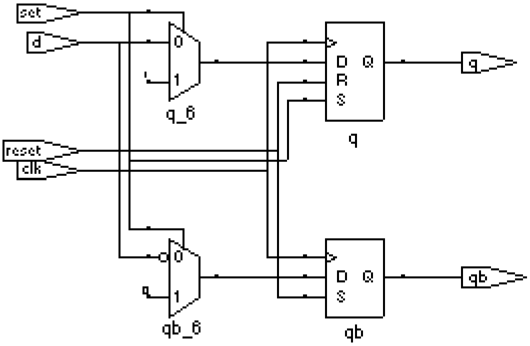
RTL View Primitives

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 – f lip-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.



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. The topics include:

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 697 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 695.
- Override the default encoding style with the syn_encoding attribute. The default encoding is determined by the number of states, where a non-default encoding is implemented if it produces better results. See Values for syn_encoding, on page 927 for a list of default and other encodings. When you specify a particular encoding style with syn_encoding, that 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 11, *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 11, *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 694.

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 Design Creation 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 -- 8-bit register
begin
-- <8-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
-- <Rotation 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 703.

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, on page 703
- Configuration Declaration, on page 707
- VHDL Configuration Statement Enhancement, on page 714

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

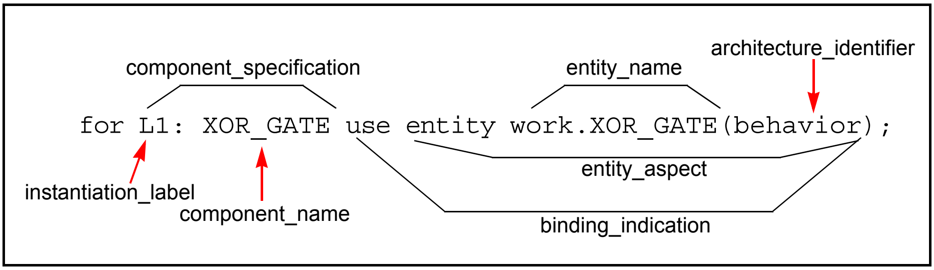
instantiation_list ::=

instantiation_label {, instantiation_label } | **others** | **all**

binding_indication ::= [**use** entity_aspect]
[generic_map_aspect]
[port_map_aspect]

entity_aspect ::=

entity entity_name [(architecture_identifier)] |
configuration configuration_name



```
for L1: XOR_GATE use entity work.XOR_GATE(behavior);
```

```
for others: AND_GATE use entity work.AND_GATE(structure);
for all: XOR_GATE use entity work.XOR_GATE;
```

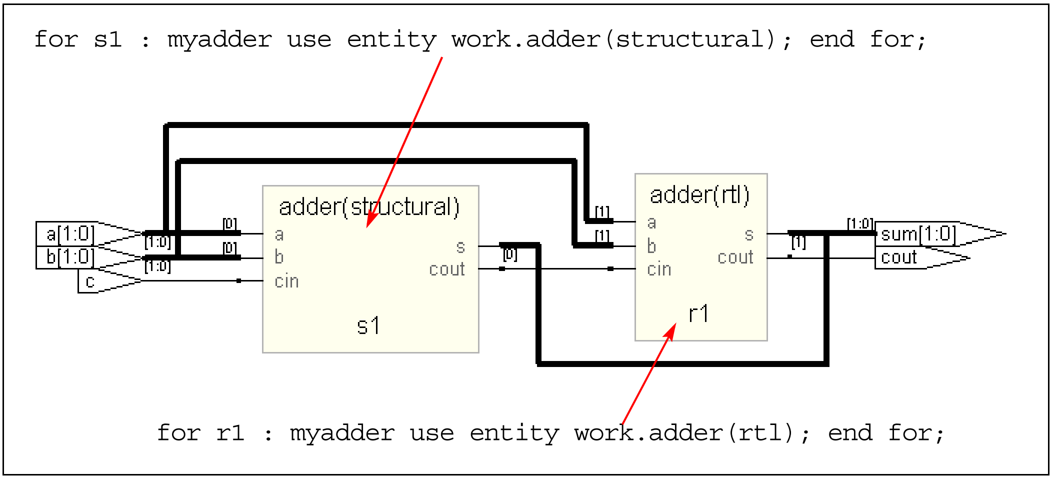
Example: Configuration Specification

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 configuration specification construct is *not* supported by the synthesis tool. An appropriate message is issued in the log file when this construct is used.

- 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 703.

Configuration Declaration within a Hierarchy

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;
```

Selection with Configuration Declaration

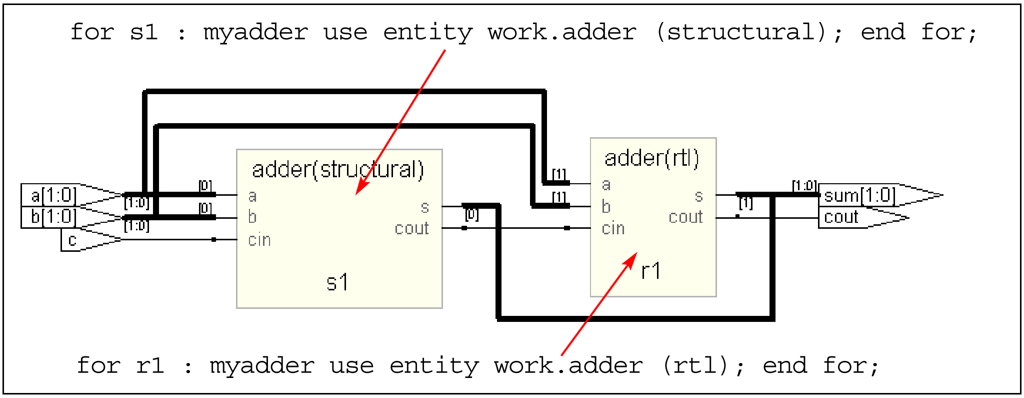
In the following example, two architectures (RTL and structural) are defined for an adder. There are two instantiations of an adder in design top. A configuration declaration defines the adder architecture to use for each instantiation. This example is similar to the configuration specification example.

```
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;
```

Direct Instantiation of Entities Using Configuration

In this method, a configured entity (i.e., an entity with a configuration declaration) is directly instantiated by writing a component instantiation statement that directly names the configuration.

Syntax

label **: configuration** configurationName
    [**generic map (**actualGeneric1, actualGeneric2, ... **)**]
    [**port map (** port1, port2, ... **)**] **;**

Example – Direct Instantiation Using Configuration Declaration

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.

VHDL Configuration Statement Enhancement

This section highlights the VHDL configuration statement support and handling component declarations with corresponding entity descriptions. Topics include:

- Generic mapping, on page 714
- Port Mapping, on page 715
- Mapping Multiple Entity Names to the Same Component, on page 717
- Generics Assigned to Configurations, on page 718
- Arithmetic Operators and Functions in Generic Maps, on page 722
- Ports in Component Declarations, on page 724

Generic mapping

Generics and ports can have different names and sizes at the entity and component levels. You use the configuration statement to bind them together with a configuration specification or a configuration declaration. The binding priority follows this order:

- Configuration specification
- Component specification
- Component declaration

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

```
entity test is
generic ( range1 : integer := 11);
   port (a, a1 : in std_logic_vector( range1 - 1 downto 0);
         b, b1 : in std_logic_vector( range1 - 1 downto 0);
         c, c1 : out std_logic_vector( range1 - 1 downto 0) );
end test;

architecture test_a of test is
component submodule1 is
generic ( size : integer := 6);
   port (a : in std_logic_vector(size -1 downto 0);
         b : in std_logic_vector(size -1 downto 0);
         c : out std_logic_vector(size -1 downto 0) );
end component;

for all : submodule1
use entity work.sub1(rtl)
generic map (size => range1);
begin
   UUT1 : submodule1 generic map (size => 4)
   port map (a => a,b => b,c => c);

end test_a;
```

If you define the following generic map for sub1, it takes priority:

```
entity sub1 is
generic(size: integer:=1);
   port (a: in std_logic_vector(size -1 downto 0);
         b : in std_logic_vector(size -1 downto 0);
         c : out std_logic_vector(size -1 downto 0 );
end sub1;
```

Port Mapping

See Generic mapping, on page 714 for information about using the configuration statement and binding priority.

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

entity test is
generic ( range1 : integer := 1);
   port (ta, ta1 : in std_logic_vector( range1 - 1 downto 0);
         tb, tb1 : in std_logic_vector( range1 - 1 downto 0);
         tc, tc1 : out std_logic_vector( range1 - 1 downto 0) );
end test;
```

```
architecture test_a of test is
component submodule1
generic ( my_size1 : integer := 6; my_size2 : integer := 6);
   port (d : in std_logic_vector(my_size1 -1 downto 0);
         e : in std_logic_vector(my_size1 -1 downto 0);
         f : out std_logic_vector(my_size2 -1 downto 0) );
end component;

for UUT1 : submodule1
use entity work1.sub1(rtl)
generic map (size1 => my_size1, size2 => my_size2)
port map ( a => d, b => e, c => f);

   begin
   UUT1 : submodule1 generic map (my_size1 => 1, my_size2 => 1)
   port map (d => ta, e => tb,f => tc);
   end test_a;
```

If you define the following port map for sub1, it overrides the previous definition:

```
entity sub1 is
generic(size1: integer:=6; size2:integer:=6);
port (a: in std_logic_vector (size1 -1 downto 0);
      b : in std_logic_vector (size1 -1 downto 0);
      c : out std_logic_vector (size2 -1 downto 0 );
end sub1:
```

Mapping Multiple Entity Names to the Same Component

When a single component has multiple entities, you can use the configuration statement and the for label clause to bind them. The following is an example:

```
entity test is
generic ( range1 : integer := 1);
   port (ta, ta1 : in std_logic_vector(range1 - 1 downto 0);
         tb, tb1 : in std_logic_vector(range1 - 1 downto 0);
         tc, tc1 : out std_logic_vector(range1 - 1 downto 0));
end test;

architecture test_a of test is
component submodule
generic (my_size1 : integer := 6; my_size2 : integer := 6);
   port (d,e : in std_logic_vector(my_size1 -1 downto 0);
         f : out std_logic_vector(my_size2 -1 downto 0));
end component;

begin
UUT1 : submodule generic map (1, 1)
   port map (d => ta, e => tb, f => tc);
UUT2 : submodule generic map (1, 1) port map
   (d => ta1, e => tb1, f => tc1)
end test_a;

configuration my_config of test is
for test_a
   for UUT1 : submodule
      use entity work.sub1(rtl)
      generic map (my_size1, my_size2)
      port map (d, e, f);
   end for;
   for others : submodule
      use entity work.sub2(rtl)
      generic map (my_size1, my_size2)
      port map ( d, e, f);
   end for;
end for;
end my_config;
```

You can map multiple entities to the same component, as shown here:

```
entity sub1 is
generic(size1: integer:=6; size2:integer:=6);
port (a: in std_logic_vector (size1 -1 downto 0);
      b : in std_logic_vector (size1 -1 downto 0);
      c : out std_logic_vector (size2 -1 downto 0 );
end sub1:
```

```
entity sub2 is
generic(width1: integer; width2:integer);
port (a1: in std_logic_vector(width1 -1 downto 0);
      b1 : in std_logic_vector (width1 -1 downto 0);
      c1 : out std_logic_vector (width2 -1 downto 0 );
end sub1:
```

Generics Assigned to Configurations

Generics can be assigned to configurations instead of entities.

Entities can contain more generics than their associated component declarations. Any additional generics on the entities must have default values to be able to synthesize.

Entities can also contain fewer generics than their associated component declarations. The extra generics on the component have no affect on the implementation of the entity.

Following are some examples.

Example1

Configuration conf_module1 contains a generic map on configuration conf_c. The component declaration for submodule1 does not have the generic use_extraSYN_ff, however, the entity has it.

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

```
entity submodule1 is
generic (width : integer := 16;
use_extraSYN_ff : boolean := false);
   port (clk : in std_logic;
         b : in std_logic_vector(width - 1 downto 0);
         c : out std_logic_vector(width - 1 downto 0) );
end submodule1;
```

```
architecture rtl of submodule1 is
signal d : std_logic_vector(width - 1 downto 0);
begin
no_resynch : if use_extraSYN_ff = false generate
   d <= b;
end generate no_resynch;

resynch : if use_extraSYN_ff = true generate
   process (clk)
   begin
      if falling_edge(clk) then
         d <= b;
      end if;
   end process;
end generate resynch;

   process (clk)
   begin
      if rising_edge(clk) then
         c <= d;
      end if;
   end process;
end rtl;

configuration conf_c of submodule1 is
   for rtl
   end for;
end configuration conf_c;

library ieee;
use ieee.std_logic_1164.all;

entity module1 is
generic ( width: integer := 16);
   port (clk : in std_logic;
         b : in std_logic_vector(width - 1 downto 0);
         c : out std_logic_vector(width - 1 downto 0) );
end module1;

architecture rtl of module1 is
component submodule1
generic (width: integer := 8);
   port (clk : in std_logic;
         b : in std_logic_vector(width - 1 downto 0);
         c : out std_logic_vector(width - 1 downto 0) );
end component;
```

```
begin
UUT2 : submodule1 port map (clk => clk,
   b => b,
   c => c);
end rtl;

library ieee;
configuration conf_module1 of module1 is
   for rtl
      for UUT2 : submodule1
         use configuration conf_c generic map( width => 16,
         use_extraSYN_ff => true);
      end for;
   end for;
end configuration conf_module1;
```

Example2

The component declaration for mod1 has the generic size, which is not in the entity. A component declaration can have more generics than the entity, however, extra component generics have no affect on the entity's implementation.

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

entity module1 is
generic (width: integer := 16;
use_extraSYN_ff : boolean := false);
   port (clk : in std_logic;
         b : in std_logic_vector ( width - 1 downto 0);
         c : out std_logic_vector( width - 1 downto 0) );
end module1;

architecture rtl of module1 is
signal d : std_logic_vector(width - 1 downto 0);
begin
   no_resynch : if use_extraSYN_ff = false generate
      d <= b ;
end generate no_resynch;
```

```
resynch : if use_extraSYN_ff = true generate -- insert pipeline
   -- registers
   process (clk)
   begin
      if falling_edge(clk) then
         d <= b;
      end if;
   end process;
end generate resynch;

   process (clk)
   begin
      if rising_edge(clk) then
         c <= d;
      end if;
   end process;
end rtl;

configuration module1_c of module1 is
   for rtl
   end for;
end module1_c;

library ieee;
use ieee.std_logic_1164.all;

entity test is
   port (clk : in std_logic;
         tb : in std_logic_vector( 7 downto 0);
         tc : out std_logic_vector( 7 downto 0) );
end test;

architecture test_a of test is
component mod1
generic (width: integer := 16;
use_extraSYN_ff: boolean := false;
size : integer := 8);
   port (clk : in std_logic;
         b : in std_logic_vector(width - 1 downto 0);
         c : out std_logic_vector(width - 1 downto 0) );
end component;

begin
UUT1 : mod1 generic map (width => 18)
   port map (clk => clk,
      b => tb,
      c => tc);
end test_a;
```

```
Configuration test_c of test is
for test_a
   for UUT1 : mod1
      use configuration module1_c
      generic map (width => 8, use_extraSYN_ff => true);
   end for;
end for;
end test_c;
```

Arithmetic Operators and Functions in Generic Maps

Arithmetic operators and functions can be used in generic maps. Following is an example.

Example

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

entity sub is
generic (width : integer:= 16);
   port (clk : in std_logic;
         a : in std_logic_vector (width - 1 downto 0);
         y : out std_logic_vector (width - 1 downto 0) );
end sub;

architecture rtl1 of sub is
begin
   process (clk, a)
   begin
      if (clk = '1' and clk'event) then
         y <= a;
      end if;
   end process;
end rtl1;

architecture rtl2 of sub is
begin y <= a;
end rtl2;

configuration sub_c of sub is
for rtl1 end for;
end sub_c;
```

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

entity test is
generic (mcu_depth : integer:=1;
mcu_width : integer:=16 );
   port (clk : in std_logic;
         a : in std_logic_vector
            ((mcu_depth*mcu_width)-1 downto 0);
         y : out std_logic_vector
            ((mcu_depth*mcu_width)-1downto 0));
end test;

architecture RTL of test is
constant CWIDTH : integer := 2;
constant size : unsigned := "100";
component sub generic ( width : integer );
   port (clk : in std_logic;
         a : in std_logic_vector (CWIDTH - 1 downto 0);
         y : out std_logic_vector (CWIDTH - 1 downto 0) );
end component;

begin i_sub : sub
generic map (width => CWIDTH ) port map (clk => clk,
   a => a,
   y => y );
end RTL;

library ieee;
use ieee.std_logic_arith.all;

configuration test_c of test is
   for RTL
      for i_sub : sub use
         configuration sub_c
         generic map(width => (CWIDTH ** (conv_integer (size))));
      end for;
   end for;
end test_c;
```

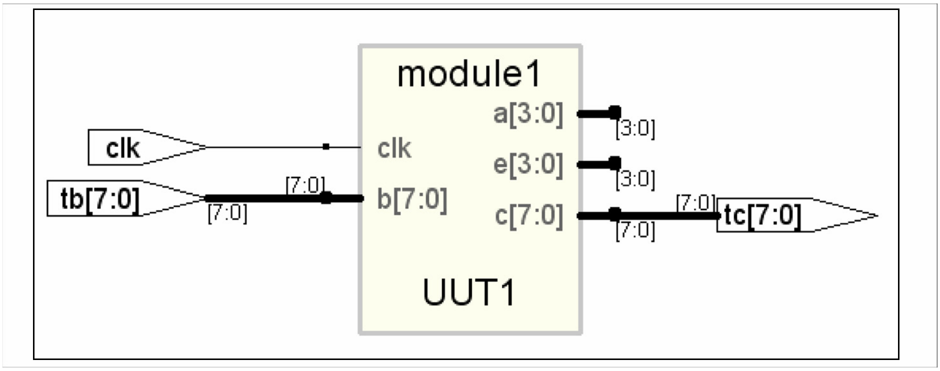
Ports in Component Declarations

Entities can contain more or fewer ports than their associated component declarations. Following are some examples.

Example1

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

entity module1 is
generic ( width: integer := 16; use_extraSYN_ff : boolean :=
false);
   port (clk : in std_logic;
         b : in std_logic_vector ( width - 1 downto 0);
         a : out integer range 0 to 15; --extra output port
            on entity
         e : out integer range 0 to 15; -- extra output port
            on entity
         c : out std_logic_vector( width - 1 downto 0));
end module1;

architecture rtl of module1 is
signal d : std_logic_vector(width - 1 downto 0);
begin
e <= width;
a <= width;
no_resynch : if use_extraSYN_ff = false generate
   d <= b ;
end generate no_resynch;

resynch : if use_extraSYN_ff = true generate
   process (clk)
   begin
      if falling_edge(clk) then
         d <= b;
      end if;
   end process;
end generate resynch;

   process (clk)
   begin
      if rising_edge(clk) then
         c <= d;
      end if;
   end process;
end rtl;
```

```
configuration module1_c of module1 is
for rtl
end for;
end module1_c;

library ieee;
use ieee.std_logic_1164.all;

entity test is
   port (clk : in std_logic;
         tb : in std_logic_vector( 7 downto 0);
         tc : out std_logic_vector( 7 downto 0) );
end test;

architecture test_a of test is
component mod1
generic (width: integer := 16);
   port (clk : in std_logic;
         b : in std_logic_vector(width - 1 downto 0);
         c : out std_logic_vector(width - 1 downto 0) );
end component;

begin
UUT1 : mod1 generic map (width => 18)
port map (clk => clk,
   b => tb,
   c => tc);
end test_a;

Configuration test_c of test is
for test_a
   for UUT1 : mod1
      use configuration module1_c
      generic map (width => 8, use_extraSYN_ff => true);
   end for;
end for;
end test_c;
```

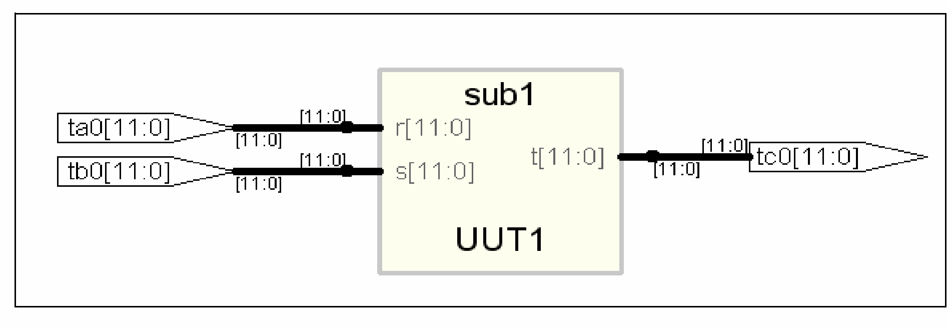


In the figure above, the entity module1 has extra ports a and e that are not defined in the corresponding component declaration mod1. The additional ports are not connected during synthesis.

Example2

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY sub1 IS
GENERIC(
size1 : integer := 11;
size2 : integer := 12);
   PORT (r : IN std_logic_vector(size1 -1 DOWNTO 0);
         s : IN std_logic_vector(size1 -1 DOWNTO 0);
         t : OUT std_logic_vector(size2 -1 DOWNTO 0) );
END sub1;

ARCHITECTURE rtl OF sub1 IS
BEGIN
   t <= r AND s;
END ARCHITECTURE rtl;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY test IS
GENERIC (range1 : integer := 12);
   PORT (ta0 : IN std_logic_vector(range1 - 1 DOWNTO 0);
         tb0 : IN std_logic_vector(range1 - 1 DOWNTO 0);
         tc0 : OUT std_logic_vector(range1 - 1 DOWNTO 0) );
END test;
```

```
ARCHITECTURE test_a OF test IS
COMPONENT submodule
GENERIC (
my_size1 : integer := 4;
my_size2 : integer := 5);
   PORT (d : IN std_logic_vector(my_size1 -1 DOWNTO 0);
         e : IN std_logic_vector(my_size1 -1 DOWNTO 0);
         ext_1 : OUT std_logic_vector(my_size1 -1 DOWNTO 0);
         ext_2 : OUT std_logic_vector(my_size1 -1 DOWNTO 0);
         f : OUT std_logic_vector(my_size2 -1 DOWNTO 0) );
END COMPONENT;

BEGIN
UUT1 : submodule
GENERIC MAP (
my_size1 => range1,
my_size2 => range1)
   PORT MAP (ext_1 => open,
      ext_2 => open,
      d => ta0,
      e => tb0,
      f => tc0 );
END test_a;

CONFIGURATION my_config OF test IS
   FOR test_a
      FOR UUT1 : submodule
      USE ENTITY work.sub1(rtl)
      GENERIC MAP (
         size1 => my_size1,
         size2 => my_size2)
      PORT MAP (r => d,
         s => e,
         t => f );
      END FOR;
   END FOR; -- test_a
END my_config;
```



In the figure above, the component declaration has more ports (ext_1 ext_2) than entity sub1. The component is synthesized based on the number of ports on the entity.

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.

- Creating a Scalable Design Using Unconstrained Vector Ports, on page 729
- Creating a Scalable Design Using VHDL Generics, on page 730
- Using a Scalable Architecture with VHDL Generics, on page 731
- Creating a Scalable Design Using Generate Statements, on page 733

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;
```

VHDL Guidelines for RAM Inference

This section provides guidelines for synthesis using RAMS and covers the following topics:

- RAM Inference in VHDL Designs, on page 735
- Limited RAM Resources, on page 736
- Additional Components, on page 737
- Synchronous READ RAMs, on page 738
- Multi-Port RAM Extraction, on page 737

RAM Inference in VHDL Designs

The synthesis tool can automatically infer synchronous and synchronously resettable RAMs from your VHDL source code and, where appropriate, generate technology-specific single or dual-port RAMs. You do not need any special input, like attributes, in your source code for this inference. To override the automatic inference or specify implementations, use the syn_ramstyle attribute (syn_ramstyle Attribute, on page 995).

Automatic RAM inference and implementation is a two-step process:

- The compiler infers an RTL-level RAM component. This always has an asynchronous READ. The RAM component that is inferred can be one of the following:

RAM1	RAM component
RAM2	RAM component with reset
NRAM	RAM component with multiple ports

- During synthesis, the mappers convert the RTL-level RAM inferred by the compiler 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 738, for information on coding your VHDL description to ensure that you get technology-specific RAM primitives with synchronous READs.

For certain technologies, the mappers implement single-port, dual-port, or multi-port RAMs. The following table lists the supported technology-specific RAMs that can be generated by the synthesis tool.

Technology vendor	Family	RAM Type
Microsemi	ProASIC3E	Single and dual port

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 ;
```

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

After inferring a RAM for some technologies, you might notice that the synthesis tool has generated a few additional components (glue logic) adjacent to the RAM. These components assure accuracy in your post placement and routing simulation results.

Multi-Port RAM Extraction

The compiler extracts multi-write port RAMs, creating an nram primitive which can have n write ports. The nrams are implemented differently depending on your technology. For information about the implementations, see the appropriate vendor chapter.

The following aspects of multi-port RAM extraction are described here:

- Prerequisites for Multi-Port RAM Extraction, on page 737
- Write Processes and Multi-Port RAM Inference, on page 737
- Multi-port RAM Recommended Coding Style, on page 739

Prerequisites for Multi-Port RAM Extraction

In order to infer the nram, you must do the following:

- The compiler will infer the nram even if you do not register the read address, as subsequent examples show. However, for many technology families, you must register the read address for the nram to be implemented as vendor RAMs.
- If the writes are in different processes, you no longer need to specify no_rw_check for the syn_ramstyle attribute in order to infer the RAM. Note that the simulator produces x's when the two writes occur at the same time while the synthesis tool assumes that one write gets priority over the other based on the code.

Write Processes and Multi-Port RAM Inference

The compiler infers the multi-port RAMs from the write processes as follows:

- When all the writes are made in one process, there are no address conflicts, and the compiler generates an nram.

- When writes are made in multiple processes, the software does not infer a multi-port RAM unless there is an explicit syn_ramstyle attribute. If there is no attribute, the software does not infer a RAM. If there is an attribute, the software infers a RAM with multiple write ports. You might get a warning about simulation mismatches when the two addresses are the same.

Multi-Port RAM Recommended Coding Style

Currently, you must use a registered read address when you code the RAM. If you do not, you could get error messages.

There are two situations which can result in the error message: "@E:MF216 : ram.v(29) | Found NRAM mem_1[7:0] with multiple processes":

- An nram with two clocks and two write addresses has syn_ramstyle set to a value of registers. The software cannot implement this, because there is a physical FPGA limitation that does not allow registers with multiple writes.
- You have a registered output for an nram with two clocks and two write addresses.

Synchronous READ RAMs

All RAM primitives that the synthesis tool generates for inferred RAMs have asynchronous READs.

Multi-port RAM Extraction

The compiler extracts multi-write port RAMs, creating an nram primitive which can have n write ports. The nrams are implemented differently depending on your technology. For more information about the implementations, see the appropriate vendor chapter.

In order for the compiler to infer an nram, you must do the following:

- Register the read address.
- Add the syn_ramstyle attribute.

The compiler then infers the multi-port RAMs from the write processes as follows:

- When all the writes are made in one process, there are no address conflicts, and the compiler generates an nram.
- When writes are made in multiple processes, the software does not infer a multi-port RAM unless there is an explicit syn_ramstyle attribute. If there is no attribute, the software does not infer a RAM. If there is an attribute, the software infers a RAM with multiple write ports. You might get a warning about simulation mismatches when the two addresses are the same.
- If the writes are in different processes, you no longer need to specify no_rw_check for the syn_ramstyle attribute in order to infer the RAM. Note that the simulator produces x's when the two writes occur at the same time while the synthesis tool assumes that one write gets priority over the other based on the code.

Multi-port RAM Recommended Coding Style

You must use a registered read address when you code the RAM or have writes to one process. If you have writes to multiple processes, you must use the syn_ramstyle attribute to infer the RAM.

There are two situations which can result in this error message: "@E:MF216 : ram.v(29) | Found NRAM mem_1[7:0] with multiple processes":

- An nram with two clocks and two write addresses has syn_ramstyle set to a value of registers. The software cannot implement this, because there is a physical FPGA limitation that does not allow registers with multiple writes.
- You have a registered output for an nram with two clocks and two write addresses.

Two-write Port RAM Example

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

entity one is
generic (data_width : integer := 4;
address_width :integer := 5 );
```

```
   port (data_a:in std_logic_vector(data_width-1 downto 0);
         data_b:in std_logic_vector(data_width-1 downto 0);
         addr_a:in std_logic_vector(address_width-1 downto 0);
         addr_b:in std_logic_vector(address_width-1 downto 0);
         wren_a:in std_logic;
         wren_b:in std_logic;
         clk:in std_logic;
         q_a:out std_logic_vector(data_width-1 downto 0);
         q_b:out std_logic_vector(data_width-1 downto 0) );
end one;

architecture rtl of one is
type mem_array is array(0 to 2**(address_width) -1) of
std_logic_vector(data_width-1 downto 0);
signal mem : mem_array;
attribute syn_ramstyle : string;
attribute syn_ramstyle of mem : signal is "no_rw_check" ;
signal addr_a_reg : std_logic_vector(address_width-1 downto 0);
signal addr_b_reg : std_logic_vector(address_width-1 downto 0);
begin
   WRITE_RAM : process (clk)
   begin
      if rising_edge(clk) then
         if (wren_a = '1') then
            mem(to_integer(unsigned(addr_a))) <= data_a;
         end if;
         if (wren_b='1') then
            mem(to_integer(unsigned(addr_b))) <= data_b;
         end if;
         addr_a_reg <= addr_a;
         addr_b_reg <= addr_b;
      end if;
   end process WRITE_RAM;
q_a <= mem(to_integer(unsigned(addr_a_reg)));
q_b <= mem(to_integer(unsigned(addr_b_reg)));
end rtl;
```

Instantiating RAMs with SYNCORE

The SYNCORE Memory Compiler is available under the IP Wizard to help you generate HDL code for your specific RAM implementation requirements. For information on using the SYNCORE Memory Compiler, see Specifying RAMs with SYNCore, on page 376 in the *User Guide*.

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 541.

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 11, *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 168 of the *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 Constraints, on page 1125, for details about constraint syntax.

VHDL Attribute and Directive Syntax

Synthesis attributes and directives can be defined in the VHDL source code to control the way the design is analyzed, compiled, and mapped. *Attributes* direct the way your design is optimized and mapped during synthesis. *Directives* control the way your design is analyzed prior to compilation. Because of this distinction, directives must be included in your VHDL source code while attributes can be specified either in the source code or in a constraint file.

The synthesis tool directives and attributes are predefined in the attributes package in the synthesis tool library. This library package contains the built-in attributes, along with declarations for timing constraints (including black-box timing constraints) and vendor-specific attributes. The file is located here:

installDirectory/lib/vhd/synattr.vhd

There are two ways to specify VHDL attributes and directives:

- Using the attributes Package, on page 746
- Declaring Attributes, on page 747

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

This is the most typical way to specify the attributes, because you only need to specify the package once. You specify the attributes package, using the following code:

```
library synplify;
use synplify.attributes.all;
-- design_unit_declarations
attribute productname_attribute of object : object_type is value ;
```

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
```

Declaring Attributes

The alternative method is to declare the attributes to explicitly define them. You must do this each time you use an attribute. Here is the syntax for declaring 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
```

Case Sensitivity

Although VHDL is case-insensitive, directives, attributes, and their values are case sensitive and must be declared in the code using the correct case. This rule applies especially for port names in directives.

For example, if a port in VHDL is defined as GIN, the following code does not work:

```
attribute black_box_tri_pin : string;
attribute black_box_tri_pin of BBDLHS : component is "gin";
```

The following code is correct because the case of the port name is correct:

```
attribute black_box_tri_pin : string;
attribute black_box_tri_pin of BBDLHS : component is "GIN";
```

VHDL Synthesis Examples

This section describes the VHDL examples that are provided with the synthesis tool. The topics include:

- Combinational Logic Examples, on page 748
- Sequential Logic Examples, on page 749

Combinational Logic Examples

The following combinational logic synthesis examples are included in the *installDirectory*/examples/vhdl/common_rtl/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 *install-Directory*/examples/vhdl/common_rtl/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 677
- Level-sensitive Latches Using Concurrent Signal Assignments, on page 681
- Level-sensitive Latches Using VHDL Processes, on page 682
- Asynchronous Sets and Resets, on page 687
- Synchronous Sets and Resets, on page 688

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 *installDirectory*/examples/vhdl/common_rtl/prep directory:

- PREP Benchmark 1, Data Path (prep1.vhd)
- PREP Benchmark 2, Timer/Counter (prep2.vhd)
- PREP Benchmark 3, Small State Machine (prep3.vhd)
- PREP Benchmark 4, Large State Machine (prep4.vhd)
- PREP Benchmark 5, Arithmetic Circuit (prep5.vhd)
- PREP Benchmark 6, 16-Bit Accumulator (prep6.vhd)
- PREP Benchmark 7, 16-Bit Counter (prep7.vhd)
- PREP Benchmark 8, 16-Bit Pre-scaled Counter (prep8.vhd)
- PREP Benchmark 9, Memory Map (prep9.vhd)

The source code for the benchmarks can be used for design examples for synthesis or for doing your own FPGA vendor comparisons.

CHAPTER 11

VHDL 2008 Language Support

This chapter describes support for the VHDL 2008 standard in the Synopsys FPGA synthesis tools. For information on the VHDL standard, see Chapter 10, *VHDL Language Support* and the IEEE 1076™-2008 standard. The following sections describe the current level of VHDL 2008 support.

Features

The support for VHDL 2008 includes:

- Operators, on page 752
- Unconstrained Data Types, on page 755
- Unconstrained Record Elements, on page 757
- Predefined Functions, on page 758
- Packages, on page 760
- Generics in Packages, on page 763
- Context Declarations, on page 763
- Case-generate Statements, on page 764
- Else/elsif Clauses, on page 767
- Syntax Conventions, on page 768

Operators

VHDL 2008 includes support for the following operators:

- Logical Reduction operators – the logic operators: and, or, nand, nor, xor, and xnor can now be used as unary operators
- Condition operator (??) – converts a bit or std_ulogic value to a boolean value
- Matching Relational operators (?=, ?/=, ?<, ?<=, ?>, ?>=) – similar to the normal relational operators, but return bit or std_ulogic values in place of Boolean values

Bit-string Literals

Bit-string literal support in VHDL 2008 includes:

- Support for characters other than 0 and 1 in the bit string, such as X or Z.

  For example:

  `X"Z45X"` is equivalent to `"ZZZZ01000101XXXX"`

  `B"0001-"` is equivalent to `"0001-"`

  `O"75X"` is equivalent to `"111101XXX"`

- Optional support for a length specifier that determines the length of the string to be assigned.

  Syntax: [*length*] *baseSpecifier* **"***bitStringvalue***"**

  For example:

  `12X"45"` is equivalent to `"000001000101"`

  `5O"17"` is equivalent to `"01111"`

- Optional support for a signed (S) or unsigned (U) qualifier that determines how the bit-string value is expanded/truncated when a length specifier is used.

  Syntax: [*length*] **S|U** *baseSpecifier* **"***bitStringvalue***"**

  For example:

  `12UB"101"` is equivalent to `"000000000101"`

`12SB"101"` is equivalent to `"111111111101"`

`12UX"96"` is equivalent to `"000010010110"`

`12SX"96"` is equivalent to `"111110010110"`

- Additional support for a base specifier for decimal numbers (D). The number of characters in the bit string can be determined by using the expression $(\log_2 n)+1$; where n is the decimal integer.

  Syntax: [*length*] **D "***bitStringvalue***"**

  For example:

  `D"10"` is equivalent to `"1010"`

  `10D"35"` is equivalent to `"0000100011"`

For complete descriptions of bit-string literal requirements, see the VHDL 2008 LRM.

Logical Reduction Operators

The logical operators and, or, nand, nor, xor, and xnor can be used as unary operators.

Example – Logical Operators

Condition Operator

The condition operator (??) converts a bit or std_ulogic value to a boolean value. The operator is implicitly applied in a condition where the expression would normally be interpreted as a boolean value as shown in the if statement in the two examples below.

Example – VHDL 2008 Style Conditional Operator

Example – VHDL 1993 Style Conditional Operator

In the VHDL 2008 example, the statement

```
if sel then
```

is equivalent to:

```
if (?? sel) then
```

The implicit use of the ?? operator occurs in the following conditional expressions:

- after if or elsif in an if statement
- after if in an if-generate statement
- after until in a wait statement
- after while in a while loop
- after when in a conditional signal statement
- after assert in an assertion statement
- after when in a next statement or an exit statement

Matching Relational Operators

The matching relational operators return a bit or std_ulogic result in place of a Boolean.

Example – Relational Operators

Unconstrained Data Types

VHDL 2008 allows the element types for arrays and the field types for records to be unconstrained. In addition, VHDL 2008 includes support for partially constrained subtypes in which some elements of the subtype are constrained, while others elements are unconstrained. Specifically, VHDL 2008:

- Supports unconstrained arrays of unconstrained arrays (i.e., element types of arrays can be unconstrained)
- Supports the VHDL 2008 syntax that allows a new subtype to be declared that constrains any element of an existing type that is not yet constrained
- Supports the ‘element attribute that returns the element subtype of an array object
- Supports the new ‘subtype attribute that returns the subtype of an object

Example – Unconstrained Element Types

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

package myTypes is
   type memUnc is array (natural range <>) of std_logic_vector;
   function summation(varx: memUnc) return std_logic_vector;
end package myTypes;

package body myTypes is
   function summation(varx: memUnc) return std_logic_vector is
      variable sum: varx’element;
   begin
      sum := (others => '0');
         for I in 0 to varx’length - 1 loop
            sum := sum + varx(I);
         end loop;
      return sum;
   end function summation;
end package body myTypes;

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

```
entity sum is
   port (in1: memUnc(0 to 2)(3 downto 0);
         out1: out std_logic_vector(3 downto 0) );
end sum;

architecture uncbehv of sum is
begin
   out1 <= summation(in1);
end uncbehv;
```

Example – Unconstrained Elements within Nested Arrays

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

package myTypes is
   type t1 is array (0 to 1) of std_logic_vector;
   type memUnc is array (natural range <>) of t1;
   function doSum(varx: memUnc) return std_logic_vector;
end package myTypes;

package body myTypes is
   function addVector(vec: t1) return std_logic_vector is
      variable vecres: vec’element := (others => '0');
   begin
      for I in vec’Range loop
         vecres := vecres + vec(I);
      end loop;
      return vecres;
   end function addVector;
   function doSum(varx: memUnc) return std_logic_vector is
      variable sumres: varx’element’element;
   begin
      if (varx’length = 1) then
         return addVector(varx(varx’low));
      end if;
      if (varx’Ascending) then
         sumres := addVector(varx(varx’high)) +
            doSum(varx(varx’low to varx’high-1));
      else
         sumres := addVector(varx(varx’low)) +
            doSum(varx(varx’high downto varx’low+1));
      end if;
      return sumres;
   end function doSum;
end package body myTypes;
```

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

entity uncfunc is
   port (in1: in memUnc(1 downto 0)(open)(0 to 3);
         in2: in memUnc(0 to 2)(open)(5 downto 0);
         in3: in memUnc(3 downto 0)(open)(2 downto 0);
         out1: out std_logic_vector(5 downto 0);
         out2: out std_logic_vector(0 to 3);
         out3: out std_logic_vector(2 downto 0) );
end uncfunc;

architecture uncbehv of uncfunc is
begin
   out1 <= doSum(in2);
   out2 <= doSum(in1);
   out3 <= doSum(in3);
end uncbehv;
```

Unconstrained Record Elements

VHDL 2008 allows element types for records to be unconstrained (earlier versions of VHDL required that the element types for records be fully constrained). In addition, VHDL 2008 supports the concept of partially constrained subtypes in which some parts of the subtype are constrained, while others remain unconstrained.

Example – Unconstrained Record Elements

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

entity unctest is
   port (in1: in std_logic_vector (2 downto 0);
         in2: in std_logic_vector (3 downto 0);
         out1: out std_logic_vector(2 downto 0) );
end unctest;
```

```
architecture uncbehv of unctest is
   type zRec is record
      f1: std_logic_vector;
      f2: std_logic_vector;
   end record zRec;
subtype zCnstrRec is zRec(f1(open), f2(3 downto 0));
subtype zCnstrRec2 is zCnstrRec(f1(2 downto 0), f2(open));
signal mem: zCnstrRec2;
begin
   mem.f1 <= in1;
   mem.f2 <= in2;
   out1 <= mem.f1 and mem.f2(2 downto 0);
end uncbehv;
```

Predefined Functions

VHDL 2008 adds the minimum and maximum predefined functions. The behavior of these functions is defined in terms of the "<" operator for the operand type. The functions can be binary to compare two elements, or unary when the operand is an array type.

Example – Minimum/Maximum Predefined Functions

```
entity minmaxTest is
   port (ary1, ary2: in bit_vector(3 downto 0);
         minout, maxout: out bit_vector(3 downto 0);
         unaryres: out bit );
end minmaxTest;

architecture rtlArch of minmaxTest is
begin
   maxout <= maximum(ary1, ary2);
   minout <= minimum(ary1, ary2);
   unaryres <= maximum(ary1);
end rtlArch;
```

Generic Types

VHDL 2008 introduces several types of generics that are not present in VHDL IEEE Std 1076-1993. These types include generic types, generic packages, and generic subprograms.

Generic Types

Generic types allow logic descriptions that are independent of type. These descriptions can be declared as a generic parameter in both packages and entities. The actual type must be provided when instantiating a component or package.

Example of a generic type declaration:

```
entity mux is
   generic (type dataType);
   port (sel: in bit; za, zb: in dataType; res: out dataType);
end mux;
```

Example of instantiating an entity with a type generic:

```
inst1:  mux generic map (bit_vector(3 downto 0))
   port map (selval,in1,in2,out1);
```

Generic Packages

Generic packages allow descriptions based on a formal package. These descriptions can be declared as a generic parameter in both packages and entities. An actual package (an instance of the formal package) must be provided when instantiating a component or package.

Example of a generic package declaration:

```
entity mux is generic (
   package argpkg is new dataPkg generic map (<>);
);
   port (sel: in bit; za, zb: in bit_vector(3 downto 0);
      res: out bit_vector(3 downto 0) );
end mux;
```

Example of instantiating a component with a package generic:

```
package memoryPkg is new dataPkg generic map (4, 16);

...

inst1: entity work.mux generic map (4, 16, argPkg => memoryPkg)
```

Generic Subprograms

Generic subprograms allow descriptions based on a formal subprogram that provides the function prototype. These descriptions can be declared as a generic parameter in both packages and entities. An actual function must be provided when instantiating a component or package.

Example of a generic subprogram declaration:

```
entity mux is
   generic (type dataType; function filter(datain: dataType)
      return dataType);
   port (sel: in bit; za, zb: in dataType; res: out dataType);
end mux;
```

Example of instantiating a component with a subprogram generic:

```
architecture myarch2 of myTopDesign is
   function intfilter(din: integer) return integer is
   begin
      return din + 1;
   end function intfilter;

...

begin
   inst1: mux generic map (integer, intfilter)
      port map (selval,intin1,intin2,intout);
```

Packages

VHDL 2008 includes several new packages and modifies some of the existing packages. The new and modified packages are located in the $LIB/vhd2008 folder instead of $LIB/vhd.

New Packages

The following packages are supported in VHDL 2008:

- fixed_pkg.vhd, float_pkg.vhd, fixed_generic_pkg.vhd, float_generic_pkg.vhd, fixed_float_types.vhd – IEEE fixed and floating point packages
- numeric_bit_unsigned.vhd – Overloads for bit_vector to have all operators defined for ieee.numeric_bit.unsigned
- numeric_std_unsigned.vhd – Overloads for std_ulogic_vector to have all operators defined for ieee.numeric_std.unsigned

String and text I/O functions in the above packages are not supported. These functions include read(), write(), to_string().

Modified Packages

The following modified IEEE packages are supported with the exception of the new string and text I/O functions (the previously supported string and text I/O functions are unchanged):

- std.vhd – new overloads
- std_logic_1164.vhd – std_logic_vector is now a subtype of std_ulogic_vector; new overloads
- numeric_std.vhd – new overloads
- numeric_bit.vhd – new overloads

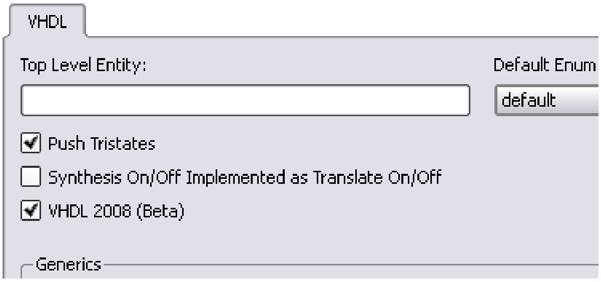
Unsupported Packages/Functions

The following packages and functions are not currently supported:

- string and text I/O functions in the new packages
- The fixed_pkg_params.vhd or float_pkg_params.vhd packages, which were temporarily supported to allow the default parameters to be changed for fixed_pkg.vhd and float_pkg.vhd packages, have been obsoleted by the inclusion of the fixed_generic_pkg.vhd or float_generic_pkg.vhd packages.

Using the Packages

A switch for VHDL 2008 is located in the GUI on the VHDL panel (Implementation Options dialog box) to enable use of these packages and the ?? operator.



VHDL

Top Level Entity:

Default Enum: default

☑ Push Tristates

□ Synthesis On/Off Implemented as Translate On/Off

☑ VHDL 2008 (Beta)

Generics

You can also enable the VHDL 2008 packages by including the following command in the compiler options section of your project file:

```
set_option -vhdl2008 1
```

Generics in Packages

In VHDL 2008, packages can include generic clauses. These generic packages can then be instantiated by providing values for the generics as shown in the following example.

Example – Including Generics in Packages

Context Declarations

VHDL 2008 provides a new type of design unit called a context declaration. A context is a collection of library and use clauses. Both context declarations and context references are supported as shown in the following example.

Example – Context Declaration

In VHDL 2008, a context clause cannot precede a context declaration. The following code segment results in a compiler error.

```
library ieee; -- Illegal context clause before a
               -- context declaration
context zcontext is
   use ieee.std_logic_1164.all;
   use ieee.numeric_std.all;
end context zcontext;
```

Similarly, VHDL 2008 does not allow reference to the library name work in a context declaration. The following code segment also results in a compiler error.

```
context zcontext is
   use work.zpkg.all; -- Illegal reference to library work
                   -- in a context declaration
   library ieee;
   use ieee.numeric_std.all;
end context zcontext;
```

VHDL 2008 supports the following two, standard context declarations in the IEEE package:

- IEEE_BIT_CONTEXT
- IEEE_STD_CONTEXT

Case-generate Statements

The case-generate statement is a new type of generate statement incorporated into VHDL 2008. Within the statement, alternatives are specified similar to a case statement. A static (computable at elaboration) select statement is compared against a set of choices as shown in the following syntax:

```
caseLabel:  case expression generate
                when choice1 =>
                        -- statement list
                 when choice2 =>
                        -- statement list
                 …
                 end generate caseLabel;
```

To allow for configuration of alternatives in case-generate statements, each alternative can include a label preceding the choice value (e.g., labels L1 and L2 in the syntax below):

```
caseLabel:  case expression generate
                when L1: choice1 =>
                        -- statement list
                 when L2: choice2 =>
                        -- statement list
                 …
                 end generate caseLabel;
```

Example – Case-generate Statement with Alternatives

```
entity myTopDesign is
   generic (instSel: bit_vector(1 downto 0) := "10");
   port (in1, in2, in3: in bit; out1: out bit);
end myTopDesign;
```

```
architecture myarch2 of myTopDesign is
component mycomp
   port (a: in bit; b: out bit);
end component;

begin
a1: case instSel generate
   when "00" =>
      inst1: component mycomp port map (in1,out1);
   when "01" =>
      inst1: component mycomp port map (in2,out1);
   when others =>
      inst1: component mycomp port map (in3,out1);
   end generate;
end myarch2;
```

Example – Case-generate Statement with Labels for Configuration

```
entity myTopDesign is
generic (selval: bit_vector(1 downto 0) := "10");
   port (in1, in2, in3: in bit; tstIn: in bit_vector(3 downto 0);
         out1: out bit);
end myTopDesign;

architecture myarch2 of myTopDesign is
   component mycomp
      port (a: in bit; b: out bit);
   end component;
begin
a1: case selval generate
   when spec1: "00" | "11"=> signal inRes: bit;
      begin
         inRes <= in1 and in3;
         inst1: component mycomp port map (inRes,out1);
      end;
   when spec2: "01" =>
      inst1: component mycomp port map (in1, out1);
   when spec3: others =>
      inst1: component mycomp port map (in3,out1);
   end generate;
end myarch2;

entity mycomp is
   port (a : in bit;
         b : out bit);
end mycomp;
```

```
architecture myarch of mycomp is
begin
   b <= not a;
end myarch;

architecture zarch of mycomp is
begin
   b <= '1';
end zarch;

configuration myconfig of myTopDesign is
for myarch2
   for a1 (spec1)
      for inst1: mycomp use entity mycomp(myarch);
      end for;
   end for;
   for a1 (spec2)
      for inst1: mycomp use entity mycomp(zarch);
      end for;
   end for;
   for a1 (spec3)
      for inst1: mycomp use entity mycomp(myarch);
      end for;
   end for;
end for;
end configuration myconfig;
```

Matching case and select Statements

Matching case and matching select statements are supported – case? (matching case statement) and select? (matching select statement). The statements use the ?= operator to compare the case selector against the case options.

Example – Use of case? Statement

Example – Use of select? Statement

Else/elsif Clauses

In VHDL 2008, else and elsif clauses can be included in if-generate statements. You can configure specific if/else/elsif clauses using configurations by adding a label before each condition. In the code example below, the labels on the branches of the if-generate statement are spec1, spec2, and spec3. These labels are later referenced in the configuration myconfig to specify the appropriate entity/architecture pair. This form of labeling allows statements to be referenced in configurations.

Example – Else/elsif Clauses in If-Generate Statements

Syntax Conventions

The following syntax conventions are supported in VHDL 2008:

- All keyword
- Comment delimiters
- Extended character set

All Keyword

VHDL 2008 supports the use of an all keyword in place of the list of input signals to a process in the sensitivity list.

Example – All Keyword in Sensitivity List

Comment Delimiters

VHDL 2008 supports the /* and */ comment-delimiter characters. All text enclosed between the beginning /* and the ending */ is treated as a comment, and the commented text can span multiple lines. The standard VHDL "--" comment-introduction character string is also supported.

Extended Character Set

The extended ASCII character literals (ASCII values from 128 to 255) are supported.

Example – Extended Character Set

CHAPTER 8

Utilities

This chapter describes some Synopsys FPGA tools and utilities that help productivity and smooth out the design process. See the following topics for information:

sdc2fdc Tcl Shell Command

The sdc2fdc Tcl shell command translates legacy FPGA timing constraints to Synopsys FPGA timing constraints. From the Tcl command line in the synthesis tool, the sdc2fdc command scans the input SDC files and attempts to convert constraints for the implementation. For details, see:

- sdc2fdc Tcl Shell Command, on page 770
- Examples of sdc2fdc Translation, on page 770
- FPGA Design Constraint (FDC) File, on page 771
- Troubleshooting, on page 773

sdc2fdc Tcl Shell Command

From the Tcl command line, type:

sdc2fdc

For information about how to run this command, see Converting SDC to FDC, on page 92.

Examples of sdc2fdc Translation

```
% sdc2fdc

INFO: Translation successful.
See:"D:/bugs/timing_88/clk_prior/scratch/FDC_constraints/rev_2/top_translated.fdc"
Replace your current *.sdc files with this one.

INFO: Automatically updating your project to reflect the new constraint file(s)
Do "Ctrl+S" to save the new settings.

% sdc2fdc

ERROR: Bad -from list for define_false_path: {my_inst}
Missing qualifier(s) (i: p: n: ...)
ERROR: Translation problems were found.
See:"D:/bugs/timing_88/clk_prior/scratch/FDC_constraints/rev_2/top_translate.log"
for details.

_translate.log
```

```
ERROR:  Bad -from list for define_false_path  {my_inst}
        Missing qualifier(s) (i: p: n: ...)

"define_false_path -from {my_inst} -to i:abc.def.g_reg -through {n:bar}”
Synplicity SDC source file: D:/bugs/timing_88/clk_prior/scratch/top.sdc.
Line number: 79
```

FPGA Design Constraint (FDC) File

FPGA Design Constraints (FDC) are input constraints for the FPGA synthesis tool. The file format is migrating from legacy FPGA timing constraints (for example, define_clock, define_input_delay, and define_false_path) to *Synopsys SDC Standard*-compliant timing constraints (for example, create_clock, set_input_delay, and set_false_path). FDC combines the FPGA design constraints (for example, define_attribute, define_scope_collection, and define_io_standard) with the Synopsys standard timing constraints.

The FDC constraint file is generated after running sdc2fdc that translates legacy FPGA design constraints (SDC) to Synopsys FPGA design constraints (FDC). This file is divided into two sections. The first section contains the FPGA design constraints that are valid (for example, define_scope_collection and define_attribute) and the legacy timing constraints that are not translated because they were specified with -disable. The second section contains the timing constraints that were translated into Synopsys SDC standard format. This file also provides the following:

- Each source sdc file has its separate subhead.
- Each compile point is treated as a top level, so its sdc file has its own _translated.fdc file.
- The translator adds the naming rule, set_rtl_ff_names, so that the synthesis tool knows these constraints are not from the Synopsys Design Compiler.

The following example shows the contents of the FDC file.

```
##################################################################################
####This file contains constraints from Synplicity SDC files that have been
####translated into Synopsys FPGA Design Constraints (FDC.
####Translated FDC output file:
####D:/bugs/timing_88/clk_prior/scratch/FDC_constraints/rev_2/top_translated.fdc
####Source SDC files to the translation:
####D:/bugs/timing_88/clk_prior/scratch/top.sdc
##################################################################################
```

```
#############################################################################
####Source SDC file to the translation:
####D:/bugs/timing_88/clk_prior/scratch/top.sdc
#############################################################################

#Legacy constraint file
#C:\Clean_Demos\Constraints_Training\top.sdc
#Written on Mon May 21 15:58:35 2012
#by Synplify Pro, Synplify Pro Scope Editor
#
#Collections
#
define_scope_collection all_grp {define_collection \
        [find -inst {i:FirstStbcPhase}] \
        [find -inst {i:NormDenom[6:0]}] \
        [find -inst {i:NormNum[7:0]}] \
        [find -inst {i:PhaseOut[9:0]}] \
        [find -inst {i:PhaseOutOld[9:0]}] \
        [find -inst {i:PhaseValidOut}] \
        [find -inst {i:ProcessData}] \
        [find -inst {i:Quadrant[1:0]}] \
        [find -inst {i:State[2:0]}] \
        }
#
#Clocks

#define_clock -disable -name {clkc} -virtual -freq 150 -clockgroup default_clkgroup_1

#Clock to Clock
#
#
#Inputs/Outputs
#
define_input_delay -disable {b[7:0]} 2.00 -ref clka:r
define_input_delay -disable {c[7:0]} 0.20 -ref clkb:r
define_input_delay -disable {d[7:0]} 0.30 -ref clkb:r
define_output_delay -disable {x[7:0]} -improve 0.00 -route 0.00
define_output_delay -disable {y[7:0]} -improve 0.00 -route 0.00
#
#Registers
#
#
#Multicycle Path
#
#
#False Path
#
#
define_false_path -disable -from {i:x[1]}
#

#Path Delay
#
#
#Attributes
#
define_io_standard -default_input -delay_type input syn_pad_type {LVCMOS_33}#

#I/O standards
#
```

```
#
#Compile Points
#
#
#Other Constraints

################################################################################
#SDC compliant constraints translated from Legacy Timing Constraints
################################################################################
#
set_rtl_ff_names {#}

create_clock -name {clka} [get_ports {clka}] -period 10 -waveform {0 5.0}
create_clock -name {clkb} [get_ports {clkb}] -period 6.666666666666667
   -waveform {0 3.3333333333333335}
set_input_delay -clock [get_clocks {clka}] -clock_fall -
add_delay 0.000 [all_inputs]
set_output_delay -clock [get_clocks {clka}] -add_delay 0.000 [all_outputs]
set_input_delay -clock [get_clocks {clka}] -
add_delay 2.00 [get_ports {a[7:0]}]
set_input_delay -clock [get_clocks {clka}] -add_delay 0 [get_ports {rst}]
set mcp 4
set_multicycle_path $mcp -start  \
     -from \
       [get_ports \
       {a* \
       b*} \
       ] \
     -to \
       [find -seq -hier {q?[*]} ]

set_multicycle_path 3 -end  \
     -from \
       [find -seq {*y*.q[*]} ]

set_clock_groups -name default_clkgroup_0 -asynchronous \
    -group [get_clocks {clka dcm|clk0_derived_clock dcm|
   clk2x_derived_clock dcm|clk0fx_derived_clock}]
set_clock_groups -name default_clkgroup_1 -asynchronous \
    -group [get_clocks {clkb}]
```

Troubleshooting

The following table contains common types of error messages you might encounter when running the sdc2fdc Tcl shell command. The error messages include descriptions of how you can resolve these problems.

Problem and Solution Examples

Cannot translate a translated file	Remove/disable D:FDC_constraints/rev_FDC/top_translated.fdc from the current implementation.
No active constraint files	Add/enable one or more SDC constraint files.

Clock not translated	• Add clock object qualifier (p: n: ...) for: "define_clock -name {clka {clka} -period 10 -clockgroup {default_clkgroup_0}" Synplicity_SDC source file: D:.../clk_prior/scratch/top.sdc. Line number: 32 • Specify -name for: "define_clock {p:clkb} -period 20 -clockgroup {default_clkgroup_1}" Synplicity SDC source file: D:.../clk_prior/scratch/top.sdc. Line number: 33
Bad -from list for define_multicycle_path {a* b*}	Missing qualifier(s) (i: p: n: ...) "define_multicycle_path 4 -from {a* b*} -to $fdc_cmd_0 -start" Synplicity SDC source file: D:.../clk_prior/scratch/top.sdc. Line number: 76
Bad -to list for define_multicycle_path {i: *y* .q[*] p:ena}	Mixing of object types not permitted "define_multicycle_path -to {i:*y*.q[*] p:ena} 3" Synplicity SDC source file: D:.../clk_prior/scratch/top.sdc. Line number: 77
Bad -from list for define_multicycle_path {i:*y* .q[*] p:ena enab}	Mixing of object types and missing qualifiers not permitted "define_multicycle_path -from {i:*y*.q[*] p:ena enab} 3" Synplicity SDC source file: D:.../clk_prior/scratch/top.sdc. Line number: 77
No period or frequency found	Default 1000. "create_clock -name {clkb} {p:clkb} -period 1000 -waveform {0 500.0}" Synplicity SDC source file: D:.../clk_prior/scratch/top.sdc. Line number: 33
Must specify both -rise and -fall or neither	"create_clock -name {clka} {p:clka} -period 10 -rise 5 -clockgroup {default_clkgroup_0" Synplicity SDC source file: D:.../clk_prior/scratch/top.sdc. Line number: 32

Besides these error messages, make sure that your files have read/write permissions set properly and there is sufficient disk space. Fix any issues in the SDC source file and rerun the sdc2fdc command.

Batch Mode

If you run sdc2fdc -batch, then the following occurs:

- The two `Clock not translated` messages in the table above are not generated.
- When the translation is successful, the SDC file is disabled and the FDC file is enabled and saved automatically in the Project file.

  However, if the -batch option is *not* used and the translation is successful, then the SDC file is disabled and the FDC file is enabled but *not* automatically saved in the Project file. A message is generated that states this condition in the Tcl shell window.

Synplicity Archive Utility

The archive utility provides a way to archive, extract, or copy your design projects. An archive file is in Synplicity proprietary format and is saved to a file name using the `sar` extension. You can also use this utility to submit your design along with a request for technical support.

The archive utility is available through the Project menu in the GUI or through the project Tcl command. See the following for details:

For information about...	See...
Archiving, un-archiving, or copying projects	Archiving Files and Projects, on page 155 in the *User Guide*
Archiving a project for Synplicity technical support	*Tech-Support Menu, on page 270*
Syntax for the associated Tcl commands	project -archive, project -unarchive, and project -copy Tcl commands

SYNCore FIFO Compiler

The SYNCore synchronous FIFO compiler offers an IP wizard that generates Verilog code for your FIFO implementation. This section describes the following:

- Synchronous FIFOs, on page 777
- FIFO Read and Write Operations, on page 778
- FIFO Ports, on page 780
- FIFO Parameters, on page 782
- FIFO Status Flags, on page 784
- FIFO Programmable Flags, on page 787

For further information, refer to the following:

- Specifying FIFOs with SYNCore, on page 370 of the *User Guide.*, for information about using the wizard to generate FIFOs
- Launch SYNCore Command, on page 187 and SYNCore FIFO Wizard, on page 189 for descriptions of the interface

Synchronous FIFOs

A FIFO is a First-In-First-Out memory queue. Different control logic manages the read and write operations. A FIFO also has various handshake signals for interfacing with external user modules.

The SYNCore FIFO compiler generates synchronous FIFOs with symmetric ports and one clock controlling both the read and write operations. The FIFO is symmetric because the read and write ports have the same width.

When the Write_enable signal is active and the FIFO has empty locations, data is written into FIFO memory on the rising edge of the clock. A Full status flag indicates that the FIFO is full and that no more write operations can be performed. See FIFO Write Operation, on page 778 for details.

When the FIFO has valid data and Read_enable is active, data is read from the FIFO memory and presented at the outputs. The FIFO Empty status flag indicates that the FIFO is empty and that no more read operations can be performed. See FIFO Read Operation, on page 779 for details.

The FIFO is not corrupted by an invalid request: for example, if a read request is made while the FIFO is empty or a write request is received when the FIFO is full. Invalid requests do not corrupt the data, but they cause the corresponding read or write request to be ignored and the Overflow or Underflow flags to be asserted. You can monitor these status flags for invalid requests. These and other flags are described in FIFO Status Flags, on page 784 and FIFO Programmable Flags, on page 787.

At any point in time, Data count reflects the available data inside the FIFO. In addition, you can use the Programmable Full and Programmable Empty status flags for user-defined thresholds.

FIFO Read and Write Operations

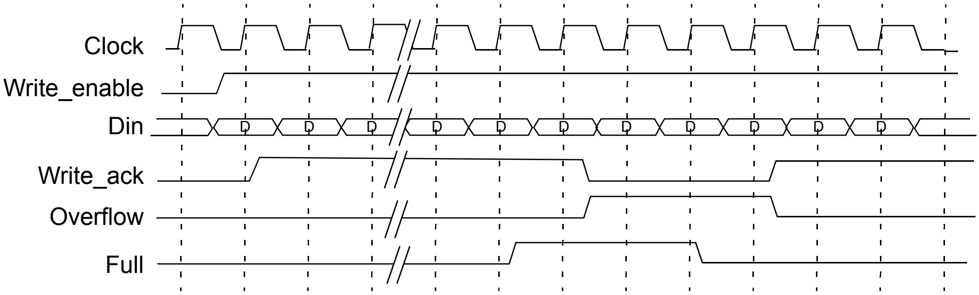
This section describes FIFO behavior with read and write operations.

FIFO Write Operation

When write enable is asserted and the FIFO is not full, data is added to the FIFO from the input bus (Din) and write acknowledge (Write_ack) is asserted. If the FIFO is continuously written without being read, it will fill with data. The status outputs are asserted when the number of entries in the FIFO is greater than or equal to the corresponding threshold, and should be monitored to avoid overflowing the FIFO.

When the FIFO is full, any attempted write operation fails and the overflow flag is asserted.

The following figure illustrates the write operation. Write acknowledge (Write_ack) is asserted on the next rising clock edge after a valid write operation. When Full is asserted, there can be no more legal write operations. This example shows that asserting Write_enable when Full is high causes the assertion of Overflow.

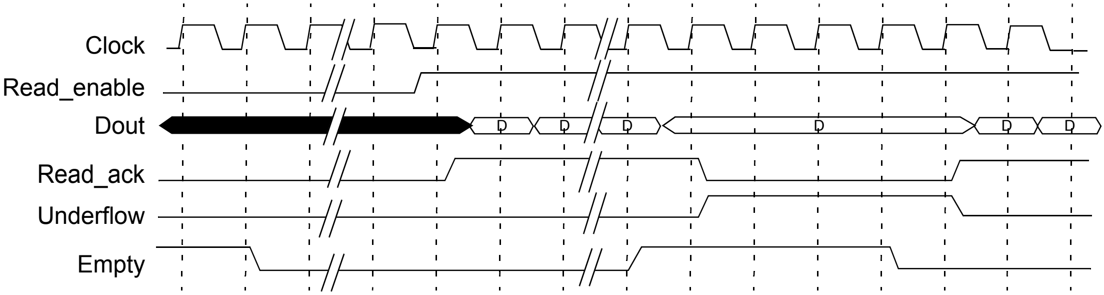


FIFO Read Operation

When read enable is asserted and the FIFO is not empty, the next data word in the FIFO is driven on the output bus (Dout) and a read valid is asserted. If the FIFO is continuously read without being written, the FIFO will empty. The status outputs are asserted when the number of entries in the FIFO are less than or equal to the corresponding threshold, and should be monitored to avoid underflowing the FIFO. When the FIFO is empty, all read operations fail and the underflow flag is asserted.

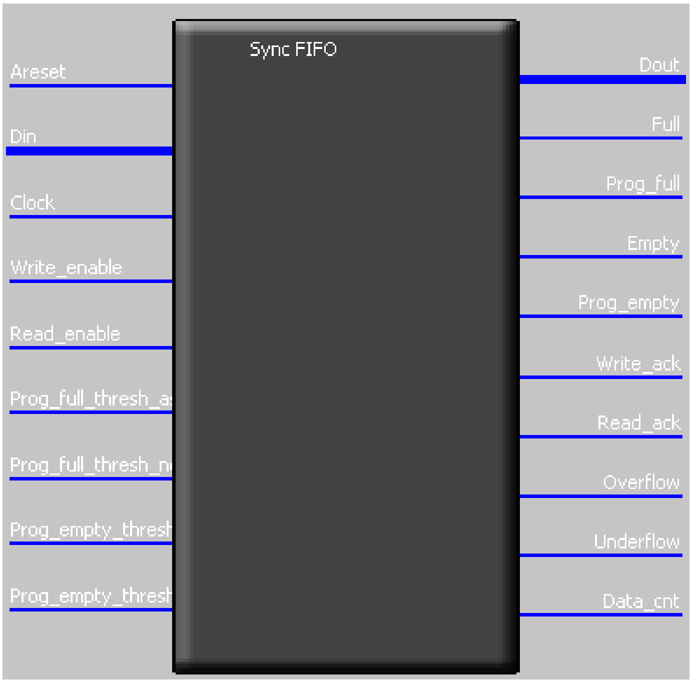
If read and write operation occur simultaneously during the empty state, the write operation will be valid and empty, and is de-asserted at the next rising clock edge. There cannot be a legal read operation from an empty FIFO, so the underflow flag is asserted.

The following figure illustrates a typical read operation. If the FIFO is not empty, Read_ack is asserted at the rising clock edge after Read_enable is asserted and the data on Dout is valid. When Empty is asserted, no more read operations can be performed. In this case, initiating a read causes the assertion of Underflow on the next rising clock edge, as shown in this figure.



FIFO Ports

The following figure shows the FIFO ports.



Port Name	Description
Almost_empty	Almost empty flag output (active high). Asserted when the FIFO is almost empty and only one more read can be performed. Can be active high or active low.
Almost_full	Almost full flag output (active high). Asserted when only one more write can be performed into the FIFO. Can be active high or active low.
AReset	Asynchronous reset input. Resets all internal counters and FIFO flag outputs.
Clock	Clock input for write and read. Data is written/read on the rising edge.
Data_cnt	Data word count output. Indicates the number of words in the FIFO in the read clock domain.

Port Name	Description
Din [width:0]	Data input word to the FIFO.
Dout [width:0]	Data output word from the FIFO.
Empty	FIFO empty output (active high). Asserted when the FIFO is empty and no additional reads can be performed. Can be active high or active low.
Full	FIFO full output (active high). Asserted when the FIFO is full and no additional writes can be performed. Can be active high or active low.
Overflow	FIFO overflow output flag (active high). Asserted when the FIFO is full and the previous write was rejected. Can be active high or active low.
Prog_empty	Programmable empty output flag (active high). Asserted when the words in the FIFO exceed or equal the programmable empty assert threshold. De-asserted when the number of words is more than the programmable full negate threshold. Can be active high or active low.
Prog_empty_ thresh	Programmable FIFO empty threshold input. User-programmable threshold value for the assertion of the Prog_empty flag. Set during reset.
Prog_empty_ thresh_assert	Programmable FIFO empty threshold assert input. User-programmable threshold value for the assertion of the Prog_empty flag. Set during reset.
Prog_empty_ thresh_negate	Programmable FIFO empty threshold negate input. User programmable threshold value for the de-assertion of the Prog_full flag. Set during reset.
Prog_full	Programmable full output flag (active high). Asserted when the words in the FIFO exceed or equal the programmable full assert threshold. De-asserted when the number of words is less than the programmable full negate threshold. Can be active high or active low.
Prog_full_thresh	Programmable FIFO full threshold input. User-programmable threshold value for the assertion of the Prog_full flag. Set during reset.
Prog_full_thresh_ assert	Programmable FIFO full threshold assert input. User-programmable threshold value for the assertion of the Prog_full flag. Set during reset.

Port Name	Description
Prog_full_thresh_ negate	Programmable FIFO full threshold negate input. User-programmable threshold value for the de-assertion of the Prog_full flag. Set during reset.
Read_ack	Read acknowledge output (active high). Asserted when valid data is read from the FIFO. Can be active high or active low.
Read_enable	Read enable output (active high). If the FIFO is not empty, data is read from the FIFO on the next rising edge of the read clock.
Underflow	FIFO underflow output flag (active high). Asserted when the FIFO is empty and the previous read was rejected.
Write_ack	Write Acknowledge output (active high). Asserted when there is a valid write into the FIFO. Can be active high or active low.
Write_enable	Write enable input (active high). If the FIFO is not full, data is written into the FIFO on the next rising edge.

FIFO Parameters

Parameter	Description
AEMPTY_FLAG_SENSE	FIFO almost empty flag sense 0 Active Low 1 Active High
AFULL_FLAG_SENSE	FIFO almost full flag sense 0 Active Low 1 Active High
DEPTH	FIFO depth
EMPTY_FLAG_SENSE	FIFO empty flag sense 0 Active Low 1 Active High
FULL_FLAG_SENSE	FIFO full flag sense 0 Active LowOVERFLOW_ 1 Active High

Parameter	Description
OVERFLOW_FLAG_ SENSE	FIFO overflow flag sense 0 Active Low 1 Active High
PEMPTY_FLAG_ SENSE	FIFO programmable empty flag sense 0 Active Low 1 Active High
PFULL_FLAG_SENSE	FIFO programmable full flag sense 0 Active Low 1 Active High
PGM_EMPTY_ ATHRESH	Programmable empty assert threshold for PGM_EMPTY_TYPE=2
PGM_EMPTY_ NTHRESH	Programmable empty negate threshold for PGM_EMPTY_TYPE=2
PGM_EMPTY_THRESH	Programmable empty threshold for PGM_EMPTY_TYPE=1
PGM_EMPTY_TYPE	Programmable empty type. See Programmable Empty, on page 791 for details. 1 Programmable empty with single threshold constant. 2 Programmable empty with multiple threshold constant 3 Programmable empty with single threshold input 4 Programmable empty with multiple threshold input
PGM_FULL_ATHRESH	Programmable full assert threshold for PGM_FULL_TYPE=2
PGM_FULL_NTHRESH	Programmable full negate threshold for PGM_FULL_TYPE=2
PGM_FULL_THRESH	Programmable full threshold for PGM_FULL_TYPE=1
PGM_FULL_TYPE	Programmable full type. See Programmable Full, on page 788 for details. 1 Programmable full with single threshold constant 2 Programmable full with multiple threshold constant 3 Programmable full with single threshold input 4 Programmable full with multiple threshold input

Parameter	Description
RACK_FLAG_SENSE	FIFO read acknowledge flag sense 0 Active Low 1 Active High
UNDERFLOW_FLAG_ SENSE	FIFO underflow flag sense 0 Active Low 1 Active High
WACK_FLAG_SENSE	FIFO write acknowledge flag sense 0 Active Low 1 Active High
WIDTH	FIFO data input and data output width

FIFO Status Flags

You can set the following status flags for FIFO read and write operations.

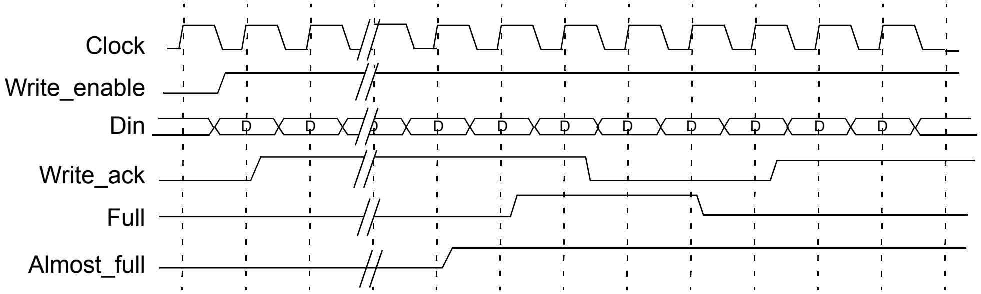
- Full/Almost Full Flags, on page 784
- Empty/Almost Empty Flags, on page 785
- Handshaking Flags, on page 786
- Programmable full and empty flags, which are described in Programmable Full, on page 788 and Programmable Empty, on page 791.

Full/Almost Full Flags

These flags indicate the status of the FIFO memory queue for write operations:

Full	Indicates that the FIFO memory queue is full and no more writes can be performed until data is read. Full is synchronous with the clock (Clock). If a write is initiated when Full is asserted, the write does not succeed and the overflow flag is asserted.
Almost_full	The almost full flag (Almost_full) indicates that there is one location left and the FIFO will be full after one more write operation. Almost full is synchronous to Clock. This flag is guaranteed to be asserted when the FIFO has one remaining location for a write operation.

The following figure displays the behavior of these flags. In this example, asserting Wriite_enable when Almost_full is high causes the assertion of Full on the next rising clock edge.



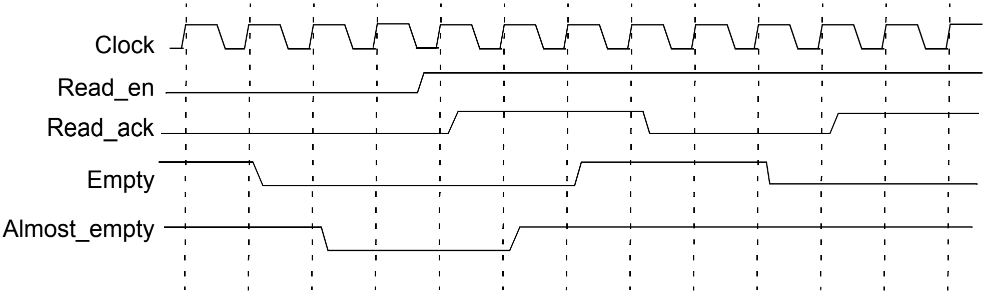
Empty/Almost Empty Flags

These flags indicate the status of the FIFO memory queue for read operations:

Empty Indicates that the memory queue for the FIFO is empty and no more reads can be performed until data is written. The output is active high and is synchronous to the clock. If a read is initiated when the empty flag is true, the underflow flag is asserted.

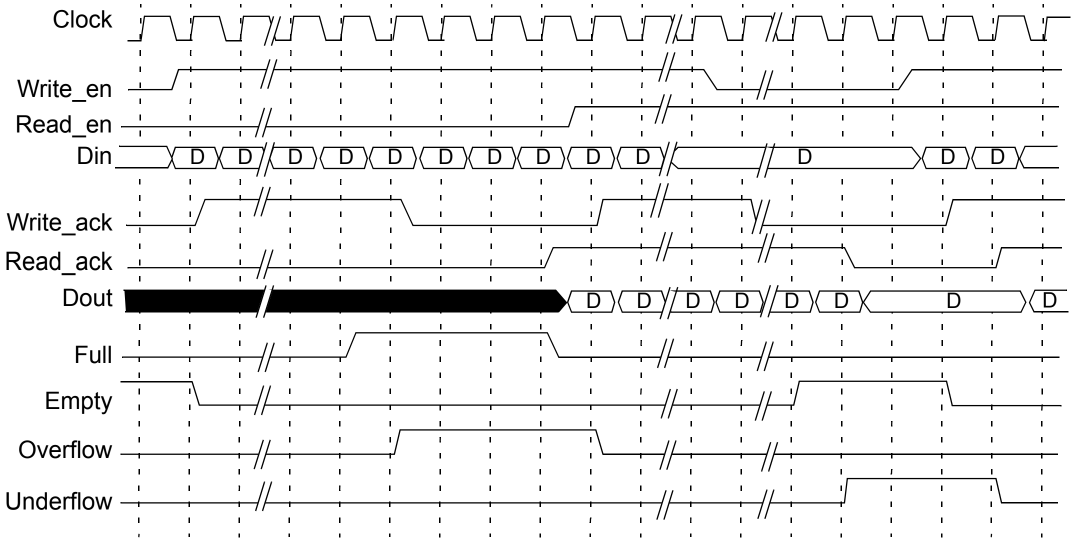
Almost_ empty Indicates that the FIFO will be empty after one more read operation. Almost_empty is active high and is synchronous to the clock. The flag is guaranteed to be asserted when the FIFO has one remaining location for a read operation.

The following figure illustrates the behavior of the FIFO with one word remaining.



Handshaking Flags

You can specify optional Read_ack, Write_ack, Overflow, and Underflow handshaking flags for the FIFO.



Read_ack Asserted at the completion of each successful read operation. It indicates that the data on the Dout bus is valid. It is an optional port that is synchronous with Clock and can be configured as active high or active low.

Read_ack is deasserted when the FIFO is underflowing, which indicates that the data on the Dout bus is invalid. Read_ack is asserted at the next rising clock edge after read enable. Read_enable is asserted when the FIFO is not empty.

Write_ack Asserted at the completion of each successful write operation. It indicates that the data on the Din port has been stored in the FIFO. It is synchronous with the clock, and can be configured as active high or active low.

Write_ack is deasserted for a write to a full FIFO, as illustrated in the figure. Write_ack is deasserted one clock cycle after Full is asserted to indicate that the last write operation was valid and no other write operations can be performed.

Overflow Indicates that a write operation was unsuccessful because the FIFO was full. In the figure, Full is asserted to indicate that no more writes can be performed. Because the write enable is still asserted and the FIFO is full, the next cycle causes Overflow to be asserted. Note that Write_ack is not asserted when FIFO is overflowing. When the write enable is deasserted, Overflow deasserts on the next clock cycle.

Underflow Indicates that a read operation was unsuccessful, because the read was attempted on an empty FIFO. In the figure, Empty is asserted to indicate that no more reads can be performed. As the read enable is still asserted and the FIFO is empty, the next cycle causes Underflow to be asserted. Note that Read_ack is not asserted when FIFO is underflowing. When the read enable is deasserted, the Underflow flag deasserts on the next clock cycle.

FIFO Programmable Flags

The FIFO supports completely programmable full and empty flags to indicate when the FIFO reaches a predetermined user-defined fill level. See the following:

Prog_full Indicates that the FIFO has reached a user-defined full threshold. See Programmable Full, on page 788 for more information.

Prog_empty Indicates that the FIFO has reached a user-defined empty threshold. See Programmable Empty, on page 791 for more information.

Both flags support various implementation options. You can do the following:

- Set a constant value
- Set dedicated input ports so that the thresholds can change dynamically in the circuit
- Use hysteresis, so that each flag has different assert and negative values

Programmable Full

The Prog_full flag (programmable full) is asserted when the number of entries in the FIFO is greater than or equal to a user-defined assert threshold. If the number of words in the FIFO is less than the negate threshold, the flag is deasserted. The following is the valid range of threshold values:

Assert threshold value	Depth / 2 to Max of Depth For multiple threshold types, the assert value should always be larger than the negate value in multiple threshold types.
Negate threshold value	Depth / 2 to Max of Depth

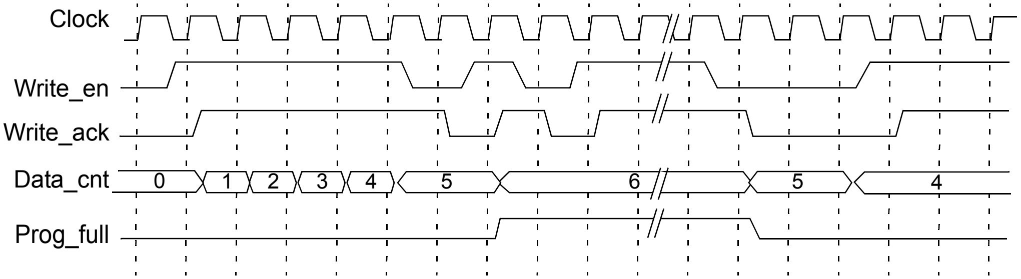
Prog_full has four threshold types:

- Programmable Full with Single Threshold Constant, on page 788
- Programmable Full with Multiple Threshold Constants, on page 789
- Programmable Full with Single Threshold Input, on page 789
- Programmable Full with Multiple Threshold Inputs, on page 790

Programmable Full with Single Threshold Constant

PGM_FULL_TYPE = 1

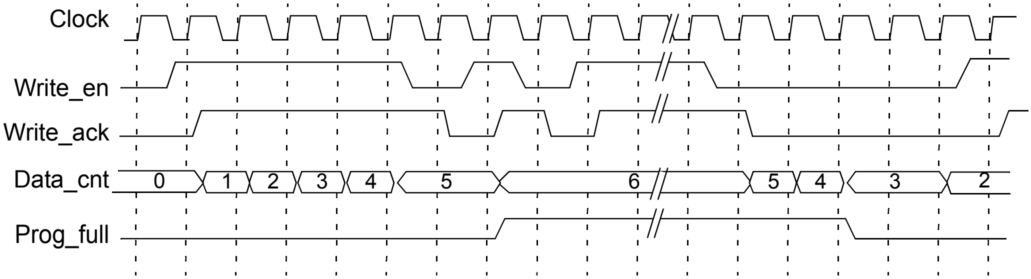
This option lets you set a single constant value for the threshold. It requires significantly fewer resources when the FIFO is generated. This figure illustrates the behavior of Prog_full when configured as a single threshold constant with a value of 6.



Programmable Full with Multiple Threshold Constants

PGM_FULL_TYPE = 2

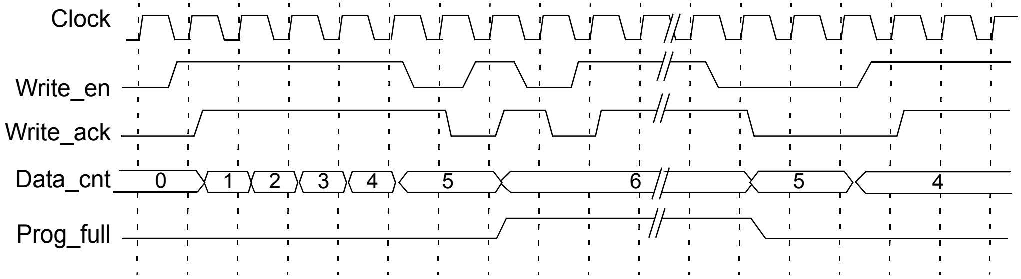
The programmable full flag is asserted when the number of words in the FIFO is greater than or equal to the full threshold assert value. If the number of FIFO words drops to less than the full threshold negate value, the programmable full flag is de-asserted. Note that the negate value must be set to a value less than the assert value. The following figure illustrates the behavior of Prog_full configured as multiple threshold constants with an assert value of 6 and a negate value of 4.



Programmable Full with Single Threshold Input

PGM_FULL_TYPE = 3

This option lets you specify the threshold value through an input port (Prog_full_thresh) during the reset state, instead of using constants. The following figure illustrates the behavior of Prog_full configured as a single threshold input with a value of 6.

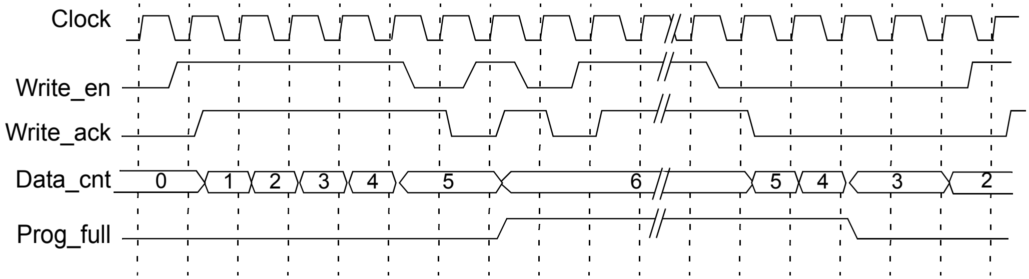


Programmable Full with Multiple Threshold Inputs

PGM_FULL_TYPE = 4

This option lets you specify the assert and negate threshold values dynamically during the reset stage using the Prog_full_thresh_assert and Prog_full_thresh_negate input ports. You must set the negate value to a value less than the assert value.

The programmable full flag is asserted when the number of words in the FIFO is greater than or equal to the Prog_full_thresh_assert value. If the number of FIFO words goes below Prog_full_thresh_negate value, the programmable full flag is deasserted. The following figure illustrates the behavior of Prog_full configured as multiple threshold inputs with an assert value of 6 and a negate value of 4.



Programmable Empty

The programmable empty flag (Prog_empty) is asserted when the number of entries in the FIFO is less than or equal to a user-defined assert threshold. If the number of words in the FIFO is greater than the negate threshold, the flag is deasserted. The following is the valid range of threshold values:

Assert threshold value	1 to Max of Depth / 2 For multiple threshold types, the assert value should always be lower than the negate value in multiple threshold types.
Negate threshold value	1 to Max of Depth / 2

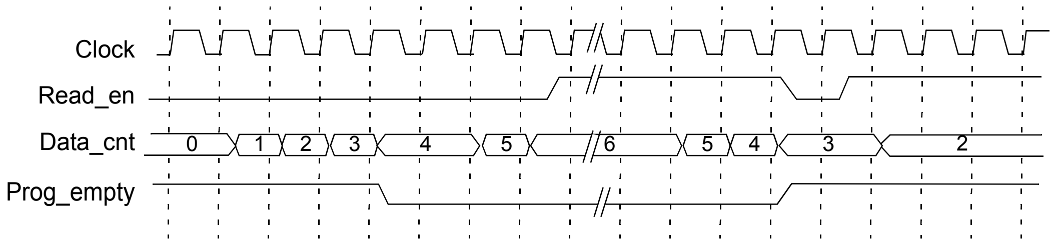
There are four threshold types you can specify:

- Programmable Empty with Single Threshold Constant, on page 791
- Programmable Empty with Multiple Threshold Constants, on page 791
- Programmable Empty with Single Threshold Input, on page 792
- Programmable Empty with Multiple Threshold Inputs, on page 793

Programmable Empty with Single Threshold Constant

PGM_EMPTY_TYPE = 1

This option lets you specify an empty threshold value with a single constant. This approach requires significantly fewer resources when the FIFO is generated. The following figure illustrates the behavior of Prog_empty configured as a single threshold constant with a value of 3.

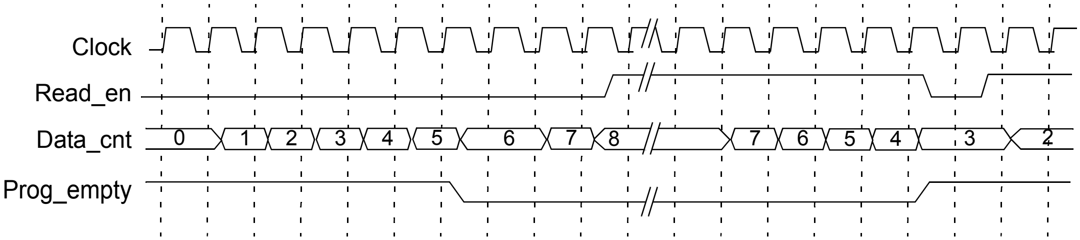


Programmable Empty with Multiple Threshold Constants

PGM_EMPTY_TYPE = 2

This option lets you specify constants for the empty threshold assert value and empty threshold negate value. The programmable empty flag asserts and deasserts in the range set by the assert and negate values. The assert value must be set to a value less than the negate value. When the number of words in the FIFO is less than or equal to the empty threshold assert value, the Prog_empty flag is asserted. When the number of words in FIFO is greater than the empty threshold negate value, Prog_empty is deasserted.

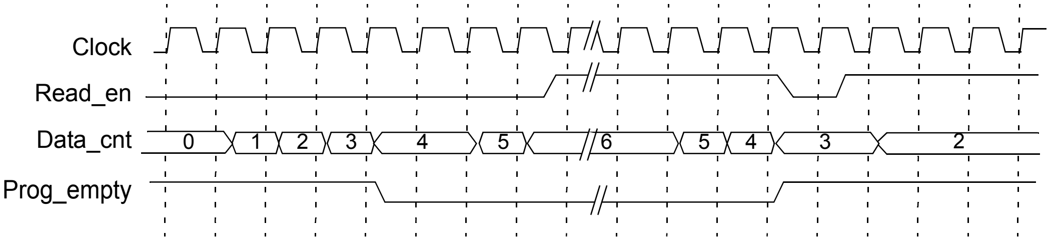
The following figure illustrates the behavior of Prog_empty when configured as multiple threshold constants with an assert value of 3 and a negate value of 5.



Programmable Empty with Single Threshold Input

PGM_EMPTY_TYPE = 3

This option lets you specify the threshold value dynamically during the reset state with the Prog_empty_thresh input port, instead of with a constant. The Prog_empty flag asserts when the number of FIFO words is equal to or less than the Prog_empty_thresh value and deasserts when the number of FIFO words is more than the Prog_empty_thresh value. The following figure illustrates the behavior of Prog_empty when configured as a single threshold input with a value of 3.

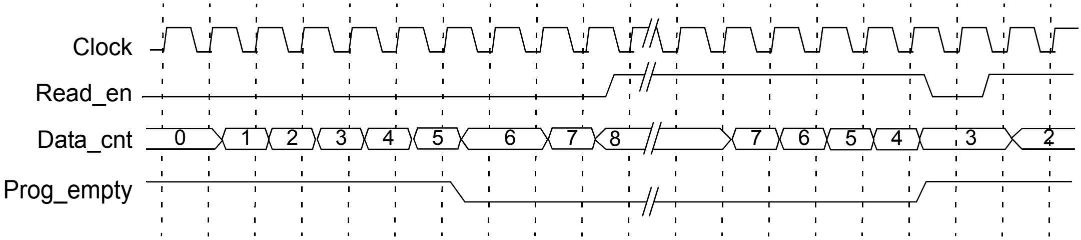


Programmable Empty with Multiple Threshold Inputs

PGM_EMPTY_TYPE = 4

This option lets you specify the assert and negate threshold values dynamically during the reset stage using the Prog_empty_thresh_assert and Prog_empty_thresh_negate input ports instead of constants. The programmable empty flag asserts and deasserts according to the range set by the assert and negate values. The assert value must be set to a value less than the negate value.

When the number of FIFO words is less than or equal to the empty threshold assert value, Prog_empty is asserted. If the number of FIFO words is greater than the empty threshold negate value, the flag is deasserted. The following figure illustrates the behavior of Prog_empty configured as multiple threshold inputs, with an assert value of 3 and a negate value of 5.



SYNCore RAM Compiler

The SYNCore RAM Compiler generates Verilog code for your RAM implementation. This section describes the following:

- Single-Port Memories, on page 794
- Dual-Port Memories, on page 796
- Read/Write Timing Sequences, on page 801

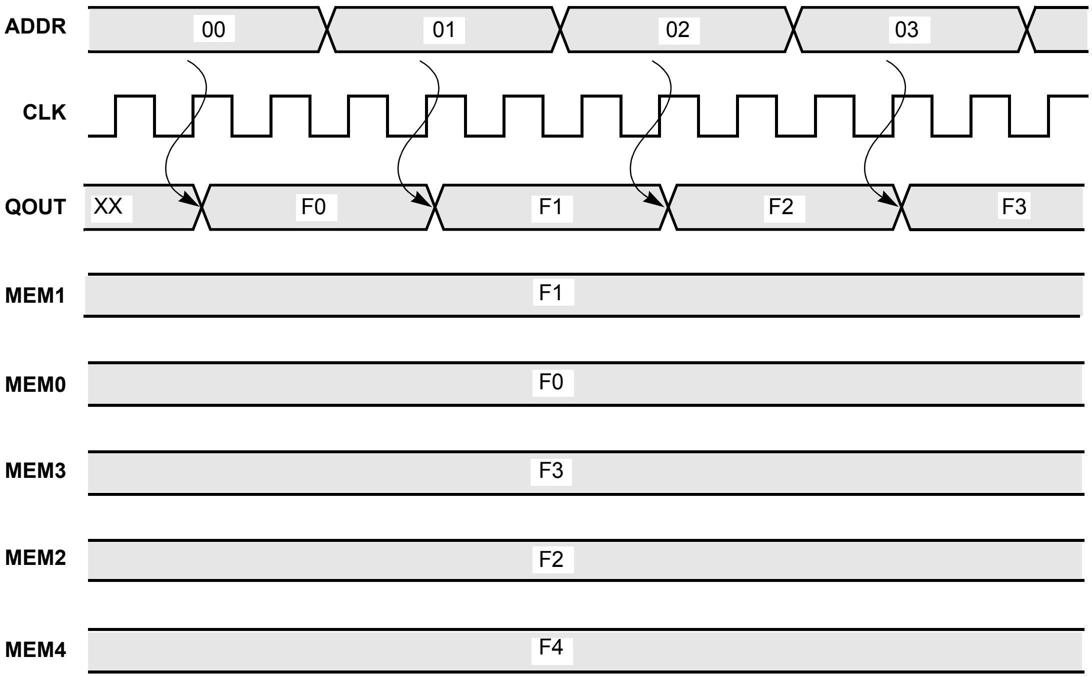
For further information, refer to the following:

- Specifying RAMs with SYNCore, on page 376 of the *User Guide*, for information about using the wizard to generate FIFOs
- Launch SYNCore Command, on page 187 and SYNCore FIFO Wizard, on page 189 for descriptions of the interface

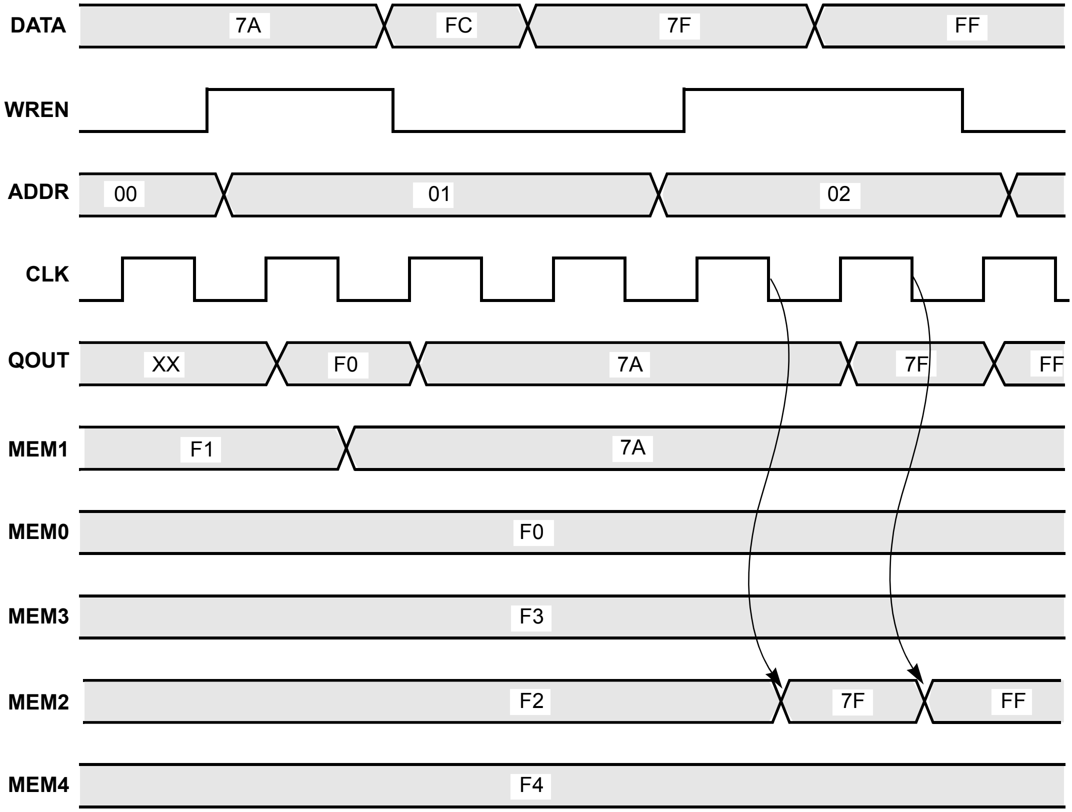
Single-Port Memories

For single-port RAM, it is only necessary to configure Port A. The following diagrams show the read-write timing for single-port memories. See Specifying RAMs with SYNCore, on page 376 in the *User Guide* for a procedure.

Single-Port Read



Single-Port Write



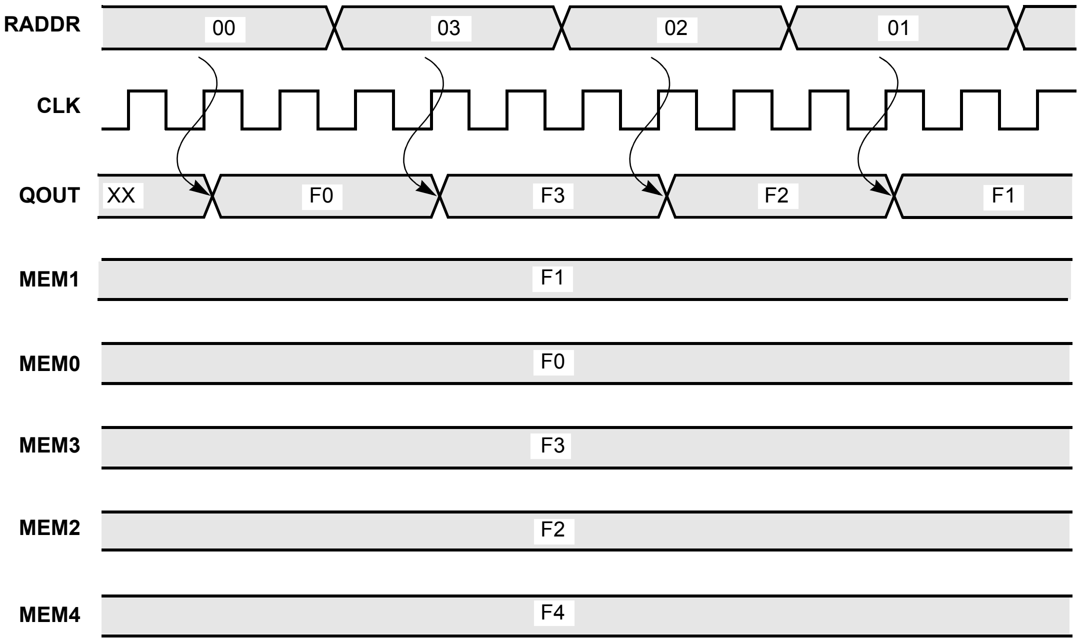
Dual-Port Memories

SYNCore dual-port memory includes the following common configurations:

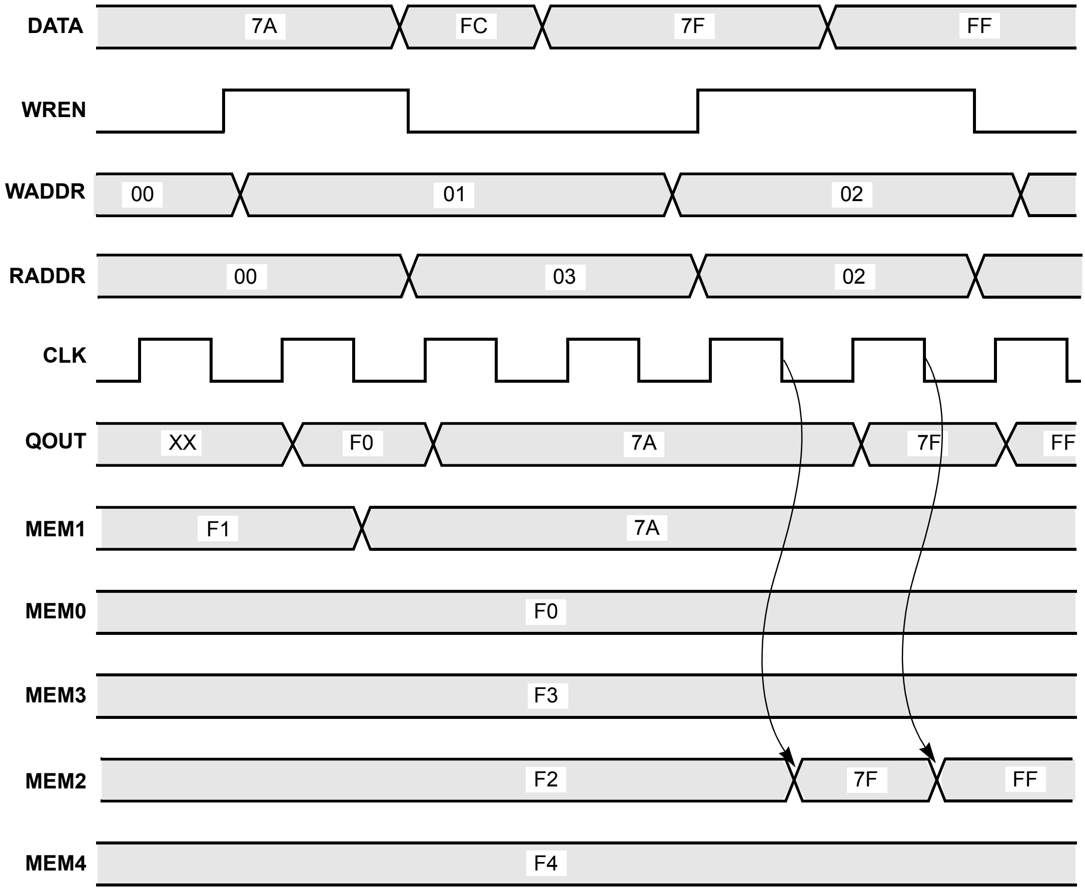
- One read access and one write access
- Two read accesses and one write access
- Two read accesses and two write accesses

The following diagrams show the read-write timing for dual-port memories. See Specifying RAMs with SYNCore, on page 376 in the *User Guide* for a procedure to specify a dual-port RAM with SYNCore.

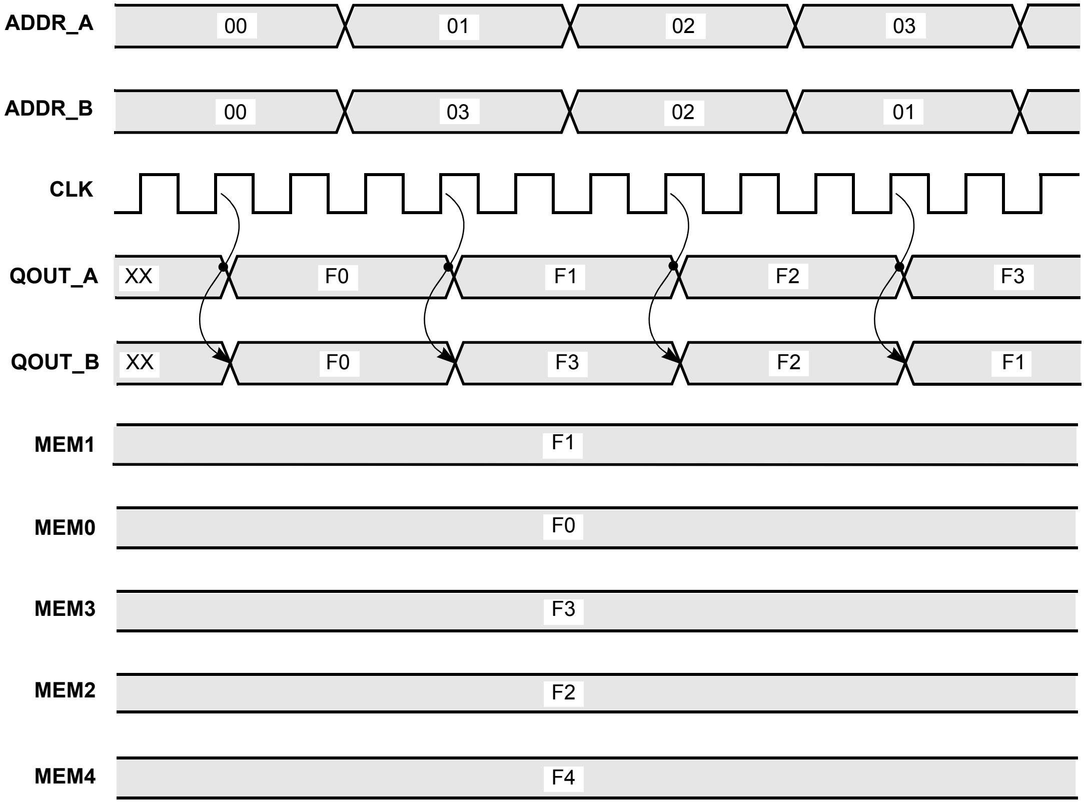
Dual-Port Single Read



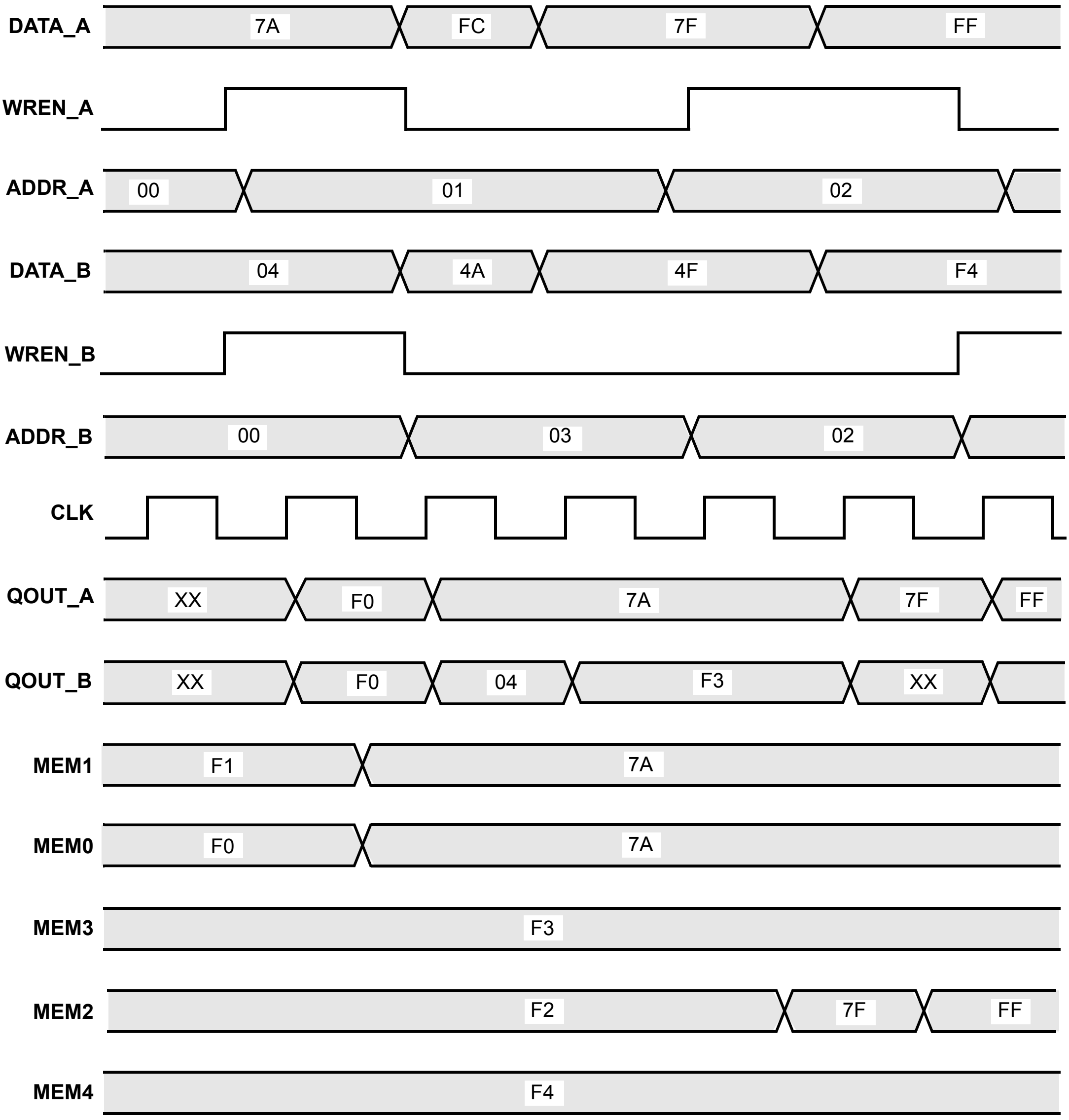
Dual-Port Single Write



Dual-Port Read



Dual-Port Write

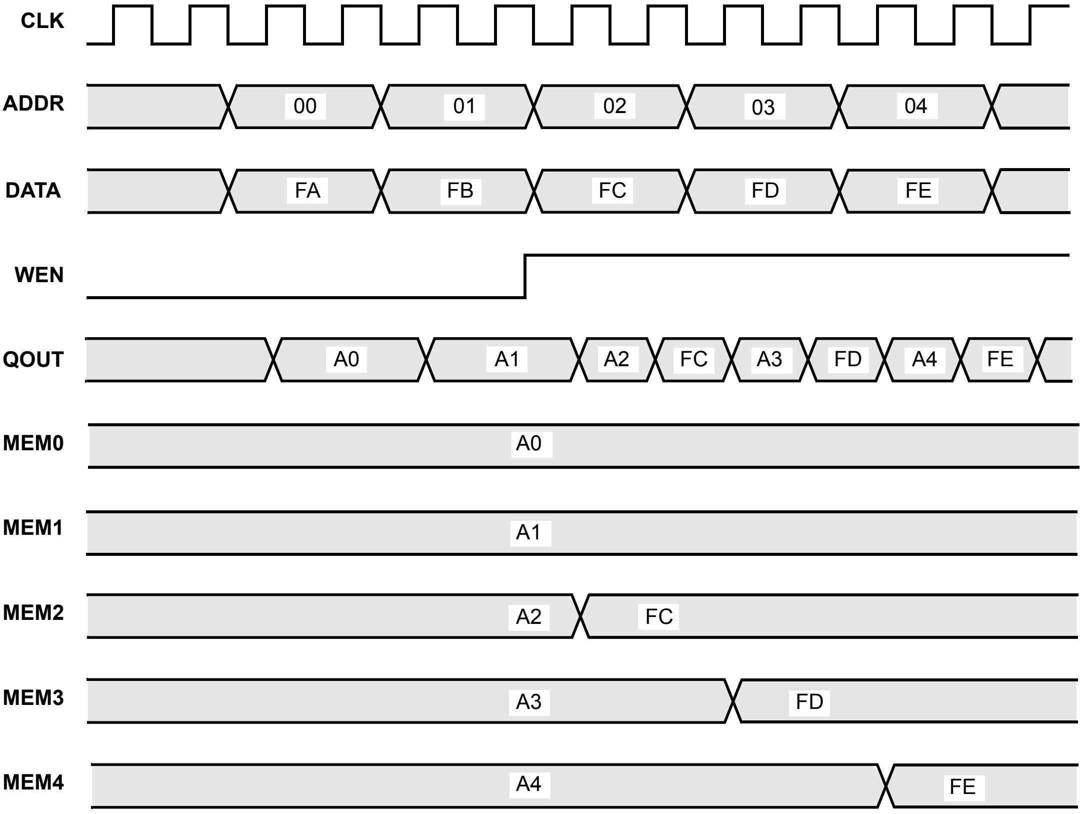


Read/Write Timing Sequences

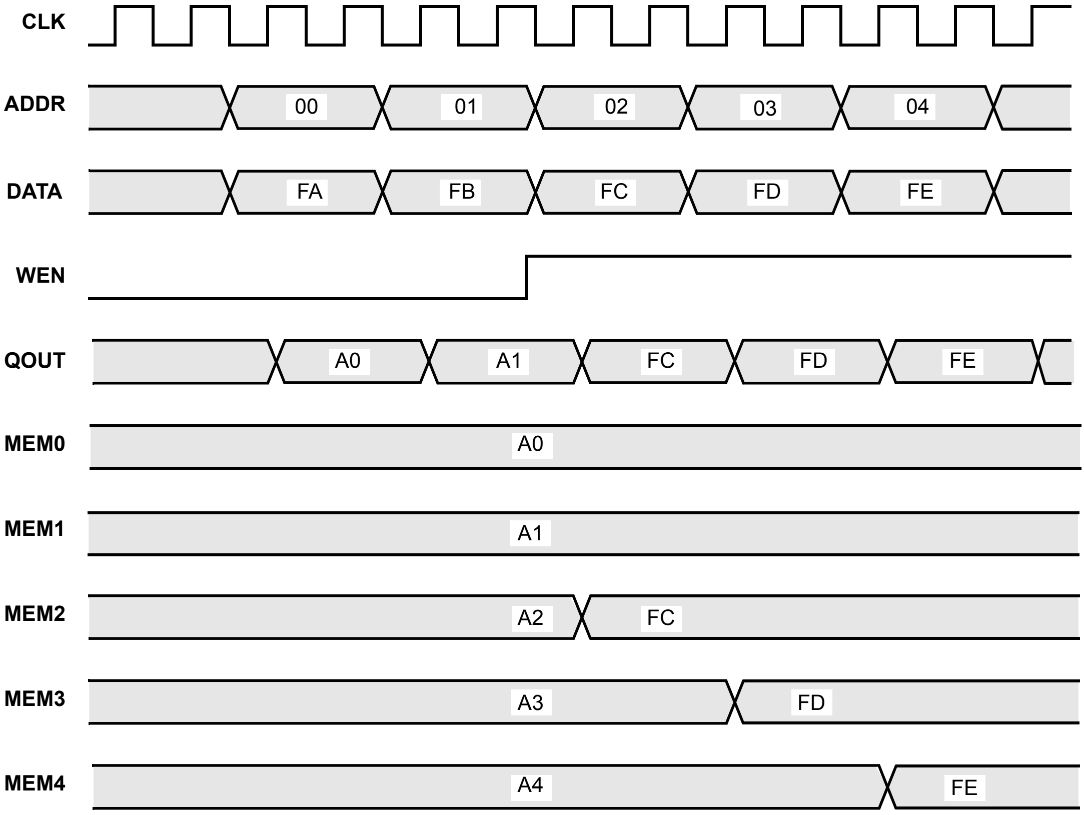
The waveforms in this section describe the behavior of the RAM when both read and write are enabled and the address is the same operation. The waveforms show the behavior when each of the read-write sequences is enabled. The waveforms are merged with the simple waveforms shown in the previous sections. See the following:

- Read Before Write, on page 801
- Write Before Read, on page 802
- No Read on Write, on page 803

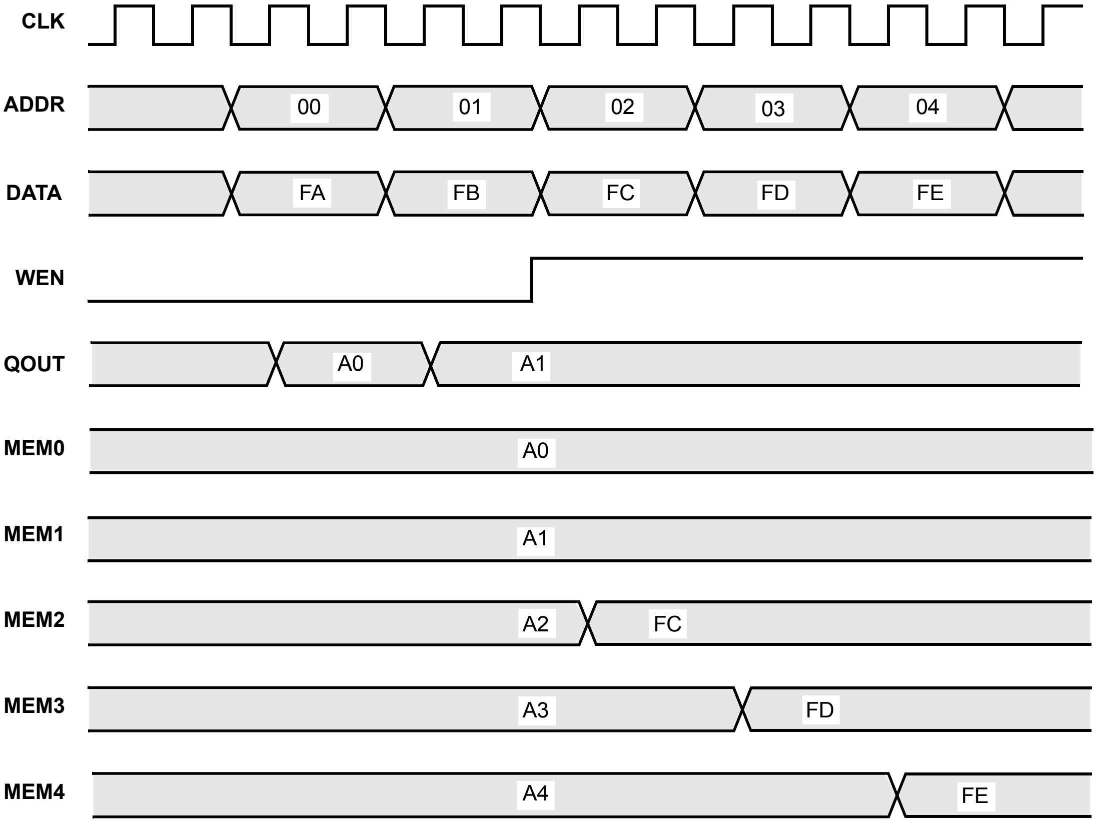
Read Before Write



Write Before Read



No Read on Write



SYNCore Byte-Enable RAM Compiler

The SYNCore byte-enable RAM compiler generates SystemVerilog code describing byte-enabled RAMs. The data width of each byte is calculated by dividing the total data width by the write enable width. The byte-enable RAM compiler supports both single- and dual-port configurations.

This section describes the following:

- Functional Overview, on page
- Write Operation, on page
- Read Operation, on page
- Parameter List, on page

For further information, refer to the following:

- Specifying Byte-Enable RAMs with SYNCore, on page 383 of the user guide for information on using the wizard to generate single- or dual-port RAM configurations.
- SYNCore Byte-Enable RAM Wizard, on page 202 for descriptions of the interface.

Functional Overview

The SYNCore byte-enable RAM component supports bit/byte-enable RAM implementations using blockRAM and distributed memory. For each configuration, design optimizations are made for optimum use of core resources. The timing diagram that follow illustrate the supported signals for byte-enable RAM configurations.

Byte-enable RAM can be configured in both single- and dual-port configurations. In the dual-port configuration, each port is controlled by different clock, enable, and control signals. User configuration controls include selecting the enable level, reset type, and register type for the read data outputs and address inputs.

Reset applies only to the output read data registers; default value of read data on reset can be changed by user while generating core. Reset option is inactive when output read data is not registered.

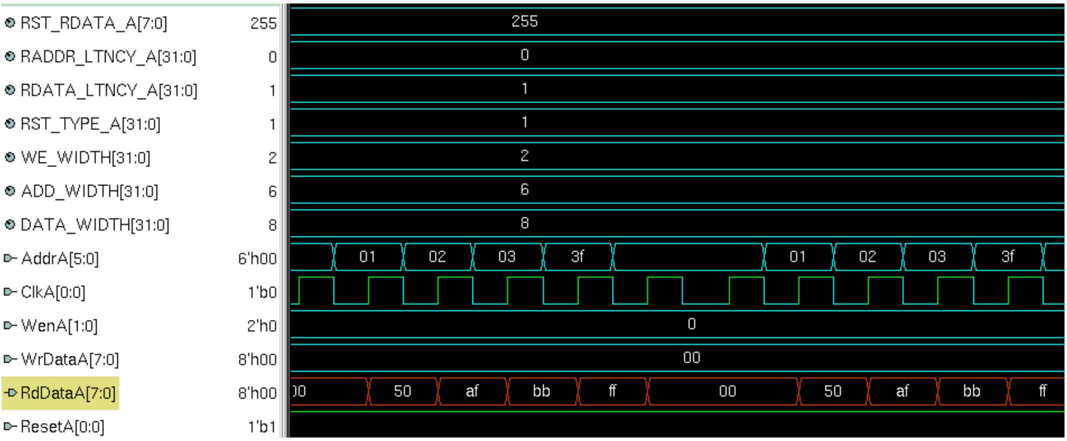
Read/Write Timing Sequences

The waveforms in this section describe the behavior of the byte-enable RAM for both read and write operations.

Read Operation

On each active edge of the clock when there is a change in address, data is valid on the same clock or next clock (depending on latency parameter values for read address and read data ports). Active reset ignores any change in input address, and data and output data are initialized to user-defined values set by parameters RST_RDATA_A and RST_RDATA_B for port A and port B, respectively.

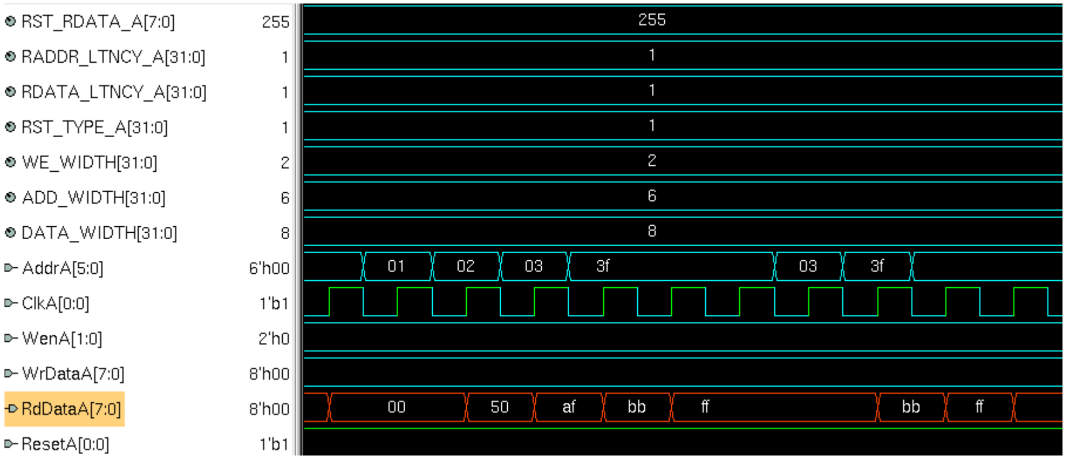
The following waveform shows the read sequence of the byte-enable RAM component with read data registered in single-port mode.



RST_RDATA_A[7:0] 255
RADDR_LTNCY_A[31:0] 0
RDATA_LTNCY_A[31:0] 1
RST_TYPE_A[31:0] 1
WE_WIDTH[31:0] 2
ADD_WIDTH[31:0] 6
DATA_WIDTH[31:0] 8
AddrA[5:0] 6'h00
ClkA[0:0] 1'b0
WenA[1:0] 2'h0
WrDataA[7:0] 8'h00
RdDataA[7:0] 8'h00
ResetA[0:0] 1'b1

As shown in the above waveform, output read data changes on the same clock following the input address changed. When the address changes from 'h00 to 'h01, read data changes to 50 on the same clock, and data will be valid on the next clock edge.

The following waveform shows the read sequence with both the read data and address registered in single-port mode.

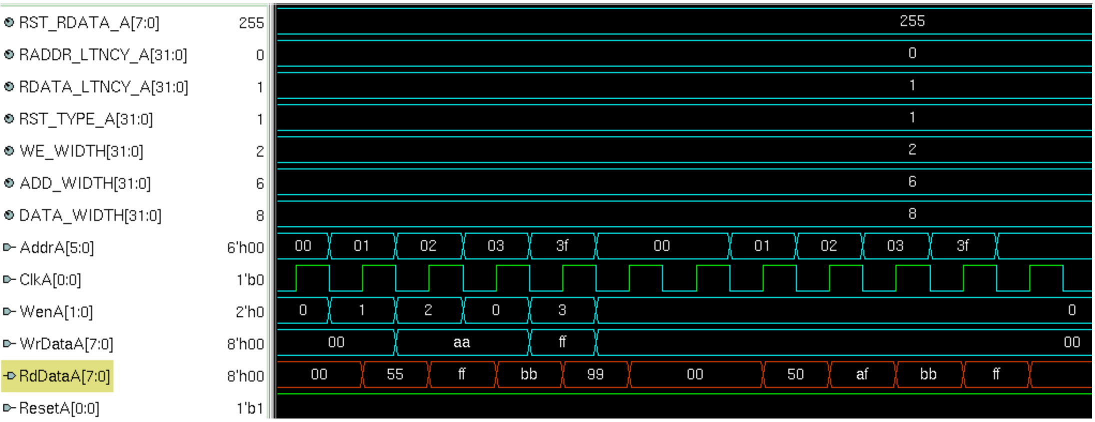


As shown in the above waveform, output read data changes on the next clock edge after the input address changes. When the address changes from 'h00 to 'h01, read data changes to 50 on the next clock, and data is valid on the next clock edge.

Note: The read sequence for dual-port mode is the same as single port; read/write conflicts occurring due to accessing the same location from both ports are the user's responsibility.

Write Operation

The following waveform shows a write sequence with read-after write in single-port mode.



On each active edge of the clock when there is a change in address with an active enable, data is written into memory on the same clock. When enable is not active, any change in address or data is ignored. Active reset ignores any change in input address and data.

The width of the write enable is controlled by the WE_WIDTH parameter. Input data is symmetrically divided and controlled by each write enable. For example, with a data width of 32 and a write enable width of 4, each bit of the write enable controls 8 bits of data (32/4=8). The byte-enable RAM compiler will error for wrong combination data width and write enable values.

The above waveform shows a write sequence with all possible values for write enable followed by a read:

- Value for parameter WE_WIDTH is 2 and DATA_WIDTH is 8 so each write enable controls 4 bits of input data.
- WenA value changes from 1 to 2, 2 to 0, and 0 to 3 which toggles all possible combinations of write enable.

The first sequence of address, write enable changes to perform a write sequence and the data patterns written to memory are 00, aa, ff. The read data pattern reflects the current content of memory before the write.

The second address sequence is a read (WenA is always zero). As shown in the read pattern, only the respective bits of data are written according to the write enable value.

Note: The write sequence for dual-port mode is the same as single port; conflicts occurring due to writing the same location from both ports are the user's responsibility.

Parameter List

The following table lists the file entries corresponding to the byte-enable RAM wizard parameters.

Name	Description	Default Value	Range
ADDR_WIDTH	Bit/byte enable RAM address width	2	multiples of 2
DATA_WIDTH	Data width for input and output data, common to both Port A and Port B	8	2 to 256
WE_WIDTH	Write enable width, common to both Port A and Port B	2	
CONFIG_PORT	Selects single/dual port configuration	1 (single port)	0 = dual-port 1 = single-port
RST_TYPE_A/B	Port A/B reset type selection	1 (synchronous)	0 = no reset 1 = synchronous
RST_RDATA_A/B	Default data value for Port A/B on active reset	All 1's	decimal value
WEN_SENSE_A/B	Port A/B write enable sense	1 (active high)	0 = active low 1 = active high
RADDR_LTNCY_A/B	Optional read address register select Port A/B	1	0 = no latency 1 = one cycle latency
RDATA_LTNCY_A/B	Optional read data register select Port A/B	1	0 = no latency 1 = one cycle latency

SYNCore ROM Compiler

The SYNCore ROM Compiler generates Verilog code for your ROM implementation. This section describes the following:

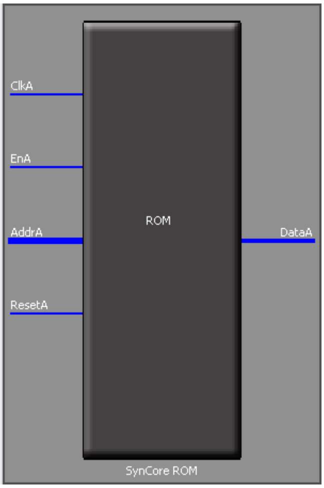
- Functional Overview, on page 809
- Single-Port Read Operation, on page 811
- Dual-Port Read Operation, on page 812
- Parameter List, on page 812
- Clock Latency, on page 814

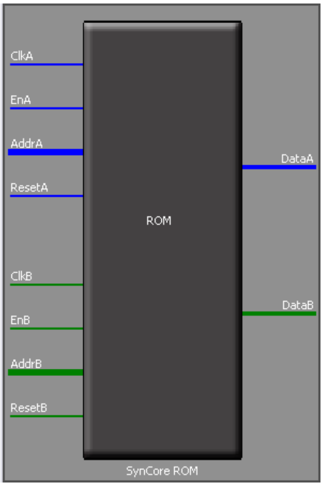
For further information, refer to the following:

- Specifying ROMs with SYNCore, on page 389 of the *User Guide*, for information about using the wizard to generate ROMs
- Launch SYNCore Command, on page 187 and SYNCore ROM Wizard, on page 205 for descriptions of the interface

Functional Overview

The SYNCore ROM component supports ROM implementations using block ROM or logic memory. For each configuration, design optimizations are made for optimum usage of core resources. Both single- and dual-port memory configurations are supported. Single-port ROM allows read access to memory through a single port, and dual-port ROM allows read access to memory through two ports. The following figure illustrates the supported signals for both configurations.





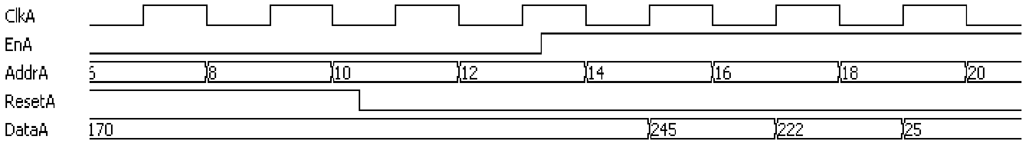
In the single-port (Port A) configuration, signals are synchronized to ClkA; ResetA can be synchronous or asynchronous depending on parameter selection. The read address (AddrA) and/or data output (DataA) can be registered to increase memory performance and improve timing. Both the read address and data output are subject to clock latency based on the ROM configuration (see Clock Latency, on page 814). In the dual-port configuration, all Port A signals are synchronized to ClkA, and all PortB signals are synchronized to ClkB. ResetA and ResetB can be synchronous or asynchronous depending on parameter selection, and both data outputs can be registered and are subject to the same clock latencies. Registering the data output is recommended.

Note: When the data output is unregistered, the data is immediately set to its predefined reset value concurrent with an active reset signal.

Single-Port Read Operation

For single-port ROM, it is only necessary to configure Port A (see Specifying ROMs with SYNCore, on page 389 in the *User Guide*). The following diagram shows the read timing for a single-port ROM.

On every active edge of the clock when there is a change in address with an active enable, data will be valid on the same clock or next clock (depending on latency parameter values). When enable is inactive, any address change is ignored, and the data port maintains the last active read value. An active reset ignores any change in input address and forces the output data to its predefined initialization value. The following waveform shows the functional behavior of control signals in single-port mode.

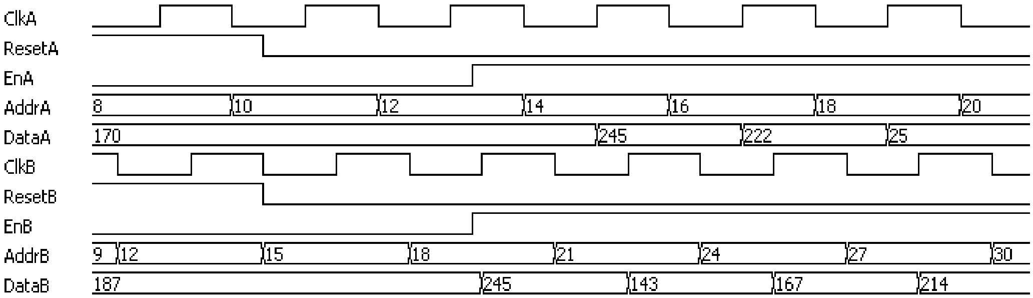


When reset is active, the output data holds the initialization value (i.e., 255). When reset goes inactive (and enable is active), data is read form the addressed location of ROM. Reset has priority over enable and always sets the output to the predefined initialization value. When both enable and reset are inactive, the output holds its previous read value.

Note: In the above timing diagram, reset is synchronous. Clock latency varies according to the implementation and parameters as described in Clock Latency, on page 814.

Dual-Port Read Operation

Dual-port ROMs allow read access to memory through two ports. For dual-port ROM, both port A and port B must be configured (see Specifying ROMs with SYNCore, on page 389 in the *User Guide*). The following diagram shows the read timing for a dual-port ROM.



When either reset is active, the corresponding output data holds the initialization value (i.e., 255). When a reset goes inactive (and its enable is active), data is read form the addressed location of ROM. Reset has priority over enable and always sets the output to the predefined initialization value. When both enable and reset are inactive, the output holds its previous read value.

Note: In the above timing diagram, reset is synchronous. Clock latency varies according to the implementation and parameters as described in Clock Latency, on page 814.

Parameter List

The following table lists the file entries corresponding to the ROM wizard parameters.

Name	Description	Default Value	Range
ADD_WIDTH	ROM address width value. Default value is 10	10	--
DATA_WIDTH	Read Data width, common to both Port A and Port B	8	2 to 256

CONFIG_PORT	Parameter to select Single/Dual configuration	dual (Dual Port)	dual (Dual), single (Single).
RST_TYPE_A	Port A reset type selection (synchronous, asynchronous)	1 - asynchronous	1(asyn), 0 (sync)
RST_TYPE_B	Port B reset type selection (synchronous, asynchronous)	1 - asynchronous	1 (asyn), 0 (sync)
RST_DATA_A	Default data value for Port A on active Reset	‘1’ for all data bits	0 – 2^DATA_WIDTH - 1
RST_DATA_B	Default data value for Port A on active Reset	‘1’ for all data bits	0 – 2^DATA_WIDTH - 1
EN_SENSE_A	Port A enable sense	1 – active high	0 - active low, 1- active high
EN_SENSE_B	Port B enable sense	1 – active high	0 - active low, 1- active high
ADDR_LTNCY_A	Optional address register select Port A	1- address registered	1(reg), 0(no reg)
ADDR_LTNCY_B	Optional address register select Port B	1- address registered	1(reg), 0(no reg)
DATA_LTNCY_A	Optional data register select Port A	1- data registered	1(reg), 0(no reg)
DATA_LTNCY_B	Optional data register select Port B	1- data registered	1(reg), 0(no reg)
INIT_FILE	Initial values file name	init.txt	--

Clock Latency

Clock latency varies with both the implementation and latency parameter values according to the following table. Note that the table reflects the values for Port A – the same values apply for Port B in dual-port configurations.

Implementation Type/Target	Parameter Value	Latency

block_rom	DATA_LTNCY_A = 0
	ADDR_LTNCY_A = 1
		1 ClkA cycle

	DATA_LTNCY_A = 1
	ADDR_LTNCY_A = 0
		1 ClkA cycle

	DATA_LTNCY_A = 1
	ADDR_LTNCY_A = 1
		2 ClkA cycles

logic	DATA_LTNCY_A = 0
	ADDR_LTNCY_A = 0
		0 ClkA cycles

	DATA_LTNCY_A = 0
	ADDR_LTNCY_A = 1
		1 ClkA cycle

	DATA_LTNCY_A = 1
	ADDR_LTNCY_A = 0
		1 ClkA cycle

	DATA_LTNCY_A = 1
	ADDR_LTNCY_A = 1
		2 ClkA cycles

SYNCore Adder/Subtractor Compiler

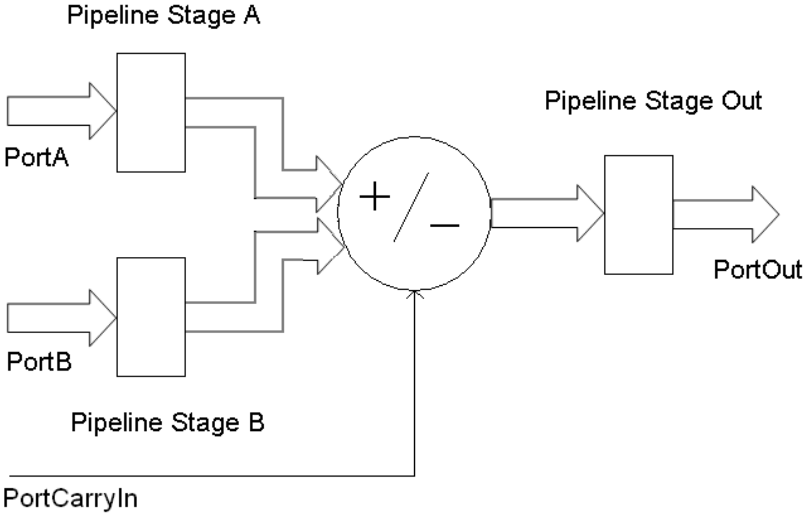
The SYNCore adder/subtractor compiler generates Verilog code for a parametrizable, pipelined adder/subtractor. This section describes the functionality of this block in detail.

Functional Description

The adder/subtractor has a single clock that controls the entire pipeline stages (if used) of the adder/subtractor.

As its name implies, this block just adds/subtracts the inputs and provides the output result. One of the inputs can be configured as a constant. The data inputs and outputs of the adder/subtractor can be pipelined; the pipeline stages can be 0 or 1, and can be configured individually. The individual pipeline stage registers include their own reset and enable ports.

The reset to all of the pipeline registers can be configured either as synchronous or asynchronous using the RESET_TYPE parameter. The reset type of the pipeline registers cannot be configured individually.



SYNCore adder/subtractor has ADD_N_SUB parameter, which can take three values ADD, SUB, or DYNAMIC. Based on this parameter value, the adder/subtractor can be configured as follows.

- Adder
- Subtractor
- Dynamic Adder and Subtractor

Adder

Based on the parameter CONSTANT_PORT, the adder can be configured in two ways.

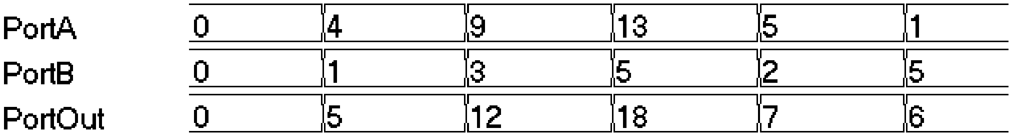
- CONSTANT_PORT='0' – adder with two input ports (port A and port B)
- CONSTANT_PORT='1' – adder with one constant port

Adder with Two Input Ports (Port A and Port B)

In this mode, port A and port B values are added. Optional pipeline stages can also be inserted at port A, port B or at both port A and port B. Optionally, pipeline stages can also be added at the output port. Depending on pipeline stages, a number of the adder configurations are given below.

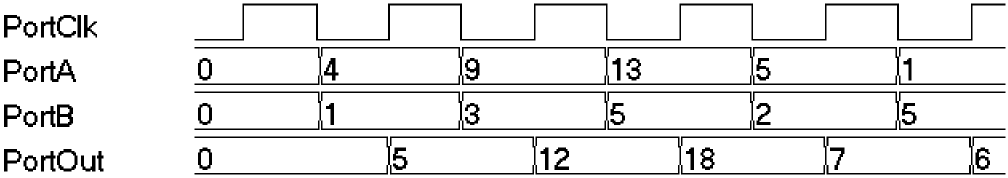
Adder with No Pipeline Stages – In this mode, the port A and port B inputs are added. The adder is purely combinational, and the output changes immediately with respect to the inputs.

Parameters: PORTA_PIPELINE_STAGE= ‘0’



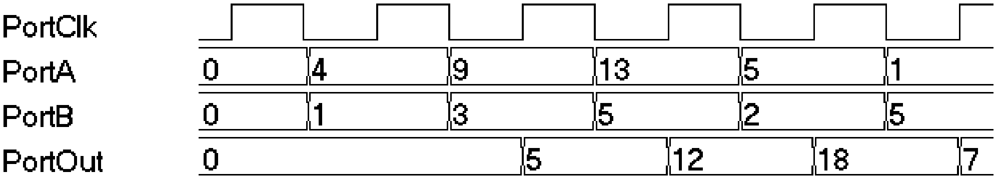
Adder with Pipeline Stages at Input Only – In this mode, the port A and port B inputs are pipelined and added. Because there is no pipeline stage at the output, the result is valid at each rising edge of the clock.

Parameters: PORTA_PIPELINE_STAGE= ‘1’
PORTB_PIPELINE_STAGE= ‘1’
PORTOUT_PIPELINE_STAGE= ‘0’



Adder with Pipeline Stages at Input and Output – In this mode, the port A and port B inputs are pipelined and added, and the result is pipelined. The result is valid only on the second rising edge of the clock.

Parameters: PORTA_PIPELINE_STAGE= ‘1’
PORTB_PIPELINE_STAGE= ‘1’
PORTOUT_PIPELINE_STAGE= ‘1’

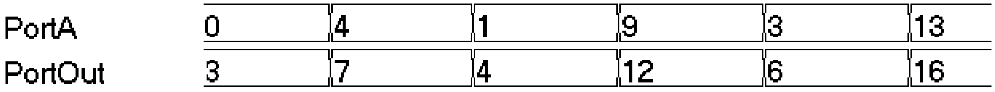


Adder with a Port Constant

In this mode, port A is added with a constant value (the constant value can be passed though the parameter CONSTANT_VALUE). Optional pipeline stages can also be inserted at port A, Optionally, pipeline stages can also be added at the output port. Depending on the pipeline stages, a number of the adder configurations are given below (here CONSTANT_VALUE=’3’)

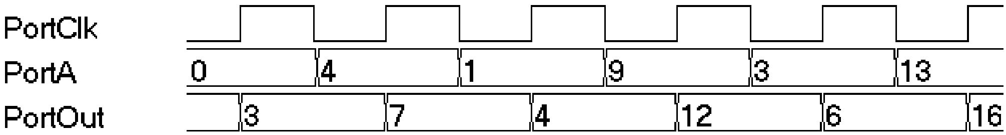
Adder with No Pipeline Stages – In this mode, input port A is added with a constant value. The adder is purely combinational, and the output changes immediately with respect to the input.

Parameters: PORTA_PIPELINE_STAGE= '0'
PORTOUT_PIPELINE_STAGE= '0'



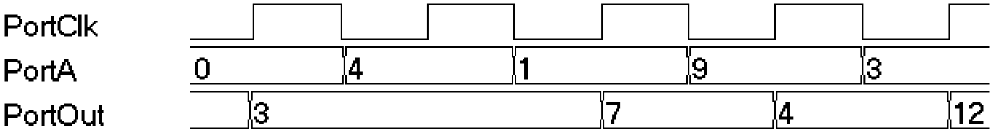
Adder with Pipeline Stage at Input Only – In this mode, input port A is pipelined and added with a constant value. Because there is no pipeline stage at the output, the result is valid at each rising edge of the clock.

Parameters: PORTA_PIPELINE_STAGE= '1'
PORTOUT_PIPELINE_STAGE= '0'



Adder with Pipeline Stages at Input and Output – In this mode, input port A is pipelined and added with a constant value, and the result is pipelined. The result is valid only on the second rising edge of the clock.

Parameters: PORTA_PIPELINE_STAGE= '1'
PORTOUT_PIPELINE_STAGE= '1'



Subtractor

Based on the parameter CONSTANT_PORT, the subtractor can be configure in two ways.

CONSTANT_PORT='0' – subtractor with two input ports (port A and port B)

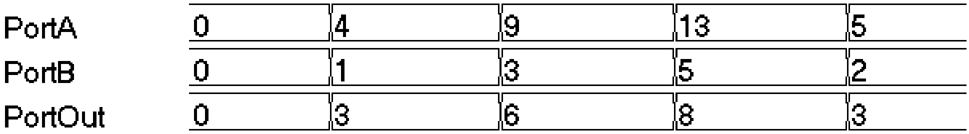
CONSTANT_PORT='1' – subtractor with one constant port

Subtractor with Two Input Ports (Port A and Port B)

In this mode, port B is subtracted from port A. Optional pipeline stages can also be inserted at port A, port B, or both ports. Optionally, pipeline stages can also be added at the output port. Depending on the pipeline stages, a number of the subtractor configurations are given below.

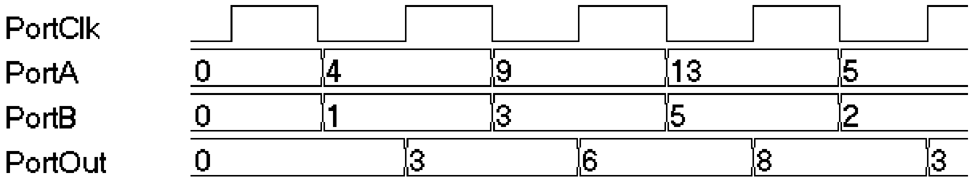
Subtractor with No Pipeline Stages – In this mode, input port B is subtracted from port A, and the subtractor is purely combinational. The output changes immediately with respect to the inputs.

Parameters: PORTA_PIPELINE_STAGE= ‘0’
PORTB_PIPELINE_STAGE= ‘0’
PORTOUT_PIPELINE_STAGE= ‘0’



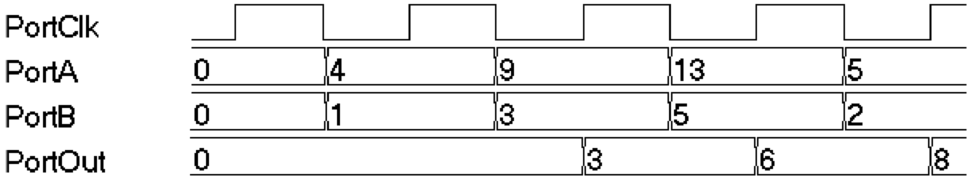
Subtractor with Pipeline Stages at Input Only – In this mode, input port B and input PortA are pipelined and then subtracted. Because there is no pipeline stage at the output, the result is valid at each rising edge of the clock.

Parameters: PORTA_PIPELINE_STAGE= ‘1’
PORTB_PIPELINE_STAGE= ‘1’
PORTOUT_PIPELINE_STAGE= ‘0’



Subtractor with Pipeline Stages at Input and Output – In this mode, input PortA and PortB are pipelined and then subtracted, and the result is pipelined. The result is valid only at the second rising edge of the clock.

Parameters: PORTA_PIPELINE_STAGE= ‘1’
PORTB_PIPELINE_STAGE= ‘1’
PORTOUT_PIPELINE_STAGE= ‘1’

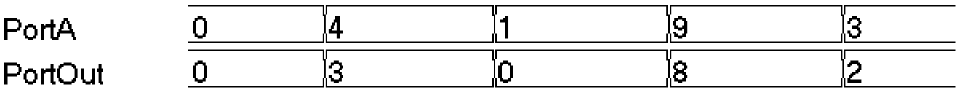


Subtractor with a Port Constant

In this mode, a constant value is subtracted from port A (the constant value can be passed though the parameter CONSTANT_VALUE). Optional pipeline stages can also be inserted at port A, Optionally, pipeline stages can also be added at the output port. Depending on pipeline stages, a number of the subtractor configurations are given below (here CONSTANT_VALUE='1').

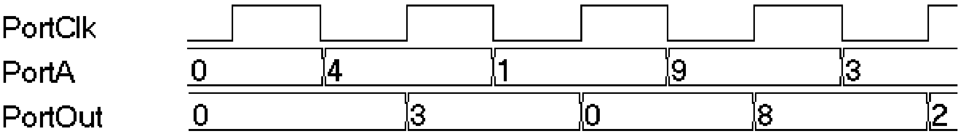
Subtractor with No Pipeline Stages – In this mode, a constant value is subtracted from port A. The subtractor is purely combinational, and the output changes immediately with respect to the input.

Parameters: PORTA_PIPELINE_STAGE= '0'
PORTOUT_PIPELINE_STAGE= '0'



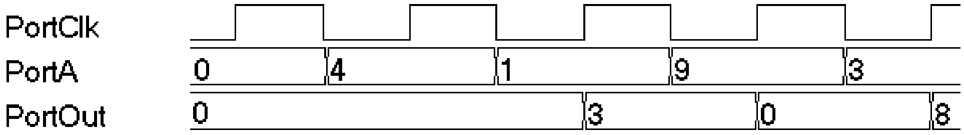
Subtractor with Pipeline Stages at Input Only – In this mode, a constant value is subtracted from pipelined input port A. Because there is no pipeline stage at the output, the output is valid at each rising edge of the clock.

Parameters: PORTA_PIPELINE_STAGE= '1'
PORTOUT_PIPELINE_STAGE= '0'



Subtractor with Pipeline Stages at Input and Output – In this mode, a constant value is subtracted from pipelined port A, and the output is pipelined. The result is valid only at the second rising edge of the clock.

Parameters: PORTA_PIPELINE_STAGE= ‘1’
PORTOUT_PIPELINE_STAGE= ‘1’



Dynamic Adder/Subtractor

In dynamic adder/subtractor mode, port PortADDnSUB controls adder/subtractor operation.

PortADDnSUB='0' – adder operation

PortADDnSUB='1' – subtractor operation

Based on the parameter CONSTANT_PORT the dynamic adder/subtractor can be configured in one of two ways:

CONSTANT_PORT='0' – dynamic adder/subtractor with two input ports

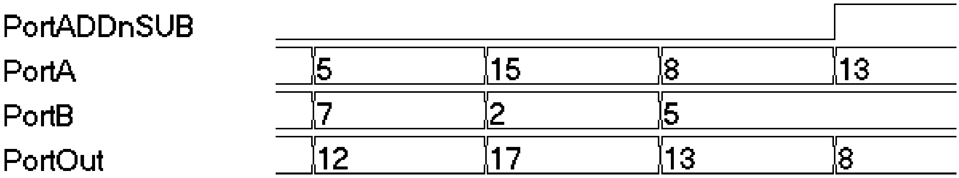
CONSTANT_PORT='1' – dynamic adder/subtractor with one constant port

Dynamic Adder/Subtractor with Two Input Ports (Port A and Port B)

In this mode, the addition and subtraction is dynamic based on the value of input port PortADDnSUB. Optional pipeline stages can also be inserted at Port A, Port B, or both Port A and Port B. Optionally, pipeline stages can also be added at the output port. Depending on pipeline stages, some of the dynamic adder/subtractor configurations are given below.

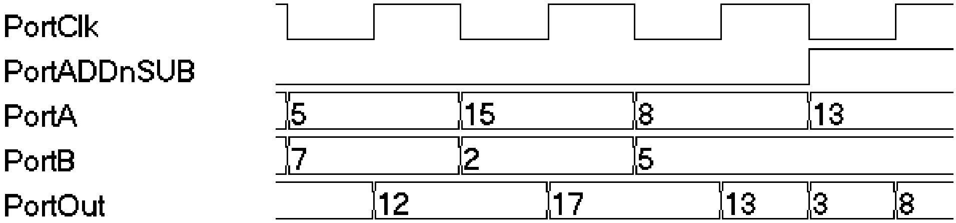
Dynamic Adder/Subtractor with No Pipeline Registers – In this mode, the dynamic adder/subtractor is a purely combinational, and output changes immediately with respect to the inputs.

Parameters: PORTA_PIPELINE_STAGE= ‘0’
PORTB_PIPELINE_STAGE= ‘0’
PORTOUT_PIPELINE_STAGE= ‘0’



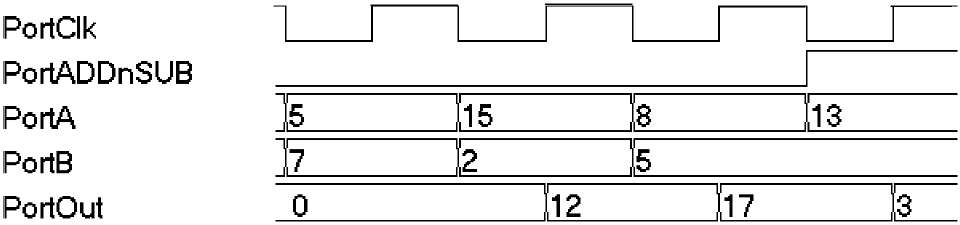
Dynamic Adder/Subtractor with Pipeline Stages at Input Only – In this mode, input port A and port B are pipelined and then added/subtracted based on the value of port PortADDnSUB. Because there is no pipeline stage at the output port, the result immediately changes with respect to the PortADDnSUB signal.

Parameters: PORTA_PIPELINE_STAGE= ‘1’
PORTB_PIPELINE_STAGE= ‘1’
PORTOUT_PIPELINE_STAGE= ‘0’



Dynamic Adder/Subtractor with Pipeline Stages at Input and Output – In this mode, input port A and port B are pipelined and then added/subtracted based on the value of port PortADDnSUB. Because the output port is pipelined, the result is valid only on the second rising edge of the clock.

Parameters: PORTA_PIPELINE_STAGE= ‘1’
PORTB_PIPELINE_STAGE= ‘1’
PORTOUT_PIPELINE_STAGE= ‘1’

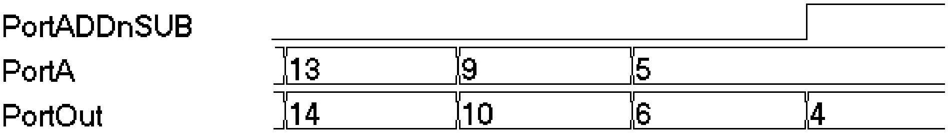


Dynamic Adder/Subtractor with a Port Constant

In this mode, a constant value is either added or subtracted from port A based on input port value PortADDnSUB (the constant value can be passed though the parameter CONSTANT_VALUE). Optional pipeline stages can also be inserted at port A, Optionally, pipeline stages can also be added at the output port. Depending on the pipeline stages, a number of the dynamic adder/subtractor configurations are given below (here CONSTANT_VALUE=’1’).

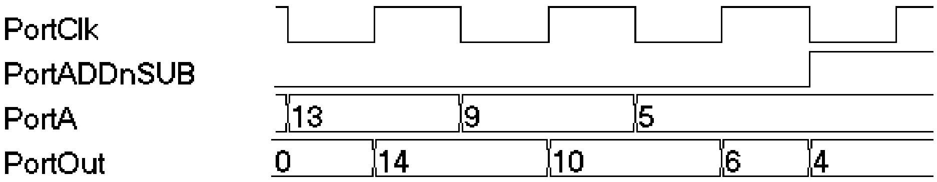
Dynamic Adder/Subtractor with No Pipeline Registers – In this mode, dynamic adder/subtractor is a purely combinational, and the output change immediately with respect to the input.

Parameters: PORTA_PIPELINE_STAGE= ‘0’
PORTOUT_PIPELINE_STAGE= ‘0’



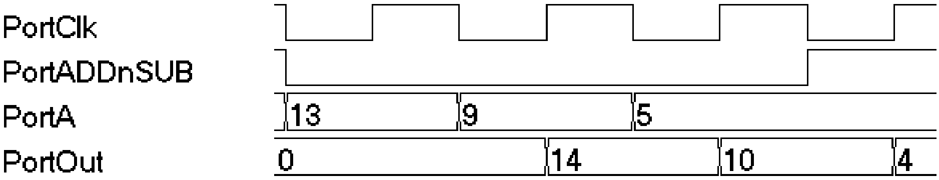
Dynamic Adder/Subtractor with Pipeline Stages at Input Only – In this mode, a constant value is either added or subtracted from the pipelined version of port A based on the value of port PortADDnSUB. Because there is no pipeline stage on the output port, the result changes immediately with respect to the PortADDnSUB signal.

Parameters: PORTA_PIPELINE_STAGE= ‘1’
PORTOUT_PIPELINE_STAGE= ‘0’



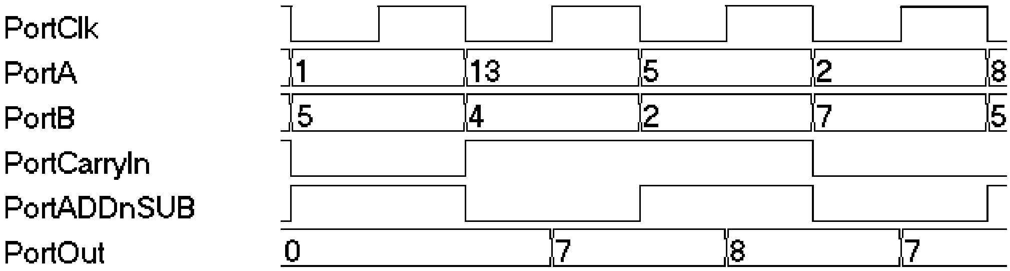
Dynamic Adder/Subtractor with Pipeline Stages at Input and Output – In this mode, a constant value is either added or subtracted from the pipelined version of port A based on the value of port PortADDnSUB. Because the output port is pipelined, the result is valid only on the second rising edge of the clock.

Parameters: PORTA_PIPELINE_STAGE= ‘1’
PORTOUT_PIPELINE_STAGE= ‘1’



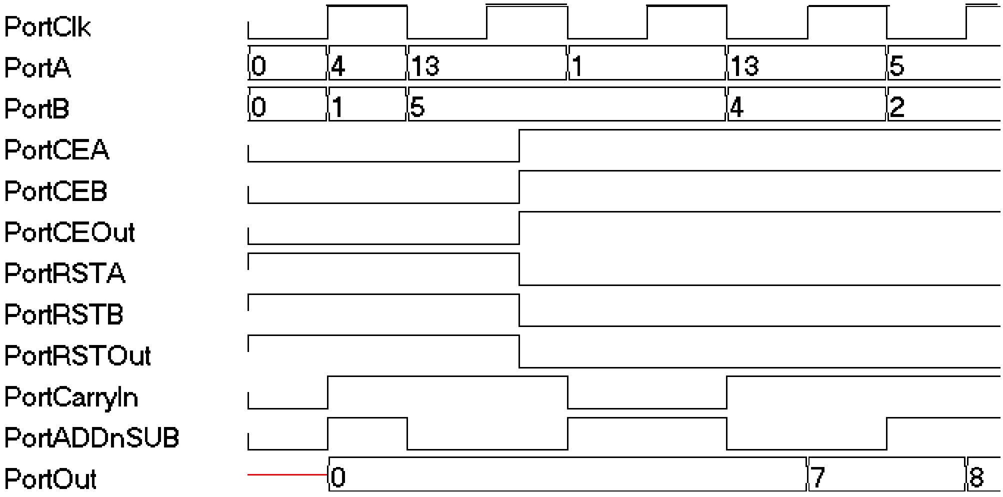
Dynamic Adder/Subtractor with Carry Input

The following waveform shows the behavior of the dynamic adder/subtractor with a carry input (the carry input is assumed to be 0).



Dynamic Adder/Subtractor with Complete Control Signals

The following waveform shows the complete signal set for the dynamic adder/subtractor. The enable and reset signals are always present in all of the previous cases.



SYNCore Counter Compiler

The SYNCore counter compiler generates Verilog code for your up, down, and dynamic (up/down) counter implementation. This section describes the following:

- Functional Overview, on page 827
- UP Counter Operation, on page 828
- Down Counter Operation, on page 828
- Dynamic Counter Operation, on page 829

For further information, refer to the following:

- Specifying Counters with SYNCore, on page 401 of the *User Guide*, for information about using the wizard to generate a counter core.
- Launch SYNCore Command, on page 187 and SYNCore Counter Wizard, on page 213 for descriptions of the interface and generating the core.

Functional Overview

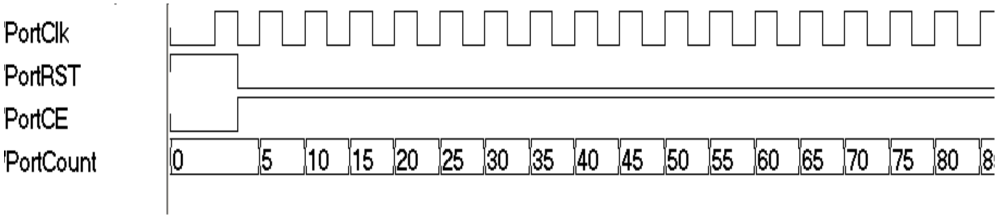
The SYNCore counter component supports up, down, and dynamic (up/down) counter implementations using DSP blocks or logic elements. For each configuration, design optimizations are made for optimum use of core resources.

As its name implies, the COUNTER block counts up (increments) or down (decrements) by a step value and provides an output result. You can load a constant or a variable as an intermediate value or base for the counter. Reset to the counter on the PortRST input is active high and can be can be configured either as synchronous or asynchronous using the RESET_TYPE parameter. Count enable on the PortCE input must be value high to enable the counter to increment or decrement.

UP Counter Operation

In this mode, the counter is incremented by the step value defined by the STEP parameter. When reset is asserted (when PostRST is active high), the counter output is reset to 0. After the assertion of PortCE, the counter starts counting upwards coincident with the rising edge of the clock. The following waveform is with a constant STEP value of 5 and no load value.

Parameters: MODE= 'Up'
LOAD= '0'

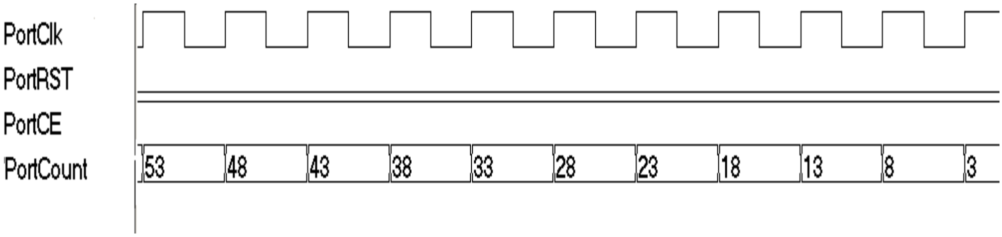


Note: Counter core can be configured to use a constant or dynamic load value in Up Counter mode (for the counter to load the PortLoadValue, PortCE must be active). This functionality is explained in Dynamic Counter Operation, on page 829.

Down Counter Operation

In this mode, the counter is decremented by the step value defined by the STEP parameter. When reset is asserted (when PostRST is active high), the counter output is reset to 0. After the assertion of PortCE, the counter starts counting downwards coincident with the rising edge of the clock. The following waveform is with a constant STEP value of 5 and no load value.

Parameters: MODE= 'Down'
LOAD= '0'



Note: Counter core can be configured to use a constant or dynamic load value in Down Counter mode (for the counter to load the PortLoadValue, PortCE must be active). This functionality is explained in Dynamic Counter Operation, on page 829.

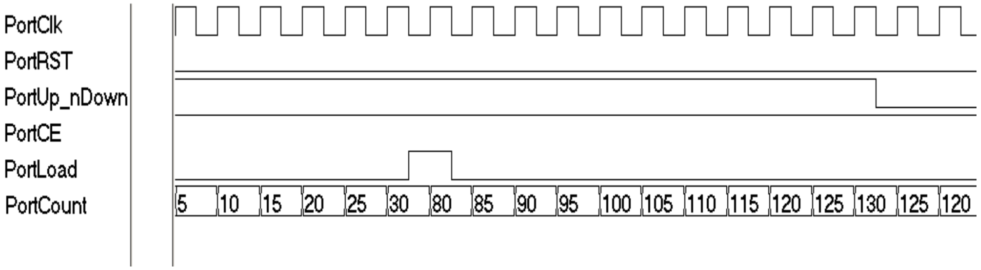
Dynamic Counter Operation

In this mode, the counter is incremented or decremented by the step value defined by the STEP parameter; the count direction (up or down) is controlled by the PortUp_nDown input (1 = up, 0 = down).

Dynamic Up/Down Counters with Constant Load Value*

On de-assertion of PortRST, the counter starts counting up or down based on the PortUp_nDown input value. The following waveform is with STEP value of 5 and a LOAD_VALUE of 80. When PortLoad is asserted, the counter loads the constant load value on the next active edge of clock and resumes counting in the specified direction.

Parameters:	MODE= ‘Dynamic’
	LOAD= ‘1’



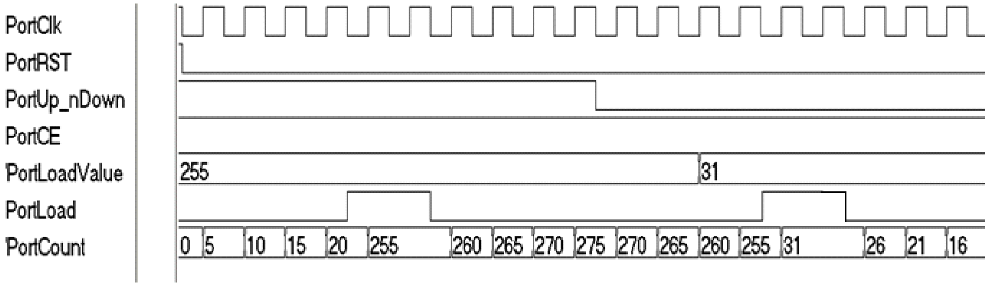
Note: *For counter to load the PortLoadValue, PortCE must be active.

Dynamic Up/Down Counters with Dynamic Load Value*

On de-assertion of PortRST, the counter starts counting up or down based on the PortUp_nDown input value. The following waveform is with STEP value of 5 and a LOAD_VALUE of 80. When PortLoad is asserted, the counter loads the constant load value on the next active edge of clock and resumes counting in the specified direction.

In this mode, the counter counts up or down based on the PortUp_nDown input value. On the assertion of PortLoad, the counter loads a new PortLoadValue and resumes up/down counting on the next active clock edge. In this example, a variable PortLoadValue of 8 is used with a counter STEP value of 5.

Parameters: MODE= ‘Dynamic’
LOAD= ‘2’



Note: * For counter to load the PortLoadValue, PortCE should be active.

CHAPTER 9

Timing Constraint Syntax

The following describe Tcl equivalents for the timing constraints you specify in the SCOPE editor or manually in a constraint file.

FPGA Timing Constraints

The FPGA synthesis tools support FPGA timing constraints for a subset of the clock definition, I/O delay, and timing exception constraints.

For more information about using FPGA timing constraints with your project, see Using the SCOPE Editor, on page 52 in the *User Guide*.

The remainder of this section describes the constraint file syntax for the following FPGA timing constraints in the FPGA synthesis tools. For constraint file syntax for the legacy timing constraints, see Synplify-Style Timing Constraints (Legacy), on page 865.

- create_clock
- create_generated_clock
- reset_path
- set_clock_groups
- set_clock_latency
- set_clock_route_delay
- set_clock_uncertainty
- set_false_path
- set_input_delay
- set_max_delay
- set_multicycle_path
- set_output_delay
- set_reg_input_delay
- set_reg_output_delay

For information on the supported design constraints, see Chapter 10, *FPGA Design Constraint Syntax*.

create_clock

Creates a clock object and defines its waveform in the current design.

Syntax

The supported syntax for the create_clock constraint is:

```
create_clock
    -name clockName [-add] {objectList} |
        -name clockName [-add] [{objectList}] |
        [-name clockName [-add]] {objectList}
    -period value
    [-waveform {riseValue fallValue}]
    [-disable]
    [-comment commentString]
```

Arguments

-name *clockName*	Specifies the name for the clock being created, enclosed in quotation marks or curly braces. If this option is not used, the clock gets the name of the first clock source specified in the *objectList* option. If you do not specify the *objectList* option, you must use the -name option, which creates a virtual clock not associated with a port, pin, or net. You can use both the -name and *objectList* options to give the clock a more descriptive name than the first source pin, port, or net. If you specify the -add option, you must use the -name option and the clocks with the same source must have different names.
-add	Specifies whether to add this clock to the existing clock or to overwrite it. Use this option when multiple clocks must be specified on the same source for simultaneous analysis with different clock waveforms. When you specify this option, you must also use the -name option.
-period *value*	Specifies the clock period in nanoseconds. This is the minimum time over which the clock waveform repeats. The *value* type must be greater than zero.

-waveform *riseValue fallValue*	Specifies the rise and fall edge times for the clock waveforms of the clock in nanoseconds, over an entire clock period. The first time is a rising transition, typically the first rising transition after time zero. There must be two edges, and they are assumed to be rise followed by fall. The edges must be monotonically increasing. If you do not specify this option, a default waveform is assumed, which has a rise edge of 0.0 and a fall edge of *periodValue*/2.
objectList	Clocks can be defined on the following objects: pins, ports, and nets The FPGA synthesis tools support nets and instances, where instances have only one output.
-disable	Disables the constraint.
-comment *textString*	Allows the command to accept a comment string. The tool honors the annotation and preserves it with the object so that the exact string is written out when the constraint is written out. The comment remains intact through the synthesis, place-and-route, and timing-analysis flows.

create_generated_clock

Creates a generated clock object.

Syntax

The supported syntax for the create_generated_clock constraint is:

```
create_generated_clock
    -name clockName [-add]] | {clockObject}
    -source masterPinName
    [-master_clock clockName]
    [-divide_by integer | -multiply_by integer [-duty_cycle value]]
    [-invert]
    [-edges {edgeList}]
    [-edge_shift {edgeShiftList}]
    [-combinational]
    [-disable]
    [-comment commentString]
```

Arguments

-name *clockName*	Specifies the name of the generated clock. If this option is not used, the clock gets the name of the first clock source specified in the -source option (*clockObject*). If you specify the -add option, you must use the -name option and the clocks with the same source must have different names.
-add	Specifies whether to add this clock to the existing clock or to overwrite it. Use this option when multiple generated clocks must be specified on the same source, because multiple clocks fan into the master pin. Ideally, one generated clock must be specified for each clock that fans into the master pin. If you specify this option, you must also use the -name and -master_clock options.
clockObject	The first clock source specified in the -source option in the absence of *clockName*. Clocks can be defined on pins, ports, and nets. The FPGA synthesis tools support nets and instances, where instances have only one output.
-source *masterPinName*	Specifies the master clock source (a clock source pin in the design), from which the clock waveform is derived. The actual delays (latency) for the generated clock are computed using its own source pins and not the *masterPin* option.

-master_clock *clockName*	Specifies the master clock to be used for this generated clock, when multiple clocks fan into the master pin.
-divide_by *integer*	Specifies the frequency division factor. If the *divideFactor* value is *2*, the generated clock period is twice as long as the master clock period.
-multiply_by *integer*	Specifies the frequency multiplication factor. If the *multiplyFactor* value is *3*, the generated clock period is one-third as long as the master clock period.
-duty_cycle *percent*	Specifies the duty cycle, as a percentage, if frequency multiplication is used. Duty cycle is the high pulse width.
-invert	Inverts the generated clock signal (in the case of frequency multiplication and division).
-edges *edgeList*	Specifies a list of integers that represents edges from the source clock that are to form the edges of the generated clock. The edges are interpreted as alternating rising and falling edges and each edge must not be less than its previous edge. The number of edges must be an odd number and not less than 3 to make one full clock cycle of the generated clock waveform. For example, 1 represents the first source edge, 2 represents the second source edge, and so on.
-edge_shift *edgeShiftList*	Specifies a list of floating point numbers that represents the amount of shift, in nanoseconds, that the specified edges are to undergo to yield the final generated clock waveform. The number of edge shifts specified must be equal to the number of edges specified. The values can be positive or negative; positive indicating a shift later in time, while negative indicates a shift earlier in time. For example, 1 indicates that the corresponding edge is to be shifted by one library time unit.
-combinational	The source latency paths for this type of generated clock only includes the logic where the master clock propagates. The source latency paths do not flow through sequential element clock pins, transparent latch data pins, or source pins of other generated clocks.
-disable	Disables the constraint.
-comment *textString*	Allows the command to accept a comment string. The tool honors the annotation and preserves it with the object so that the exact string is written out when the constraint is written out. The comment remains intact through the synthesis, place-and-route, and timing-analysis flows.

reset_path

Resets the specified paths to single-cycle timing.

Syntax

The supported syntax for the reset_path constraint is:

reset_path [**-setup**]
[**-from {***objectList***}**]
[**-through {***objectList***}** [**-through {***objectList***} ...**]]
[**-to {***objectList***}**]
[**-disable**]
[**-comment** *commentString*]

Arguments

-setup	Specifies that setup checking (maximum delay) is reset to single-cycle behavior.
-from	Specifies the names of objects to use to find path start points. The -from *objectList* includes: • Clocks • Registers • Top-level input or bi-directional ports) • Black box outputs When the specified object is a clock, all flip-flops, latches, and primary inputs related to that clock are used as path start points

-through	Specifies the intermediate points for the timing exception. The -through *objectList* includes: • Combinational nets • Hierarchical ports • Pins on instantiated cells By default, the through points are treated as an OR list. The constraint is applied if the path crosses any points in *objectList*. If more than one object is included, the objects must be enclosed either in quotation marks ("") or in braces ({}). If you specify the -through option multiple times, reset_path applies to the paths that pass through a member of each *objectList*. If you use the -through option in combination with the -from or -to options, reset_path applies only if the -from or -to and the -through conditions are satisfied.
-to	Specifies the names of objects to use to find path end points. The -to *objectList* includes: • Clocks • Registers • Top-level output or bi-directional ports • Black box inputs If a specified object is a clock, all flip-flops, latches, and primary outputs related to that clock are used as path end points.
-disable	Disables the constraint.
-comment *textString*	Allows the command to accept a comment string. The tool honors the annotation and preserves it with the object so that the exact string is written out when the constraint is written out. The comment remains intact through the synthesis, place-and-route, and timing-analysis flows.

set_clock_groups

Specifies clock groups that are mutually exclusive or asynchronous with each other in a design so that the paths between these clocks are not considered during timing analysis.

Syntax

The supported syntax for the set_clock_groups constraint is:

set_clock_groups
-asynchronous
-group {*clockList***}** [**-group {***clockList***}** …]
-derive
[**-disable**]
[**-comment** *commentString*]

Arguments

-asynchronous	Specifies that the clock groups are asynchronous to each other (the FPGA synthesis tools assume all clock groups are synchronous). Two clocks are asynchronous with respect to each other if they have no phase relationship at all.
-group *clockList*	Specifies the clocks in a group (*clockList*). You can use the -group option more than once in a single command execution. However, you can include a clock only in one group in a single command execution. To include a clock in multiple groups, you must execute the set_clock_groups command multiple times. By default, a generated clock and its master clock are not in the same group when the exclusive or asynchronous clock groups are defined. The -derive option can be used to override this behavior and allow generated or derived clocks to inherit the clock group of their parent source clock. Each -group instance specifies a group of clocks, which are exclusive or asynchronous with the clocks in all other groups. If you specify only one group, it means that the clocks in that group are exclusive or asynchronous with all other clocks in the design. A default other group is created for this single group. Whenever a new clock is created, it is automatically included in this group.
-derive	Specifies that generated and derived clocks inherit the clock group of the parent clock.

-disable	Disables the constraint.
-comment *textString*	Allows the command to accept a comment string. The tool honors the annotation and preserves it with the object so that the exact string is written out when the constraint is written out. The comment remains intact through the synthesis, place-and-route, and timing-analysis flows.

Limitations

Clock grouping in the FPGA synthesis environment is inclusionary or exclusionary (for example, clk2 and clk3 can each be related to clk1 without being related to each other).

The following set_clock_groups constraint specifies that clk1 and clk2 are asynchronous, but clk1 and clk2 are synchronous to clk3.

```
create_clock [get_ports {c1}] -name clk1 -period 10
create_clock [get_ports {c2}] -name clk2 -period 16
create_clock [get_ports {c3}] -name clk3 -period 5
set_clock_groups -asynchronous -group [get_clocks {clk1}]
   -group [get_clocks {clk2}]
```

set_clock_latency

Specifies clock network latency.

Syntax

The supported syntax for the set_clock_latency constraint is:

```
set_clock_latency
    -source
    [-clock {clockList}]
    delayValue
    {objectList}
```

Arguments

-source	Indicates that the specified delay is applied to the clock source latency.
-clock *clockList*	Indicates that the specified delay is applied with respect to the specified clocks. By default, the specified delay is applied to all specified objects.
delayValue	Specifies the clock latency value.
objectList	Specifies the input ports for which clock latency is to be set

Description

In the FPGA synthesis tools, the set_clock_latency constraint accepts both clock objects and clock aliases. Applying a set_clock_latency constraint on a port can be used to model the off-chip clock delays in a multi-chip environment. Clock latency is forward annotated in the top-level constraint file as part of the time budgeting that takes place in the Certify/HAPS flow. The annotated values represent the arrival times for clocks on specific ports of any particular FPGA in a HAPS design.

In the above syntax, *objectList* references either input ports with defined clocks or clock aliases defined on the input ports. When more than one clock is defined for an input port, the -clock option can be used to apply different latency values to each alias.

Restrictions

The following limitations are present in the FPGA synthesis environment:

- Clock latency can only be applied to clocks defined on input ports.
- The set_clock_latency constraint is only used for source latency.
- The constraint only applies to port clock objects.
- Latency on clocks defined with create_generated_clock is not supported.

set_clock_route_delay

Translates the -route option for the legacy define_clock constraint.

Syntax

The supported syntax for the set_clock_route_delay constraint is:

set_clock_route_delay {*clockAliasList***} {***delayValue***}**

Arguments

clockAliasList	Lists the clock aliases to include the route delay.
delayValue	Specifies the route delay value.

Description

The sdc2fdc translator performs a translation of the -route option for the legacy define_clock constraint and places a set_clock_route_delay constraint in the *_translated.fdc file using the following format:

```
set_clock_route_delay [get_clocks {clk_alias_1  clk_alias_2  ...}]
   {delay_in_ns}
```

set_clock_uncertainty

Specifies the uncertainty (skew) of the specified clock networks.

Syntax

The supported syntax for the set_clock_uncertainty constraint is:

set_clock_uncertainty
{*objectList***}**
-from *fromClock* **|-rise_from** *riseFromClock* **| -fall_from** *fallFromClock*
-to *toClock* **|-rise_to** *riseToClock* **| -fall_to** *fallToClock*
value

Arguments

objectList	Specifies the clocks for simple uncertainty. The uncertainty is applied to the capturing latches clocked by one of the specified clocks. You must specify either this argument or a clock pair with the -from/-rise_from/-fall_from and -to/-rise_to/-fall_to options; you cannot specify both an object list and a clock pair.
-from *fromClock*	Specifies the source clocks for interclock uncertainty. You can use only one of the -from, -rise_from, and -fall_from options and you must specify a destination clock with one of the -to, -rise_to, and -fall_to options.
-rise_from *riseFromClock*	Specifies that the uncertainty applies only to the rising edge of the source clock. You can use only one of the -from, -rise_from, and -fall_from options and you must specify a destination clock with one of the -to, -rise_to, and -fall_to options.
-fall_from *fallFromClock*	Specifies that the uncertainty applies only to the falling edge of the source clock. You can use only one of the -from, -rise_from, and -fall_from options and you must specify a destination clock with one of the -to, -rise_to, and -fall_to options.
-to *toClock*	Specifies the destination clocks for interclock uncertainty. You can use only one of the -to, -rise_to, and -fall_to options and you must specify a source clock with one of the -from, -rise_from, and -fall_from options.
-rise_to *riseToClock*	Specifies that the uncertainty applies only to the rising edge of the destination clock. You can use only one of the -to, -rise_to, and -fall_to options and you must specify a source clock with one of the -from, -rise_from, and -fall_from options.

-fall_to *fallToClock*	Specifies that the uncertainty applies only to the falling edge of the destination clock. You can use only one of the -to, -rise_to, and -fall_to options and you must specify a source clock with one of the -from, -rise_from, and -fall_from options.
value	Specifies a floating-point number that indicates the uncertainty value. Typically, clock uncertainty should be positive. Negative uncertainty values are supported for constraining designs with complex clock relationships. Setting the uncertainty value to a negative number could lead to optimistic timing analysis and should be used with extreme care.

set_false_path

Removes timing constraints from particular paths.

Syntax

The supported syntax for the set_false_path constraint is:

```
set_false_path
   [-setup]
   [-from {objectList}]
   [-through {objectList} [-through {objectList} ...] ]
   [-to {objectList}]
   [-disable]
   [-comment commentString]
```

Arguments

-setup	Specifies that setup checking (maximum delay) is reset to single-cycle behavior.
-from	Specifies the names of objects to use to find path start points. The -from *objectList* includes: • Clocks • Registers • Top-level input or bi-directional ports • Black box outputs When the specified object is a clock, all flip-flops, latches, and primary inputs related to that clock are used as path start points.

-through

Specifies the intermediate points for the timing exception. The -through *objectList* includes:

- Combinational nets
- Hierarchical ports
- Pins on instantiated cells

By default, the through points are treated as an OR list. The constraint is applied if the path crosses any points in *objectList.* If more than one object is included, the objects must be enclosed either in quotation marks ("") or in braces ({}). If you specify the -through option multiple times, set_path applies to the paths that pass through a member of each *objectList*. If you use the -through option in combination with the -from or -to options, set_false_path applies only if the -from or -to and the -through conditions are satisfied.

-to

Specifies the names of objects to use to find path end points. The -to *objectList* includes:

- Clocks
- Registers
- Top-level output or bi-directional ports
- Black box inputs

If a specified object is a clock, all flip-flops, latches, and primary outputs related to that clock are used as path end points.

-disable

Disables the constraint.

-comment *textString*

Allows the command to accept a comment string. The tool honors the annotation and preserves it with the object so that the exact string is written out when the constraint is written out. The comment remains intact through the synthesis, place-and-route, and timing-analysis flows.

set_input_delay

Sets input delay on pins or input ports relative to a clock signal.

Syntax

The supported syntax for the set_input_delay constraint is:

set_input_delay
[**-clock** *clockName* [**-clock_fall**]]
[**-rise**|**-fall**]
[**-min**|**-max**]
[**-add_delay**]
delayValue
{*portPinList***}**
[**-disable**]
[**-comment** *commentString*]

Argument

-clock *clockName*	Specifies the clock to which the specified delay is related. If -clock_fall is used, -clock *clockName* must be specified. If -clock is not specified, the delay is relative to time zero for combinational designs. For sequential designs, the delay is considered relative to a new clock with the period determined by considering the sequential cells in the transitive fanout of each port.
-clock_fall	Specifies that the delay is relative to the falling edge of the clock. The default is the rising edge.
-rise	Specifies that *delayValue* refers to a rising transition on the specified ports of the current design. If neither -rise nor -fall is specified, rising and falling delays are assumed to be equal.
-fall	Specifies that *delayValue* refers to a falling transition on the specified ports of the current design. If neither -rise nor -fall is specified, rising and falling delays are assumed equal.
-min	Specifies that *delayValue* refers to the shortest path. If neither -max nor -min is specified, maximum and minimum input delays are assumed equal.
-max	Specifies that *delayValue* refers to the longest path. If neither -max nor -min is specified, maximum and minimum input delays are assumed equal.

-add_delay	Specifies if delay information is to be added to the existing input delay or if is to be overwritten. The -add_delay option enables you to capture information about multiple paths leading to an input port that are relative to different clocks or clock edges.
-disable	Disables the constraint.
-comment *textString*	Allows the command to accept a comment string. The tool honors the annotation and preserves it with the object so that the exact string is written out when the constraint is written out. The comment remains intact through the synthesis, place-and-route, and timing-analysis flows.
delayValue	Specifies the path delay. The *delayValue* must be in units consistent with the technology library used during optimization. The *delayValue* represents the amount of time the signal is available after a clock edge. This represents a combinational path delay from the clock pin of a register.
portPinList	Specifies a list of input port names in the current design to which *delayValue* is assigned. If more than one object is specified, the objects are enclosed in quotes ("") or in braces ({}).

set_max_delay

Specifies a maximum delay target for paths in the current design.

Syntax

The supported syntax for the set_max_delay constraint is:

```
set_max_delay
    [-from {objectList}]
    [-through{objectList} [-through {objectList} ...] ]
    [-to {objectList}]
    delayValue
    [-disable]
    [-comment commentString]
```

Arguments

-from

Specifies the names of objects to use to find path start points. The -from *objectList* includes:

- Clocks
- Registers
- Top-level input or bi-directional ports
- Black box outputs

When the specified object is a clock, all flip-flops, latches, and primary inputs related to that clock are used as path start points. All paths from these start points to the end points in the -from *objectList* are constrained to *delayValue*. If a -to *objectList* is not specified, all paths from the -from *objectList* are affected. If you include more than one object, you must enclose the objects in quotation marks ("") or braces ({}).

-through

Specifies the intermediate points for the timing exception. The -through *objectList* includes:

- Combinational nets
- Hierarchical ports
- Pins on instantiated cells

By default, the through points are treated as an OR list. The constraint is applied if the path crosses any points in *objectList*. The max delay value applies only to paths that pass through one of the points in the -through *objectList*. If more than one object is included, the objects must be enclosed either in quotation marks ("") or in braces ({}). If you specify the -through option multiple times, set_max_delay applies to the paths that pass through a member of each *objectList*. If you use the -through option in combination with the -from or -to options, set_max_delay applies only if the -from or -to and the -through conditions are satisfied.

-to

Specifies the names of objects to use to find path end points. The -to *objectList* includes:

- Clocks
- Registers
- Top-level output or bi-directional ports
- Black box inputs

If a specified object is a clock, all flip-flops, latches, and primary outputs related to that clock are used as path end points. All paths to the end points in the -to *objectList* are constrained to *delayValue*. If a -from *objectList* is not specified, all paths to the -to *objectList* are affected. If you include more than one object, you must enclose the objects in quotation marks ("") or braces ({}).

-disable

Disables the constraint.

-comment *textString*

Allows the command to accept a comment string. The tool honors the annotation and preserves it with the object so that the exact string is written out when the constraint is written out. The comment remains intact through the synthesis, place-and-route, and timing-analysis flows.

delayValue	Specifies the value of the desired maximum delay for paths between start and end points. You must express *delayValue* in the same units as the technology library used during optimization. If a path start point is on a sequential device, clock skew is included in the computed delay. If a path start point has an input delay specified, that delay value is added to the path delay. If a path end point is on a sequential device, clock skew and library setup time are included in the computed delay. If the end point has an output delay specified, that delay is added into the path delay.

set_multicycle_path

Modifies the single-cycle timing relationship of a constrained path.

Syntax

The supported syntax for the set_multicycle_path constraint is:

```
set_multicycle_path
   [-start |-end]
   [-from {objectList}]
   [-through {objectList} [-through {objectList} ...] ]
   [-to {objectList}]
   pathMultiplier
   [-disable]
   [-comment commentString]
```

Arguments

-start | -end

Specifies if the multi-cycle information is relative to the period of either the start clock or the end clock. These options are only needed for multi-frequency designs; otherwise start and end are equivalent. The start clock is the clock source related to the register or primary input at the path start point. The end clock is the clock source related to the register or primary output at the path endpoint. The default is to move the setup check relative to the end clock, and the hold check relative to the start clock. A setup multiplier of 2 with -end moves the relation forward one cycle of the end clock. A setup multiplier of 2 with -start moves the relation back one cycle of the start clock. A hold multiplier of 1 with -start moves the relation forward one cycle of the start clock. A hold multiplier of 1 with -end moves the relation back one cycle of the end clock.

-from	Specifies the names of objects to use to find path start points. The -from *objectList* includes: • Clocks • Registers • Top-level input or bi-directional ports • Black box outputs When the specified object is a clock, all flip-flops, latches, and primary inputs related to that clock are used as path start points. If a -to *objectList* is not specified, all paths from the -from *objectList* are affected. If you include more than one object, you must enclose the objects in quotation marks ("") or braces ({}).
-through	Specifies the intermediate points for the timing exception. The -through *objectList* includes: • Combinational nets • Hierarchical ports • Pins on instantiated cells The multi-cycle values apply only to paths that pass through one of the points in the -through *objectList*. If more than one object is included, the objects must be enclosed either in double quotation marks ("") or in braces ({}). If you specify the -through option multiple times, set_multicycle_delay applies to the paths that pass through a member of each *objectList*. If the -through option is used in combination with the -from or -to options, the multi-cycle values apply only if the -from or -to conditions and the -through conditions are satisfied.
-to	Specifies the names of objects to use to find path end points. The -to *objectList* includes: • Clocks • Registers • Top-level output or bi-directional ports • Black box inputs If a specified object is a clock, all flip-flops, latches, and primary outputs related to that clock are used as path end points. If a -from *objectList* is not specified, all paths to the -to *objectList* are affected. If you include more than one object, you must enclose the objects in quotation marks ("") or braces ({})..
-disable	Disables the constraint.

-comment *textString*	Allows the command to accept a comment string. The tool honors the annotation and preserves it with the object so that the exact string is written out when the constraint is written out. The comment remains intact through the synthesis, place-and-route, and timing-analysis flows.
pathMultiplier	Specifies the number of cycles that the data path must have for setup or hold relative to the start point or end point clock before data is required at the end point. When used with -setup, this value is applied to setup path calculations. When used with -hold, this value is applied to hold path calculations. If neither -hold nor -setup are specified, *pathMultiplier* is used for setup, and 0 is used for hold. Changing the *pathMultiplier* for setup also affects the hold check.

set_output_delay

Sets output delay on pins or output ports relative to a clock signal.

Syntax

The supported syntax for the set_output_delay constraint is:

set_output_delay
[**-clock** *clockName* [**-clock_fall**]]
[**-rise**|[**-fall**]
[**-min**|**-max**]
[**-add_delay**]
delayValue
{*portPinList***}**
[**-disable**]
[**-comment** *commentString*]

Arguments

-clock *clockName*	Specifies the clock to which the specified delay is related. If -clock_fall is used, -clock *clockName* must be specified. If -clock is not specified, the delay is relative to time zero for combinational designs. For sequential designs, the delay is considered relative to a new clock with the period determined by considering the sequential cells in the transitive fanout of each port.
-clock_fall	Specifies that the delay is relative to the falling edge of the clock. If -clock is specified, the default is the rising edge.
-rise	Specifies that *delayValue* refers to a rising transition on the specified ports of the current design. If neither -rise nor -fall is specified, rising and falling delays are assumed to be equal.
-fall	Specifies that *delayValue* refers to a falling transition on the specified ports of the current design. If neither -rise nor -fall is specified, rising and falling delays are assumed equal.
-min	Specifies that *delayValue* refers to the shortest path. If neither -max nor -min is specified, maximum and minimum output delays are assumed equal.
-max	Specifies that *delayValue* refers to the longest path. If neither -max nor -min is specified, maximum and minimum output delays are assumed equal.

-add_delay
Specifies whether to add delay information to the existing output delay or to overwrite. The -add_delay option enables you to capture information about multiple paths leading to an output port that are relative to different clocks or clock edges.

-disable
Disables the constraint.

-comment *textString*
Allows the command to accept a comment string. The tool honors the annotation and preserves it with the object so that the exact string is written out when the constraint is written out. The comment remains intact through the synthesis, place-and-route, and timing-analysis flows.

delayValue
Specifies the path delay. The *delayValue* must be in units consistent with the technology library used during optimization. The *delayValue* represents the amount of time that the signal is required before a clock edge. For maximum output delay, this usually represents a combinational path delay to a register plus the library setup time of that register. For minimum output delay, this value is usually the shortest path delay to a register minus the library hold time

portPinList
A list of output port names in the current design to which *delayValue* is assigned. If more than one object is specified, the objects are enclosed in double quotation marks ("") or in braces ({}).

set_reg_input_delay

Speeds up paths feeding a register by a given number of nanoseconds.

Syntax

set_reg_input_delay {*registerName***}** [**-route** *ns*] [**-disable**] [**-comment** *textString*]

Arguments

registerName	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.
-comment	Allows the command to accept a comment string. The tool honors the annotation and preserves it with the object so that the exact string is written out when the constraint is written out. The comment remains intact through the synthesis, place-and-route, and timing-analysis flows.
-disable	Disables the constraint.

Description

The set_reg_input_delay timing constraint speeds up paths feeding a register by a given number of nanoseconds. The Synopsys FPGA synthesis tool attempts to meet the global clock frequency goals for a design as well as the individual clock frequency goals (set with create_clock). Use this constraint to speed up the paths feeding a register.

set_reg_output_delay

Speeds up paths coming from a register by a given number of nanoseconds.

Syntax

set_reg_output_delay {*registerName***}** [**-route** *ns*] [**-disable**] [**-comment** *textString*]

Arguments

registerName	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.
-comment	Allows the command to accept a comment string. The tool honors the annotation and preserves it with the object so that the exact string is written out when the constraint is written out. The comment remains intact through the synthesis, place-and-route, and timing-analysis flows.
-disable	Disables the constraint.

The set_reg_output_delay constraint speeds up paths coming from a register by a given number of nanoseconds. The synthesis tool attempts to meet the global clock frequency goals for a design as well as the individual clock frequency goals (set with create_clock). Use this constraint to speed up the paths coming from a register.

Object Qualifiers

The following are the supported object qualifiers for the FPGA constraints. These qualifiers can be used with the Tcl collection commands (see find Command (Batch), on page 1098). The identifier in parentheses is the corresponding *typespec* value used in the constraint file.

- get_clocks (c:)
- get_registers (r:)
- get_nets (n:)
- get_ports (p:)
- get_cells (i:)
- get_pins (t:)

Currently, wildcards (*) are not supported with get_clocks.

Naming Rule Syntax Commands

The FPGA synthesis environment uses a set of naming conventions for design objects in the RTL when your project contains constraint files. The following naming rule commands are added to the constraint file to change the expected default values. These commands must appear at the beginning of the constraint file before any other constraints. Similarly, when multiple constraint files are included in the project, the naming rule commands must be in the first constraint file read.

set_hierarchy_separator Command

The set_hierarchy_separator command redefines the hierarchy separator character (the default separator character is the period in the FPGA synthesis environment). For example, the following command changes the separator character to a forward slash:

```
set_hierarchy_separator {/}
```

set_rtl_ff_names Command

The set_rtl_ff_names command controls the stripping of register suffixes in the object strings of delay-path constraints (for example, set_false_path, set_multicycle_path). Generally, it is only necessary to change this value from its default when constraints that target ASIC designs are being imported from the Design Compiler (in the Design Compiler, inferred registers are given a _reg suffix during the elaboration phase; constraints targeting these registers must include this suffix). When importing constraints from the Design Compiler, include the following command to change the value of this naming rule to {_reg} to automatically recognize the added suffix.

```
set_rtl_ff_names {_reg}
```

For example, using the above value allows the DC exception

```
set_false_path -to [get_cells {register_bus_reg[0]}]
```

to apply to the following object without having to manually modify the constraint:

```
[get_cells {register_bus[0]}]
```

bus_naming_style Command

The bus_naming_style command redefines the format for identifying bits of a bus (by default, individual bits of a bus are identified by the bus name followed by the bus bit enclosed in square brackets). For example, the following command changes the bus-bit identification from the default *busName*[*busBit*] format to the *busName_busBit* format:

```
bus_naming_style {%s_%d}
```

bus_dimension_separator_style Command

The bus_dimension_separator_style command redefines the format for identifying multi-dimensional arrays (by default, multidimensional arrays such as row 2, bit 3 of array ABC[*n* x *m*] are identified as ABC[2][3]). For example, the following command changes the bus-dimension separator from individual square bracket sets to an underscore:

```
bus_dimension_separator_style {_}
```

The resulting format for the above example is:

```
ABC[2_3]
```

read_sdc Command

Reads in a script in Synopsys FPGA constraint format. The supported syntax for the read_sdc constraint is:

read_sdc *fileName*

Synplify-Style Timing Constraints (Legacy)

This section describes the constraint file syntax for the Synplify-style legacy timing constraints.

Note: For tool versions 2012.09 and later, it is recommended that you use the Synopsys Standard timing constraint format instead of the Synplify-style legacy format. Use the sdc2fdc command to convert your constraints. See FPGA Timing Constraints, on page 834 for more information.

The legacy constraints are included in your project through an SDC file. This section describes the constraint file syntax for the legacy FPGA timing constraints, which include:

- define_clock, on page 866
- define_clock_delay, on page 869
- define_false_path, on page 870
- define_input_delay, on page 873
- define_multicycle_path, on page 875
- define_output_delay, on page 879
- define_path_delay, on page 881
- define_reg_input_delay, on page 884
- define_reg_output_delay, on page 885

For information on the supported design constraints, see Chapter 10, *FPGA Design Constraint Syntax*.

define_clock

Defines a clock with a specific duty cycle and frequency or clock period goal.

Syntax

define_clock [**-disable**] [**-virtual**] **{***clockObject***}**
[**-freq** *MHz* |**-period** *ns*] [**-clockgroup** *domain*]
[**-rise** *value* |**-fall** *value*] [**-route** *ns*]
[**-name** *clockName*] [**-comment** *textString*]

Arguments

-disable Disables a previous clock constraint.

-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.

clockObject A required parameter that specifies the clock object name. Clocks can be defined on the following objects:

- Top-level input ports (p:)
- Nets (n:)
- Instances (i:)
- Output pins of instantiated cells (t:)

Clocks defined on any of the following objects *WILL NOT* be honored:

- Top-level output ports
- Input pins of instantiated gates
- Pins of inferred instances

-name Specifies a name for the clock if you want to use a name other than the clock object name. This alias name is used in the timing reports.

-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 assign related (synchronized) clocks to 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).

See Clock Groups, on page 412 for more information.

-rise|-fall

Specifies a non-default duty cycle. By default, the 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 413.

-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 414.

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.

-comment

Allows the command to accept a comment string. The tool honors the annotation and preserves it with the object so that the exact string is written out when the constraint is written out. The comment remains intact through the synthesis, place-and-route, and timing-analysis flows.

Description

The define_clock timing constraint defines a clock with a specific duty cycle and frequency or clock period goal. 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. If you are constraining a differential clock, you only need to constrain the positive input.

For the equivalent SCOPE spreadsheet interface and descriptions of the options, see Clocks (Legacy), on page 409 and Clocks Panel Description, on page 410.

Examples

define_clock Syntax

```
define_clock {CLK1} -period 10.0 -clockgroup default_clkgroup

define_clock {CLK3} -period 5.0 -clockgroup default_clkgroup
   -name INT_REF3

define_clock -virtual {CLK2} -period 20.0 -clockgroup g2

define_clock {CLK4} -period 20.000 -clockgroup g3
   -rise 1.000 -fall 11.000 -ref_rise 0.000 -ref_fall 10.000
```

define_clock Pin-Level Constraint

```
define_clock {i:myInst1.Q} -period 10.000
   -clockgroup default -rise 0.200 –fall 5.200 -name myff1

define_clock {i:myInst2.Q} -period 12.000
   -clockgroup default -rise 0.400 –fall 5.400 -name myff2
```

In the example above, a clock is defined on the Q pins of instances myInst1 and myInst2.

define_clock_delay

Defines the delay between the clocks in the design.

Syntax

define_clock_delay -rise|-fall {*clockName1***} -rise|-fall {***clockName2***}** *delayValue*

Arguments

-rise|fall	**Specifies the clock edge**
clockName	Specifies the clocks to constrain The clock must be already defined with define_clock.
delayValue	Specifies the delay, in nanoseconds, between the two clocks. You can also specify a value false which defines the path as a false path.

Description

The define_clock_delay timing constraint defines the 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 Clock Relationships section of the Timing Report are based on calculations made using this constraint.

Note: Maintenance for the define_clock_delay command is limited. It is recommended that you use the define_false_path and define_path_delay commands instead, since these commands support clock aliases.

define_false_path

Defines paths to ignore (remove) during timing analysis.

Syntax

define_false_path [**-disable**] {**-from** *startPoint* |**-to** *endPoint* |**-through** *throughPoint*}
[**-comment** *textString*]

Arguments

-from *startPoint*

Specifies the starting point for the false path. The from point defines a timing start point and can be any of the following:

- Clocks (c:) (See Clocks as From/To Points, on page 401.)
- Registers (i:)
- Top-level input or bi-directional ports (p:)
- Black box outputs (i:)

For more information, see the following:

- syn_black_box Directive, on page 921
- Specifying From, To, and Through Points, on page 396
- Defining From/To/Through Points for Timing Exceptions, on page 70 in the *User Guide*.

-to *endPoint*

Specifies the ending point for the false path. The to point defines a timing end point and can be any of the following objects:

- Clocks (c:) (See Clocks as From/To Points, on page 401.)
- Registers (i:)
- Top-level output or bi-directional ports (p:)
- Black box inputs (i:)

For more information, see the following:

- syn_black_box Directive, on page 921
- Specifying From, To, and Through Points, on page 396
- Defining From/To/Through Points for Timing Exceptions, on page 70 in the *User Guide*

-through *throughPoint*	Specifies the intermediate points for the timing exception. Intermediate points can be any of the following objects: • Combinational nets (n:) • Hierarchical ports (t:) • Pins on instantiated cells (t:) By default, the through points are treated as an OR list. The constraint is applied if the path crosses any points in the list. For more information, see the following: • syn_black_box Directive, on page 921 • Specifying From, To, and Through Points, on page 396 • Defining From/To/Through Points for Timing Exceptions, on page 70 in the *User Guide* To keep the signal name intact through synthesis, set the syn_keep directive (Verilog or VHDL) on the signal.
-disable	Disables the constraint.
-comment	Allows the command to accept a comment string. The tool honors the annotation and preserves it with the object so that the exact string is written out when the constraint is written out. The comment remains intact through the synthesis, place-and-route, and timing-analysis flows.

Description

The define_false_path timing constraint defines paths to ignore (remove) during timing analysis and gives 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, on page 393. See False Path Constraint Examples, on page 395, and Priority of False Path Constraints, on page 394, for additional information.

Example

The following example shows the syntax for setting a define_false_path constraint between registers:

```
define_false_path -from {i:myInst1_reg} -through {n:myInst2_net}
   -to {i:myInst3_reg}
```

The constraint is defined from the output pin of myInst1_reg, through net myInst2_net, to the input of myInst3_reg. If an instance is instantiated, a pin-level constraint applies on the pin, as defined. However, if an instance is inferred, the pin-level constraint is transferred to the instance.

For through points specified on pins, the constraint is transferred to the connecting net. You cannot define a through point on a pin of an instance that has multiple outputs. When specifying a pin on a vector of instances, you cannot refer to more than one bit of that vector.

define_input_delay

Specifies the external input delays on top-level ports in the design.

Syntax

define_input_delay [**-disable**] **{***inputportName* **| -default** *ns* [**-route** *ns*]
[**-ref** *clockName***:***edge*] [**-comment** *textString*]

Arguments

-disable	Disables the constraint.
inputportName	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
-route	An advanced option, which includes route delay when the 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 419 for an example of its use.
-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
-comment	Allows the command to accept a comment string. The tool honors the annotation and preserves it with the object so that the exact string is written out when the constraint is written out. The comment remains intact through the synthesis, place-and-route, and timing-analysis flows.

Description

The define_input_delay timing constraint specifies the external input delays on top-level ports in the design. This external delay 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 interface, see Inputs/Outputs, on page 373.

Example

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}
```

The following example shows how all the bits of the datain input bus are constrained with a delay of 2 ns, except for bit 16:

```
define_input_delay {datain[0:15]} 2.00 -ref clk:r
define_input_delay {datain[17:99]} 2.00 -ref clk:r
```

define_multicycle_path

Specifies a path that uses multiple clock cycles as a timing exception.

Syntax

define_multicycle_path [**-disable**] [**-start** |**-end**]
{**-from** *startPoint* |**-to** *endPoint* |**-through** *throughPoint*}
clockCycles [**-comment** *textString*]

Arguments

-start | -end

Specifies the clock cycles to use for paths with different start and end clocks. This option determines the clock period to use as the multiplicand in the calculation for clock distance. If you do not specify a start or end option, the end clock is the default. See Multi-cycle Path with Different Start and End Clocks, on page 390 for more information.

-from *startPoint*

Specifies the start point for the multi-cycle timing exception. The From point in the constraint defines a timing start point and can be any of the following:

- Clocks (c:) (See Clocks as From/To Points, on page 401.)
- Registers (i:)
- Top-level input or bi-directional ports (p:)
- Black box outputs (i:)

For more information, see the following:

- syn_black_box Directive, on page 921
- Specifying From, To, and Through Points, on page 396
- Defining From/To/Through Points for Timing Exceptions, on page 70 in the *User Guide*.

-to *endPoint*

Specifies the end point for the multi-cycle timing exception. The to point defines a timing end point and can be any of the following objects:

- Clocks (c:) (See Clocks as From/To Points, on page 401.)
- Registers (i:)
- Top-level output or bi-directional ports (p:)
- Black box outputs (i:)

For more information, see the following:

- syn_black_box Directive, on page 921
- Specifying From, To, and Through Points, on page 396
- Defining From/To/Through Points for Timing Exceptions, on page 70 in the *User Guide*.

-through *throughPoint*

Specifies the intermediate points for the timing exception. Intermediate points can be:

- Combinational nets (n:)
- Hierarchical ports (t:)
- Pins on instantiated cells (t:)

By default, the intermediate points are treated as an OR list, the exception is applied if the path crosses any points in the list. For more information, see the following:

- syn_black_box Directive, on page 921
- Specifying From, To, and Through Points, on page 396
- Defining From/To/Through Points for Timing Exceptions, on page 70 in the *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 use this option, set the syn_keep directive (Verilog or VHDL) on the signal.

clockCycles

The number of clock cycles to use for the path constraint.

-disable

Disables the constraint.

-comment

Allows the command to accept a comment string. The tool honors the annotation and preserves it with the object so that the exact string is written out when the constraint is written out. The comment remains intact through the synthesis, place-and-route, and timing-analysis flows.

Description

The define_multicycle_path timing constraint specifies a path that uses multiple clock cycles as a timing exception. This constraint provides extra clock cycles to the designated paths for timing analysis and optimization. For information about the equivalent SCOPE interface, see Multicycle Paths, on page 390.

Timing Exceptions Object Types

Timing exception constraints must contain object types in the specification. Timing exceptions, such as multi-cycle path and false path constraints, require that you explicitly specify the object type (n: or i:) in the instance name parameter. For example:

```
define_multicycle_path -from {i:inst2.lowreg_output[7]}
   -to {i:inst1.DATA0[7]} 2
```

If you use the SCOPE UI to specify timing exceptions, it automatically attaches object type qualifiers to the object names. For more information, see Specifying From, To, and Through Points, on page 396.

Example

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
```

```
define_multicycle_path -from {i:special_regs.trisc[7:0]}
   -through {t:uc_alu.aluz.Q} -through {t:special_net.Q} 2
```

The following example shows the syntax for setting a multi-cycle path constraint between registers:

```
define_multicycle_path -from {i:myInst1_reg}
   -through {n:myInst2_net} -to {i:myInst3_reg} 2
```

The constraint is defined from the output of myInst1_reg, through net myInst2_net, to the input pin myInst3_reg. If the instance is instantiated, a pin-level constraint applies on the pin, as defined. However, if the instance is inferred, the pin-level constraint is transferred to the instance. For through points specified on pins, the constraint is transferred to the connecting net. You cannot define a through point on a pin of an instance that has multiple outputs. When specifying a pin on a vector of instances, you cannot refer to more than one bit of that vector.

For additional examples of multi-cycle paths, see Multicycle Path Examples, on page 391.

define_output_delay

Specifies the delay of the logic outside the FPGA driven by the top-level outputs.

Syntax

define_output_delay [**-disable**] **{***outputportName***}** **|-default** *ns* [**-route** *ns*]
[**-ref** *clockName***:***edge*] [**-comment** *textString*]

Arguments

-disable	Disables the constraint.
outputportName	The name of the output port.
-default	Sets a default output 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

-route

An advanced option, which includes route delay when the synthesis tool tries to meet the clock frequency goal. See Route Option, on page 419 for an example of its use with an input register.

-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

See *Output Pad Clock Domain Default*, below for more information on this option.

-comment

Allows the command to accept a comment string. The tool honors the annotation and preserves it with the object so that the exact string is written out when the constraint is written out. The comment remains intact through the synthesis, place-and-route, and timing-analysis flows.

Description

The define_output_delay constraint models 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, on page 373.

Output Pad Clock Domain Default

By default, define_output_delay constraints with no reference clock are constrained against the global frequency, instead of the start clock for the path to the port. The tool assumes the register and pad are not in the same clock domain. This change affects the timing report and timing driven optimizations on any logic between the register and the pad.

You must specify the clock domain for all output pads on which you have set output delay constraints. For the pads for which you do not specify a clock, add the -ref option to the define_output_delay constraint. For example:

```
define_output_delay {LDCOMP} 0.50
   -route 0.25 -ref {CLK1:r}
```

define_path_delay

Specifies point-to-point delay for maximum delay constraints.

Syntax

define_path_delay [**-disable**]
-from {*startPoint*} | **-to** {*endPoint*} | **-through** {*throughPoint*}
-max *delayValue* [**-comment** *textString*]

Arguments

-disable

Disables the constraint.

-from *startPoint*

Specifies the starting point of the path. The From point defines a timing start point and can be any of the following objects:

- Clocks (c:) (See Clocks as From/To Points, on page 401.)
- Registers (i:)
- Top-level input or bi-directional ports (p:)
- Black box outputs (i:)

For more information, see the following:

- syn_black_box Directive, on page 921
- Specifying From, To, and Through Points, on page 396
- Defining From/To/Through Points for Timing Exceptions, on page 70 in the *User Guide*.

-to *endPoint*

Specifies the ending point of the path. The to point in the constraint must be a timing end point and can be any of the following objects:

- clocks (c:) (See Clocks as From/To Points, on page 401.)
- registers (i:)
- top-level output or bi-directional ports (p:)
- black box inputs (i:)

For more information, see:

- Defining From/To/Through Points for Timing Exceptions, on page 70 in the *User Guide*.

You can combine this option with -from or -through to get a specific path.

-through *throughPoint*	Specifies the intermediate points for the timing exception. Intermediate points can be: • combinational nets (n:) • hierarchical ports (t:) • pins on instantiated cells (t:) By default, the intermediate points are treated as an OR list, the exception is applied if the path crosses any points in the list. For more information, see: • Defining From/To/Through Points for Timing Exceptions, on page 70 in the *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 use this option, set the syn_keep directive (Verilog or VHDL) on the signal.
-max *delayValue*	Sets the maximum allowable delay for the specified path. This is an absolute value in nanoseconds and is shown as max analysis in the timing report.
-comment	Allows the command to accept a comment string. The tool honors the annotation and preserves it with the object so that the exact string is written out when the constraint is written out. The comment remains intact through the synthesis, place-and-route, and timing-analysis flows.

The define_path_delay timing constraint specifies point-to-point delay, in nanoseconds, for maximum delay constraints. You can specify the start, end, or through points using the -from, -to, or -through options or any combination of these options.

Example

Here are examples of the path delay constraint.

```
define_path_delay -from {i:dmux.alua[5]}
   -to {i:regs.mem_regfile_15[0]} -max 0.800
```

The following constraint sets a max delay of 2 ns on all paths to the falling edge of the flip-flops clocked by clk1.

```
define_path_delay -to {c:clk1:f} -max 2
```

Here is an example of setting the path delay constraint on the pins between registers:

```
define_path_delay –from {i:myInst1_reg} –through {t:myInst2_net.Y}
   –to {i:myInst3_reg} –max 0.123
```

The constraint is defined from the output pin of myInst1, through pin Y of net myInst2, to the input pin of myInst3. If the instance is instantiated, a pin-level constraint applies on the pin, as defined. If the instance is inferred, the pin-level constraint is transferred to the instance.

Limitations

The following limitations are present:

- For through points specified on pins, the constraint is transferred to the connecting net. You cannot define a through point on a pin of an instance that has multiple outputs.
- When specifying a pin on a vector of instances, you cannot refer to more than one bit of that vector.

define_reg_input_delay

Speeds up paths feeding a register by a given number of nanoseconds.

Syntax

define_reg_input_delay {*registerName***}** [**-route** *ns*] [**-comment** *textString*]

Arguments

registerName	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 419 for an example of its use.
-comment	Allows the command to accept a comment string. The tool honors the annotation and preserves it with the object so that the exact string is written out when the constraint is written out. The comment remains intact through the synthesis, place-and-route, and timing-analysis flows.

Description

The define_reg_input_delay timing constraint speeds up paths feeding a register by a given number of nanoseconds. The Synopsys FPGA 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 (Legacy), on page 421.

define_reg_output_delay

Speeds up paths coming from a register by a given number of nanoseconds.

Syntax

define_reg_output_delay {*registerName***}** [**-route** *ns*] [**-comment** *textString*]

Arguments

registerName	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 419.
-comment	Allows the command to accept a comment string. The tool honors the annotation and preserves it with the object so that the exact string is written out when the constraint is written out. The comment remains intact through the synthesis, place-and-route, and timing-analysis flows.

The define_reg_output_delay constraint speeds up paths coming from a register by a given number of nanoseconds. The 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 (Legacy), on page 421.

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.

The objects in the SCOPE pull-down menus automatically use the correct prefix, but if you drag and drop an object from an HDL Analyst or other view, make sure you use the appropriate prefix for the language as described below:

- Verilog Naming Syntax, on page 886
- VHDL Naming Syntax, on page 887
- Object Naming Examples, on page 888

Verilog Naming Syntax

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, port, 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 888*.

VHDL Naming Syntax

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.

objpath	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 888*.

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 entries that correspond to the previous source code examples:

```
define_attribute {statemod|i:statereg[*]} syn_encoding sequential
define_multicycle_path -to {*.*slowreg[*]} 2
```

CHAPTER 10

FPGA Design Constraint Syntax

The following describe the Tcl syntax for the design constraints specified with a SCOPE editor or entered manually into a constraint file. These constraints are common between the two supported constraint file formats.

The remainder of this section describes the constraint file syntax for the following FPGA design constraints:

- define_compile_point, on page 890
- define_current_design, on page 891
- define_io_standard, on page 892

For information on timing constraints, see either FPGA Timing Constraints, on page 834 or Synplify-Style Timing Constraints (Legacy), on page 865.

define_compile_point

The define_compile_point command defines a compile point in a top-level constraint file. You use one define_compile_point command for each compile point you define. For the equivalent SCOPE spreadsheet interface, see Compile Points, on page 382. (Compile points are only available for certain technologies.)

This is the syntax:

define_compile_point [**-disable**] **{***moduleName***}**
-type {soft|hard|locked|} [**-comment** *textString*]

-disable	Disables a previous compile point definition.
-type	Specifies the type of compile point. This can be soft, hard, or locked. See Compile Point Types, on page 419 for more information.

Refer to Object Naming Syntax, on page 885 for details about the syntax and prefixes for naming objects.

Here is a syntax example:

```
define_compile_point {v:work.prgm_cntr} -type {locked}
```

define_current_design

The define_current_design command specifies the module to which the constraints that follow it apply. It must be the first command in a block-level or compile-point constraint file. The specified module becomes the top level for objects defined in this hierarchy and the constraints applied in the respective block-level or compile-point constraint file.

This is the syntax:

define_current_design {*regionName* **|** *libraryName***.***moduleName* **}**

Refer to Object Naming Syntax, on page 885 for details about the syntax and prefixes for naming objects.

Here is an example:

```
define_current_design {lib1.prgm_cntr}
```

Objects in all constraints that follow this command relate to prgm_cntr.

define_io_standard

Specifies a standard I/O pad type to use for various Microsemi families. See I/O Standards, on page 380 for details of the SCOPE equivalent.

define_io_standard [**-disable**] **{p:***portName***} -delay_type input|output|bidir**
syn_pad_type {*IO_standard***}** [*parameter* **{***value***}**...]

In the above syntax:

portName is the name of the input, output, or bidirectional port.

-delay_type identifies the port direction which must be input, output, or bidir.

syn_pad_type is the I/O pad type (I/O standard) to be assigned to *portName*.

parameter is one or more of the parameters defined in the following table. Note that these parameters are device-family dependent.

Parameter	Function
syn_io_termination	The termination type; typical values are pullup and pulldown.
syn_io_drive	The output drive strength; values include low and high or numerical values in mA.
syn_io_dv2	Switch to use a 2x impedance value.
syn_io_dci	Switch for digitally-controlled impedance (DCI).
syn_io_slew	The slew rate for single-ended output buffers; values include slow and fast or low and high.

Example:

```
define_io_standard {p:DATA1[7:0]} -delay_type input
   syn_pad_type {LVCMOS_33} syn_io_slew {high}
   syn_io_drive {12} syn_io_termination {pulldown}
```

CHAPTER 11

Synthesis Attributes and Directives

Synthesis attributes and directives let you direct the way a design is analyzed, optimized, and mapped during synthesis. This chapter shows you how to specify attributes and directives. It contains individual attribute and directive descriptions, syntax definitions, and examples. It contains the following information:

How Attributes and Directives are Specified

By definition, *attributes* control mapping optimizations and *directives* control compiler optimizations.

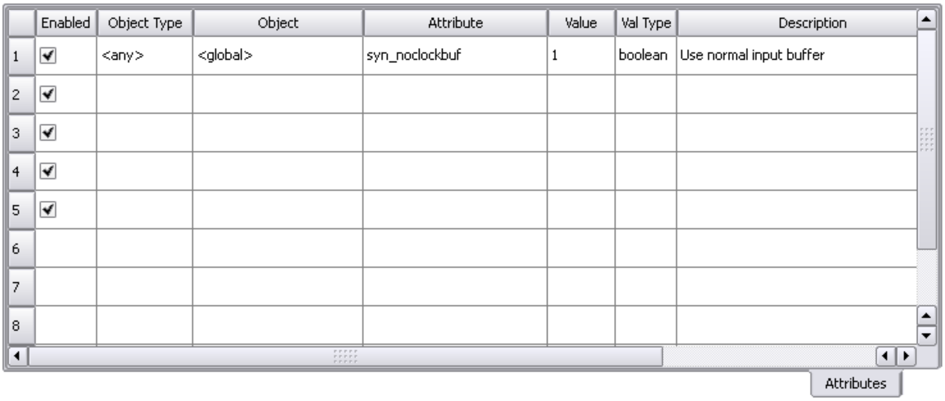
Because of this difference, directives must be entered directly in the HDL source code. Attributes can be entered either in the source code, in the SCOPE Attributes tab, or manually in a constraint file. For detailed procedures on different ways to specify attributes and directives, see Entering Attributes and Directives, on page 144 in the *User Guide*.

Verilog files are case sensitive, so attributes and directives must be entered exactly as presented in the syntax descriptions.

The SCOPE Attributes Tab

This section describes how to enter attributes using the SCOPE Attributes tab. To use the SCOPE spreadsheet, use this procedure:

1. Start with a compiled design, then open the SCOPE window.
2. Scroll if needed and click the Attributes tab.



	Enabled	Object Type	Object	Attribute	Value	Val Type	Description
1	☑	<any>	<global>	syn_noclockbuf	1	boolean	Use normal input buffer
2	☑						
3	☑						
4	☑						
5	☑						
6							
7							
8							

3. Click in the Attribute cell and use the pull-down menus to enter the appropriate attributes and their values.

The Attributes panel includes the following columns.

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 pull-down 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. You can also drag and drop an object from the RTL or Technology view into this column.
Attribute	(Required) Specifies the attribute name. You can choose from a pull-down 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 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	Contains any comments you want to add about the attributes.

For more details on how to use the Attributes panel of the SCOPE spreadsheet, see Specifying Attributes Using the SCOPE Editor, on page 147 in the *User Guide.*

When you use the SCOPE spreadsheet to create and modify a constraint file, the proper define_attribute or define_global_attribute statement is automatically generated for the constraint file. The following shows the syntax for these statements as they appear in the constraint file.

define_attribute {*object***}** *attributeName* **{***value***}**

define_global_attribute *attributeName* **{***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 1121 for details about the syntax conventions. If you have mixed input files, use the object naming syntax appropriate for the format in which the object is defined. Global attributes, since they apply to an entire design, do not use an *object* argument.
attribute_name	The name of the synthesis attribute. This must be an attribute, not a directive, as directives are not supported in constraint files.
value	String, integer, or boolean value.

See Summary of Global Attributes, on page 1032 for more details on specifying global attributes in the synthesis environment.

Summary of Attributes and Directives

The following section summarizes the synthesis attributes and directives:

- Attribute and Directive Summary (Alphabetical), on page 897

For detailed descriptions of individual attributes and directives, see the individual attributes and directives, which are listed in alphabetical order.

Attribute and Directive Summary (Alphabetical)

The following table summarizes the synthesis attributes and directives. For detailed descriptions of each one, you can find them listed in alphabetical order.

Attribute/Directive	Description	Default
alsloc Attribute	Preserves relative placement in Microsemi designs.	
alspin Attribute	Assigns scalar or bus ports to I/O pin numbers in Microsemi designs.	
alspreserve Attribute	Specifies nets that must be preserved by the Microsemi place-and-route tool.	
black_box_pad_pin Directive	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 Directive	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 Directive	Specifies that a Verilog case statement has covered all possible cases.	

Attribute/Directive	Description	Default
loop_limit Directive	Specifies a loop iteration limit for for loops.	
parallel_case Directive	Specifies a parallel multiplexed structure in a Verilog case statement, rather than a priority-encoded structure.	
pragma translate_off/pragma translate_on Directive	Specifies sections of code to exclude from synthesis, such as simulation-specific code.	
syn_allow_retiming Attribute	Specifies whether registers can be moved during retiming.	
syn_black_box Directive	Defines a black box for synthesis.	
syn_encoding Attribute	Specifies the encoding style for state machines.	Based on number of states.
syn_enum_encoding Directive	Specifies the encoding style for enumerated types (VHDL only).	
syn_global_buffers Attribute	Determines the number of global buffers available.	
syn_hier Attribute	Determines hierarchical control across module or component boundaries.	soft
syn_isclock Directive	Specifies that a black-box input port is a clock, even if the name does not indicate it is one.	
syn_keep Directive	Prevents the internal signal from being removed during synthesis.	
syn_loc Attribute	Specifies pin locations for I/O pins and cores, and forward-annotates this information to the place-and-route tool.	
syn_looplimit Directive (VHDL)	Specifies a loop iteration limit for while loops.	
syn_maxfan Attribute	Overrides the default fanout guide for an individual input port, net, or register output.	

Attribute/Directive	Description	Default
syn_multstyle Attribute	Determines how multipliers are implemented.	block_mult
syn_netlist_hierarchy Attribute	Controls hierarchy generation in EDIF output files	1
syn_noarrayports Attribute	Specifies ports as individual signals or bus arrays.	1
syn_noclockbuf Attribute	Disables automatic clock buffer insertion.	0
syn_noprune Directive	Controls the automatic removal of instances that have outputs that are not driven.	
syn_pad_type Attribute	Specifies an I/O buffer standard for certain technology families.	
syn_preserve Directive	Preserves registers that can be optimized due to redundancy or constraint propagation.	
syn_probe Attribute	Adds probe points for testing and debugging.	
syn_radhardlevel Attribute	Specifies the radiation-resistant design technique to use.	
syn_ramstyle Attribute	Determines how RAMs are implemented.	registers
syn_reference_clock Attribute	Specifies a clock frequency other than that implied by the signal on the clock pin of the register.	
syn_replicate Attribute	Controls replication, either globally or on registers.	0
syn_resources Attribute	Specifies resources used in black boxes.	
syn_sharing Directive	Enables/disables resource sharing of operators inside a module.	
syn_state_machine Directive	Determines if the FSM Compiler extracts a structure as a state machine.	

Attribute/Directive	Description	Default
syn_tco<n> Directive	Defines timing clock to output delay through a black box. The *n* indicates a value between 1 and 10.	
syn_tpd<n> Directive	Specifies timing propagation for combinational delay through a black box. The *n* indicates a value between 1 and 10.	
syn_tristate Directive	Specifies that a black-box pin is a tristate pin.	
syn_tsu<n> Directive	Specifies the timing setup delay for input pins, relative to the clock. The *n* indicates a value between 1 and 10.	
syn_useenables Attribute	Generates clock enable pins for registers.	1
translate_off/translate_on Directive	Specifies sections of code to exclude from synthesis, such as simulation-specific code.	

Attribute/Directive Descriptions

Attribute and directive descriptions are provided in the remaining sections. Each description contains a definition, and for the constraint, Verilog and VHDL files, syntax and examples. Verilog files are case-sensitive, so the syntax must be entered exactly as shown.

alsloc Attribute

Attribute; Microsemi (except 500K and PA). Preserves relative placements of macros and IP blocks in the Microsemi Designer place-and-route tool. The alsloc attribute has no effect on synthesis, but is passed directly to Microsemi Designer.

Constraint File Syntax and Example

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:

```
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. See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

```
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

Attribute; Microsemi. The alspin attribute assigns the scalar or bus ports of the design to Microsemi I/O pin numbers (pad locations). Refer to the Microsemi 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 attribute. See *Specifying Locations for Microsemi Bus Ports, on page 351* of the *User Guide* for information on assigning pin numbers to buses and slices.

Constraint 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 Microsemi 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 Microsemi 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

See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

attribute alspin of *object* **:** *objectType* **is "***pin_number***" ;**

Where *object* is the port, *objectType* is signal, and *pin_number* is the Microsemi 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

Attribute; Microsemi (except 500K and PA). Specifies a net that you do not want removed (optimized away) by the Microsemi Designer place-and-route tool. The alspreserve attribute has no effect on synthesis, but is passed directly to the Microsemi 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 directive.

Constraint 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.

```
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:

```
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

See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

attribute alspreserve of ***object*** **: signal is true ;**

Where *object* is the name of the net to preserve.

For example:

```
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;
```

black_box_pad_pin Directive

Directive. Used with the syn_black_box directive and specifies that the pins on black box are I/O pads visible to the outside environment. To specify more than one port as an I/O pad, list the ports inside double-quotes ("), separated by commas (,), and without enclosed spaces.

To instantiate an I/O from your programmable logic vendor, you usually do not need to define a black box or this directive. The synthesis tool provides predefined black boxes for vendor I/Os. For more information, refer to your vendor section under FPGA and CPLD Support.

black_box_pad_pin is one of several directives that you can use with syn_black_box to define timing for a black box. See syn_black_box Directive, on page 921 for a list of the associated directives.

Verilog Syntax and Example

object **/* synthesis syn_black_box black_box_pad_pin = "***portList***" */ ;**

where *portList* 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* **:** *objectType* **is "***portList***" ;**

where *object* is an architecture or component declaration of a black box. Data type is string; *portList* is a spaceless, comma-separated list of the black-box port names that are I/O pads.

See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

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

Entity top is
generic ( width : integer := 4);
   port (in1,in2 : in std_logic_vector(width downto 0);
   clk : in std_logic;
   q : out std_logic_vector (width downto 0)
   );
end top;

architecture top1_arch of top is
component test is
   generic (width1 : integer := 2);
      port (in1,in2 : in std_logic_vector(width1 downto 0);
      clk : in std_logic;
      q : out std_logic_vector (width1 downto 0)
   );
end component;

attribute syn_black_box : boolean;
attribute black_box_pad_pin : string;
attribute syn_black_box of test : component is true;
attribute black_box_pad_pin of test : component is "in1(4:0),
in2[4:0], q(4:0)";

begin
   test123 : test generic map (width) port map (in1,in2,clk,q);
end top1_arch;
```

black_box_tri_pins Directive

Directive. Used with the syn_black_box directive and specifies that an output port on a black box component is a tristate. This directive eliminates multiple driver errors when the output of a black box has more than one driver. To specify more than one tristate port, list the ports inside double-quotes ("), separated by commas (,), and without enclosed spaces.

The black_box_tri_pins directive is one of several directives that you can use with the syn_black_box directive to define timing for a black box. See syn_black_box Directive, on page 921 for a list of the associated directives.

Verilog Syntax and Examples

object /* **synthesis syn_black_box black_box_tri_pins = "***portList***" */ ;**

where *portList* 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* **:** *objectType* **is "***portList***" ;**

where *object* is a component declaration or architecture. Data type is string, and *portList* is a spaceless, comma-separated list of the tristate output port names.

See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

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

package my_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 package my_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, specify all the bits on the bus:

```
attribute black_box_tri_pins of bb1 : component is "bus1[7:0]";
```

full_case Directive

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.

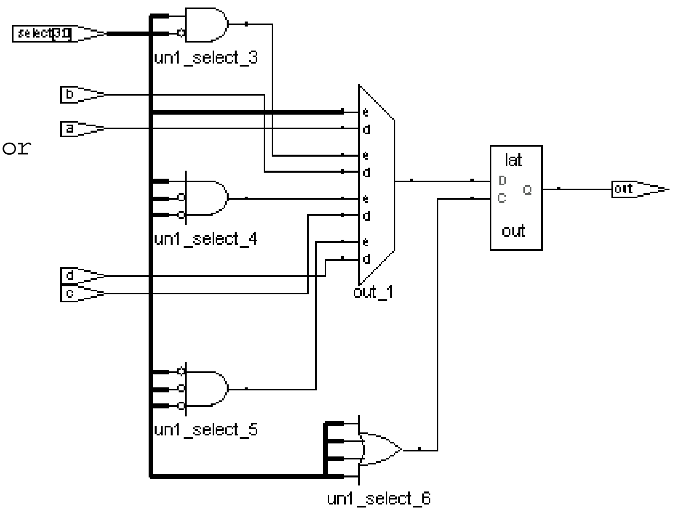
Verilog Syntax and Example

object /* **synthesis full_case** */

where *object* is 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	Must 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	Can cause mismatches between pre- and post-synthesis simulation because the simulator does not use full_case

loop_limit Directive

Directive. For Verilog designs only. Specifies a loop iteration limit for for loops in the design when the loop index is a variable, not a constant. 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.

Note: VHDL applications use the syn_looplimit directive (see syn_looplimit Directive (VHDL), on page 957).

Verilog Syntax and Example

beginning_of_loop_statement /* **synthesis loop_limit** *integer* */

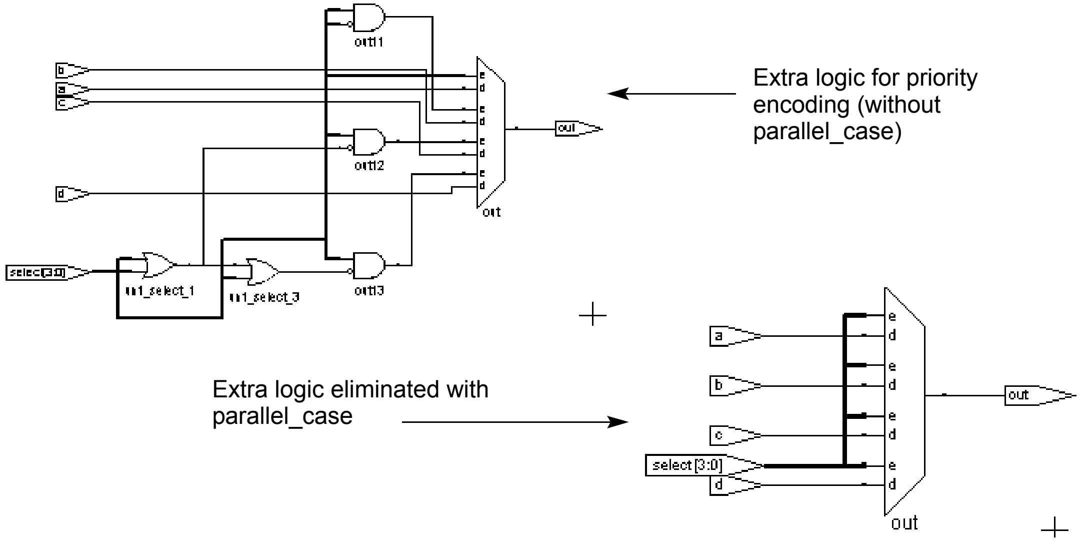
The following is an example where the loop limit is set to 2000:

```
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

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 priority-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.

pragma translate_off/pragma translate_on Directive

Directive. Allows you to synthesize designs originally written for use with other synthesis tools without needing to modify source code. All source code that is between these two directives is ignored 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 pragma translate_off/translate_on to skip over simulation-specific lines of code that are not synthesizable.

When you use pragma translate_off in a module, synthesis of all source code that follows is halted until pragma translate_on is encountered. Every pragma translate_off must have a corresponding pragma translate_on. These directives cannot be nested, therefore, the pragma translate_off directive can only be followed by a pragma translate_on directive.

Note: See also, translate_off/translate_on Directive, on page 1030. These directives are implemented the same in the source code.

Verilog Syntax and Example

The Verilog syntax for these directives is as follows:

/* pragma translate_off */

/* pragma translate_on */

For example:

```
module real_time (ina, inb, out);
input ina, inb;
output out;

/* pragma translate_off */
realtime cur_time;
/* pragma translate_on */

assign out = ina & inb;
endmodule
```

VHDL Syntax and Example

The following is the VHDL syntax for these directives:

pragma translate_off

pragma translate_on

For example:

```
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
signal a1:std_logic;

--pragma translate_off

constant a1:std_logic:='0';

--pragma translate_on

begin
   sum <= (a xor b xor cin);
   cout <= (a and b) or (a and cin) or (b and cin); end behave;
```

syn_allow_retiming Attribute

Attribute; The syn_allow_retiming attribute determines if registers can be moved across combinational logic to improve performance.

A value of 1 (true) allows moving registers during retiming, and a value of 0 (false) ensures that registers are not moved. Typically, you enable the Retiming option and use the syn_allow_retiming attribute to disable retiming for specific objects that you do not want moved. Note, do not use the syn_allow_retiming attribute with the Fast Synthesis flow. The attribute can be applied either globally or to specific registers.

Constraint File Syntax and Example

define_attribute {*register*} syn_allow_retiming {1|0}

define_global_attribute syn_allow_retiming {1|0}

For example:

```
define_attribute {reg1} syn_allow_retiming {0}
```

Verilog Syntax and Examples

object **/* synthesis syn_allow_retiming = 0 | 1 */ ;**

where *object* is a register.

Here is an example of applying it to a register:

```
reg [7:0] app_reg /* synthesis syn_allow_retiming=0 */
```

VHDL Syntax and Example

attribute syn_allow_retiming of *object* **:** *objectType* **is true | false ;**

where *object* is a register. The data type is Boolean. Here is an example of applying it to a register:

```
signal app_reg : std_logic_vector (7 downto 0);
attribute syn_allow_retiming : boolean;
attribute syn_allow_retiming of app_reg : signal is true;

-- Other code
```

See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

syn_black_box Directive

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. 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 synthesis tool provides pre-defined black boxes for the vendor macros.

A module can be a black box whether or not it is empty. In mixed language designs, if you have a black box defined in one language at the top level but also have a description for it in another language (Verilog or VHDL,) the tool checks the existing black box definition. If the tool can replace the black box declaration with the description from the other language, it does not use the syn_black_box directive you set. If you truly want a black box in your design, do one of the following:

- Set a syn_black_box directive on the module or entity in the HDL file that contains the description, not at the top level.
- If your project includes black box descriptions in .srs, .ngc, or .edf formats, the tool uses these black box descriptions even if you have specified syn_black_box at the top level. If you want to black box such a module, you must remove its .srs or .edf description from the project.

Once you define a black box with syn_black_box, you use other source-code directives to define timing for the black box. You must add the directives to the source code because the models are specific to individual instances. There are no corresponding Tcl directives you can use in a constraint file.

Black-box Source Code Directives

Use the following directives with syn_black_box to characterize black-box timing:

syn_isclock Directive	Specifies a clock port on a black box.
syn_tco<n> Directive	Sets timing clock to output delay through the black box.
syn_tpd<n> Directive	Sets timing propagation for combinational delay through the black box.
syn_tsu<n> Directive	Sets timing setup delay required for input pins relative to the clock.

Black Box Pin Definitions

You define the pins on a black box with these directives in the source code:

black_box_pad_pin Directive	Indicates that a black box is an I/O pad for the rest of the design.
black_box_tri_pins Directive	Indicates tristates on black boxes.

For more information on black boxes, see Instantiating Black Boxes in Verilog, on page 542, and Instantiating Black Boxes in VHDL, on page 744.

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* **:** *objectType* **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. See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

```
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 coding
```

The syn_black_box directive can also be specified on the architecture of an entity which must be defined as a black box as shown below:

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

entity bbx is
   port (bbx_clk : in std_logic;
   bbx_in : in std_logic;
   result : out std_logic );
attribute syn_isclock : boolean;
attribute syn_isclock of bbx_clk: signal is true;
end bbx;

architecture bbx of bbx is
attribute syn_black_box : boolean;
attribute syn_black_box of bbx : architecture is true;

begin
end bbx;

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

```
entity bbx_parent is
   port (data, bob: in std_logic;
         q: out std_logic );
end bbx_parent;

architecture bbx_parent of bbx_parent is
component bbx
   port (bbx_clk : in std_logic;
         bbx_in : in std_logic;
         result : out std_logic );
end component;

begin
-- Simple component instantiation
bbx1: bbx
   port map(bbx_clk => bob, bbx_in => data, result => q);
end bbx_parent;
```

syn_direct_enable Attribute/Directive

Attribute; Microsemi Axcelerator only. 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.

As a directive, you use syn_direct_enable 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.

Constraint File Syntax and Examples

define_attribute {*object***} syn_direct_enable {1}**

For example:

```
define_attribute {n:e2} {syn_direct_enable} {1}
```

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* **:** *objectType* **is true;**

See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

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

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 84, for more information on FSM Compiler.

Note the following:

- You cannot set this attribute at a global level.
- For the encoding specified with this attribute to take effect, the design must contain state machines that have been inferred by the FSM Compiler. Setting this attribute when syn_state_machine is set to 0 will have no effect.
- The encoding specified by this attribute apply to the final mapped netlist. For other kinds of enumerated encoding, use syn_enum_encoding. See syn_enum_encoding Directive, on page 931 and syn_encoding Compared to syn_enum_encoding, on page 935 for more information.

The default encoding style automatically assigns encoding based on the number of states in the state machine (see the table in Values for syn_encoding, on page 927 for details). 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 for just those specific registers.

Values for syn_encoding

Value	Description
default	The mapper automatically picks an encoding style that results in the best performance. To ensure that a particular encoding style is used, explicitly specify that style.
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 Because onehot 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 slower than a gray style if you have a large output decoder following a state machine.

Value	Description
gray	More than one of the state registers can be hot. The 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.
original	This respects the encoding you set, but the software still does state machine and reachability analysis.

Encoding Style Implementation

The encoding style is implemented during the mapping phase. A message appears when the synthesis tool extracts a state machine, for example:

```
@N: CL201 : "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. This information is also available in the FSM Viewer (see FSM Viewer Window, on page 74).

See also the following:

- For information on enabling state machine optimization for individual modules, see syn_state_machine Directive, on page 1010.
- For VHDL designs, see syn_encoding Compared to syn_enum_encoding, on page 935 for comparative usage information.

Constraint File Syntax and Examples

The constraint file syntax for the attribute is

define_attribute {*object***} syn_encoding {onehot|gray|safe|sequential|original}**

The *object* must be an instance, with the syntax **i**:*instance*. The instance must be a sequential instance with a view name of statemachine.

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. For descriptions of these values, see Values for syn_encoding, on page 927.

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 synthesis tool will reset the values to a valid state.

If you specify the syn_encoding attribute in Verilog, all instances of that FSM use the same syn_encoding value. To have unique syn_encoding values for each FSM instance, use different entities or modules, or specify the syn_encoding attribute in the sdc file.

VHDL Syntax and Example

attribute syn_encoding of *object* **:** *objectType* **is "***string***" ;**

where *object* is a signal that holds the state values of the state machines. See Values for syn_encoding, on page 927 for descriptions of the values for this attribute.

Here is an example of using syn_encoding to define the gray encoding style for the signal s1. See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

```
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;
```

If you specify the syn_encoding attribute in VHDL, all instances of that FSM use the same syn_encoding value. To have unique syn_encoding values for each FSM instance, use different entities or modules, or specify the syn_encoding attribute in the fdc file.

syn_enum_encoding Directive

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, this directive has no effect on the encoding styles of extracted state machines; the tool uses the values specified in 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 to syn_enum_encoding, on page 935 for more information. For step-by-step details about setting coding styles with this attribute see Defining State Machines in VHDL, on page 178 of the *User Guide*.

Values for syn_enum_encoding

Values for syn_enum_encoding are as follows:

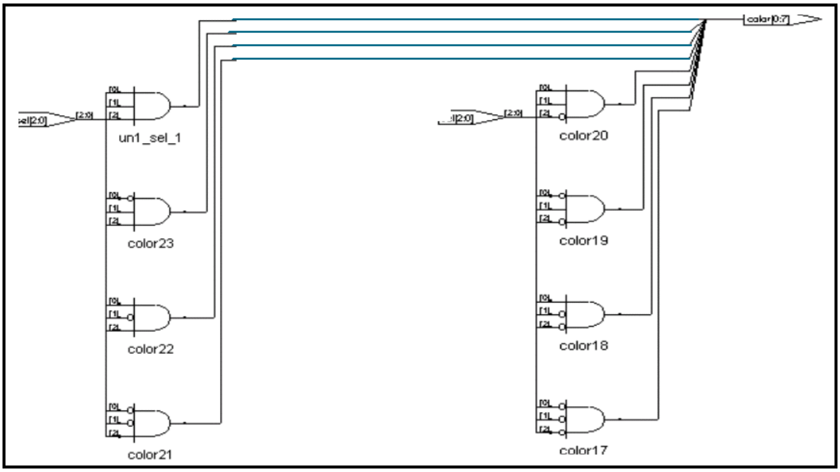
- default – Automatically assigns an encoding style that results in the best performance.
- sequential – More than one bit of the state register can change at a time, but because more than one bit can be hot, the value must be decoded to determine the state. For example: 000, 001, 010, 011, 100
- onehot – 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 935.

A message appears in the log file when you use the syn_enum_encoding directive; for example:

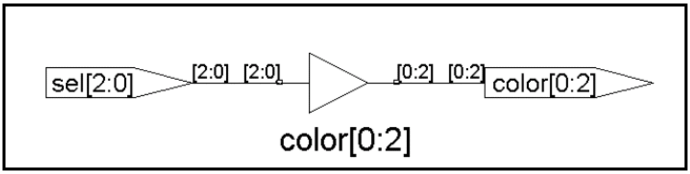
```
@N: CD231: "c:\design\..":17:11:17:12|Using onehot encoding for
type mytype (red="10000000")
```

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 eight 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* **:** *objectType* **is "***value***" ;**

Where *object* is an enumerated type and *value* is one of the following: default, sequential, onehot or gray. See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

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_enum_encoding, enum_encoding, and syn_encoding

Custom attributes are attributes that are not defined in the IEEE specifications, but which you or a tool vendor define for your own use. They provide a convenient back door in VHDL, and are used to better control the synthesis and simulation process. enum_encoding is one of these custom attributes that is widely used to allow specific binary encodings to be attached to objects of enumerated types.

The enum_encoding attribute is declared as follows:

```
attribute enum_encoding: string;
```

This can be either written directly in your VHDL design description, or provided to you by the tool vendor in a package. Once the attribute has been declared and given a name, it can be referenced as needed in the design description:

```
type statevalue is (INIT, IDLE, READ, WRITE, ERROR);
attribute enum_encoding of statevalue: type is
   "000 001 011 010 110";
```

When this is processed by a tool that supports the enum_encoding attribute, it uses the information about the statevalue encoding. Tools that do not recognize the enum_encoding attribute ignore the encoding.

Although it is recommended that you use syn_enum_encoding, the Synopsys FPGA tools recognize enum_encoding and treat it just like syn_enum_encoding. The tool uses the specified encoding when the FSM compiler is disabled, and ignores the value when the FSM Compiler is enabled.

If enum_encoding and syn_encoding are both defined and the FSM compiler is enabled, the tool uses the value of syn_encoding. If you have both syn_enum_encoding and enum_encoding defined, the value of syn_enum_encoding prevails.

syn_encoding Compared to 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_global_buffers Attribute

Attribute; Microsemi ProASIC3E. Specifies the number of global buffers to be used in a design. The 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/entity or view. For Microsemi designs, it can be any integer between 6 and 18. If you specify an integer less than 6, the software infers six global buffers.

Constraint File Syntax and Example

define_attribute {*view***} syn_global_buffers {***maximum***}**

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* **:** *objectType* **is** *maximum***;**

See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

```
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

Attribute. Lets you control the amount of hierarchical transformation across boundaries on module or component instances during optimization.

During synthesis, the synthesis tool dissolves as much hierarchy as possible to allow efficient logic optimization across hierarchical boundaries while maintaining optimal run times. The tool then rebuilds the hierarchy as close as possible to the original source to preserve the topology of the design. Use the syn_hier attribute to address specific needs to maintain the original design hierarchy during optimization. This attribute gives you manual control over flattening/preserving instances, modules, or architectures in the design.

Constraint File Syntax and Example

define_attribute {*object***} syn_hier {***value***}**

where *object* is a view, and *value* can be any of the values described in syn_hier Values, on page 942. Note however, if you are defining syn_hier globally, it is recommended that you use the SCOPE collection to apply syn_hier on all views instead. For example:

```
define_scope_collection all_views {find -hier -view {*}}
```

```
define_attribute {$all_views} {syn_hier} {fixed}
```

Check the attribute values to determine where to attach the attribute. Here is an example:

```
define_attribute {v:fifo} syn_hier {hard}
```

Make sure to specify the attribute on the view (**v:** object type). See syn_hier in the SCOPE Window, on page 943 for details.

Verilog Syntax and Examples

object **/* synthesis syn_hier = "***value***" */ ;**

where *object* can be a module declaration and *value* can be any of the values described in syn_hier Values, on page 942. Check the attribute values to determine where to attach the attribute.

This is the Verilog syntax:

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

// Other code
```

VHDL Syntax and Examples

attribute syn_hier of *object* **: architecture is "***value***" ;**

where *object* is an architecture name and *value* can be any of the values described in syn_hier Values, on page 942. Check the attribute values to determine the level at which to attach the attribute.

See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives. This is the VHDL syntax:

```
architecture struct of cpu is

attribute syn_hier : string;
attribute syn_hier of struct: architecture is "hard";

-- Other code
```

syn_hier Values

The following table shows the values you can use for syn_hier. For additional information about using this attribute in HDL Analyst, see Controlling Hierarchy Flattening, on page 209 and Preserving Hierarchy, on page 209 in the *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. However, when there is cell packing across the boundary, it changes the interface and does not guarantee the exact RTL interface. This attribute affects only the design unit in which it is specified.
hard	Preserves the interface of the design unit and prevents most optimizations across the hierarchy. However, the boundary optimization for constant propagation is performed. This attribute affects only the specified design units.

fixed	Preserves the interface of the design unit with no exceptions. Fixed prevents all optimizations performed across hierarchical boundaries and retains the port interfaces as well. For more information, see Using syn_hier fixed, on page 944.
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 Technology view schematics.
macro	Preserves the interface and contents of the design with no exceptions. This value can only be set on structural netlists. (In the .fdc file, or using the SCOPE editor, set syn_hier to macro on the view (the **v:** object type).
flatten	Flattens the hierarchy of all levels below, but not the one where it is specified. This only affects synthesis optimization. The hierarchy is reconstructed in the netlist and Technology view schematics. To create a completely flattened netlist, use the syn_netlist_hierarchy attribute (syn_netlist_hierarchy Attribute, on page 963), set to false. You can use flatten in combination with other syn_hier values; the effects are described in Using syn_hier flatten with Other Values, on page 946. If you apply syn_hier to a compile point, flatten is the only valid attribute value. All other values only apply to the current level of hierarchy. The compile point hierarchy is determined by the type of compile point specified, so a syn_hier value other than flatten is redundant and is ignored.

syn_hier in the SCOPE Window

If you use the SCOPE window to specify the syn_hier attribute, do not drag and drop the object into the SCOPE spreadsheet. Instead, first select syn_hier in the Attribute column, and then use the pull-down menu in the Object column to select the object. This is because you must set the attribute on a view (v:). If you drag and drop an object, you might not get a view object. Selecting the attribute first ensures that only the appropriate objects are listed in the Object column.

Using syn_hier fixed

When you use the fixed value with syn_hier, hierarchical boundaries are preserved with no exceptions. For example, optimizations such as constant propagation are not performed across these boundaries.

Note: It is recommended that you do not use syn_hier with the fixed value on modules that have ports driven by tri-state gates. For details, see When Using Tri-states, on page 944.

When Using Tri-states

It is advised that you avoid using syn_hier="fixed" with tri-states. However, if you do, here is how the software handles the following conditions:

- Tri-states driving output ports

  If a module with syn_hier="fixed" includes tri-state gates that drive a primary output port, then the synthesis software retains a tri-state buffer so that the P&R tool can pack the tri-state into an output port.

- Tri-states driving internal logic

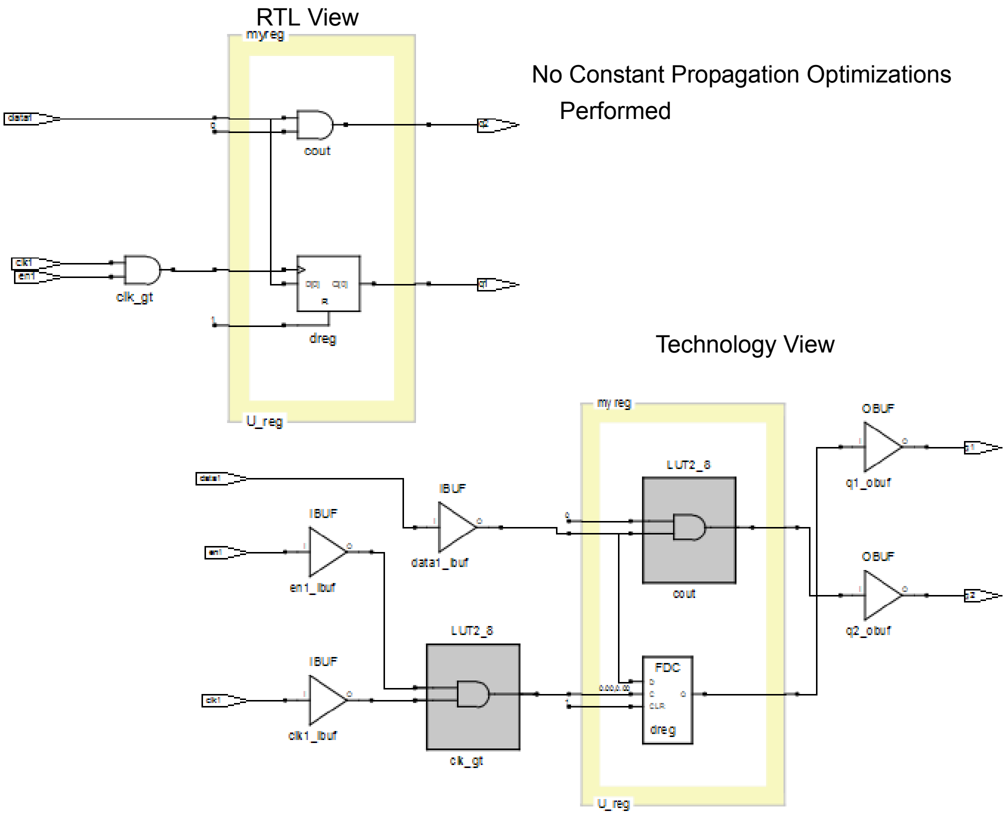
  If a module with syn_hier="fixed" includes tri-state gates that drive internal logic, then the synthesis software converts the tri-state gate to a MUX and optimizes within the module accordingly.

In the following code example, myreg has syn_hier set to fixed.

```
module top(
   clk1,en1, data1,
   q1, q2
   );

input clk1, en1;
input data1;
output q1, q2;

wire cwire, rwire;
wire clk_gt;

assign clk_gt = en1 & clk1;

// Register module
```

```
myreg U_reg (
   .datain(data1),
   .rst(1'b1),
   .clk(clk_gt),
   .en(1'b0),
   .dout(rwire),
   .cout(cwire)
   );

assign q1 = rwire;
assign q2 = cwire;

endmodule


module myreg (
   datain,
   rst,
   clk,
   en,
   dout,
   cout
   ) /* synthesis syn_hier = "fixed" */;

input clk, rst, datain, en;
output dout;
output cout;

   reg dreg;

   assign cout = en & datain;

   always @(posedge clk or posedge rst)
      begin
         if (rst)
            dreg <= 'b0;
         else
            dreg <= datain;
      end

assign dout = dreg;

endmodule
```

The HDL Analyst views show that myreg preserves its hierarchical boundaries without exceptions and prevents constant propagation optimizations.



RTL View

No Constant Propagation Optimizations Performed

Technology View

Using syn_hier 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.

These example demonstrate the use of the flatten and remove values to flatten the current level of the hierarchy and all levels below it (unless you have defined another syn_hier attribute at a lower level).

Verilog

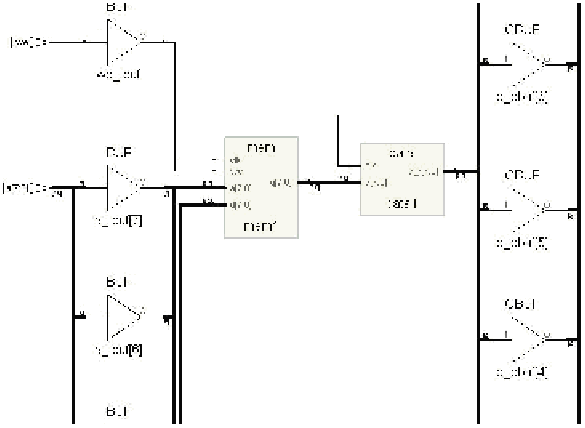
```
module top1 (Q, CLK, RST, LD, CE, D)
/* synthesis syn_hier = "flatten,remove" */;

// Other code
```

VHDL

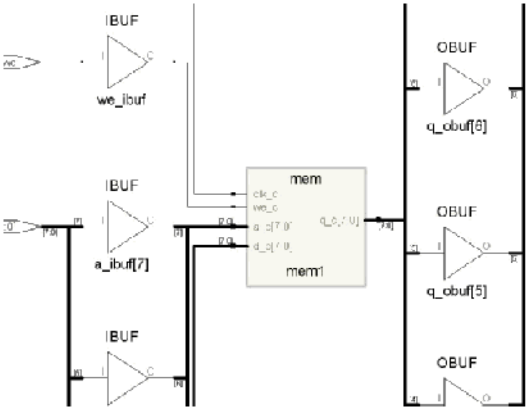
```
architecture struct of cpu is

attribute syn_hier : string;
attribute syn_hier of struct: architecture is "flatten,remove";

-- Other code
```

Example of syn_hier hard

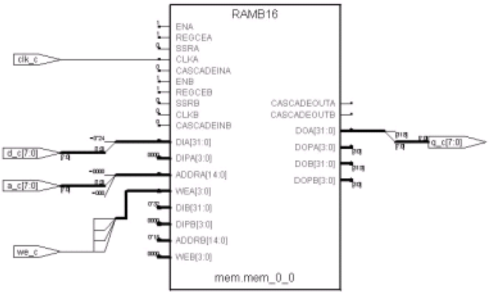
Here is an example of two versions of a design: one with syn_hier set on modules mem and data; the other shows what happens to those modules with the automatic flattening that occurs during synthesis.



With syn_hier=”hard”



Without syn_hier="hard"
This is the default.



syn_isclock Directive

Directive. Used with the syn_black_box directive and 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.

The syn_isclock directive is one of several directives that you can use with the syn_black_box directive to define timing for a black box. See syn_black_box Directive, on page 921 for a list of the associated directives.

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*: *objectType* **is true ;**

where *object* is a black-box input port.

See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

```
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

Directive. Keeps the specified net intact during optimization and 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 951 for an example). However the buffer is not part of the final netlist, so no extra logic is generated. There are various situations where this directive is useful:

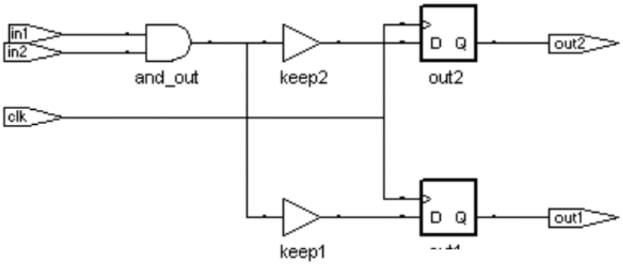
- To preserve a net that would otherwise be removed as a result of optimization. You might want to preserve the net for simulation results or to obtain a different synthesis implementation.
- To prevent duplicate cells from being merged during optimization. You apply the directive to the nets connected to the input of the cells you want to preserve.
- As a placeholder to apply the -through option of the define_multicycle_path or define_false_path timing constraint. This allows you to specify a unique path as a multiple-cycle or false path. Apply the constraint to the keep buffer.
- To prevent the absorption of a register into a macro. If you apply syn_keep to a reg or signal that will become a sequential object, the tool keeps the register and does not absorb it into a macro.

Note: Do not use the syn_keep directive on nets with SystemVerilog data types, such as bit, logic, longint, or shortint. For more information, see Data Types, on page 561.

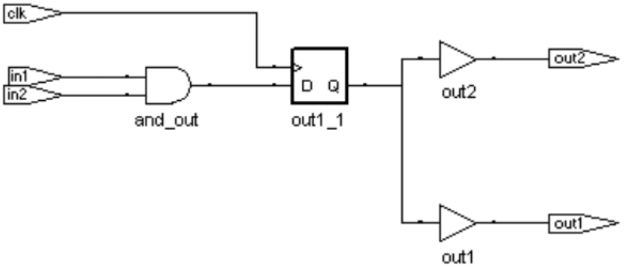
Effects of Using syn_keep

When you use syn_keep, the tool retains duplicate logic instead of optimizing it away. The following figure shows the technology view for two versions of a design.

In the first, syn_keep is set on the nets connected to the inputs of the registers out1 and out2, to prevent sharing. The second figure shows the same design without syn_keep. Setting syn_keep on the input wires for the registers ensures that you get duplicate registered outputs for out1 and out2. If you do not apply syn_keep to keep1 and keep2, the software optimizes out1 and out2 to one register.



With syn_keep



Without syn_keep

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 works only on nets and combinational logic. It ensures that
  - The wire is kept during synthesis
  - No optimizations cross the wire

  This directive is usually used to break unwanted optimizations and to ensure manually created replications. When applied to a register, the register is preserved and not absorbed into a macro.
- syn_preserve ensures that registers are not optimized away. However, it allows the register to be absorbed into a macro. If you do not want this, use syn_keep on the register to preserve it outside the macro.
- 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).

Applying syn_keep on Multiple Nets in Verilog

The syn_keep attribute is only applied on net c, which is the last variable in the wire declaration or variable associated with the directive declaration in the following syntax.

```
wire a,b,c /* synthesis syn_keep=1 */;
```

To apply syn_keep to all the nets, use one of the following methods:

- Declare each individual net separately as shown below.

  ```
  wire a /* synthesis syn_keep=1 */;
  wire b /* synthesis syn_keep=1 */;
  wire c /* synthesis syn_keep=1 */;
  ```

- Use the Verilog 2001 Parenthetical Comments, to declare the syn_keep attribute as a single line statement.

  ```
  (* syn_keep=1 *) wire a,b,c;
  ```

  For more information, see Attribute Examples Using Verilog 2001 Parenthetical Comments, on page 553.

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 951.

```
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* **:** *objectType* **is true ;**

where *object* is a single or multiple-bit signal. See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

Here is the source code used to produce the schematics shown in Effects of Using syn_keep, on page 951.

```
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_loc Attribute

Attribute: Microsemi. Specifies pin locations for I/O pins and cores, and forward-annotates this information to the place-and-route tool.

This attribute can only be specified in a top-level source file or a constraint (`fdc`) file.

The PDC2SDC utility sets this attribute when it translates Microsemi PDC set_io and set_location constraints.

In the syntax, *pinNumbers* is a comma-separated list of pin or placement numbers. Refer to the vendor databook for valid values.

Constraint File Syntax and Examples

define_attribute {*portName***} syn_loc {***pinNumbers***}**

The following are examples of using this attribute:

Microsemi	`define_attribute {CR_DIN[3:0]} syn_loc {M7, Y6, B6, D10}`

You can also specify locations for individual bus bits with this attribute:

Microsemi

`define_attribute {CR_DIN[3]} syn_loc {M7}`
`define_attribute {CR_DIN[2]} syn_loc {Y6}`
`define_attribute {CR_DIN[1]} syn_loc {B6}`
`define_attribute {CR_DIN[0]} syn_loc {D10}`
	Specify the b: prefix and the bit slice:
	`define_attribute {b:CR_DIN[0]} syn_loc {D10}`
	`define_attribute {b:CR_DIN[1]} syn_loc {B6}`
	`define_attribute {b:CR_DIN[2]} syn_loc {Y6}`
	`define_attribute {b:CR_DIN[3]} syn_loc {M7}`

Verilog Syntax and Examples

object **/* synthesis syn_loc = "***pinNumbers***" */ ;**

The following shows pin assignments in Verilog:

Microsemi

```
module prep2_2 (...);
   .
   .
   .
input [3:0] CR_DIN /* synthesis syn_loc = "M7,Y6, B6, D10"
*/;
```

VHDL Syntax and Examples

attribute syn_loc of *object* **:** *objectType* **is "***pinNumbers***" ;**

See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

The following shows VHDL examples:

Microsemi

```
entity prep2_2 is
   port (CR_DIN : in std_logic_vector (3 downto 0);
      .
      .
      .
   );
attribute syn_loc of CR_DIN : signal is "M7,Y6, B6, D10";
end prep2_2;
```

syn_looplimit Directive (VHDL)

Directive. Specifies a loop iteration limit for while loops in the design when the loop index is a variable, not a constant. If your design requires a variable loop index, use the syn_looplimit directive to specify a limit for the compiler. If you do not, you can get a "while loop not terminating" compiler error. The limit cannot be an expression. The higher the value you set, the longer the runtime.

Verilog applications use the loop_limit directive (see loop_limit Directive, on page 914).

VHDL Syntax and Example

attribute syn_looplimit : *integer*;
attribute syn_looplimit of *labelName* **: label is** *value***;**

The following is an example where the loop limit is set to 5000:

```
library IEEE;
use std.textio.all;
use ieee.std_logic_textio.all;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity initram is
   port (rAddr, wAddr, dataIn  : in integer;
         clk: in bit;
         we : in bit;
         dataOut  : out integer );
end;

architecture rtl of initram is
subtype smallint is integer range 0 to 3000;
type intAry is array (0 to 3000) of smallint;
function load( name : string) return intAry is
attribute syn_looplimit : integer;
attribute syn_looplimit of myloop: label is 5000;
```

```
variable t : intAry ;
variable data : smallint ;
variable dataLine : line ;
variable i : natural ;
file dataFile : text open READ_MODE is name ;

begin
   myloop: while ( not endfile(dataFile) ) loop
      readline(dataFile,dataLine);
      read(dataLine,data);
      t(i) := data;
      i := i + 1;
   end loop myloop;

return t;
end load;

-----------------------------------------------------

signal ram : intAry := load("data.txt");
signal rAddr_reg : integer ;
begin
   process (clk) begin
      if (clk'event and clk='1') then
         rAddr_reg <= rAddr;
         if(we = '1') then
            ram(wAddr) <= dataIn;
         end if;
      end if;
end process;

dataOut <= ram(rAddr_reg);
end RTL ;
```

The data.txt file in the example is a large data file with each entry representing an iteration for the loop.

syn_maxfan Attribute

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 Implementation Options dialog box or with the set_option -fanout_limit command or -fanout_guide 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. The synthesis tool does not respect the syn_maxfan limit if the limit imposes constraints that interfere with optimization.

You can apply the syn_maxfan attribute to the following:

- Registers or instances. You can also apply it to a module or entity.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.
- Ports or nets. If you apply the attribute to a net, the synthesis tool creates a KEEPBUF component and attaches the attribute to it to prevent the net itself from being optimized away during synthesis .

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.

Constraint 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* **:** *objectType* **is "***value***" ;**

See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

```
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_multstyle Attribute

Attribute; This attribute determines how multipliers are implemented: as dedicated hardware DSP blocks or as logic. You can apply it globally or to modules of Microsemi devices.

Constraint File Syntax and Example

define_attribute {*object*} syn_multstyle {dsp|logic}

define_global_attribute syn_multstyle {dsp|logic}

This example specifies that multipliers are implemented as logic.

```
define_attribute {temp[15:0]} syn_multstyle {logic}
```

Verilog Syntax and Example

object **/* synthesis syn_multstyle = "dsp | logic" */ ;**

```
module mult(a,b,c,r,en);
input [7:0] a,b;
output [15:0] r;
input [15:0] c;
input en;
wire [15:0] temp /* synthesis syn_multstyle="logic" */;
assign temp = a*b;
assign r = en ? temp : c;
endmodule
```

VHDL Syntax and Example

attribute syn_multstyle of *object* **:** *objectType* **is "dsp | logic " ;**

See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

```
library ieee ;
use ieee.std_logic_1164.all ;
USE ieee.numeric_std.all;
```

```
entity mult is
   port (clk : in std_logic ;
         a : in std_logic_vector(7 downto 0) ;
         b : in std_logic_vector(7 downto 0) ;
         c : out std_logic_vector(15 downto 0)) ;
end mult ;

architecture rtl of mult is
signal mult_i : std_logic_vector(15 downto 0) ;
attribute syn_multstyle : string ;
attribute syn_multstyle of mult_i : signal is "logic" ;
begin
mult_i <= std_logic_vector(unsigned(a)*unsigned(b)) ;
   process(clk)
   begin
      if (clk'event and clk = '1') then
         c <= mult_i ;
      end if ;
   end process ;
end rtl ;
```

syn_netlist_hierarchy Attribute

Attribute. A global attribute that controls the generation of hierarchy in the output netlist (result file) when you assign it to the top-level module of your design. The default (true) allows hierarchy generation, and setting it to false flattens the hierarchy and results in a flattened output netlist.

For information about flattening techniques, see Controlling Hierarchy Flattening, on page 209 in the *User Guide*.

Constraint 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* **:** *objectType* **is true | false ;**

where *object* can be a top-level architecture name. The data type is Boolean. See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

```
architecture top of top is
attribute syn_netlist_hierarchy : boolean;
attribute syn_netlist_hierarchy of top : architecture is false;

-- Other code
```

syn_hier flatten and syn_netlist_hierarchy

The syn_hier=flatten attribute and the syn_netlist_hierarchy=false attributes both flatten hierarchy, but work slightly differently. Use the syn_netlist_hierarchy attribute if you want a completely flattened netlist, because this attribute flattens all levels of hierarchy. When you set syn_hier=flatten, you flatten the hierarchical levels below the component on which it is set, but you do not flatten the current hierarchical level where it is set. Refer to syn_hier Attribute, on page 941 for information about this attribute.

syn_noarrayports Attribute

Attribute. Specifies that the ports of a design unit be treated as individual signals (scalars), not as buses (arrays) in the output file.

Constraint 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* is 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* **:** *objectType* **is true | false ;**

where *object* is an architecture name. The data type is Boolean. See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

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

Attribute. The 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 nets or specific input ports. Set the Boolean value to 1 or true to turn off automatic clock buffering.

You can attach this attribute to a port or net in any hard architecture or module whose hierarchy will not be dissolved during optimization.

Constraint File Syntax and Example

define_attribute {*clock_port*} syn_noclockbuf {0|1}

define_global_attribute syn_noclockbuf {0|1}

For example:

```
define_attribute {clk} syn_noclockbuf {1}

define_global_attribute syn_noclockbuf {1}
```

Verilog Syntax and Examples

object **/* synthesis syn_noclockbuf = 1 | 0 */ ;**

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* **:** *objectType* **is true | false ;**

Where *object* is a hard architecture or an input (clock) port. See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

This example turns off automatic clock buffering for an entire architecture using the syn_noclockbuf attribute from the Synopsys FPGA 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.
```

The following example demonstrates turning off automatic clock buffering for an individual signal:

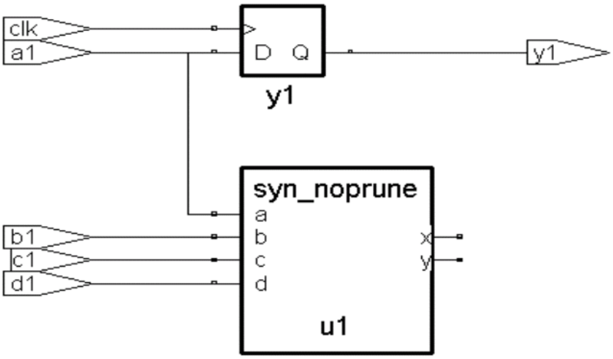
```
library ieee, synplify;
use ieee.std_logic_1164.all;

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_noprune Directive

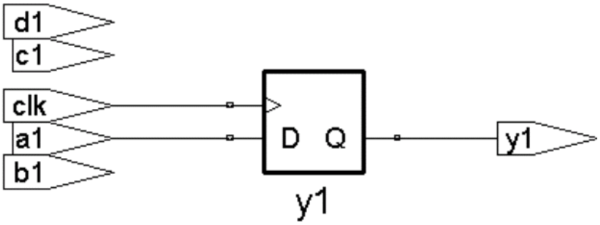
Directive. Prevents optimizations for instances and black-box modules (including technology-specific primitives) with unused output ports. During optimization, if a module does not drive any logic, it is removed by the synthesis tool. If you want to keep the instance of the module in the design, use 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 HDL Analyst view for two versions of a design: one version using syn_noprune on black-box instance U1, one version without syn_noprune.



With syn_noprune



Without syn_noprune

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 on nets and combinational logic.
- syn_preserve ensures that registers are not optimized away.
- syn_noprune ensures that an instance or black box is not optimized away when its outputs are unused (i.e., when its outputs do not drive any logic).

Using syn_noprune – Example 1

The following examples show the Verilog and VHDL source code used for the previous schematics.

Verilog Syntax and Example

object **/* synthesis syn_noprune = 1 */ ;**

where object is a module declaration or an instance. The data type is Boolean.

```
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* **:** *objectType* **is true ;**

where the data type is boolean, and *object* is an architecture, a component, or a label of an instantiated component. See Architectures, on page 971, Component Declaration, on page 972, Component Instance, on page 972, for details of the objects. See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

```
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 744, 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;
```

Using syn_noprune – Example 2

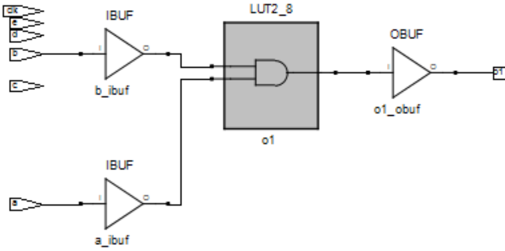
In the following Verilog code example, syn_noprune is applied on both an instance and black box module with unused outputs.

Verilog Example

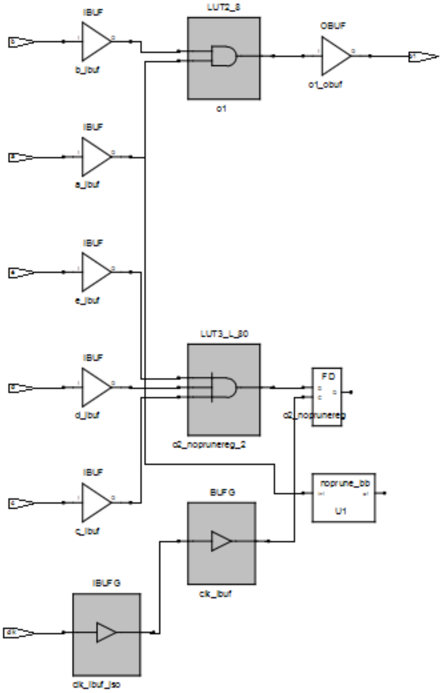
```
module top
   (input a, b, c, d, e, clk,
   output o1);
reg o2_noprunereg /* synthesis syn_noprune = 1*/ ;
wire o3_wire;
assign o1 = a & b;
always @(posedge clk)
   begin
      o2_noprunereg = c & d & e;
   end
noprune_bb U1 (a, o3_wire) /* synthesis syn_noprune = 1*/ ;
endmodule
module noprune_bb ( input in1, output o1 );
endmodule
```

The following Technology views show that the instance and black box module have not been optimized away when their outputs are unused if syn_noprune is applied.

Without syn_noprune



With syn_noprune

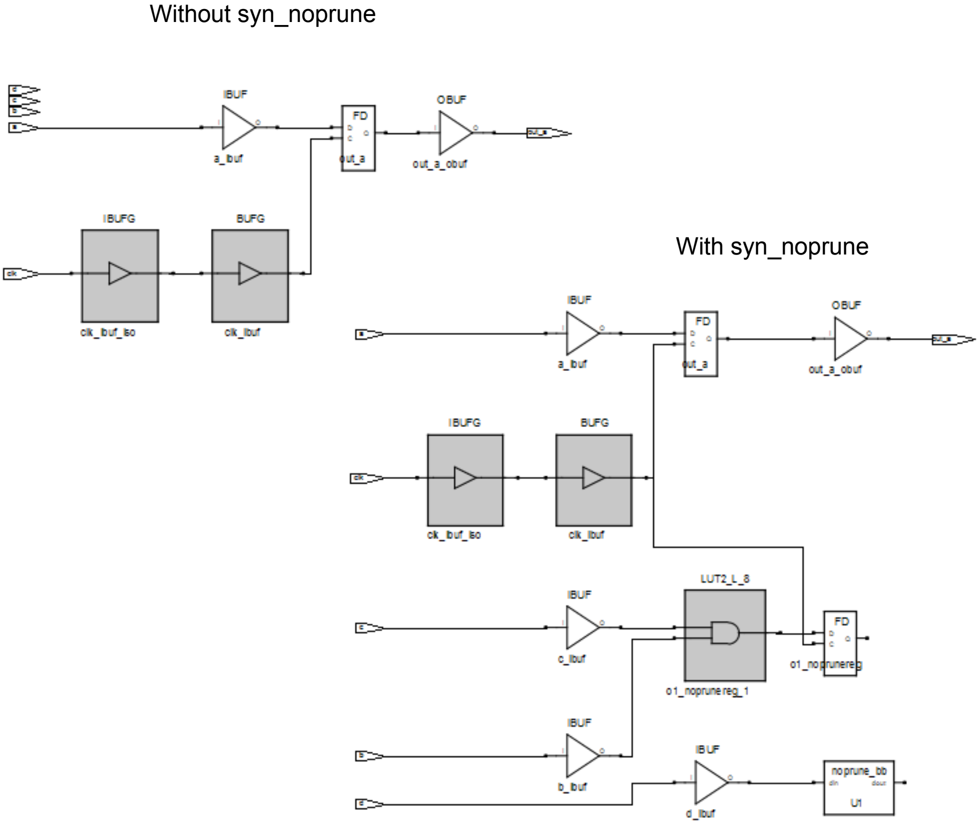


VHDL Example

In the following VHDL code example, syn_noprune is applied on both an instance and black box module with unused outputs.

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

entity top is
   port (
   clk : in  std_logic;
   a, b, c, d : in std_logic;
   out_a : out std_logic);
end entity top;

architecture arch of top is
   component noprune_bb
      port(
      din : in std_logic;
      dout : out std_logic);
   end component noprune_bb;

   signal o1_noprunereg : std_logic;
   signal o2_reg : std_logic;

   attribute syn_noprune : boolean;
   attribute syn_noprune of U1: label is true;
   attribute syn_noprune of o1_noprunereg : signal is true;

begin
   process(clk)
      begin
         if rising_edge(clk) then
            o1_noprunereg <= b and  c;
            out_a <= a;
      end if;
   end process;
   U1: noprune_bb port map (d, o2_reg);
end architecture arch;
```

The following Technology views show that the instance and black box module have not been optimized away when their outputs are unused if syn_noprune is applied.



Without syn_noprune

With syn_noprune

syn_pad_type Attribute

Attribute; Microsemi (Axcelerator, ProASIC families). Specifies an I/O buffer standard. Refer to I/O Standards, on page 380 and to the vendor-specific documentation for a list of I/O buffer standards available for the selected device family.

Constraint File Syntax

define_io_standard -default_*portType***|{***port***} -delay_type** *portType*
syn_pad_type {*io_standard***}**

-default_*portType*	*PortType* can be input, output, or bidir. Setting default_input, default_output, or default_bidir causes all ports of that type to have the same I/0 standard applied to them.
-delay_type *portType*	*PortType* can be input, output, or bidir.
syn_pad_type {*io_standard***}**	Specifies I/O standard (see following table).

.fdc File Examples

To set...	Use this syntax...
The default for all input ports to the AGP1X pad type	`define_io_standard -default_input -delay_type input syn_pad_type {AGP1X}`
All output ports to the GTL pad type	`define_io_standard -default_output -delay_type output syn_pad_type {GTL}`
All bidirectional ports to the CTT pad type	`define_io_standard -default_bidir -delay_type bidir syn_pad_type {CTT}`

The following are examples of pad types set on individual ports. Note that you cannot assign pad types to bit slices.

```
define_io_standard {in1} -delay_type input
   syn_pad_type {LVCMOS_15}

define_io_standard {out21} -delay_type output
   syn_pad_type {LVCMOS_33}

define_io_standard {bidirbit} -delay_type bidir
   syn_pad_type {LVTTL_33}
```

Verilog Syntax

object /* **synthesis syn_pad_type =** *io_standard* */ **;**

See the tables under I/O Standards, on page 380 for a list of the valid I/O standards.

VHDL Syntax

attribute syn_pad_type of *object*: *objectType* **is** *io_standard*;

See the tables under I/O Standards, on page 380 for a list of the valid I/O standards. See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

syn_preserve Directive

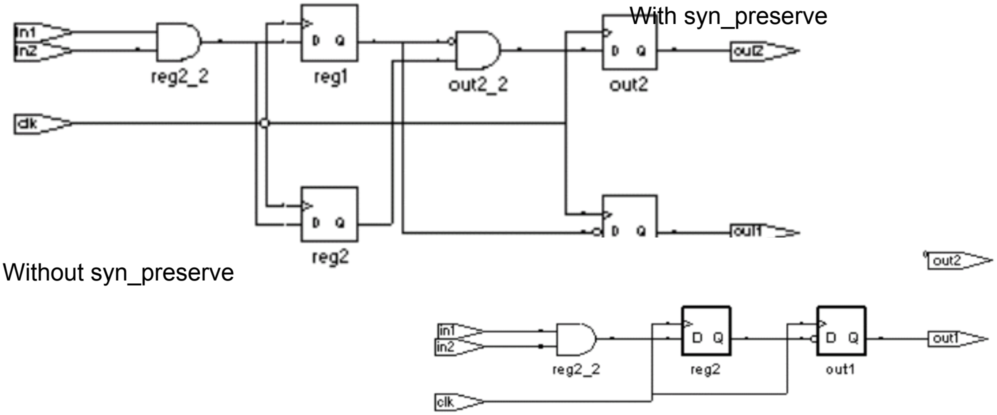
Directive. Prevents sequential optimization such as constant propagation, inverter push-through, and FSM extraction.

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 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.



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 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_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 Verilog source code used for the schematics at the beginning of the syn_preserve attribute description.

```
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* **:** *objectType* **is true ;**

where *object* is an output port or an internal signal that holds the value of a state register or architecture. See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

```
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 VHDL source code used for the schematics at the beginning of the syn_preserve attribute description.

```
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_preserve_sr_priority Attribute

Attribute; Microsemi ACT 1 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 Microsemi 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 synthesis tool automatically add hardware to force the priority that you have coded in the source code of your design.

syn_pipeline Summary

.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* **:** *objectType* **is true ;**

Where *object* is the entity. See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

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_probe Attribute

Attribute. Works as a debugging aid, inserting probe points for testing and debugging the internal signals of a design, causing them to appear as ports at the top level. Use of this attribute implies the use of syn_keep, too.

You can specify values to name probe ports and assign pins to named ports for selected technologies. Pin-locking properties of probed nets will be transferred to the probe port and pad. If empty square brackets [] are used, probe names will be automatically indexed, according to the index of the bus being probed.

The table below shows how to apply syn_probe values to nets, buses, and bus slices. It indicates what port names will appear at the top level. When the syn_probe value is 0, probe generation is disabled; when syn_probe is 1, the probe port name is derived from the net name.

Net Name	syn_probe Value	Probe Port Name	Comments
n:ctrl	1	ctrl_probe_1	Probe port name generated by the synthesis tool.
n:ctr	test_pt	test_pt	For string values on a net, the port name is identical to the syn_probe value.
n:aluout[2]	test_pt	test_pt	For string values on a bus slice, the port name is identical to the syn_probe value.
n:aluout[2]	test_pt[ ]	test_pt[2]	The empty, square brackets [ ] in the syn_probe value indicate that port names will be indexed to net names.
n:aluout[2:0]	test_pt[ ]	test_pt[2] test_pt[1] test_pt[0]	The empty, square brackets [ ] in the syn_probe value indicate that port names will be indexed to net names.
n:aluout[2:0]	test_pt	test_pt, test_pt_0, test_pt_1	If a syn_probe value without brackets is applied to a bus, the port names are adjusted.

Constraint File Syntax and Example

define_attribute {n:*netName*} syn_probe {*probePortname*|1|0}

The following examples show how to insert a probe signal into a net and assign pin locations to the ports.

```
define_attribute {n:inst2.DATA0_*[7]} syn_probe {test_pt[]}
define_attribute {n:inst2.DATA0_*[7]} syn_loc
   {14,12,11,5,21,18,16,15}
```

To insert a probe into the design using the spreadsheet, do the following:

1. In an RTL view, select the net in which you want to insert the probe point, then drag and drop the net into the Object column under the Attributes panel.
2. Click the Enabled box.
3. Choose syn_probe from the list of attributes in the corresponding Attribute column. Enter a value for the attribute (1, 0, or an alphanumeric string) in the adjacent Value column (see the previous table value formats).
4. Save the constraint file (close the spreadsheet) and add the file to your project. After synthesis, you can use the Find command in the Technology view to locate the probe ports and output pads.

Verilog Syntax and Example

object /* **synthesis syn_probe = "***string***" | 1 | 0 */ ;**

where *object* is a signal.

The following example shows how to insert probes on bus alu_tmp [7:0] and assign pin locations to each of the ports inserted for the probes.

```
module alu(out1, opcode, clk, a, b, sel);
output [7:0] out1;
input [2:0] opcode;
input [7:0] a, b;
input clk, sel;
reg [7:0] alu_tmp /* synthesis syn_probe="alu1_probe[]"
   syn_loc="A5,A6,A7,A8,A10,A11,A13,A14" */;
reg [7:0] out1;
// Other code
always @(opcode or a or b or sel)
begin
   case (opcode)
      3'b000:alu_tmp <= a+b;
      3'b000:alu_tmp <= a-b;
      3'b000:alu_tmp <= a^b;
      3'b000:alu_tmp <= sel ? a:b;
      default:alu_tmp <= a|b;
   endcase
end

always @(posedge clk)
out1 <= alu_tmp;
endmodule
```

VHDL Syntax and Example

attribute syn_probe of *object* **: signal is "***string***" | 1 | 0 ;**

where *object* is a signal. See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

The following example shows how to insert probes on bus alu_tmp(7 downto 0) and assign pin locations to each of the ports inserted for the probes.

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

entity alu 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;
         alu_out : out std_logic_vector(7 downto 0) );
end alu;

architecture rtl of alu is
signal alu_tmp : std_logic_vector (7 downto 0);
attribute syn_probe : string;
attribute syn_probe of alu_tmp : signal is "alu2_probe";
attribute syn_loc : string;
attribute syn_loc of alu_tmp : signal is
   "A5,A6,A7,A8,A10,A11,A13,A14";

begin
-- other code
end rtl;
```

syn_radhardlevel Attribute

Attribute; Microsemi Anti-fuse (RT, RH and RD radhard devices), ProASIC3E and later devices.

For details about support for the syn_radhardlevel attribute, see Microsemi Support, on page 990.

Microsemi Support

For Microsemi designs, specifies the radiation-resistant technique to use on an object. You can apply syn_radhardlevel globally to the top-level module/architecture or a register output signal (inferred register in VHDL), and is used in conjunction with the Microsemi macro files supplied with the software. The tmr option of this attribute can potentially affect the area and timing quality of results (QoR) for the design because of the additional registers or modules inserted. Be sure to keep this in mind when enabling the TMR feature and set the attribute appropriately for the area and timing goals for your design.

Some techniques are not available or appropriate for all Microsemi families. The design technique must be one that is valid for the project. Contact Microsemi technical support for more information. You can also control the design technique to apply on individual registers. For more details about setting this attribute, see Usage for Microsemi Designs, on page 993. See also Working with Radhard Designs, on page 347 in the *User Guide*.

Attribute Parameters

Values for syn_radhardlevel with Microsemi devices are as follows:

Value	Description
none	Standard design techniques are used; no triple register logic is inserted. This is the default. Note: For Microsemi designs, specify the value of none for Microsemi Anti-fuse and ProASIC3E devices. This is the default.
cc	Combinational cells with feedback are used to implement storage rather than flip-flop or latch primitives. Specify the value of cc for Microsemi Anti-fuse devices only.
tmr	Triple modular 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. Note: For Microsemi designs, specify the value of tmr for Microsemi Anti-fuse and ProASIC3E devices.
tmr_cc	Triple modular redundancy is used where each voting register is composed of combinational cells with feedback rather than flip-flop or latch primitives Specify the value of tmr_cc for Microsemi Anti-fuse devices only.

Constraint File Syntax and Example

The following syntax applies for Microsemi designs:

define_attribute {*object*} syn_radhardlevel {none|cc|tmr|tmr_cc}

where *object* is a module/architecture/register. For example:

```
define_attribute {dataout[3:0]} syn_radhardlevel {tmr}
```

Verilog Syntax and Example

The following syntax applies for Microsemi designs:

object /* **synthesis syn_radhardlevel ="none|cc|tmr|tmr_cc" */ ;**

Where *object* is an output signal. For example:

```
//Top level
module top (clk, dataout, a, b);
input clk;
input a;
input b;
output [3:0] dataout;
M1 inst_M1 (a1, M3_out1, clk, rst, M1_out);
// Other code

//Sub modules subjected to DTMR
module M1 (a1, a2, clk, rst, q)
   /* synthesis syn_radhardlevel="distributed_tmr" */;
input clk;
input signed [15:0] a1,a2;
input clk, rst;
output signed [31:0] q;
// Other code
```

VHDL Syntax and Example

The following syntax applies for Microsemi designs:

attribute syn_radhardlevel of *object* **:** *objectType* **is "none|cc|tmr|tmr_cc" ;**

Where *object* is a module/architecture/register. *objectType* is architecture or signal. See VHDL Attribute and Directive Syntax, on page 746 for alternate methods for specifying VHDL attributes and directives.

```
library synplify;
architecture top of top is
attribute syn_radhardlevel : string;
attribute syn_radhardlevel of top: architecture is "tmr";

-- Other code
```

Usage for Microsemi Designs

Apply the syn_radhardlevel attribute to a module or architecture in the source code or in the Attributes panel of the SCOPE window. For a register, however, you can only apply this attribute in the source code.

To set attributes in SCOPE, see How Attributes and Directives are Specified, on page 894.

The following procedure outlines how to set this attribute in the source code.

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

VHDL

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

Verilog

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

Note: Add the appropriate Microsemi macro file (tmr.v or tmr.vhd) to the project. The macro files are in the *installDirectory*/lib/microsemi.For ProASIC3E devices only, you do not need to add the Microsemi macro file to your project.

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

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

VHDL	Verilog

`library synplify;`
`use synplify.attributes.all;`
`attribute syn_radhardlevel of`
`   bl_int: signal is "tmr"`
	`reg [15:0] a1_int, b1_int`
	`/* synthesis syn_radhardlevel`
	`= "tmr" */;`

- Add the appropriate Microsemi macro file (tmr.v or tmr.vhd for this example) to the project.

3. For ProASIC3E devices only, you do not need to add the Microsemi macro file to your project.To set a global or default value, make sure that the corresponding Microsemi macro file is the first file listed in the project, if required.

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

syn_ramstyle Attribute

Attribute. The syn_ramstyle attribute specifies the implementation to use for an inferred RAM. You can apply syn_ramstyle globally, to a module, or to a RAM instance. To turn off RAM inferencing, set the value to registers.

The values for syn_ramstyle vary with the technology being used. The following table lists all the valid syn_ramstyle values, some of which apply only to certain technologies. See Values for syn_ramstyle by Technology, on page 996 for information about the values and their corresponding RAM implementations in different technologies. For details about using syn_ramstyle, see Specifying RAM Implementation Styles, on page 187 in the *User Guide*.

Values for syn_ramstyle

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. By default, the software uses deep block RAM configurations instead of wide configurations to get better timing results. Using the deeper RAMs reduces the output data delay timing by reducing the MUX logic at the output of the RAMs. With this option, the software also does not use the parity bit for data by default.
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.

no_rw_check	You can use this option independently or in conjunction with a RAM type value such as M512, or with the power value for supported technologies. When enabled, the synthesis tool does not insert bypass logic around the RAM. If you know your design does not read and write to the same address simultaneously, use no_rw_check to eliminate bypass logic. You can use this option to selectively not insert bypass logic on the RAM for read write checks, which is only used when the Read Write Check on RAM option is turned on (enabled) on the Device tab of the Implementation Options panel. Otherwise, this option is not honored. Use this value when you cannot simultaneously read and write to the same RAM location and you want to minimize overhead logic. *Note:* The no_rw_check and rw_check options for the syn_ramstyle attribute are mutually exclusive and must not be used together. Whenever synthesis conflicts exist, the software uses the following order of precedence: first the syn_ramstyle attribute, the syn_rw_conflict attribute, and then the Read Write Check on RAM option on the Implementation Option panel.
rw_check	When enabled, the synthesis tool inserts bypass logic around the RAM to prevent a simulation mismatch between the RTL and post-synthesis simulations. If you know your design has RAM that read and write to the same address simultaneously, use rw_check to insert bypass logic. You can use this option to selectively generate bypass logic on the RAM for read write checks, which is only used when the Read Write Check on RAM option is turned off (disabled) on the Device tab of the Implementation Options panel. Otherwise, this option is not honored. *Note:* The no_rw_check and rw_check options for the syn_ramstyle attribute are mutually exclusive and must not be used together. Whenever synthesis conflicts exist, the software uses the following order of precedence: first the syn_ramstyle attribute, the syn_rw_conflict attribute, and then the Read Write Check on RAM option on the Implementation Option panel.

Values for syn_ramstyle by Technology

The attribute is only available for certain technologies and has different values, depending on the technology. The following table lists the valid values and implementations for each supported technology. For an explanation of the values, see Values for syn_ramstyle, on page 995.

Vendor	Technology	Values	Implementation
Microsemi	ProASIC3E		Default: Vendor-specific memories
		registers	Registers
		block_ram	Vendor-specific memories
		no_rw_check	RAMs implemented without read/write address conflict (glue) logic.
		rw_check	RAM implemented with read/write address conflict (glue) logic.

Constraint File Syntax and Example

define_attribute {_signalName_**[**_bitRange_**]} syn_ramstyle {**_string_**}**

define_global_attribute 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};

define_attribute {mem[7:0]} syn_ramstyle {block_ram};
```

Verilog Syntax and Example

object **/* synthesis syn_ramstyle = "**_string_**" */ ;**

where _object_ is a register definition (reg) signal. The data type is string.

Here is an example:

```
module ram4 (datain,dataout,clk);
output [31:0] dataout;
input clk;
input [31:0] datain;
reg [7:0] dataout[31:0] /* synthesis syn_ramstyle="block_ram" */;
// Other code
```

VHDL Syntax and Example

attribute syn_ramstyle of _object_ **:** _objectType_ **is "**_string_**" ;**

where *object* is a signal that defines a RAM or a label of a component instance. Data type is string. See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

```
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
```

Effect if Using syn_ramstyle with Read Write Check on RAM

When syn_ramstyle is set to rw_check, the synthesis software inserts bypass logic around the RAM to prevent a read/write address collision condition. Conversely, if syn_ramstyle is set to no_rw_check, then the synthesis software does not insert any bypass logic around the RAM to prevent a read/write address collision condition.

Constraint File Syntax and Example

define_attribute {*signalName***[***bitRange***]} syn_ramstyle {***string***}**

define_global_attribute syn_ramstyle {*string***}**

When you apply syn_ramstyle, you must include the range of the signal with the signal name. For example:

```
define_attribute {mem[7:0]} syn_ramstyle {rw_check};
define_attribute {mem1[7:0]} syn_ramstyle {no_rw_check};
```

Verilog Syntax and Example

object /* **synthesis syn_ramstyle = "***string***" */ ;**

where *object* is a register definition (ram4) signal. The data type is string.

Verilog Example 1: syn_ramstyle="no_rw_check"

```
module ram4 (datain,dataout,clk);
output [31:0] dataout;
input clk;
input [31:0] datain;
reg [7:0] dataout[31:0]
   /* synthesis syn_ramstyle="no_rw_check" */;
// Other code
```

Verilog Example 2: syn_ramstyle="rw_check"

```
module ram4 (datain,dataout,clk);
output [31:0] dataout;
input clk;
input [31:0] datain;
reg [7:0] dataout[31:0] /* synthesis syn_ramstyle="rw_check" */;
// Other code
```

VHDL Syntax and Example

attribute syn_ramstyle of *object* **: objectType is "***string***" ;**

where *object* is a signal the defines a RAM or label of a component instance. The data type is string.

See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

VHDL Example 1: syn_ramstyle="no_rw_check"

```
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 "no_rw_check";
-- Other code
```

VHDL Example 2: syn_ramstyle="rw_check"

```
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 "rw_check";
-- Other code
```

syn_reference_clock Attribute

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.

Constraint File Syntax and Examples

define_attribute {*register***} syn_reference_clock {***clockName***}**

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. Assign the find command to a collection (clock_enable_col), then refer to the collection when applying the syn_reference_clock constraint.

The following example shows how you can apply the constraint to all registers with the enable signal en40:

```
define_scope_collection clock_enable_col {find -seq * -filter
   (@clock_enable==en40)}
define_attribute {$clock_enable_col} syn_reference_clock {clk2}
```

Note: Apply syn_reference_clock only in an fdc file; you cannot use it in source code.

syn_replicate Attribute

Attribute; Controls replication. The 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.

Constraint File Syntax and Example

define_attribute {*object***} syn_replicate {0|1}**

define_global_attribute syn_replicate {0|1}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* **:** *objectType* **is true | false ;**

See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

```
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_resources Attribute

Attribute; Microsemi ProASIC3E. Specifies the resources used inside a black box. It is applied to Verilog black-box modules and VHDL architectures or component definitions.

The value of the attribute is any combination of the following:

Value	Description
blockrams=*integer*	number of RAM resources
corecells=*integer*	number of core cells for Microsemi families only.

The Microsemi families only support resource values of blockrams and corecells.

Constraint File Syntax and Example

define_attribute {v:*moduleName***} syn_resources**
{luts=*integer***|regs=***integer***|blockrams=***integer***|dsp_blocks=***integer***|**
blockmults=*integer***}**

define_attribute {v:*moduleName***} syn_resources**
{blockrams=*integer***|corecells=***integer***}**

You can apply the attribute to more than one kind of resource at a time by separating assignments by a comma (**,**). For example:

```
define_attribute {v:bb} syn_resources {luts=500, regs=400,
   blockrams=10,blockmults=2}
```

```
define_attribute {v:bb} syn_resources {corecells=50,blockrams=20}
```

Verilog Syntax and Example

object /* **synthesis syn_resources = "***value***" */ ;**

In Verilog, you can only attach this attribute to a module. Here is an example:

```
module bb (o,i) /* synthesis syn_black_box syn_resources =
   "luts=500,regs=463,blockrams=10,blockmults=2" */;
input i;
output o;
endmodule

module top_bb (o,i);
input i;
output o;
bb u1 (o,i);
endmodule
```

Verilog Syntax and Example

object /* **synthesis syn_resources = "***value***" */ ;**

In Verilog, you can only attach this attribute to a module. Here is an example:

```
module bb (o,i) /* synthesis syn_black_box syn_resources =
   "corecells=10,blockrams=5" */;
input i;
output o;
endmodule

module top_bb (o,i);
input i;
output o;
bb u1 (o,i);
endmodule
```

VHDL Syntax and Example

attribute syn_resources of *object* **:** *objectType* **is "***string***" ;**

See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives. In VHDL, this attribute can be placed on either an architecture or a component declaration.

```
architecture top of top is
component decoder
   port (clk : in bit;
         a, b : in bit;
         qout : out bit_vector(7 downto 0) );
end component;

attribute syn_resources : string;
attribute syn_resources of decoder: component is
   "luts=500,regs=463,blockrams=10,blockmults=2";

-- Other code
```

VHDL Syntax and Example

attribute syn_resources of *object* **:** *objectType* **is "***string***" ;**

See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives. In VHDL, this attribute can be placed on either an architecture or a component declaration.

```
architecture top of top is
component decoder
  port (clk : in bit;
      a, b : in bit;
      qout : out bit_vector(7 downto 0) );
end component;

attribute syn_resources : string;
attribute syn_resources of decoder: component is
  "corecells=500,blockrams=10";

-- Other code
```

syn_sharing Directive

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 215 in the *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* is a module definition.

```
module my_design(out,in,clk_in) /* synthesis syn_sharing=off */;
// Other code
```

VHDL Syntax and Example

attribute syn_sharing of *object* **:** *objectType* **is " true | false " ;**

where *object* is an architecture name. See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

```
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;
```

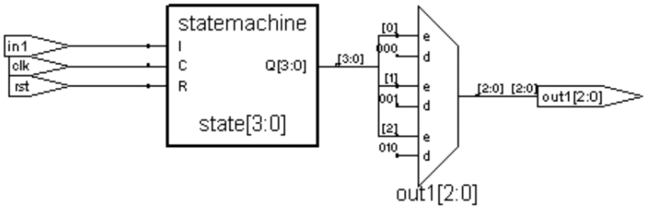
syn_state_machine Directive

Directive. Enables/disables state-machine optimization on individual state registers in the design. When you disable the FSM Compiler, state-machines are not automatically extracted. To extract some state machines, use this directive with a value of 1 on just those individual state-registers to be extracted. Conversely, when the FSM Compiler is enabled and there are state machines in your design that you do not want extracted, use syn_state_machine with a value of 0 to override extraction on just those individual state registers.

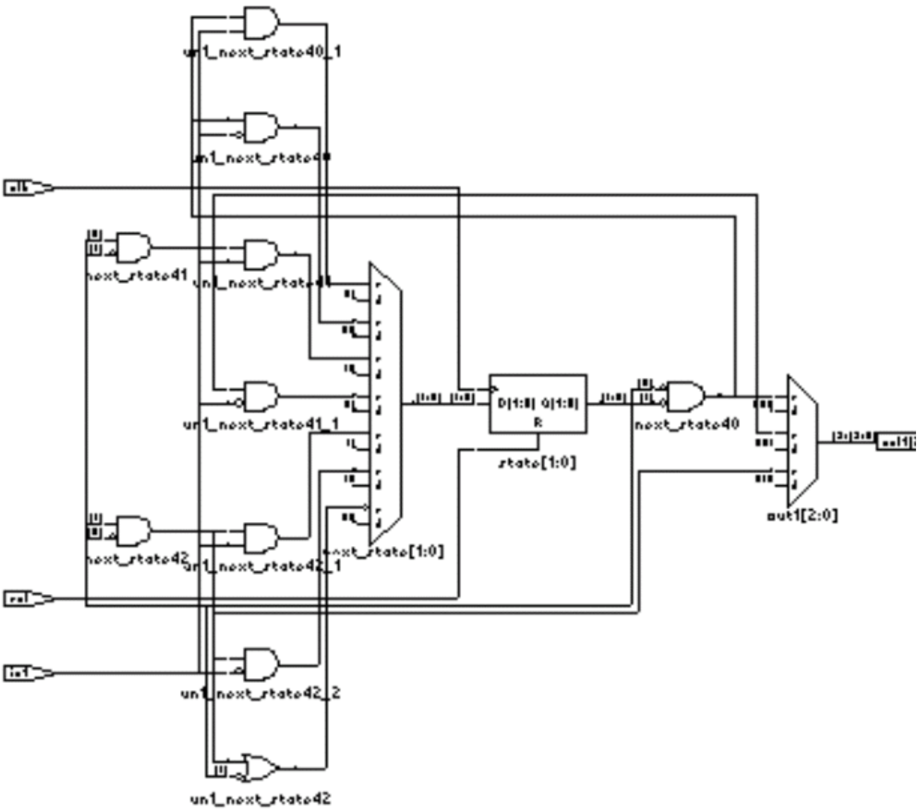
Also, when the FSM Compiler is enabled, all state machines are usually detected during synthesis. However, on occasion there are cases in which certain state machines are not detected. You can use this directive to declare those undetected registers as state machines.

The following figure shows an example of two implementations of a state machine: one with the syn_state_machine directive enabled, the other with the directive disabled.

syn_state_machine=0



syn_state_machine=1



See the following HDL syntax and example sections for the source code used to generate the schematics above. See also:

- syn_encoding Attribute, on page 927 for information on overriding default encoding styles for state machines.
- For VHDL designs, syn_encoding Compared to syn_enum_encoding, on page 935 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* **:** *objectType* **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;
```

See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

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

Directive. Used with the syn_black_box directive; supplies the clock to output timing-delay through a black box.

The syn_tco<n> directive is one of several directives that you can use with the syn_black_box directive to define timing for a black box. See syn_black_box Directive, on page 921 for a list of the associated directives.

Constraint File Syntax and Example

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. This is the constraint file syntax for the directive:

define_attribute {v:*blackboxModule***} 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.
blackboxModule	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 bundle is 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. To assign values to bundles, use the following syntax. The values are in ns. [**!**]*clock***->***bundle***=***value*
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}
```

Verilog Syntax and Example

object /* **syn_tco***n* **= "**[**!**]*clock* **->** *bundle* **=** *value***"** */ **;**

See Constraint File Syntax and Example, on page 1016 for syntax explanations. The following example defines syn_tco<*n*> and other black-box constraints:

```
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* **:** *objectType* **is "**[**!**]*clock* **->** *bundle* **=** *value***" ;**

In VHDL, there are ten predefined instances of each of these directives in the synplify library: 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_tco11 : string;
attribute syn_tco12 : string;
```

See Constraint File Syntax and Example, on page 1016 for other syntax explanations.

See VHDL Attribute and Directive Syntax, on page 746 for alternate methods for specifying VHDL attributes and directives.

The following example defines syn_tco<*n*> and 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 Dpram10240x8
   port (
-- Port A
      ClkA, EnA, WeA: in  std_logic;
      AddrA : in  std_logic_vector(13 downto 0);
      DinA  : in  std_logic_vector(7 downto 0);
      DoutA : out std_logic_vector(7 downto 0);
-- Port B
      ClkB, EnB: in  std_logic;
      AddrB : in  std_logic_vector(13 downto 0);
      DoutB : out std_logic_vector(7 downto 0) );
end component;

attribute syn_black_box : boolean;
attribute syn_tsu1      : string;
attribute syn_tsu2      : string;
attribute syn_tco1      : string;
attribute syn_tco2      : string;
attribute syn_isclock   : boolean;
attribute syn_black_box of Dpram10240x8 : component is true;
attribute syn_tsu1 of Dpram10240x8 : component is
   "EnA,WeA,AddrA,DinA -> ClkA = 3.0";
```

```
attribute syn_tco1 of Dpram10240x8 : component is
   "ClkA -> DoutA[7:0] = 6.0";
attribute syn_tsu2 of Dpram10240x8 : component is
   "EnB,AddrB -> ClkB = 3.0";
attribute syn_tco2 of Dpram10240x8 : component is
   "ClkB -> DoutB[7:0] = 13.0";

-- Other code
```

Verilog-Style Syntax in VHDL for Black Box Timing

In addition to the syntax used in the code above, you can also use the following Verilog-style syntax to specify black-box timing constraints:

```
attribute syn_tco1 of inputfifo_coregen : component is
   "rd_clk->dout[48:0]=3.0";
```

syn_tpd<*n*> Directive

Directive. Used with the syn_black_box directive; supplies information on timing propagation for combinational delay through a black box.

The syn_tpd<n> directive is one of several directives that you can use with the syn_black_box directive to define timing for a black box. See syn_black_box Directive, on page 921 for a list of the associated directives.

Constraint File Syntax and Example

You can enter the syn_tpd<*n*> directive as an attribute using the Attribute panel of the SCOPE editor. The information in the object, attribute, and value fields must be manually entered. This is the constraint file syntax:

define_attribute {v:*blackboxModule***} 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.
blackboxModule	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 bundle is 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 values are in ns. **"***bundle***->***bundle***=***value***"**
value	Input to output delay value in ns.

Constraint file example:

```
define_attribute {v:MEM} syn_tpd1 {MEM_RD->DATA_OUT[63:0]=20}
```

Verilog Syntax and Example

object **/* syn_tpd***n* **= "***bundle* **->** *bundle* **=** *value***" */ ;**

See Constraint File Syntax and Example, on page 1020 for an explanation of the syntax. This is an example of 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* **:** *objectType* **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";
```

See Constraint File Syntax and Example, on page 1020 for an explanation of the syntax.

See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

The following is an example of assigning syn_tpd<*n*> along with some of the black box constraints. See Verilog-Style Syntax in VHDL for Black Box Timing, on page 1019 for another way.

```
-- 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
```

syn_tristate Directive

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

Directive. Used with the syn_black_box directive; supplies information on timing setup delay required for input pins (relative to the clock) in the black box.

The syn_tsu<n> directive is one of several directives that you can use with the syn_black_box directive to define timing for a black box. See syn_black_box Directive, on page 921 for a list of the associated directives.

Constraint File Syntax and Example

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. The constraint file syntax for the directive is:

define_attribute {v:*blackboxModule***} 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.
blackboxModule	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 values are in ns. This is the syntax to define a bundle: *bundle***->**[**!**]*clock***=***value*
!	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}
```

Verilog Syntax and Example

object /* **syn_tsu***n* **=** "*bundle* **->** [**!**]*clock* **=** *value*" */ ;

For syntax explanations, see Constraint File Syntax and Example, on page 1024.

This is an example that defines 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* **:** *objectType* **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:

```
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";
```

For other syntax explanations, see Constraint File Syntax and Example, on page 1024.

See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives. For this directive, you can also use the following Verilog-style syntax to specify it, as described in Verilog-Style Syntax in VHDL for Black Box Timing, on page 1019.

The 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
```

syn_useenables Attribute

Attribute; Generates register instances with clock enable pins. By default, the synthesis tool tries to use the enable pin. You set syn_useenables to 0 to turn off clock-enable extraction.

Constraint 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* is 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* **:** *objectType* **is true | false ;**

where *object* is a label of a component register or signal.

See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

```
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;
```

translate_off/translate_on Directive

Directive. Allows you to synthesize designs originally written for use with other synthesis tools without needing to modify source code. All source code that is between these two directives is ignored 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, synthesis of all source code that follows is halted 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.

Note: See also, pragma translate_off/pragma translate_on Directive, on page 917. These directives are implemented the same in the source code.

Verilog Syntax and Example

The Verilog syntax for these directives is as follows:

/* synthesis translate_off */

/* synthesis translate_on */

For example:

```
module test(input a, b, output c);

//synthesis translate_off
assign c=a&b

//synthesis translate_on
assign c=a|b;
endmodule
```

For SystemVerilog designs, you can alternatively use the synthesis_off/synthesis_on directives. The directives function the same as the translate_off/translate_on directives to ignore all source code contained between the two directives during synthesis.

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 546 for information.

VHDL Syntax and Example

For VHDL designs, you can alternatively use the synthesis_off/synthesis_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.

See VHDL Attribute and Directive Syntax, on page 746 for different ways to specify VHDL attributes and directives.

The following is the VHDL syntax for translate-off/translate_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
```

Summary of Global Attributes

Design attributes in the synthesis environment can be defined either globally, (values are applied to all objects of the specified type in the design), or locally, values are applied only to the specified design object (module, view, port, instance, clock, and so on). When an attribute is set both globally and locally on a design object, the local specification overrides the global specification for the object.

In general, the syntax for specifying a global attribute in the `fdc` file is:

define_global_attribute *attribute_name* **{***value***}**

The table below contains a list of attributes that can be specified globally in the synthesis environment.

For complete descriptions of any of the attributes listed below, see Summary of Attributes and Directives, on page 897.

Global Attribute	Can Also Be Set On Design Objects
syn_allow_retiming	x
syn_hier	x
syn_multstyle	x
syn_netlist_hierarchy	
syn_noarrayports	
syn_noclockbuf	x
syn_ramstyle	x
syn_replicate	x

CHAPTER 12

Batch Commands and Scripts

This chapter describes Tcl commands and scripts. This chapter discusses the following topics:

Introduction to Tcl

Tcl (Tool Command Language) is a popular scripting language for controlling software applications. Synopsys has extended the Tcl command set with additional commands that you can use to run the Synopsys FPGA 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 and contain a set of Tcl commands designed to complete a task or set of tasks. In the Synplify Pro tool, you can also run Tcl scripts through the Tcl window (see Tcl Script Window, on page 62).

The Synopsys FPGA 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 Working with Tcl Scripts and Commands, on page 456 in the *User Guide*:

- Running Batch Mode on a Project File, on page 450
- Running Batch Mode with a Tcl Script, on page 451
- Generating a Job Script, on page 457
- Creating a Tcl Synthesis Script, on page 458
- Using Tcl Variables to Try Different Clock Frequencies, on page 460
- Running Bottom-up Synthesis with a Script, on page 463

Batch 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 1036
- add_folder, on page 1040
- command_history, on page 1040
- constraint_file, on page 1041get_env, on page 1042
- get_option, on page 1044
- hdl_define, on page 1044
- hdl_param, on page 1045
- impl, on page 1047
- job, on page 1048
- open_design, on page 1049
- open_file, on page 1051
- partdata, on page 1051

- project, on page 1053
- project_data, on page 1059
- project_file, on page 1059
- project_folder, on page 1061
- recording, on page 1062
- run_tcl, on page 1063
- set_option, on page 1065
- Technology-specific Tcl Commands, on page 1086

add_file

The add_file command adds one or more files to a project.

Syntax

add_file [**-***filetype*] *fileName* [*fileName* [...]]

add_file -verilog *fileName* [*fileName* [...]] [**-folder** *folderName*]

add_file -vhdl [**-lib** *libName*[*libName*]] *fileName* [*fileName* [...]] [**-folder** *folderName*]

add_file -vlog_std *standard fileName* [*fileName* [...]]

-filetype	Specifies the type of file being added to the project (files are placed in folders according to their file types; including this argument overrides automatic filename-extension placement). See Filename Extensions, on page 1039 for a list of the recognized file types.
fileName	Specifies the name of the file being added to the project. Files are added to the individual project folders according to their filename extensions (View Project Files in Folders must be set in the Project View Options dialog box). You can add multiple files by separating individual filenames with a space, and you can specify different file types (extensions) within the same command.
-verilog or -vhdl	Adds HDL files with non-standard extensions to the Verilog or VHDL directory, 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` If you do not specify -verilog, the file is added to the Other directory (new is not a recognized Verilog extension), and the file would not be compiled with the files in the Verilog directory.
[-lib *libName*]	Specifies the library associated with VHDL files. The default library is work. The -lib option sets the VHDL library to *libName*. Note: You can also specify multiple libraries for VHDL files. For example: `add_file -vhdl -lib {mylib,work} "ff.vhd"` Both the logical and physical libraries must be specified in the Project file (if you only specify the logical library associated with the VHDL files, the compiler treats the module as a black box).

[-folder *folderName*]	Creates logical folders with custom files in various hierarchy groupings within your Project view. For example:
	`add_file -verilog -folder memory "ram_1.v"`
	`add_file -verilog -folder memory "C:/examples/verilog/common_rtl/memory/ram_1.v"`

-include	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.
-vlog_std *standard*	Overrides the global Verilog standard for an individual file. The accepted values for *standard* are v95 (Verilog 95), v2001 (Verilog 2001), and sysv (SystemVerilog). The file (*fileName*) is added to the Verilog folder in the project; the specified standard is listed after the filename in the project view and is enclosed in angle brackets (for example, commchip.v <sysv>). Note that when you add a SystemVerilog file (a file with an sv extension) to a project, the add_file entry in the project file includes the -vlog_std *standard* string. The default standard for new projects is SystemVerilog. For Verilog 2005 extensions, use sysv (SystemVerilog).

Filename Extensions

Files with the following extensions are automatically added to their corresponding project directories; files with any other extension are added to the Other directory. The ***-filetype*** argument overrides automatic filename extension placement.

Extension	-Filetype	Project Folder
.adc	-analysis_constraint	Analysis Design Constraint
.edf, .edn	-edif	EDIF
.fdc	-constraint	Logic Constraints (FDC)
.sdc	-constraint	Logic Constraints (SDC)
.sv[a]	-verilog	Verilog
.tcl	-tcl	Tcl Script
.v	-verilog	Verilog
.vhd, .vhdl	-vhdl	VHDL

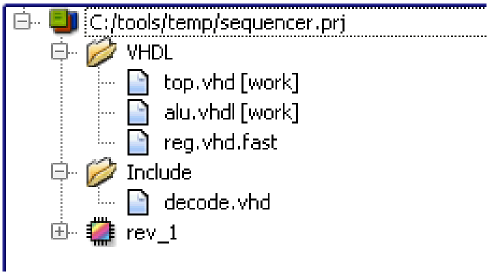
[a] Use the .sv format for SystemVerilog keyword support. Both Verilog and SystemVerilog formats are added to the Verilog folder.

Example: Add Files

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:



add_folder

The add_folder command adds a custom folder to a project.

Syntax

add_folder *folderName*

Creates logical folders with files in various custom hierarchy groupings within your Project view. These custom folders can be specified with any name or hierarchy level.

```
add_folder verilog
add_folder verilog/common_rtl
add_folder verilog/common_rtl/prep
```

For more information about custom folders, see Project File Hierarchy Management, on page 120 in the *User Guide*.

command_history

Displays a list of the Tcl commands executed during the current session.

Syntax

command_history [-save *filename*]

Arguments and Options

-save

Writes the list of Tcl commands to the specified *filename*.

Description

The command_history command displays a list of the Tcl commands executed during the current session. Including the -save option, saves the commands to the specified file to create Tcl scripts.

Examples

```
command_history -save C:/DesignsII/cert_tut/proto/myTclScript.tcl
```

See Also

- recording, on page 1062

constraint_file

The constraint_file command 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 arguments.

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.fdc clocks1.fdc clocks2.fdc eight_bit_uc.fdc

% constraint_file -disable eight_bit_uc.fdc
```

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.fdc
```

get_env

The get_env command reports the value of a predefined system variable.

Syntax

get_env *systemVariable*

Use this command in the Tcl window to view system variable values. 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 or a Tcl script, 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 $*systemVariable* 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

The get_option command reports the settings of predefined project and device options. The options are the same as those for set_option. See set_option, on page 1065 for details.

Syntax

get_option -*optionName*

hdl_define

For Verilog designs, this command displays the current HDL parameters in the design or is used to enter new values for the compiler directives. You can define directives that you would normally enter using the Verilog `ifdef and `define. The hdl_define command is the same as the set_option -hdl_define command. The parameter value is valid for the current implementation only.

Syntax

hdl_define
-set "*directive***=***value***"**
-list

Example:

```
hdl_define -set "SIZE=32"
```

This statement specifies the value 32 for the SIZE directive; the following statement is written to the project file:

```
set_option -hdl_define -set "SIZE=32"
```

To define multiple directive values using hdl_define, enclose the directives in quotes and use a space delimiter. For example:

```
hdl_define -set "SIZE=32 WIDTH=8"
```

The software writes the following statement to the prj file:

```
set_option -hdl_define -set "size=32 width=8"
```

See Also

Compiler Directives and Design Parameters, on page 172 for information on specifying compiler directives in the GUI.

hdl_param

The hdl_param command shows or sets 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
 -overrides

The following table describes the command arguments.

Option	Description
-add	Adds a parameter override to the project.
-list	Shows parameters for the top-level module only and lists values for parameters if there is a parameter override.
-set	Sets a parameter override and its value for the active implementation. Only the parameter value is enclosed within curly braces.
-clear	Clears all parameter overrides of the active implementation.
-overrides	Lists all the parameter override values used in this project.

Examples

In batch mode, to set generic values using the set_option command in a project file, specify the hdl_param generic with quotes and enclose it within {}. For example:

```
    set_option -hdl_param -set ram_file {"init.mem"}

    set_option -hdl_param -set simulation {"false"}
```

Suppose the following parameter is set for the top-level module.

```
    set_option -hdl_param -set {"width=8"}
```

Add a parameter override and its value, then list the parameter override.

```
hdl_param -add {"size=32"}
hdl_param -list "size=32"
```

impl

The impl command adds, removes, or modifies an implementation.

Syntax

```
impl
    -add [implName] [model]
    -name implName
    -remove implName
    -active [implName]
    -list
    -type implType
    -result_file
    -dir
```

The following table describes the command arguments.

Option	Description
-add	Adds a new device implementation. If: • *implName* is not specified, creates a unique implementation name by incrementing the name of the active implementation. • you want to add a new implementation copied from implementation *model*.
-name	Changes the name of the active implementation.
-remove	Removes the specified implementation.
-active	Reports the active implementation. If you specify an implementation name, changes the specified name to the active implementation.
-list	Lists all the implementations used in this project.
-type	Specifies the type of implementation to add. For example, the: • -type fpga option creates an FPGA implementation. • -type identify option creates an Identify implementation.
-result_file	Displays the implementation results file.
-dir	Displays the implementation directory.

Examples

List all implementations, report the active implementation, then make a different implementation active.

```
% impl -list
design_worst design_typical design_best

% impl -active
design_best

%impl -active design_typical

% impl -active
design_typical

% impl -add rev_1_identify mixed -type identify
```

job

The job command, for place and route job support, creates, removes, identifies, runs, cancels, and sets/gets options for named P&R jobs.

Syntax

job *jobName* [**-add** *jobType* |**-remove** |**-type** |**-run** [*mode*] |**-cancel** |
 -option *optionName* [*optionValue*]]

job -list

The following table describes the command options.

Option	Description
-run	Runs the P&R job, according to the specified options:
-add *jobType*	Creates a new P&R job for the active implementation.
-cancel	Cancels a P&R job in progress.
-remove	Removes a P&R job from an active implementation
-list	Returns a list of the P&R jobs in the active implementation.
-remove	Removes a P&R job from the active implementation

Option	Description
-option *optionName* [*optionValue*]	Get/set options for *jobName*.
-type	Returns the P&R job type.

Examples

```
% job pr_2 -add par
% job pr_2 -option enable_run 1
% job pr_2 -run
```

open_design

The open_design command writes a netlist (`srs` or `srm`) to the database so that you can use the find command in batch mode. With open_design, you can use find without having to open an RTL or Technology view. Use open_design to read in the `srs` or `srm` file before issuing the find command. See the example below.

Syntax

open_design [**-shared 0**|**1**] *filename*

Where:

- -shared — indicates if the database is to be shared (read-only). Setting this switch to 0 allows the database being searched to be updated which requires additional memory resources.
- *filename* is the RTL (`srs`) or Technology (`srm`) file to write to the database.

Example

```
project -load ../examples/vhdl/prep2_2.prj
open_design prep2_2.srs
set a [find -inst *]
c_print $a -file a.txt
open_design prep2_2.srm
set b [find -net *]
c_print $b -file b.txt
```

In the example above, prep2_2 is loaded and the information from the RTL view file is read in. Then, the find command searches for all instances in the design and prints them to file a. Next, the technology view file is read in, then find searches for all nets in the design and prints them to file b.

See Also

- find Command (Batch), on page 1098.

open_file

The open_file command opens views within the tool. The command accepts two arguments: -rtl_view and -technology_view.

Syntax

open_file -rtl_view |-technology_view

The -rtl_view option displays the RTL view for the current implementation, and the -technology_view option displays the technology view for the current implementation. Views remain displayed until overwritten and multiple views can be displayed.

partdata

The partdata command loads part files and returns information regarding a part such as available families, family parts, vendors, attributes, grades, packages.

Syntax

partdata
-load *filename*
-family
-part *family*
-vendor *family*
-attribute *attribute family*
-grade [*family*:]*part*
-package [*family*:]*part*
-oem [*family*:]*part*

Option	Description
-load *filename*	Loads part file.
-family	Lists available technology families.
-part *family*	Lists all parts in specified family.
-vendor *family*	Returns vendor name for the specified family.
-attribute *attribute family*	Returns the value of the job attribute for the specified family.

Option	Description
-grade [*family*:]*part*	Lists the speed grades available for the specified part.
-package [*family*:]*part*	Lists the packages available for the specified part.
-oem [*family*:]*part*	Returns true if the part entered is an OEM part.

Example

The following example prints out the available vendors, their supported families, and the parts for each family.

```
% foreach vendor [partdata -vendorlist]
% puts VENDOR:$vendor;
% foreach family [partdata -family $vendor]
% puts \tFAMILY:$family;
% puts \t\tPARTS:;
% foreach part [partdata -part $family]
% puts \t\t$part;
```

program_terminate

Immediately terminates the tool session without saving any data.

Syntax

program_terminate

Arguments and Options

None

Description

The program_terminate command terminates a tool session without prompting or saving data.

Examples

```
program_terminate
```

program_version

Returns software product version.

Syntax

program_version

Arguments and Options

None

Description

The program_version command returns the software product version number.

Examples

```
% program_version
Synplify Pro G-2012.09
```

project

The project command runs job flows to create, load, save, and close projects, to change and examine project status, and to archive projects.

Syntax

project -**run** [**-fg**] [**-all**] [*mode*]

project {**-new** [*projectPath*] **|-load** *projectPath* **| -close** [*projectPath*] **|-save** [*projectPath*] **|-insert** *projectPath*} **|**

project {**-active** [*projectName*] **|-dir |-file |-name |-list |-filelist | -fileorder** *filepath1 filepath2* [... *filepathN*] **|-addfile** *filepath* **| -movefile** *filepath1* [*filepath2*] **|-removefile** *filepath*}

project {**-result_file** *resultFilePath* **|-log_file** [*logfileName*] }

project -copy [**-project** *filename*] [**-implement** *implementationName*] [**-dest_dir** *pathname*] [**-copy_type** {**full | local | customize**}] [**-add_srs** [*fileList*] **-no_input**]

project -unarchive [**-archive_file** *pathname/filename*] [**-dest_dir** *pathname*]

The following table describes the command options.

Option	Description
-run [**-fg**] [**-all**] [*mode*]	Synthesizes the project, according to the specified options: • **-fg** – Synthesizes in the foreground. • **-all** – Synthesizes all implementations. The *mode* is one of the following keywords: • **compile** – Compiles the active project, but does not map it. • **constraint_check** - Validates the syntax and applicability of constraints defined in one or more fdc files. • **syntax_check** – Verifies that the HDL is syntactically correct; errors are reported in the log file. • **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. • **timing** – Runs the Timing Analyst. This is equivalent to clicking the Generate Timing button in the Timing Report Generation dialog box with user-specified values. • **write_netlist** – Writes the mapped output netlist to structural Verilog (vm) or VHDL (vhm) format. You can also use this command in an incremental timing analysis flow. For details, see Run Menu, on page 180 and Generating Custom Timing Reports with STA, on page 331.

Option	Description
-new [*projectPath*]	Creates a new project in the current working directory. If *projectPath* is specified, creates the project in the specified directory. For the Hierarchical Project Management flow, you can create a new sub-project for the top-level project.
-load *projectPath*	Loads the project file specified by *projectPath*.
-close [*projectPath*]	Closes the currently active project. If *projectPath* is specified, closes the specified project.
-save [*projectPath*]	Saves the currently active project. If *projectPath* is specified, saves the specified project.
-insert *projectPath*	Adds the specified project to the workspace project. For the Hierarchical Project Management flow, adds the specified sub-project.
-active [*projectName*]	Shows the active project. If *projectName* is specified, makes the specified project the active project.
-dir	Shows the project directory for the active project.
-file	Returns the path to the active project.
-name	Returns the filename (prj) of the active project.
-list	Returns a list of the loaded projects.
-filelist	Returns the pathnames of the files in the active project.
-fileorder *filepath1 filepath2* [... *filepathN*]	Reorders files by adding the specified files to the end of the project file list.
-addfile *filepath*	Adds the specified file to the project.
-movefile *filepath1* [*filepath2*]	Moves *filepath1* to follow *filepath2* in project file list. If *filepath2* is not specified, moves *filepath1* to top of list.
-removefile *filepath*	Removes the specified file from the project.
-result_file *resultFilePath*	Changes the name of the synthesis result file to the path specified.
-log_file [*logfileName*]	Reports the name of the project log file. If *logfileName* is specified, changes the base name of the log file.

Option	Description
-archive **-project** *filename* [**-root_dir** *pathname*] **-archive_file** *filename*.**sar** **-archive_type** {**full** | **local** | **customize**} **-add_srs** [*fileList*] **-no_input**	• **project** *filename* – copies a project other than the active project. If you do not use this option, by default the active project is copied. • **root_dir** *pathname* – specifies the top-level directory containing the project files. • **archive_file** *filename* – is the name of the archived project file. • **archive_type** - specifies the type of archive: • full – performs a complete archive; all input and result files are contained in the archive file. • customize – performs a partial archive; only the project files that you select are included in the archive. • local – includes only project input files in the archive; does not include result files. • **add_srs** – adds the listed `srs` files to the archived project. Use the -no_input option with this command. If *fileList* is omitted, adds all `srs` files for the project/implementations. The `srs` files are the RTL schematic views that are output when the design is compiled (Run->Compile Only). For examples using the project -archive command, see Archive Examples, on page 1058.

Option	Description
-copy **-project** *filename* **-implement** *implementationName* **-dest_dir** *pathname* **-copy_type** {**full | local | customize**} **-add_srs** [*fileList*] **-no_input**	• **project** *filename* – copies a project other than the active project. If you do not use this option, by default the active project is copied. • **implement** *implementation_name* – archives all files in the specified implementation. • **dest_dir** *directory_pathname* – specifies the directory in which to copy the project files. • **copy_type** – specifies the type of file/project copy: • full – performs a complete copy; all input and result files are contained in the archive file. • customize – performs a partial copy; only the project files that you select are included in the archive. • local – includes only project input files in the copy; does not include result files. • **add_srs** – adds the listed `srs` files to the archived project. Use the -no_input option with this command. If *fileList* is omitted, adds all `srs` files for the project/implementations. The `srs` files are the RTL schematic views that are output when the design is compiled (Run->Compile Only).
-unarchive **-archive_file** *pathname/filename* **-dest_dir** *pathname*	• **archive_file** *pathname/filename* – is the name of the archived project file. • **dest_dir** *pathname* – specifies the directory in which to write the project files. For examples using the project -unarchive command, see Un-Archive Example, on page 1058.

Examples

Load the project top.prj and compile the design without mapping it. Compiling makes it possible to create a constraint 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.

Insert sub-projects to a top-level design.

```
% project -insert "./block2/block2.prj"
% project -insert "./block3/block3.prj"
% project -insert "./block1/block1.prj"
% project -insert "./control/control.prj"
```

Archive Examples

The following example archives all files in the project and stores the files in the specified sar file:

```
project -archive -project c:/proj1.prj
      -archive_file c:/archive/proj1.sar
```

The next example archives the project file (prj) and all local input files into the specified sar file.

```
project -archive -project c:/proj1.prj -archive_type local
      -archive_file c:/archive/proj1.sar
```

The following example archives the project file (prj) only for selected srs files into the specified sar file. Any input source files that are in the project are not included.

```
project -archive -project c:/proj1.prj -archive_type customize
      -add_srs -no_input -archive_file c:/archive/proj1.sar
```

Un-Archive Example

The following example extracts the project files from c:/archive/proj1.sar to directory c:/proj1. All directories and sub-directories are created if they do not already exist.

```
project -unarchive -archive_file c:/archive/proj1.sar
         -dest_dir c:/proj1
```

Copy Examples

The following example copies only selected srs files for the project to the destination project file directory.

```
project -copy -project d:/test/proj_2.prj -copy_type customize
         –add_srs –no_input -dest_dir d:/test_1
```

The next example copies all input source files and `srs` files selected for the project to the destination project file directory.

```
project -copy -project d:/test/proj_2.prj -copy_type customize
        -dest_dir d:/test_1
```

project_data

The project_data command shows or sets properties of a project.

Syntax

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

The following table describes the 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

The project_file command manipulates and examines project files.

Syntax

project_file {**-lib** *fileName* [*libName*] **|** **-name** *fileName* [*newPath*] **|**
-time *fileName* [*format*] **|** **-date** *fileName* **|** **-type** *fileName* **|**
-savetype *fileName* [**relative | absolute**] **-move** *fileName1* [*fileName2*] **|**
-remove *fileName* **|** **-top** *topModule* **|**
-tooltag *applicationTagName* **|** **- toolargs** [*arguments*] *fileName* }

The following table describes the 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 t1he 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.
-savetype	Sets or shows whether a file is saved relative to the project or its absolute path.
-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.
-top	Sets or shows the top-level module of the specified file for the active implementation.

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
```

project_folder

The project_folder command manipulates and examines attributes for project folders.

Syntax

project_folder [*folderName*] [**-folderlist**] [**-filelist**] [**-printout**] [**-add**] [**-remove**] [**-r**] [**-tooltag**] [**-toolargs**]

The following table describes the command options.

Option	Description
folderName	Specifies the name of the folder for which attributes are examined.
-folderlist	Lists folders contained in the specified project folder.
-filelist	Lists files contained in the specified project folder.
-printout	Prints the specified project folder hierarchy including its files.
-add	Adds a new project folder.
-remove	Removes the specified project folder.
-r	Removes the specified project folder and all its containing sub-folders. Files are removed from the project folder, but are not deleted.

Examples

Add a folder and list the files added to a project folder.

```
% project_folder -add newfolder
% project_folder -filelist newfolder
```

recording

Allows you to record and store the Tcl commands generated when you work on your projects in the GUI. You can use this command for creating job scripts. The complete syntax for the recording command is:

recording { **-on** | **-off** **-file** [*historyLogFile]* | **-save** [*historyLogFile*] } **-state**

In the command line:

- **on|off**– turns Tcl command recording on (1) or off (0). Recording mode is off by default.
- **file** – if you specify a history log file name, this option uses the specified file in which to store the recorded Tcl commands for the current session. If you do not specify a history log name, reports the name of the current history log file.
- **save** – if you do not specify a file name, updates the current history log. If you specify a history log file name, saves Tcl command history to the specified file.
- **state** – returns the Boolean value of recording mode.

Examples

Turn on recording mode and save the Tcl commands in the `cpu_tcl_log` file created.

```
% recording -on
% recording -file cpu_tcl_log
```

report_clocks

Reports the clocks in the design database.

Syntax

report_clocks -netlist [*srsNetlistFile*] [**-csv_format**] [**-out** *fileName*]

Arguments and Options

srsNetlistFile
The name of the srs netlist file. If this optional argument is not specified, the netlist file is taken from the active project implementation.

-csv_format
Displays the report in spread-sheet format.

-out
Specifies the name of the output report file (default name is *desigName*_clk.rpt).

Description

The report_clocks command generates a report of the clocks found in the design database. The report includes a listing of the clock domain, parent clock, and clock type for each clock. If the -csv_format option is included, the report is output in spread-sheet format.

Examples

```
report_clocks c:/designs/mem_ctrl/mem_ctrl.srs -csv_format
```

run_tcl

The run_tcl command lets you synthesize your project using a Tcl script file from the Tcl Script window of the synthesis tool.

Syntax

run_tcl [**-fg**] *tclFile*

You can also use the following command:

source *tclFile*

These commands are equivalent.

The following table describes the run_tcl command options.

Option	Description
-fg	Synthesizes the project in foreground mode.
TclFile	Specifies the name of the Tcl file used to synthesize the project. To create a Tcl Script file, see Creating a Tcl Synthesis Script, on page 458.

set_option

The set_option command sets options for the technology (device) as well as for the design project.

Syntax

set_option **-***optionName optionValue*

For syntax and descriptions of the options and related values, see one of the following tables:

- Device Options for set_option/get_option
- Project Options for set_option/get_option

Device Options for set_option/get_option

The following table lists *generic* device arguments for the technology, part, and speed grade. These are the options on the Implementation Options-> Device tab.

Information on all other Implementation Options tabs are listed in the next section, Project Options for set_option/get_option, on page 1066.

Option Name	Description
-technology *parameter*	Sets the target technology for the implementation. *parameter* is the string for the vendor architecture. Check the Device panel in the GUI or see Device Panel, on page 159, for a list of supported families.
-part *part_name*	Specifies a part for the implementation. Check the Device panel of the Implementation Options dialog box (see Device Panel, on page 159) for available choices.
-speed_grade -*value*	Sets the speed grade for the implementation. Check the Device panel of the Implementation Options dialog box (see Device Panel, on page 159) for available choices.

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 Implementation Options dialog box (see Device Panel, on page 159) for available choices.
-grade -*value*	Same as -speed_grade. Included for backwards compatibility.

In general, device options are technology-specific, or have technology-specific defaults or limitations. For vendor-specific details, see Technology-specific Tcl Commands, on page 1086.

Project Options for set_option/get_option

Below is a list of options for the set_option and get_option commands. Click on the option below for the corresponding description and GUI equivalents. Options set through the Device tab are listed in Device Options for set_option/get_option, on page 1065.

analysis_constraint
areadelay
area_delay_percent
auto_constrain_io
autosm
block
compiler_compatible
compiler_constraint
constraint
default_enum_encoding
disable_io_insertion
dup
enable64bit
fanout_limit
frequency
frequency auto
globalthreshold
hdl_define
help
identify_debug_mode
ignore_undefined_libs
include_path
job (PR)
library_path
maxfan
maxfan_hard
max_parallel_jobs
multi_file_compilation_unit
num_critical_paths
num_startend_points
opcond
preserve_registers
project_relative_includes
report_path
reporting_*type*
resolve_multiple_driver
resource_sharing
result_file
retiming
run_prop_extract
symbolic_fsm_compiler
synthesis_onoff_pragma
top_module
update_models_cp
use_fsm_explorer
vlog_std
write_apr_constraint
write_verilog
write_vhdl

Option	Description	GUI Equivalent
-analysis_constraint *path/filename*.adc	Specifies the analysis design constraint file (adc) you can use to modify constraints for the stand-alone Timing Analyst only.	Constraint File section on the Timing Report Generation Parameters dialog box
-areadelay *percentValue* **-area_delay_percent** *percentValue*	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
-auto_constrain_io 1|0	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.	Use clock period for unconstrained IO check box, Constraints Panel
-autosm 1|0 **-symbolic_fsm_compiler 1|0**	Enables/disables the FSM compiler.	FSM Compiler check box, Options Panel
-beta_vfeatures 1 | 0	Enables/disables the use of Verilog compiler beta features.	Beta Features for Verilog, Verilog 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

Option	Description	GUI Equivalent
-compiler_compatible 1|0	Disables pushing of tristates across process/block boundaries.	*Complement* of the Push Tristates Across Process/ Block Boundaries check box, VHDL Panel and Verilog Panel
compiler_constraint *constraintFile*	When multiple constraint files are defined, specify which constraint files are to be used from the Constraints tab of the Implementation Options panel.	Constraints Files, Constraints Panel
constraint -*option*	Manipulates constraint files in the project: -enable/disable *filename* – adds or removes constraint file from active implementation -list – lists all enabled constraint files in active implementation -all – enables all constraint files in active implementation -clear – disables all constraint files in active implementation	Constraint Files, Constraints Panel
continue_on_error 1|0	Supports some Microsemi technologies. The continue_on_error option serves the following function. **Mapper** – when enabled during compile-point synthesis, allows the mapping operation to continue on error and synthesize the remaining compile points. The default for this option (0) is to stop on any compilation or synthesis error.	Continue on Error, Project View checkbox or Options Panel, or Configure Compile Point Process Command
-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)

Option	Description	GUI Equivalent
-disable_io_insertion 1|0 **-block 1|0**	Enables/disables I/O insertion in some technologies.	Disable I/O Insertion, Device Panel
-dup	For Verilog designs, allows the use of duplicate module names. When true, the last definition of the module is used by the software and any previous definitions are ignored. You should not use duplicate module names in your Verilog design, therefore, this option is disabled by default. However, if you need to, you can allow for duplicate modules by setting this option to 1.	Allow Duplicate Modules, Verilog Panel
-enable64bit 1|0	Enables/disables the 64-bit mapping switch. When enabled, this switch allows you to run client programs in 64-bit mode, if available on your system.	Enable 64-bit Synthesis, Options Panel
-fanout_limit *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
-frequency auto	Enables/disables auto constraints.	Auto Constrain, Constraints Panel
-globalthreshold *value*	This option applies only to the Microsemi ProASIC3E technology: Sets the minimum number of fanout loads. Signals that exceed the load value are promoted to global signals. Global buffers are assigned by the synthesis tool to drive the global signals.	Device Panel

Option	Description	GUI Equivalent
-hdl_define	For Verilog designs; used for extracting design parameters and entering compiler directives.	Compiler Directives and Design Parameters, Verilog Panel
-**hdl_param**	Shows or sets HDL parameter overrides. See hdl_param, on page 1045 for command syntax.	Use this command in the Tcl window of the UI.
-help	This option is useful for getting syntax help on the various implementation options used for compiling and mapping a design. For examples, see help for set_option, on page 1079.	Use this command in the Tcl window of the UI.
-identify_debug_mode 1|0	When set option to 1, creates an Identify implementation in the Project view. Then, you can launch the Identify Instrumentor or Debugger from within the FPGA synthesis tools.	Select the Identify implementation, then launch: • Launch Identify Instrumentor or • Launch Identify Debugger
-ignore_undefined_libs 1|0	(VHDL only) When enabled (default), the compiler will ignore any declared library files not included with the source file. In previous releases, the missing library file would cause the synthesis tool to error out. To set this option to error out when a library file is missing (as in previous releases), use 0 for the command value.	Not available in the UI

Option	Description	GUI Equivalent
-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 where the included design files can be found. The software searches for include files in the following order: • First, the source file directory. • Then, looks in the included path directory order and stops at the first occurrence of the included file it finds. • Finally, the project directory. The include paths are relative. Use the project_relative_includes option to update older project files.	Include Path Order, Verilog panel (see Verilog Panel, on page 169)
-job *PR_job_name* **-option enable_run 1|0**	If enabled, runs the specified place-and-route job with the appropriate vendor-specific place-and-route tool after synthesis.	Specify the place-and-route job you want to run for the specified implementation. See Place and Route Panel.
-library_path *directory_pathname*	For Verilog designs, specifies the paths to the directories which contain the library files to be included in your design for the project. Defines the search path used by the tool to include all the Verilog design files for your project. The argument *directory_pathname* is a string that specifies the directories where these included library files can be found. The software searches for all included Verilog files and the tool determines the top-level module.	Library Directories on Verilog Panel.

Option	Description	GUI Equivalent
-log_file *logFileName*	Allows you to change the name for a default log file (both the `srr` and `htm` files). For example: `set_option -log_file test` generates the the following files in the Implementation Directory after synthesis is run: • `test.htm` • `synlog\test_premap.srr` • `synlog\test_fpga_mapper.srr` • `synlog\test_fpga_mapper.srr_Min`	Enter command from the Tcl window
-maxfan *value* **-fanout_guide** *value*	Sets the fanout limit for the current project. The limit value is a guideline for the tool rather than a hard limit.	Fanout Guide, Device Panel
-maxfan_hard 1	*Available in the Synplify Pro and Synplify tools for Microsemi designs.* This option specifies that the -maxfan value is a hard fanout limit that the synthesis tool must not exceed it.	Hard Limit to Fanout, Device Panel
max_parallel_jobs *n*	Lets you run multiprocessing with compile points. This allows the synthesis software to run multiple, independent compile point jobs simultaneously, providing additional runtime improvements for the compile point synthesis flow. For information on setting the maximum number of parallel synthesis jobs, see Setting Number of Parallel Jobs, on page 457 in the *User Guide*.	Maximum number of parallel mapper jobs, on the Configure Compile Point Process dialog box.

Option	Description	GUI Equivalent
-multi_file_compilation_unit 1|0	When you enable the Multiple File Compilation Unit switch, the Verilog compiler uses the compilation unit for modules defined in multiple files.	Verilog 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. Number of Start/End Points, Timing Report Generation dialog box.
-opcond *value*	This option applies only to the Microsemi Axcelerator and ProASIC families of technologies. Sets the operating condition for device performance in the areas of optimization, timing analysis, and timing reports. Values are Default, MIL-WC, IND-WC, COM-WC, and Automotive-WC. See Operating Condition Device Option, on page 1139 for more information.	Device Panel
-preserve_registers 1|0	This option is available only for Microsemi technologies. When enabled, the software uses less restrictive register optimizations during synthesis if area is not as great a concern for your device. The default for this option is disabled (0).	Conservative Register Optimization switch on the Device Panel

Option	Description	GUI Equivalent
-project_relative_includes 1|0	Enables/disables the Verilog include statement to be relative to the project, rather than a verilog file. For projects built with software after 8.0, the include statement is no longer relative to the files but is relative to the project: project_relative (1). See Updating Verilog Include Paths in Older Project Files, on page 119 in the *User Guide* for information about updating older project files.	Include Path Order, Verilog Panel
-report_path *integer*	Available for Microsemi 500K and ProASIC family of technologies. Sets the maximum number of critical paths in a forward-annotated SDF constraint file	Max Number of Critical Paths in SDF, Device Panel
-**reporting_***type*	Sets parameters for the stand-alone Timing Analyst report. See Timing Report Parameters for set_option, on page 1077 for details.	Analysis->Timing Analyst command: Timing Report Generation Parameters
-resolve_multiple_driver 1|0	When a net is driven by a VCC or GND and active drivers, enable this option to connect the net to the VCC or GND driver. The default for this option is disabled (0). See Resolve Mixed Drivers Option, on page 1080 for details.	Resolve Multiple Drivers, 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

Option	Description	GUI Equivalent
-retiming 1|0	When enabled (1), registers may be moved into combinational logic to improve performance. The default value is 0 (disabled).	Retiming, Device Panel
-run_prop_extract 1|0	Enables/disables the annotation of certain generated properties relating to clocks and expansion onto the RTL view. This enables the Tcl expand and find commands to work correctly with clock properties.	Options Panel
-RWCheckOnRam 1 | 0	If read or write conflicts exist for the RAM, enable this option to insert bypass logic around the RAM to prevent simulation mismatch. Disabling this option does not generate bypass logic. For more information about using this option in conjunction with the syn_ramstyle attribute, see syn_ramstyle Attribute, on page 995.	Read Write Check on RAM, Device Panel
-supporttypedflt 1|0	When enabled (1), the compiler passes init values through a syn_init property to the mapper. For more information, see VHDL Implicit Data-type Defaults, on page 668.	Implicit Initial Value Support, VHDL Panel

Option	Description	GUI Equivalent
-symbolic_fsm_compiler 1|0 **-autosm 1|0**	Enables/disables the FSM compiler. Controls the use of FSM synthesis for state machines. The default is false (FSM Compiler disabled). Value can be 1 or true, 0 or false. When this option is true, the FSM Compiler automatically recognizes and optimizes state machines in the design. The FSM Compiler extracts the state machines as symbolic graphs, and then optimizes them by re-encoding the state representations and generating a better logic optimization starting point for the state machines. However, if you turn off sequential optimizations for the design, FSM Compiler and/or the syn_state_machine directive and syn_encoding attribute are effectively disabled. See -no_sequential_opt 1|0 for more information on turning off sequential optimizations.	FSM Compiler check box, Device Panel
-synthesis_onoff_pragma 1|0	Determines whether code between synthesis on/off directives is ignored. When enabled, the software ignores any VHDL code between synthesis_on and synthesis_off directives. It treats these third-party directives like translate_on/ off directives (see translate_off/translate_on Directive, on page 1030 for details).	Synthesis on/off Implemented as Translate on/Off, VHDL Panel

Option	Description	GUI Equivalent
-top_module *name*	Specifies the top-level module. If the top-level entity does not use the default work library to compile the VHDL files, you must specify the library file where the top-level entity can be found. To do this, the top-level entity name must be preceded by the VHDL library followed by the dot (.).	Top-level Entity/Module, VHDL Panel or Verilog Panel
-update_models_cp 1|0	Determines whether (1) or not (0) changes inside a compile point can cause the compile point (or top-level) containing it to change accordingly.	Update Compile Point Timing Data, Device Panel
-use_fsm_explorer 1|0	Enables/disables the FSM Explorer.	FSM Explorer, Device Panel
-vlog_std v2001 | v95 | sysv	The default Verilog standard for new projects is SystemVerilog. Turning off both options in the Verilog panel defaults to v95.	Verilog 2001, SystemVerilog, 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

Timing Report Parameters for set_option

The following lists the parameters for the stand-alone timing report (ta file).

async_clock
filename
filter
gen_output_srm
margin
netlist
output_srm

Reporting Option	Description
-reporting_async_clock	Generates a report for paths that cross between clock groups using the stand-alone Timing Analyst.
-reporting_filename *filename*.**ta**	Specifies the standard timing report file (ta) generated from the stand-alone Timing Analyst.
-reporting_filter *filter options*	Generates the standard timing report based on the filter options you specify for paths, such as: • From points • Through points • To points For more information, see: • Timing Report Generation Parameters, on page 227. • Combining Path Filters for the Timing Analyzer, on page 232 • Timing Analyzer Through Points, on page 229. • Specifying From, To, and Through Points, on page 396.
-**reporting_gen_output_srm 1|0**	Specifies the new name of the output SRM File when you change the default name. If this option is set to 1, this new name is used for the output srm file after you run the stand-alone Timing Analyst.
-reporting_margin *value*	You can specify a slack margin to obtain a range of paths within the worst slack time for the design after you run the stand-alone Timing Analyst.
-reporting_netlist *filename*.**srm**	Specifies the associated gate-level netlist file (srm) generated from the stand-alone Timing Analyst.
-reporting_output_srm 1|0	Allows you to change the name of the output srm file. If you enable the output SRM File option, you can change this default name.

For GUI equivalent switches for these parameters, see Timing Report Generation Parameters, on page 227.

help for set_option

This option is useful for getting syntax help on the various implementation options used for compiling and mapping a design, especially since this list of options keeps growing.

Syntax

```
% set_option -help
```

Usage:

set_option *optionName optionValue* [**-help** [*value*]]

Where:

- *optionName*—specifies the option name.
- *optionValue*—specifies the option value.
- -help [*value*]—to get help on options. Use:
  - -help * for the list of options
  - -help *optionName* for a description of the option

Examples

To list all option commands in the Tcl window:

```
set_option -help *
```

To list all option commands beginning with the letters fi in the Tcl window:

```
% set_option -help sy*
```

```
symbolic_fsm_compiler
synthesis_onoff_pragma
```

To get help on a specific option in the Tcl window:

```
% set_option -help symbolic_fsm_compiler
```

```
Extracts and optimizes finite state machines.
```

Use the following Tcl commands to print a description of the options:

```
% set_option -help c*
% set hl [set_option -help c*]
% puts $hl
% foreach option $hl { puts "$option:\t [set_option -help
$option]"; }
```

This example will print a list of set_option options that begin with the letter c.

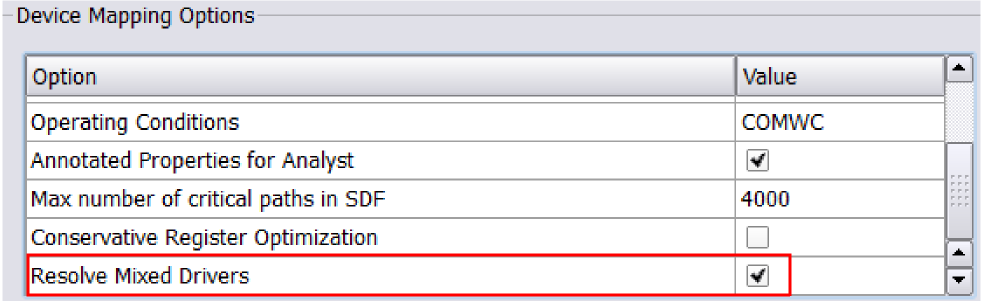
Resolve Mixed Drivers Option

Use the Resolve Mixed Drivers option when mapping errors are generated for input nets with mixed drivers. You might encounter the following messages in the log file:

```
@A:BN313 | Found mixed driver on pin pin:data_out inst:dpram_lut3
of work.dpram(verilog), use option "Resolve Mixed Drivers" in
"Device" tab of "Implementation Options" to automatically resolve
this
@E:BN314 | Net "GND" in work.test(verilog) has mixed drivers

@A:BN313 | Found mixed driver on pin pin:Q[0] inst:dff1.q of
PrimLib.sdffr(prim), use option "Resolve Mixed Drivers" in
"Device" tab of "Implementation Options" to automatically resolve
this
@E:BN314) | Net "VCC" in work.test(rtl) has mixed drivers
```

Whenever a constant net (GND or VCC) and an active net are driving the same output net, enable the Resolve Mixed Drivers option so that synthesis can proceed. To set this switch:

- Check Resolve Mixed Drivers on the Device tab of the Implementation Options panel.



Device Mapping Options

Option	Value
Operating Conditions	COMWC
Annotated Properties for Analyst	☑
Max number of critical paths in SDF	4000
Conservative Register Optimization	□
Resolve Mixed Drivers	☑

- Use the Tcl command, `set_option -resolve_multiple_driver 1`.

  By default this option is disabled and set to:

  `set_option -resolve_multiple_driver 0.`

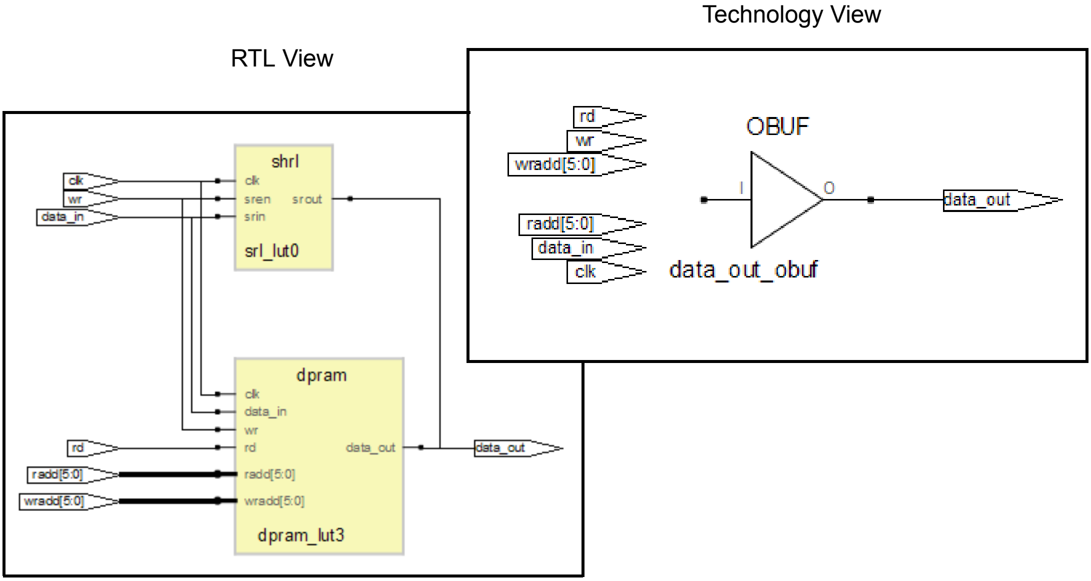
When you rerun synthesis, you should now see messages like the following in the log file:

```
@W:BN312 | Resolving mixed driver on net GND, connecting output
pin:data_out inst:dpram_lut3 of work.dpram(verilog) to GND
@N:BN116) | Removing sequential instance dpram_lut3.dout of
view:PrimLib.dffe(prim) because there are no references to its
outputs
@N:BN116 | Removing sequential instance dpram_lut3.mem of
view:PrimLib.ram1(prim) because there are no references to its
outputs

@W:BN312 | Resolving mixed driver on net VCC, connecting output
pin:Q[0] inst:dff1.q of PrimLib.sdffr(prim) to VCC
@N:BN116 | Removing sequential instance dff1.q of
view:PrimLib.sdffr(prim) because there are no references to its
outputs
```

Example – Active Net and Constant GND Driving Output Net (Verilog)

```
module test(clk,data_in,data_out,radd,wradd,wr,rd);
input clk,wr,rd;
input data_in;
input [5:0]radd,wradd;
output data_out;
// component instantiation for shift register module
shrl srl_lut0 (
   .clk(clk),
   .sren(wr),
   .srin(data_in),
   .srout(data_out)
   );
// Instantiation for ram
dpram dpram_lut3 (
   .clk(clk),
   .data_in(data_in),
   .data_out(data_out),
   .radd(radd),
   .wradd(wradd),
   .wr(wr),
   .rd(rd)
   );

endmodule

module shrl (clk,sren,srin,srout);
input clk;
input sren;
input srin;
output srout;

parameter width = 32;
reg [width-1:0] sr;

always@(posedge clk)
begin
   if (sren == 1)
   begin
      sr <= {sr[width-2:0], srin};
   end
end
// Constant net driving

// the output net
assign srout = 1'b0;
endmodule

module dpram(clk,data_in,data_out,radd,wradd,wr,rd);
input clk,wr,rd;
input data_in;
input [5:0]radd,wradd;
output data_out;

reg dout;
reg [0:0]mem[63 :0];

always @ (posedge clk)
begin
   if(wr)
      mem[wradd] <= data_in;
end

always @ (posedge clk)
begin
   if(rd)
      dout <= mem[radd];
   end

assign  data_out = dout;

endmodule
```

See the following RTL and Technology views; the Technology view shows the constant net tied to the output.



RTL View

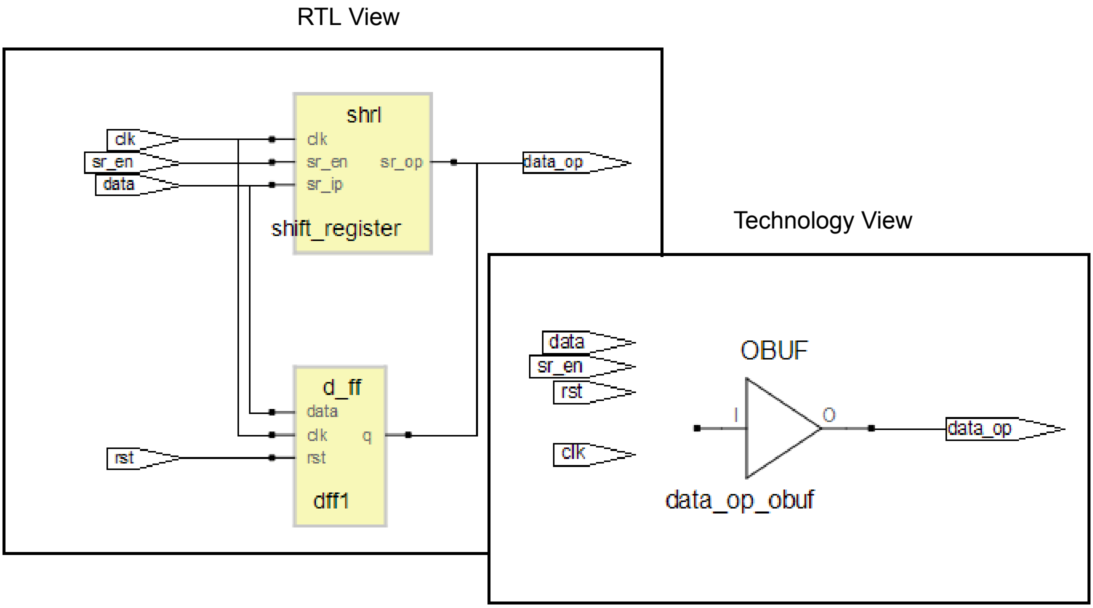
Technology View

Example – Active Net and Constant VCC Driving Output Net (VHDL)

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

entity test is
port (clk,rst : in std_logic;
      sr_en : in std_logic;
      data : in std_logic;
      data_op : out std_logic );
end entity test;

architecture rtl of test is
component shrl
generic (sr_length : natural);
port (clk : in std_logic;
      sr_en : in std_logic;
      sr_ip : in std_logic;
      sr_op : out std_logic );
end component shrl;

component d_ff
port (data, clk, rst : in std_logic;
      q : out std_logic );
end component d_ff;

begin
-- instantiation of shift register
shift_register : shrl
generic map (sr_length => 64)
port map (clk => clk,
          sr_en => sr_en,
          sr_ip => data,
          sr_op => data_op );
-- instantiation of flipflop
dff1 : d_ff
port map (data => data,
          clk => clk,
          rst => rst,
          q => data_op );
end rtl;

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

```
entity shrl is
generic (sr_length : natural);
port (clk : in std_logic;
      sr_en : in std_logic;
      sr_ip : in std_logic;
      sr_op : out std_logic );
end entity shrl;

architecture rtl of shrl is
signal sr_reg : std_logic_vector(sr_length-1 downto 0);
begin
   shreg_lut: process (clk)
   begin
      if rising_edge(clk) then
         if sr_en = '1' then
            sr_reg <= sr_reg(sr_length-2 downto 0) & sr_ip;
         end if;
      end if;
   end process shreg_lut;
-- Constant net driving output net
sr_op <= '1';
end architecture rtl;

library IEEE;
use IEEE.std_logic_1164.all;

entity d_ff is
port (data, clk, rst : in std_logic;
      q : out std_logic );
end d_ff;

architecture behav of d_ff is
begin
   FF1:process (clk) begin
      if (clk'event and clk = '1') then
         if (rst = '1') then
            q <= '0';
         else q <= data;
         end if;
      end if;
   end process FF1;
end behav;
```

See the following RTL and Technology views; the Technology view shows the constant net tied to the output.



Technology-specific Tcl Commands

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

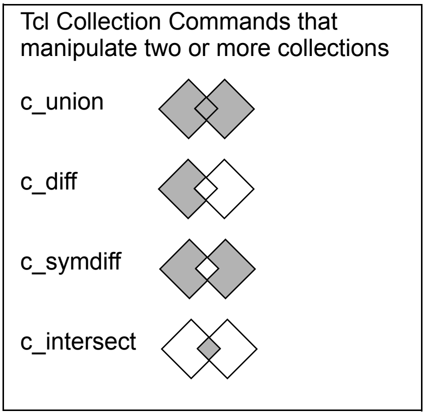
Vendor/Family	Tcl Commands Described in...
Microsemi	Microsemi Tcl set_option Command Options, on page 1146

Collection Commands

The following commands and operators work on collections created by the find and expand commands (see expand Command (Batch), on page 1095 and find Command (Batch), on page 1098). For information on using these commands and collections, see Creating and Using Collections (SCOPE Window), on page 77 in the *User Guide*.

Use the collection commands to create, copy, evaluate, traverse and filter collections. You can add to (union) and subtract from (difference) collections using the operators. Once you have defined a collection or a list, you can apply the same constraint to all the objects in the collection. You can do this from both the SCOPE editor (see Collections, on page 371) or in the Tcl file.

Command	**Description**
Creation	
define_collection	Creates a collection from a list
set modules	Creates a collection
set modules_copy $*modules*	Copies a collection
Comparison and Analysis	
c_diff	Identifies differences between lists or collections
c_intersect	Identifies objects common to a list and a collection
c_symdiff	Identifies objects that belong exclusively to only one list or collection
c_union	Concatenates a list to a collection
Evaluation and Statistics	
c_info	Prints statistics for a collection
c_list	Converts a collection to a Tcl list for evaluation
c_print	Displays collections or properties for evaluation


Tcl Collection Commands that manipulate two or more collections
c_union
c_diff
c_symdiff
c_intersect

c_diff

The c_diff command lets you difference a list to a collection. For this command to work, the design must be open in the GUI.

Example

```
%set mycollection {i:reg1 i:reg2}
%set subcollection [c_diff $mycollection {i:reg1}]
%c_print $subcollection
{i:reg2}
```

You can also difference two or more collections using the c_diff command as shown in the following command-line syntax:

set *reducedCollectionName* **[c_diff $***collection1* **$***collection2***]**

This command also includes a -print option to display the result.

c_info

The c_info command returns specifics of a collection including database name, number of objects per type, and total number of objects. You can save results to a Tcl variable (array) using the -array *name* option.

Syntax

c_info $*mycollection* [**-array** *name*]

c_intersect

Intersect defines the objects that belong to all collections. You intersect a list to a collection using the c_intersect command.

Example

```
%set mycollection {i:reg1 i:reg2}
%set intercollection [c_intersect $mycollection {i:reg1 i:reg3}]
%c_print $intercollection
   {i:reg1}
```

You intersect two or more collections using the c_intersect command as shown in the following command-line syntax:

set *reducedCollectionName* **[c_intersect $***collection1* **$***collection2***]**

This command also includes a -print option to display the result.

c_list

The c_list command converts a collection to a Tcl list of objects. Any collection can be evaluated using this command. You can manipulate the list of objects by assigning the collection to a variable and then using standard Tcl list commands (lappend, lsort, etc). You can optionally specify object properties to add to the resulting list using the -prop option:

```
(object prop_value ... prop_value)...
   (object prop_value ... prop_value)
```

Syntax

c_list $*collection***|***list* [**-prop** *propertyName*]*

Example

```
$set modules [find -view *]
%c_list $modules
{v:top}{v:block_a}{v:block_b}

%c_list $modules –prop is_vhdl –prop is_verilog

Name          is_vhdl     is_verilog
{v:top}       0           1
{v:block_a}   1           0
{v:block_b}   1           0
```

c_print

The c_print command displays collections or properties in column format. Object properties are printed using one or more –prop *propertyName* options.

To print to a file, use the -file option. The following command in an `fdc` file prints the whole collection to a file:

```
c_print –file foo.txt $col
```

Note that the command prints the file to the current working directory. If you have multiple projects loaded, check that the file is written to the correct location. You can use the `pwd` command in the Tcl window to echo the current directory and then use cd *directoryName* to change the directory as needed.

Syntax

c_print $_collection_|_list_ [**-prop** _propertyName_]* [**-file** _filename_]

Example

```
%set modules [find -view *]
%c_print $modules
{v:top}
{v:block_a}
{v:block_b}
```

```
%c_print -prop is_vhdl -prop is_verilog $modules
Name is_vhdl is_verilog
{v:top}0 1
{v:block_a}1 0
{v:block_b}1 0
```

c_symdiff

Symmetrical difference defines the objects that exclusively belong to only one collection. This is the compliment of intersect.

You symmetrically difference a list to a collection using the c_symdiff command:

```
%set mycollection {i:reg1 i:reg2}
%set diffcollection [c_symdiff $mycollection {i:reg1 i:reg3}]
%c_list $diffcollection
   {i:reg2} {i:reg3}
```

You can symmetrically difference two or more collections using the c_symdiff command as shown in the following command-line syntax:

set *reducedCollectionName* **[c_symdiff $***collection1* **$***collection2***]**

This command also includes a -print option to display the result.

c_union

The c_union command adds a list to a collection. For this command to work, the design must be open in the GUI.

Example

```
%set mycollection [find -instance {reg?} -print]
   {i:reg1}
   {i:reg2}
%set sumcollection [c_union $mycollection {i:reg3}]
%c_list $sumcollection
   {i:reg1} {i:reg2} {i:reg3}
```

You can concatenate two collections using the c_union command as shown in the following syntax:

set *combinedCollectionName* **[c_union $***collection1* **$***collection2***]**

The c_union function automatically removes redundant elements

```
%set mycollection {i:reg1 i:reg2}
%set sumcollection [c_union $mycollection {i:reg2 i:reg3}]
%c_list $sumcollection
   {i:reg1} {i:reg2} {i:reg3}
```

This command also includes a -print option to display the result.

get_prop

Returns a single property value in a Tcl list for each member of the collection.

Example

```
get_prop -prop clock [find -seq *]
```

define_collection

Creates a collection from any combination of single elements, Tcl lists, and collections. You get a warning message about empty collections if you define a collection with a leading asterisk and then define an attribute for it, as shown here:

```
define_scope_collection  noretimesh {find -hier -inst -seq
   *uc_alu.*}
define_attribute {$noretimesh} {syn_allow_retiming} {0}
```

To avoid the error message, remove the leading asterisk and change `*uc_alu` to `uc_alu`.

Example

```
set modules [define_collection {v:top} {v:cpu} $mycoll $mylist]
```

define_scope_collection

Same as define_collection, on page 1093.

set

The set command is used to create or copy a collection. Note that set copies the collection (not the collection handle) and that changes to $*collectionName* do not update the copied collection.

Syntax

set *collectionName collectionCriteria*

set *copyName* **$***collectionName*

Arguments and Options

collectionName

The name of the new collection.

collectionCriteria

The collection criteria for defining the elements to be included in the collection (see examples below).

copyName

The name assigned to the copied collection.

$*collectionName*

The name of an existing collection.

Examples

Using the set command with find and a variable name to create a collection:

```
set my_module [find -view *]
```

Using the set command with define_collection to create a collection:

```
set my_module [define_collection {v:top} {v:cpu} $col_l $mylist]
```

Using the set command to copy a collection:

```
set my_mod_copy $my_module
```

expand Command (Batch)

The expand command provides a way to generate a connectivity-based collection of objects. For more information, see Using the Expand Tcl Command to Define Collections, on page 86 of the *User Guide*.

The syntax for the expand command is as follows:

expand [-*objectType*] [**-from** *object*] [**-thru** *object*] [**-to** *object*] [**-level** *integer*]
[**-hier**] [**-leaf**] [**-seq**] [**-print**]

See Tcl Expand Examples, on page 1097.

Argument	Description
-from *object*	Specifies a list or collection of ports, instances, pins, or nets for expansion forward from all the pins listed. Instances and input pins are automatically expanded to all output pins of the instances. Nets are expanded to all output pins connected to the net. If you do not specify this argument, backward propagation stops at all sequential elements.
-hier	Modifies the range of any expansion to any level below the current view. The default for the current view is the top level and is defined with the define_current_design command as in the compile-point flow.
-leaf	Returns only non-hierarchical instances.
-level *integer*	Limits the expansion to N logic levels of propagation. You cannot specify more than one -from, -thru, or -to point when using this option.
-*objectType*	Optionally specifies the type of object to be returned by the expansion. If you do not specify an *objectType*, all objects are returned. The object type is one of the following: • **instance** returns all instances between the expansion points. This is the default. • **pin** returns all instance pins between the expansion points. • **net** returns all nets between the expansion points. • **port** returns all top-level ports between the expansion points.

Argument	Description

-print	Evaluates the expand function and prints the first 20 results. If you use this command from HDL Analyst, these results are printed to the Tcl window; for fdc file commands, the results are printed to the log file at the start of the Mapper section.
	For a full list of objects found, you must use c_print or c_list. Reported object names have prefixes that identify the object type. There are double quotes around each name to allow for spaces in the names. For example:
	`"i:reg1"`
	`"i:reg2"`
	`"i:\weird_name[foo$]"`
	`"i:reg3"`
	`<<found 233 objects. Displaying first 20 objects. Use c_print or c_list for all. >>`

-seq	Modifies the range of any expansion to include only sequential elements. By default, the expand command returns all object types. If you want just sequential instances, make sure to define the *object_type* with the -inst argument, so that you limit the command to just instances.
-thru *object*	Specifies a list or collection of instances, pins, or nets for expansion forward or backward from all listed output pins and input pins respectively. Instances are automatically expanded to all input/output pins of the instances. Nets are expanded to all input/output pins connected to the net. You can have multiple -thru lists for product of sum (POS) operations.
-to *object*	Specifies a list or collection of ports, instances, pins, or nets for expansion backward from all the pins listed. Instances and output pins are automatically expanded to all input pins of the instances. Nets are expanded to all input pins connected to the net. If you do not specify this argument, forward propagation stops at all sequential elements.

Tcl Expand Examples

Example	Description
expand -hier -from {i:reg1} -to {i:reg2}	Expands the cone of logic between two registers. Includes hierarchical instances below the current view.
expand -inst -from {i:reg1}	Expands the cone of logic from one register. Does not include instances below the current view.
expand -inst -hier -to {i:reg1}	Expands the cone of logic to one register. Includes hierarchical instances below the current view.
expand -pin -from {t:i_and2.z} -level 1	Finds all pins driven by the specified pin. Does not include pins below the current view.
expand -hier -to {t:i_and2.a} -level 1	Finds all instances driving an instance. Includes hierarchical instances below the current view.
expand -hier -from {n:cen}	Finds all elements in the transitive fanout of a clock enable net, across hierarchy.
expand -hier -from {n:cen} -level 1	Finds all elements directly connected to a clock enable net, across hierarchy.
expand -hier -thru {n:cen}	Finds all elements in the transitive fanout and transitive fanin of a clock enable net, across hierarchy.

find Command (Batch)

The Tcl find command lets you search for design objects. Use this command to form collections and then apply constraints to the group. This command operates on the RTL database. The following provide more information:

For...	See...
Command syntax	Tcl Find Syntax, on page 1098
Syntax details, like objects, expressions, and special characters	Tcl Find Command Object Types, on page 1101 Tcl Find Command Regular Expression Syntax, on page 1102 Tcl Find Command Special Characters, on page 1103 Tcl Find Command Case Sensitivity, on page 1104 Find Filter Properties, on page 1105
Brief examples of find syntax	synhooks File Syntax, on page 1110
Details on forming collections and apply constraints	Using the Tcl Find Command to Define Collections, on page 83 in the *User Guide*.
Using find in batch mode	open_design, on page 1049.

Tcl Find Syntax

find [-*objectType*] [*patterns*]
  [**-in $***collectionName* | *listName*]
  [**-namespace techview | netlist**]
  [**-hier**]
  [**-flat**]
  [**-leaf**]
  [**-hsc** *separator*]
  [**-regexp**]
  [**-rtl | -tech**]
  [**-nocase**]
  [**-exact**]
  [**-print**]
  [**-filter** *expression*] ** the -filter option must appear last in the syntax.

Argument	Description
-*objectType*	Specifies the type of objects to be found such as views, ports, instances, pins, or nets. For details, see Tcl Find Command Object Types, on page 1101. You can further match names to patterns, which can include the * and ? wildcard characters.
-in **$***collectionName* | *listName*	Restricts the search to the specified list or collection.
-namespace techview | netlist	Determines the location to search for the find operation. When you specify techview, the command searches the mapped (srm) database; when you specify netlist, it searches the output netlist. This option is not available for an RTL view.
-hier	Restricts the search to any level below the current view. The default for the current level is the top level, so the command searches the entire design. See Tcl Find Command Special Characters, on page 1103 for additional information.
-flat	Allows wildcard * to match the hierarchy separator.
-leaf	Returns only non-hierarchical instances.
-hsc *separator*	Specifies the hierarchy delimiter character. The default is the dot (.). In some cases, using the dot as a delimiter and as a normal character results in ambiguity. For example, block1.u1 could be an instance u1 in block1, or a record named block1.u1. You can use the -hsc *separator* option to specify an unambiguous character as the hierarchy delimiter. For example, find -hsc "@" [block1@u1] finds the hierarchical instance in the block, while find -hsc "@" [block1.u1] finds the record. SeeTcl Syntax Guidelines for Constraint Files, on page 47 for more information.
-regexp	Treats the pattern as a regular expression instead of a simple wildcard pattern. It also uses the == and != filter operators to compare regular expressions rather than simple wildcard patterns. You cannot use this argument with -nocase or -exact. See Tcl Find Command Regular Expression Syntax, on page 1102 and Tcl Find Command Special Characters, on page 1103 for details about regular expressions.
-rtl | -tech	Uses the most recently activated RTL or Technology view, and if none are available, opens a new view.

Argument	Description
-nocase	Ignores case when matching patterns. The default is to take case into account (-case). You cannot use the -nocase argument with -exact or -regexp. See Tcl Find Command Case Sensitivity, on page 1104 for more information.
-exact	Disables simple pattern matching. Use it to search for objects that contain the * and ? wildcard characters. You cannot use this argument with -nocase or -regexp. See Tcl Find Command Special Characters, on page 1103 for additional information.

-print	Evaluates the `find` function and prints the first 20 results. If you use this command from HDL Analyst, these results are printed to the Tcl window; for `fdc` file commands, the results are printed to the log file, at the start of the Mapper section.
	For a full list of objects found, you must use c_print or c_list. Reported object names have prefixes that identify the object type. There are double quotes around each name to allow for spaces in the names. For example:
	`"i:reg1"`
	`"i:reg2"`
	`"i:\weird_name[foo$]"`
	`"i:reg3"`
	`<<found 233 objects. Displaying first 20 objects. Use c_print or c_list for all. >>`

-filter *expression*	Filters objects based on their properties. This option must appear last in the find syntax.
	Syntax and examples:
	find *pattern* **-filter** *@prop|operator|value*
	`find * -filter @fanout>8`
	`find * -hier -filter (@view!=dff) -inst *`
	`find * -hier -filter @view!=andv || @view!=orv`
	`find -hier -inst * -filter @inst_of==statemachine`
	`find -hier -inst * -filter @kind==statemachine`
	Boolean expressions on multiple properties are supported:
	`find * -filter @fanout>8 && @slack<0`
	Use find to check for the absence of a property:
	`c_print [find -hier -view {*} -filter (!@is_black_box)]`
	The example above finds all objects that are not black boxes.
	See Find Filter Properties, on page 1105 for a summary of supported properties.

Tcl Find Command Examples

The following are examples of find syntax:

Example	Description
find {a*}	Finds any object in the current view that starts with a
find {a*} -hier -nocase	Finds any object that starts with a or A
find -net {*synp*} -hier	Finds any net the contains synp
find -seq * -filter {@clock==myclk}	Finds any register in the current view that is clocked by myclk
find -flat -seq {U1.*}	Finds all sequential elements at any hierarchical level under U1 (* matches hierarchy separator)

Tcl Find Command Object Types

You can specify the following types of objects:

Object	Prefix	Description and Example	Synopsys
view	v:	A design. v:work.cpu.rtl is the master cell of the cpu entity, rtl architecture, compiled in the VHDL work library.	lib_cell
instance	i:	An instance in a view. This is the default object type. i:core.i_cpu.reg1 points to the reg1 instance inside i_cpu.	cell
port	p:	A port. p:data_in[3] points to bit 3 of the primary data_in port. work.cpu.rt1|p:rst is the hierarchical rst port in the cpu view. This eventually points to all instances of cpu.	port

Object	Prefix	Description and Example	Synopsys
pin	t:	A pin. t:core.i_cpu.rst points to the hierarchical rst pin of instance i_cpu.	pin
net	n:	A net. n:core.i_cpu.rst points to the rst net driven in i_cpu.	net
seq	i:	A sequential instance. i:core.i_cpu.reg[7:0]	

Tcl Find Command Regular Expression Syntax

You can use the following meta characters:

Syntax	Matches...
Meta characters: Used to match certain conditions in a string	
^	At the beginning of the string
$	At the end of the string
.	Any character. If you want to use the dot (.) as a hierarchy delimiter, you must escape it with a backslash (\), because it has a special meaning in regular expressions.
\k	The specified non-alphanumeric character (where *k* is the non-alphanumeric character) when it is used as an ordinary character. For example, \$ matches a dollar character.
\c	The specified non-alphanumeric character (where *c* is the non-alphanumeric character) when it is used in an escape sequence.
|	Equivalent to an OR.
Character Class: A list of characters to match	
[*list*]	Any single character from the *list*. For example, [abc] matches a lower-case a, b, or c. To specify a range of characters in the list, use a dash. For example, [A-Za-z] matches any alphabetical character.
[^*list*]	Characters not in the list. You can specify a range, as described above. For example, [^0-9] matches any non-numeric character.

Syntax	Matches...
\	Used as a prefix to escape special characters like the following: ^ $ \ . | ( ) [ ] { } ? + *
Escape Sequence: Shortcuts for common character class	
\d	A digit between 0 and 9
\D	A non-numeric character
\s	A white space character
\S	A non-white space character
\w	A word character; i.e., alphanumeric characters or underscores
\W	A non-word character
Quantifiers: Number of times to match the preceding pattern	
*	A sequence of 0 or more matches
+	A sequence of 1 or more matches
?	A sequence of 0 or 1 match
{*N*}	A sequence of exactly *N* matches
{*N*,}	A sequence of *N* or more matches
{*N*,*P*}	A sequence of *N* through *P* matches (*P* included); *N*<=*P*

Tcl Find Command Special Characters

The following describe how the Tcl find command handles special characters:

- Tcl uses square brackets [] and dollar signs $ as special characters. Use curly brackets {} or double quotes to prevent the interpretation of special characters within a pattern.
- When -regexp is not specified, there are only two special characters that are used as wildcards: * and ?. See Tcl Find Command Regular Expression Syntax, on page 1102 for information about regular expressions.

- When -hier is not specified, the search is limited to the current view and wildcards do not match the hierarchy delimiter character. You can still traverse downward through the hierarchy by adding the delimiter in the pattern. Thus, "*" matches any object at the current level, but "*.*" matches any object one level below the current view.
- The find command significantly differs from a simple Tcl search. A simple Tcl search does not treat any character (except for the backslash, \) as a special character, so * matches everything in a string. This means that it searches through hierarchies with forward slash (/) separators, because it does not treat them as special characters.

  On the other hand, the Tcl find command confines searches to within the levels of hierarchy. If you specify the following command, the tool searches each level of hierarchy individually within the search pattern:

  ```
  find -hier *abc*addr_reg[*]
  ```

  If you want to search through the hierarchy, you must add hierarchy separators to the search pattern"

  ```
  find {*.*.abc.*.*.addr_reg[*]
  ```

- The dot character (.) is treated as a hierarchy separator. If a dot is part of an instance name, you must use an escape character to indicate this:

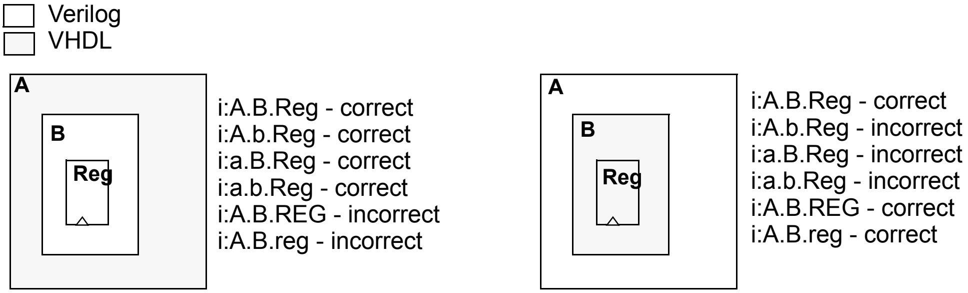
  ```
  a.b.c\.d.e
  ```

  If not, the find command treats it as hierarchy separator when it is trying to match a pattern.

Tcl Find Command Case Sensitivity

Case sensitivity matches the language from which the object was generated: case-insensitive for VHDL, and case-sensitive for Verilog.

In mixed-language designs, the case-sensitivity rules for the parent object are used when the path of a lower-level object crosses a language boundary.



i:A.B.Reg - correct
i:A.b.Reg - correct
i:a.B.Reg - correct
i:a.b.Reg - correct
i:A.B.REG - incorrect
i:A.B.reg - incorrect

i:A.B.Reg - correct
i:A.b.Reg - incorrect
i:a.B.Reg - incorrect
i:a.b.Reg - incorrect
i:A.B.REG - correct
i:A.B.reg - correct

Find Filter Properties

These properties are design or constraint oriented. They are used to qualify searches and build collections. To generate these properties, open Project->Implementation Options->Device and enable the Annotated Properties for Analyst check box. The tool creates `.sap` and `.tap` files (design and timing properties, respectively) in the project folder.

The table below lists the common filter object properties. It does not include some vendor-specific properties. Use the table to determine properties that you want to filter. Here is how to read the columns:

- Property Name – name to use in the find -filter syntax; property name must be preceded with the @ character, for example, if clock is the property name: {@clock==myclk}.
- Property Value – value to specify when using operators in the filter expression; value can be object names such as {@clock==myclk}, where myclk is the name of the object, or values, such as {@fanout>=60}. See find Command (Batch), on page 1098, syntax on -filter option for details.

- View – find properties display in the Tcl window when either HDL Analyst RTL or Technology view is active. However, if you want to use find in collections to apply constraints during synthesis, see Comparing Methods for Defining Collections, on page 76 for more information. All specifies that the property is available in both views.
- Comment – additional information about the design object.

Property Name	Property Value	HDL View	Comment
Common Properties			
type	view | port | net | instance | pin	All	
View Properties			
compile_point	locked	Tech	
is_black_box	1	All	
is_verilog	0 | 1	All	
is_vhdl	0 | 1	All	
syn_hier	remove | flatten | soft | firm | hard	Tech	
Port Properties			
direction	input | output | inout	All	
fanout	*value*	All	Total fanout (integer)
Instance Properties			
area	*area_value*	Tech	
arrival_time	*value*	Tech	Corresponds to worst slack
async_reset	**n:***netName*	All	Idem
async_set	**n:***netName*	All	Idem
clock	*clockName*	All	Could be a list if there are multiple clocks

Property Name	Property Value	HDL View	Comment
clock_edge	rise | fall | high | low	All	Could be a list if there are multiple clocks
clock_enable	**n:***netName*	All	Highest branch name in the hierarchy, and closest to the driver
compile_point	locked	Tech	Automatically inherited from its view
hier_rtl_name	*hierInstanceName*	All	
inout_pin_count	value	All	
input_pin_count	value	All	
inst_of	viewName	All	
is_black_box	1 (Property added)	All	Automatically inherited from its view

is_hierarchical	1 (Property added)	All	Use the -filter option to check for the property's existence:
			`-filter @is_combinational`
			checks for the existence
			`-filter !@is_combinational`
			checks for the absence
			Also see, the -filter option for the Tcl Find Syntax, on page 1098.

is_sequential	1 (Property added)	All	
is_combinational	1(Property added)	All	
is_pad	1(Property added)	All	
is_tristate	1(Property added)	All	
is_keepbuf	1(Property added)	All	
is_clock_gating	1(Property added)	All	
is_vhdl	0 | 1	All	Automatically inherited from its view
is_verilog	0 | 1	All	Automatically inherited from its view
kind	*primitive* For example: inv | and |dff | mux | statemachine | ...)	All	Tech view contains vendor-specific primitives

Property Name	Property Value	HDL View	Comment
location	(*x*, *y*)	Tech	Format can differ
name	*instanceName*	All	
orientation	N | S | E | W	Tech	
output_pin_count	*value*	All	
pin_count	*value*	All	
placement_type	unplaced | placed	All	
rtl_name	*nonhierInstanceName*	All	
slack	*value*	Tech	Worst slack of all arcs
slow	1	Tech	
sync_reset	**n:***netName*	All	Idem
sync_set	**n:***netName*	All	Idem
syn_hier	remove | flatten | soft | firm | hard	Tech	Automatically inherited from its view
view	*viewName*	All	
Pin Properties			
arrival_time	*timingValue*	Tech	
clock	*clockName*	All	Could be a list if there are multiple clocks
clock_edge	rise | fall | high | low	All	Could be a list if there are multiple clocks
direction	input | output | inout	All	
fanout	*value*	All	Total fanout (integer)
is_clock	0 | 1	All	
is_gated_clock	0 | 1	All	Set in addition to is_clock
slack	*value*	Tech	

Property Name	Property Value	HDL View	Comment
Net Properties			
clock	*clockName*	All	Could be a list if there are multiple clocks
is_clock	0|1	All	
is_gated_clock	0|1	All	Set in addition to is_clock
fanout	*value*	All	Total fanout (integer)

Tcl Find Command Examples

The following are examples of find syntax:

Example	Description
find {a*}	Finds any object in the current view that starts with a
find {a*} -hier -nocase	Finds any object that starts with a or A
find -net {*synp*} -hier	Finds any net the contains synp
find -seq * -filter {@clock==myclk}	Finds any register in the current view that is clocked by myclk
find -flat -seq {U1.*}	Finds all sequential elements at any hierarchical level under U1 (* matches hierarchy separator)

synhooks File Syntax

The Tcl hooks commands provide an advanced user with callbacks to customize a design flow or integrate with other products. To enable these callbacks, set the environment variable SYN_TCL_HOOKS to the location of the Tcl hooks file(synhooks.tcl), then customize this file to get the desired customization behavior. For more information on creating scripts using synhooks.tcl, see Automating Flows with synhooks.tcl, on page 463.

Tcl Callback Syntax	Function
proc syn_on_set_project_template {*projectPath***} {***yourDefaultProjectSettings***}**	Called when creating a new project. *projectPath* is the path name to the project being created.
proc syn_on_new_project {*projectPath***} {***yourCode***}**	Called when creating a new project. *projectPath* is the path name to the project being created.
proc syn_on_open_project {*projectPath***} {***yourCode***}**	Called when opening a project. *projectPath* is the path name to the project being created.
proc syn_on_close_project {*projectPath***} {***yourCode***}**	Called after closing a project. *projectPath* is the path name to the project being created.
proc syn_on_start_application {*applicationName version currentDirectory***} {***yourCode***}**	Called when starting the application. • *applicationName* is the name of the software. For example synplify_pro. • *version* is the name of the version of the software. For example 8.4 • *currentDirectory* is the name of the software installation directory. For example C:\synplify_pro\bin\synplify_pro.exe.
proc syn_on_exit_application {*applicationName version***} {***yourCode***}**	Called when exiting the application. • *applicationName* is the name of the software. For example synplify_pro. • *version* is the name of the version of the software. For example 8.4.

Tcl Callback Syntax	Function
proc syn_on_start_run {*runName projectPath implementationName***}** **{***yourCode***}**	Called when starting a run. • *runName* is the name of the run. For example compile or synthesis. • *projectPath* is the location of the project. • *implementationName* is the name of the project implementation. For example, rev_1.
proc syn_on_end_run {*runName projectPath implementationName***}** **{***yourCode***}**	Called at the end of a run. • *runName* is the name of the run. For example, compile or synthesis. • *projectPath* is the location of the project. • *implementationName* is the name of the project implementation. For example, rev_1.
proc syn_on_press_ctrl_F8 {} **{***yourCode***}**	Called when Ctrl-F8 is pressed. See Tcl Hook Command Example next.
proc syn_on_press_ctrl_F9 {} **{***yourCode***}**	Called when Ctrl-F9 is pressed.
proc syn_on_press_ctrl_F8 {} **{***yourCode*	Called when Ctrl-F11 is pressed.

Tcl Hook Command Example

Create a modifier key (ctrl-F8) to get all the selected files from a project browser and project directory.

```
set sel_files [get_selected_files]
while {[expr [llength $sel_files] > 0]} {
   set file_name [lindex $sel_files 0]
      puts $file_name
   set sel_files [lrange $sel_files 1 end]
}
```

Log File Commands

The Synplify Pro software supports Tcl commands that let you filter messages in the log file. The commands are:

- log_filter Tcl Command, on page 1112
- log_report Tcl Command, on page 1113

log_filter Tcl Command

This command lets you filter errors, notes, and warning messages. The GUI equivalent of this command is the Warning Filter dialog box, which you access by selecting the Warnings tab in the Tcl window and then clicking Filter. For information about using this command, see Filtering Messages in the Message Viewer, on page 247 in the *User Guide*.

Syntax

log_filter -field *fieldName***==***value*
log_filter -show_matches
log_filter -hide_matches
log_filter -enable
log_filter -disable
log_filter -clear

The following table shows valid *fieldName* and *value* values for the -field option:

Fieldname	Value
type	Error | Warning | Note
id	The message ID number. For example, MF138
message	The text of the message. You can use wildcards.
source_loc	The name of the HDL file that generated the message.
log_loc	The corresponding srr file (log).
time	The time the message was generated.
report	The log file section. For example, Compiler or Mapper.

Example

```
log_filter -hide_matches
log_filter -field type==Warning -field message==*Una*
   -field source_loc==sendpacket.v -field log_loc==usbHostSlave.srr
   -field report=="Compiler Report"
log_filter -field type==Note
log_filter -field id==BN132
log_filter -field id==CL169
log_filter -field message=="Input *"
log_filter -field report=="Compiler Report"
```

log_report Tcl Command

This command lets you write out the results of the log_filter command to a file. For information about using this command, see Filtering Messages in the Message Viewer, on page 247 in the *User Guide*.

Syntax

You specify this command after the log_filter commands.

log_report -print *fileName*

Example

```
log_report -print output.txt
```

Tcl Script Examples

This section provides the following examples of Tcl scripts:

- Using Target Technologies, on page 1114
- Different Clock Frequency Goals, on page 1115
- Setting Options and Timing Constraints, on page 1116

Using 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 {

         ProASIC ProASIC3E AX # 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 PROASIC3E
   set_option -part A3PE600
   set_option -package PQFP208
   set_option -speed_grade -2

# 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.fdc"

# 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.fdc," is read by "test3.tcl"
# with the add_file -constraint "design.fdc" 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
```

APPENDIX A

Designing with Microsemi

The following topics describe how to design and synthesize with the Microsemi technology:

Basic Support for Microsemi Designs

This section describes the use of the synthesis tool with Microsemi devices. It describes

- Microsemi Device-specific Support, on page 1120
- Microsemi Features, on page 1121
- Synthesis Constraints and Attributes for Microsemi, on page 1121

Microsemi Device-specific Support

The synthesis tool creates technology-specific netlists for a number of Microsemi 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 Implementation Options dialog box (see Device Panel, on page 159).

Synthesis supports the following technologies:

Low-Power FPGAs

- ProASIC3E

Antifuse FPGAs

- Axcelerator
- eX

Legacy PFGAs

- ACT1, ACT2/1200L, and ACT3
- ProASIC^{PLUS}
- 3200DX, 40MX, 42MX, 54SX, and 54SXA

After synthesis, the synthesis tool generates EDIF netlists as well as a constraint file that is forward annotated as input into the Microsemi Libero tool.

Microsemi Features

The synthesis tool contains the following Microsemi-specific features:

- Direct mapping to Microsemi 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 1136 for more information.

Synthesis Constraints and Attributes for Microsemi

The synthesis tools let you specify timing constraints, general HDL attributes, and Microsemi-specific attributes to improve your design. You can manage the attributes and constraints in the SCOPE interface. Microsemi has vendor-specific I/O standard constraints it supports for synthesis. For a list of supported I/O standards, see Microsemi I/O Standards, on page 1121.

Microsemi I/O Standards

The following table lists the supported I/O standards for the ProASIC3E, ProASIC and ProASIC^{PLUS} families and Axcelerator family. Some I/O standards have associated modifiers you can set, such as slew, termination, drive, power, and Schmitt, which allow the software to infer the correct buffer types.

ProASIC3E	ProASIC, ProASIC^{PLUS}	Axcelerator
GTL25	LVCMOS_25	GTL+25
GTL+25	LVCMOS_33	GTL+33
GTL33	LVTTL	HSTL_Class_I
GTL+33	PCI33	HSTL_Class_II
HSTL_Class_I		LVCMOS_15
HSTL_Class_II		LVCMOS_18

ProASIC3E	ProASIC, ProASIC^{PLUS}	Axcelerator
LVCMOS_12		LVCMOS_5
LVCMOS_15		LVDS
LVCMOS_18		LVPECL
LVCMOS_33		LVTTL
LVCMOS_5		PCI
LVDS		PCIX
LVPECL		SSTL_2_Class_I
LVTTL		SSTL_2_Class_II
PCI		SSTL_3_Class_I
PCIX		SSTL_3_Class_II
SSTL_2_Class_I		
SSTL_2_Class_II		
SSTL_3_Class_I		
SSTL_3_Class_II		

See Also:

- Industry I/O Standards, on page 387 for a list of industry I/O standards.
- Microsemi Attribute and Directive Summary, on page 1149 for a list of Microsemi attributes and directives.

Microsemi Components

The following topics describe how the synthesis tools handle various Microsemi components, and show you how to work with or manipulate them during synthesis to get the results you need:

- Macros and Black Boxes in Microsemi Designs, on page 1123
- DSP Block Inference, on page 1124
- Microsemi RAM Implementations, on page 1127
- Instantiating RAMs with SYNCORE, on page 1132

Macros and Black Boxes in Microsemi Designs

You can instantiate Smartgen[1] macros or other Microsemi macros like gates, counters, flip-flops, or I/Os by using the supplied Microsemi macro libraries to pre-define the Microsemi macro black boxes. For certain technologies, the following macros are also supported:

- SIMBUF Macro
- MATH18X18 Block

For general information on instantiating black boxes, see Instantiating Black Boxes in VHDL, on page 744, and Instantiating Black Boxes in Verilog, on page 542. For specific procedures about instantiating macros and black boxes and using Microsemi black boxes, see the following sections in the *User Guide*:

- Defining Black Boxes for Synthesis, on page 168
- Using Predefined Microsemi Black Boxes, on page 346
- Using Smartgen Macros, on page 347

[1] Smartgen macros now replace the ACTgen macros. ACTgen macros were available in the previous Designer 6.x place-and-route tool.

SIMBUF Macro

The synthesis software supports instantiation of the SIMBUF macro. The SIMBUF macro provides the flexibility to probe signals without using physical locations, such as possible from the Identify tool. The Resource Summary will report the number of SIMBUF instantiations in the IO Tile section of the log file.

SIMBUF macros are supported for ProASIC3E devices.

MATH18X18 Block

The synthesis software supports instantiation of the MATH18X18 block. The MATH18X18 block is useful for mapping arithmetic functions for RTAXSDSP devices.

DSP Block Inference

This feature allows the synthesis tools to infer DSP or MATH18x18 blocks for Axcelerator RTAX2000D and RTAX4000D devices only. The following structures are supported:

- Multipliers
- Mult-adds — Multiplier followed by an Adder
- Mult-subs — Multiplier followed by a Subtractor
- Wide multiplier inference

  A multiplier is treated as wide, if any of its inputs is larger than 18 bits signed or 17 bits unsigned. The multiplier can be configured with only one input that is wide, or else both inputs are wide. Depending on the number of wide inputs for signed or unsigned multipliers, the synthesis software uses the cascade feature to determine how many math blocks to use and the number of Shift functions it needs.
- MATH block inferencing across hierarchy

  This enhancement to MATH block inferencing allows packing input registers, output registers, and any adders or subtractors into different hierarchies. This helps to improve QoR by packing logic more efficiently into MATH blocks.

By default, the synthesis software maps the multiplier to DSP blocks if all inputs to the multiplier are more than 2-bits wide; otherwise, the multiplier is mapped to logic. You can override this default behavior using the syn_multstyle attribute. See syn_multstyle Attribute, on page 961 for details.

The following conditions also apply:

- Signed and unsigned multiplier inferencing is supported.
- Registers at inputs and outputs of multiplier/multiplier-adder/multiplier-subtractor are packed into DSP blocks.
- Synthesis software fractures multipliers larger than 18X18 (signed) and 17X17 (unsigned) into smaller multipliers and packs them into DSP blocks.
- When multadd/multsub are fractured, the final adder/subtractor are packed into logic.

DSP Cascade Chain Inference

The MATH18x18 block cascade feature supports the implementation of multi-input Mult-Add/Sub for devices with MATH blocks. The software packs logic into MATH blocks efficiently using hard-wired cascade paths, which improves the QoR for the design.

Prerequisites include the following requirements:

- The input size for multipliers is *not* greater than 18x18 bits (signed) and 17x17 bits (unsigned).
- Signed multipliers have the proper sign-extension.
- All multiplier output bits feed the adder.
- Multiplier inputs and outputs can be registered or not.

Multiplier-Accumulators (MACs) Inference

The Multiplier-Accumulator structures use internal paths for adder feedback loops inside the MATH18x18 block instead of connecting it externally.

Prerequisites include the following requirements:

- The input size for multipliers is *not* greater than 18x18 bits (signed) and 17x17 bits (unsigned).

- Signed multipliers have the proper sign-extension.
- All multiplier output bits feed the adder.
- The output of the adder must be registered.
- The registered output of the adder feeds back to the adder for accumulation.
- Since the Microsemi MATH block contains one multiplier, only Multiplier-Accumulator structures with one multiplier can be packed inside the MATH block.

The other Multiplier-Accumulator structure supported is with Synchronous Loadable Register.

Prerequisites include the following requirements:

- All the requirements mentioned above apply for this structure as well.
- For the Loading Multiplier-Accumulator structure, new Load data should be passed to input C.
- The LoadEn signal should be registered.

DSP Limitations

Currently, DSP inferencing does not support the following functions:

- SIMD mode
- Overflow extraction
- Arithmetic right shift for operand C
- Dynamic Mult-AddSub

Note: For more information about Microsemi DSP math blocks along with a comprehensive set of examples, see the *Inferring Microsemi RTAX-DSP MATH Blocks* application note on SolvNet.

Microsemi RAM Implementations

Refer to the following topics for Microsemi RAM implementations:

- ProASIC (500K) and ProASICPLUS (PA)
- ProASIC3E
- ProASIC (500K) and ProASICPLUS (PA)
- Axcelerator

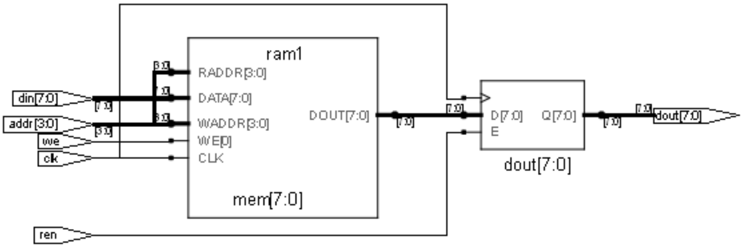
RAM Read Enable Extraction

RAM Read Enable extraction currently supports RAMs for output registers, with an enable. This feature is available for ProASIC3E and Axcelerator devices only.

The following example contains a RAM with read enable.

```
`timescale 100 ps/100 ps
/* Synchronous write and read RAM */

module test (dout, addr, din, we, clk, ren);

parameter data_width = 8;
parameter address_width = 4;
parameter ram_size = 16;

output [data_width-1:0] dout;
input [data_width-1:0] din;
input [address_width-1:0] addr;
input we, clk, ren;

reg [data_width-1:0] mem [ram_size-1:0];
reg [data_width-1:0] dout;

always @(posedge clk) begin
   if(we)
      mem[addr] <= din;
```

```
            if (ren)
            dout = mem[addr];
    end

    endmodule
```



ProASIC3E

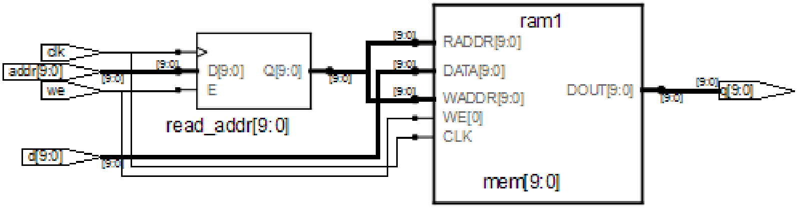
The synthesis software extracts single-port and dual-port versions of the following RAM configurations:

RAM4K9	Synchronous write, synchronous read, transparent output
RAM512X18	Synchronous write, synchronous read, registered output

The architecture of the inferred RAM for the ProASIC3E can be registers, block_ram, rw_check, or no_rw_check. You set these values in the SCOPE interface using the syn_ramstyle attribute.

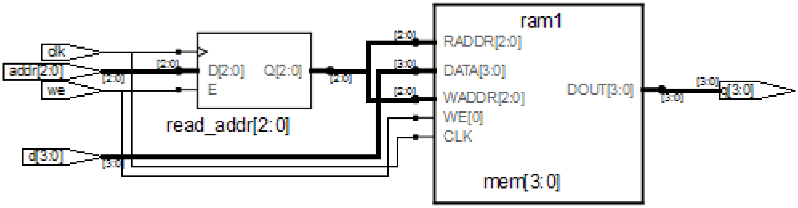
The following is an example of the RAM4K9 configuration:

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

entity ramtest is
port (q : out std_logic_vector(9 downto 0);
   d : in std_logic_vector(9 downto 0);
   addr : in std_logic_vector(9 downto 0);
   we : in std_logic;
   clk : in std_logic);
end ramtest;

architecture rtl of ramtest is
type mem_type is array (1023 downto 0) of std_logic_vector (9
downto 0);

signal mem : mem_type;
signal read_addr : std_logic_vector(9 downto 0);

begin

q <= mem(conv_integer(read_addr));

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

end rtl;
```



The following is an example of the RAM512X18 configuration:

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

entity ramtest is
port (q : out std_logic_vector(3 downto 0);
   d : in std_logic_vector(3 downto 0);
   addr : in std_logic_vector(2 downto 0);
   we : in std_logic;
   clk : in std_logic);
end ramtest;

architecture rtl of ramtest is
type mem_type is array (7 downto 0) of std_logic_vector (3 downto
0);

signal mem : mem_type;
signal read_addr : std_logic_vector(2 downto 0);

begin

q <= mem(conv_integer(read_addr));

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

end rtl;
```


clk
addr[2:0]
we
d[3:0]
[2:0]
[2:0]
D[2:0]
E
Q[2:0]
[2:0]
read_addr[2:0]
[3:0]
ram1
[2:0]
RADDR[2:0]
[3:0]
DATA[3:0]
[2:0]
WADDR[2:0]
WE[0]
CLK
DOUT[3:0]
mem[3:0]
[3:0]
[3:0]
q[3:0]

ProASIC (500K) and ProASIC^{PLUS} (PA)

The synthesis software extracts single-port and dual-port versions of the following RAM configurations:

SA	Synchronous write, asynchronous read
SST	Synchronous write, synchronous read, transparent output
SSR	Synchronous write, synchronous read, registered output

Axcelerator

The synthesis software extracts single-port and dual-port versions of the following RAM configurations. The memory blocks operate in synchronous mode for both read and write operations. There are two read modes and one write mode:

Read Nonpipelined	Synchronous – one clock edge
Read Pipelined	Synchronous – two clock edges
Write	Synchronous – one clock edge

The architecture of the inferred RAM for the Axcelerator device can be registers, block_ram rw_check, or no_rw_check. You set these values in the SCOPE interface using the syn_ramstyle attribute.

Instantiating RAMs with SYNCORE

The SYNCORE Memory Compiler is available under the IP Wizard to help you generate HDL code for your specific RAM implementation requirements. For information on using the SYNCORE Memory Compiler, see Specifying RAMs with SYNCore, on page 376 in the *User Guide*.

Output Files and Forward-annotation for Microsemi

After synthesis, the software generates a log file and output files for Microsemi. The following describe some of the reports with Microsemi-specific information, or files that forward-annotate information for the Microsemi P&R tools.

- Forward-annotating Constraints for Placement and Routing, on page 1133
- Synthesis Reports, on page 1134

Forward-annotating Constraints for Placement and Routing

For Microsemi Axcelerator, ProASIC (500K), ProASIC^{PLUS} (PA), and ProASIC3E technology families, the synthesis tool forward-annotates timing constraints to placement and routing through an Microsemi constraint (*filename*_sdc.sdc) file. These timing constraints include clock period, max delay, multiple-cycle paths, input and output delay, and false paths. During synthesis, the Microsemi constraint file is generated using synthesis tool attributes and constraints.

By default, Microsemi constraint files are generated. You can disable this feature in the Project view. To do this, bring up the Implementation Options dialog box (Project -> Implementation Options), then, on the Implementation Results panel, disable Write Vendor Constraint File.

Forward-annotated Constraints

The constraint file generated for Microsemi's place-and-route tools has an _sdc.sdc file extension. Constraints files that properly specify either Synplify-style timing constraints or Synopsys SDC timing constraints can be forward annotated to support the Microsemi P&R tool.

Synthesis	Microsemi
define_clock	create_clock The create_clock constraint is allowed for all NGT families. No wildcards are accepted. The pin or port must exist in the design. The -name argument is not supported as that would define a virtual clock. However, for backward compatibility, a -name argument does not generate an error or warning when encountered in a .fdc file.
define_path_delay	set_max_delay The set_max_delay constraint is allowed for all NGT families, however, it will not drive ProASIC (500K) TDPR because of conversion issues to GCF (the whole path is needed in GCF, not the start and end point). Wildcards are accepted.
define_multicycle_path	set_multicycle_path You must specify at least one of the -from or -to arguments, however, it is best to specify both. Wildcards are accepted. Multicycle constraints with -from and/or -to arguments only are supported for Microsemi Axcelerator, ProASIC, ProASIC^{PLUS}, and ProASIC3E technologies. Multicycle constraints with a -through argument are not supported for any NGT family.
define_false_path	set_false_path Only false path constraints with a -through argument are supported for NGT families. However, it might not drive ProASIC (500K) TDPR because of conversion issues to GCF (some non supported objects). False path constraints with either -from and/or -to arguments are not supported for any NGT family. Wildcards are accepted.

Synthesis Reports

The synthesis tool generates a resource usage report, a timing report, and a net buffering report for the Microsemi designs that you synthesize.To view the synthesis reports, click View Log.

Optimizations for Microsemi Designs

The synthesis tools offer various optimizations for Microsemi designs. The following describe the optimizations in more detail:

The syn_maxfan Attribute in Microsemi Designs

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 Synopsys FPGA 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.

Promote Global Buffer Threshold

The Promote Global Buffer Threshold option is for the ProASIC3E technology families only. This option is for both ports and nets.

The Tcl command equivalent is set_option -globalthreshold *value,* where the value refers to the minimum number of fanout loads. The default value is 1.

Only signals with fanout loads larger than the defined value are promoted to global signals. The synthesis tool assigns the available global buffers to drive these signals using the following priority:

1. Clock
2. Asynchronous set/reset signal
3. Enable, data

I/O Insertion

The Synopsys FPGA 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 Microsemi 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 for the ProASIC (500K), ProASIC^{PLUS} (PA) and ProASIC3E technology families. 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. Microsemi 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.

Retiming

Retiming is the process of automatically moving registers (register balancing) across combinational gates to improve timing, while ensuring identical logic behavior. Currently, retiming is available for the Axcelerator, ProASIC (500K), ProASIC^{PLUS} (PA), and ProASIC3E technology families.

You enable/disable global retiming with the Retiming device mapping option (Project view or Device panel). You can use the syn_allow_retiming attribute to enable or disable retiming for individual flip-flops. See *syn_allow_retiming Attribute, on page 919* and the *Synplify User Guide* for more information.

Update Compile Point Timing Data Option

In ProASIC (500K), ProASIC^{PLUS} (PA) and ProASIC3E designs, the Synopsys FPGA compile-point synthesis flow lets you break down a design into smaller synthesis units, called *compile points*, making incremental synthesis possible. See Synthesizing Compile Points, on page 433 in the *User Guide*.

The Update Compile Point Timing Data option controls whether or not changes to a locked compile point force remapping of its parents, taking into account the new timing model of the child.

Note: To simplify this description, the term *child* is used here to refer to a compile point that is contained inside another; the term *parent* is used to refer to the compile point that contains the child. These terms are thus not used here in their strict sense of direct, immediate containment: If a compile point A is nested in B, which is nested in C, then A and B are both considered children of C, and C is a parent of both A and B. The top level is considered the parent of all compile points.

Disabled

When the Update Compile Point Timing Data option is *disabled* (the default), only (locked) compile points that have changed are remapped, and their remapping does *not* take into account changes in the timing models of any of their children. The old (pre-change) timing model of a child is used, instead, to map and optimize its parents.

An exceptional case occurs when the option is disabled and the *interface* of a locked compile point is changed. Such a change requires that the immediate parent of the compile point be changed accordingly, so both are remapped. In this exceptional case, however, the *updated* timing model (not the old model) of the child is used when remapping this parent.

Enabled

When the Update Compile Point Timing Data option is *enabled*, locked compile-point changes are taken into account by updating the timing model of the compile point and resynthesizing all of its parents (at all levels), using the updated model. This includes any compile point changes that took place prior to enabling this option, and which have not yet been taken into account (because the option was disabled).

The timing model of a compile point is updated when either of the following is true:

- The compile point is remapped, and the Update Compile Point Timing Data option is enabled.
- The interface of the compile point is changed.

Operating Condition Device Option

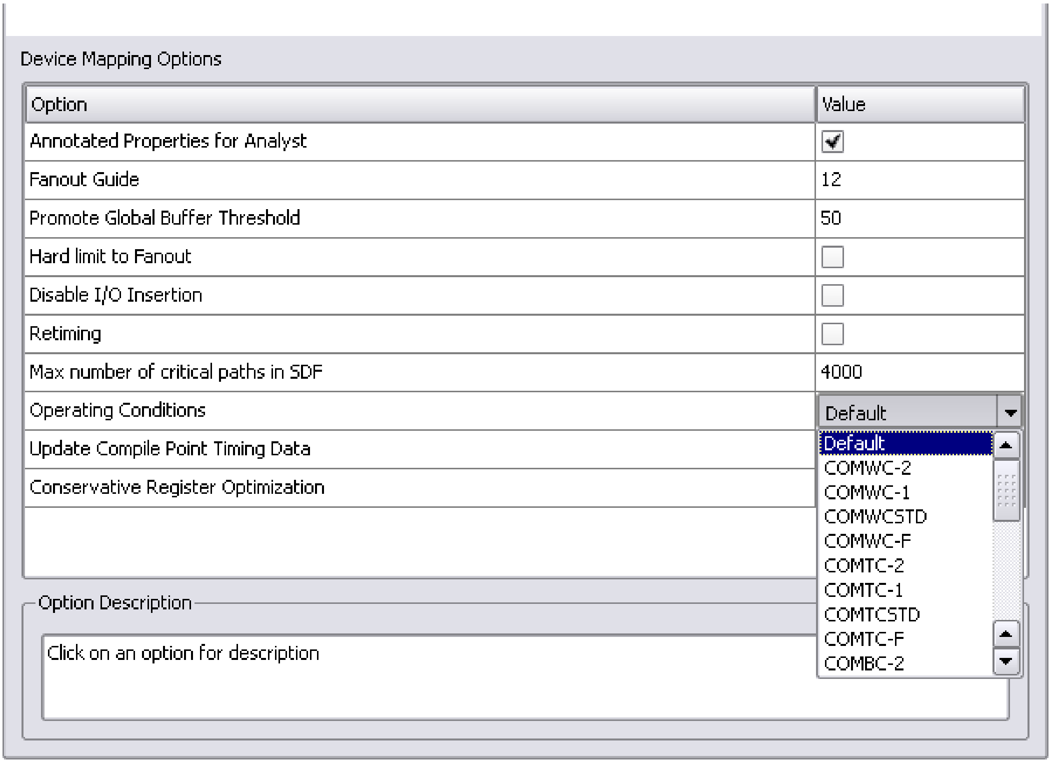
You can specify an operating condition for certain Microsemi technologies:

- ProASIC (500K)
- ProASIC^{PLUS} (PA)ProASIC3E
- Axcelerator

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.



Device Mapping Options

Option	Value
Annotated Properties for Analyst	☑
Fanout Guide	12
Promote Global Buffer Threshold	50
Hard limit to Fanout	□
Disable I/O Insertion	□
Retiming	□
Max number of critical paths in SDF	4000
Operating Conditions	Default
Update Compile Point Timing Data	
Conservative Register Optimization	

Option Description

Click on an option for description

To set an operating condition in a project or Tcl file, use the command:

```
set_option -opcond value
```

where *value* can be specified like the following typical operating conditions:

Default	Typical timing
MIL-WC	Worst-case Military timing
MIL-TC	Typical-case Military timing
MIL-BC	Best-case Military timing
Automotive-WC	Worst-case Automotive timing

For Example

The Microsemi operating condition can contain any of the following specifications:

- MIL—military
- COM—commercial
- IND—Industrial
- TGrade1
- TGrade2

as well as, include one of the following designations:

- WC—worst case
- BC—best case
- TC—typical case

For specific operating condition values for your required technology, see the Device tab on the Implementation Options dialog box.

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 Microsemi's documentation or software for information on valid combinations and more information on the meaning of each operating condition.

Radiation-tolerant Applications

You can specify the radiation-resistant design technique to use on an object for a design with the syn_radhardlevel attribute. 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 Microsemi macro files supplied with the software.

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

For details, see:

- Working with Radhard Designs, on page 347
- syn_radhardlevel Attribute, on page 990

Integration with Microsemi Tools and Flows

The following describe how the synthesis tools support various tools and flows for Microsemi designs:

- Compile Point Synthesis, on page 1143
- Microsemi Place-and-Route Tools, on page 1144

Compile Point Synthesis

Compile-point synthesis is available for the Synplify Pro tool and only for the Microsemi Axcelerator, ProASIC (500K), ProASIC^{PLUS} (PA), and ProASIC3E technology families. The compile-point synthesis flow lets you achieve incremental design and synthesis without having to write and maintain sets of complex, error-prone scripts to direct synthesis and keep track of design dependencies. See Synthesizing Compile Points, on page 433 for a description, and *Working with Compile Points, on page 413* in the *User Guide* for a step-by-step explanation of the compile-point synthesis flow.

In device technologies that can take advantage of compile points, you break down your design into smaller synthesis units or *compile points*, in order to make incremental synthesis possible. A compile point is a module that is treated as a block for incremental mapping: When your design is resynthesized, compile points that have already been synthesized are not resynthesized, unless you have changed:

- the HDL source code in such a way that the design logic is changed,
- the constraints applied to the compile points, or
- the device mapping options used in the design.

(For details on the conditions that necessitate resynthesis of a compile point, see Compile Point Basics, on page 414, and Update Compile Point Timing Data Option, on page 1137.)

Microsemi Place-and-Route Tools

You can run place and route automatically after synthesis. For details on how to set options, see Running Place-and-Route after Synthesis, on page 356 in the *User Guide*.

For details about the place-and-route tools, refer to the Microsemi documentation.

Microsemi Device Mapping Options

You select device mapping options for Microsemi technologies, select Project -> Implementation Options->Device and set the options.

Option	**For details, see...**
Conservative Register Optimization	See the Microsemi Tcl set_option Command Options, on page 1146 for more information about the preserve_registers option.
Disable I/O Insertion	I/O Insertion, on page 1136.
Fanout Guide	Setting Fanout Limits, on page 211 of the *User Guide* and The syn_maxfan Attribute in Microsemi Designs, on page 1135.
Hard Limit to Fanout	Setting Fanout Limits, on page 211 of the *User Guide* and The syn_maxfan Attribute in Microsemi Designs, on page 1135.
Max number of critical paths in SDF (certain technologies)	Number of Critical Paths, on page 1137.
Operating Conditions (certain technologies)	Operating Condition Device Option, on page 1139
Promote Global Buffer Threshold	Controlling Buffering and Replication, on page 213 of the *User Guide* and Promote Global Buffer Threshold, on page 1136.

Option	For details, see...
Read Write Check on RAM	Lets the synthesis tool insert bypass logic around the RAM to prevent a simulation mismatch between the RTL and post-synthesis simulations. The synthesis software globally inserts bypass logic around the RAM that read and write to the same address simultaneously. Disable this option, when you cannot simultaneously read and write to the same RAM location and want to minimize overhead logic. For details about using this option in conjunction with the syn_ramstyle attribute, see syn_ramstyle Attribute, on page 995.
Retiming	Retiming, on page 198 of the *User Guide and* Retiming, on page 1137*.*
Resolve Mixed Drivers	When a net is driven by a VCC or GND and active drivers, enable this option to connect the net to the VCC or GND driver.
Update Compile point Timing Data	Update Compile Point Timing Data Option, on page 1137

Microsemi Tcl set_option Command Options

You can use the set_option Tcl command to specify the same device mapping options as are available through the Implementation Options dialog box displayed in the Project view with Project -> Implementation Options (see Implementation Options Command, on page 158).

This section describes the Microsemi-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 Microsemi architectures. For a complete list of options for this command, refer to set_option, on page 1065. You cannot specify a package (-package option) for some Microsemi technologies in the synthesis tool environment. You must use the Microsemi back-end tool for this.

.

Option	Description
-technology *keyword*	Sets the target technology for the implementation. Keyword must be one of the following Microsemi architecture names: 3200DX, 40MX, 42MX, 500K, 54SX, 54SXA, ACT1, ACT2, ACT3, AXCELERATOR, EX, PA, and ProASIC3E. For the 1200XL architecture, use ACT2.
-part *part_name*	Specifies a part for the implementation. Refer to the Implementation Options dialog box for available choices.
-package *package_name*	Specifies the package. Refer to Project-> Implementation Options->Device for available choices.
-speed_grade *value*	Sets the speed grade for the implementation. Refer to the Implementation Options dialog box for available choices. This option is not supported by the ProASIC (500K) technology.
-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 1136.

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 The syn_maxfan Attribute in Microsemi Designs, on page 1135.
-globalthreshold *value*	Sets the minimum fanout load value. This option applies only to the ProASIC3E technologies. For more information, see Promote Global Buffer Threshold, on page 1136.
-maxfan_hard 1	Specifies that the specified -fanout_guide value is a hard fanout limit that the synthesis tool must not exceed. To set a guideline limit, see the -fanout_guide option. For more information about fanout limits, see The syn_maxfan Attribute in Microsemi Designs, on page 1135.
-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), ProASIC^{PLUS} (PA), and ProASIC3E technologies. Values are Default, MIL-WC, IND-WC, COM-WC, and Automotive-WC. See Operating Condition Device Option, on page 1139 for more information.
-preserve_registers 1 | 0	When enabled, the software uses less restrictive register optimizations during synthesis if area is not as great a concern for your device. The default for this option is disabled (0).
-resolve_multiple_driver 1 | 0	When a net is driven by a VCC or GND and active drivers, enable this option to connect the net to the VCC or GND driver.
-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), ProASIC^{PLUS} (PA), and ProASIC3E technologies. For information about setting critical paths, see Number of Critical Paths, on page 1137.

Option	Description
-retiming 1 | 0	When enabled (1), registers may be moved into combinational logic to improve performance. The default value is 0 (disabled). This option is only available for the Axcelerator, ProASIC (500K), ProASIC^{PLUS} (PA), and ProASIC3E technologies. For additional information about retiming, see Retiming, on page 1137
-RWCheckOnRam 1 | 0	If read or write conflicts exist for the RAM, enable this option to insert bypass logic around the RAM to prevent simulation mismatch. Disabling this option does not generate bypass logic. For more information about using this option in conjunction with the syn_ramstyle attribute, see syn_ramstyle Attribute, on page 995.
-update_models_cp 1 | 0	Determines whether (1) or not (0) changes to a locked compile point force remapping of its parents, taking into account the new timing model of the child. This option is only available for the ProASIC (500K), ProASIC^{PLUS} (PA), and ProASIC3E technologies. See Update Compile Point Timing Data Option, on page 1137, for details.

Microsemi Attribute and Directive Summary

The following table summarizes the synthesis and Microsemi-specific attributes and directives available with the Microsemi technology. Complete descriptions and examples are in Chapter 11, *Synthesis Attributes and Directives*.

Attribute/Directive	Description
alsloc Attribute	Forward-annotates the relative placements of macros and IP blocks to Microsemi Designer. This attribute does not apply to ProASIC (500K) and ProASIC^{PLUS} (PA) designs.
alspin Attribute	Assigns scalar or bus ports to Microsemi I/O pin numbers. This attribute does not apply to ProASIC (500K) and ProASIC^{PLUS} (PA) designs.
alspreserve Attribute	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 Directive (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 Directive (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 Directive (D)	Specifies that a Verilog case statement has covered all possible cases.
loop_limit Directive (D)	Specifies a loop iteration limit for for loops.
parallel_case Directive (D)	Specifies a parallel multiplexed structure in a Verilog case statement, rather than a priority-encoded structure.
syn_allow_retiming Attribute	Specifies whether registers can be moved during retiming.
syn_black_box Directive (D)	Defines a black box for synthesis.

(D) indicates directives; all others are attributes.

Attribute/Directive	Description
syn_direct_enable Attribute/Directive	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 Attribute	Specifies the encoding style for state machines.
syn_enum_encoding Directive (D)	Specifies the encoding style for enumerated types (VHDL only).
syn_global_buffers Attribute	Sets the number of global buffers to use in a ProASIC3E.
syn_hier Attribute	Controls the handling of hierarchy boundaries of a module or component during optimization and mapping.
syn_isclock Directive (D)	Specifies that a black-box input port is a clock, even if the name does not indicate it is one.
syn_keep Directive (D)	Prevents the internal signal from being removed during synthesis and optimization.
syn_maxfan Attribute	Overrides the default fanout guide for an individual input port, net, or register output.
syn_multstyle Attribute	Determines how multipliers are implemented for Microsemi devices.
syn_netlist_hierarchy Attribute	Determines whether the EDIF output netlist is flat or hierarchical.
syn_noarrayports Attribute	Prevents the ports in the EDIF output netlist from being grouped into arrays, and leaves them as individual signals.
syn_noclockbuf Attribute	Turns off the automatic insertion of clock buffers.
syn_noprune Directive (D)	Controls the automatic removal of instances that have outputs that are not driven.
syn_pad_type Attribute	Specifies an I/O buffer standard.
syn_preserve Directive (D)	Prevents sequential optimizations across a flip-flop boundary during optimization, and preserves the signal.

(D) indicates directives; all others are attributes.

Attribute/Directive	Description
syn_preserve_sr_priority Attribute	Forces set/reset flip-flops to honor the coded priority for the set or reset. ACT 1 and 40MX architectures only.
syn_probe Attribute	Adds probe points for testing and debugging.
syn_radhardlevel Attribute	Specifies the radiation-resistant design technique to apply to a module, architecture, or register.
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.
syn_reference_clock Attribute	Specifies a clock frequency other than that implied by the signal on the clock pin of the register.
syn_replicate Attribute	Controls replication.
syn_resources Attribute	Specifies resources used in black boxes.
syn_sharing Directive (D)	Specifies resource sharing of operators.
syn_state_machine Directive (D)	Determines if the FSM Compiler extracts a structure as a state machine.
syn_tco<n> Directive (D)	Defines timing clock to output delay through the black box. The *n* indicates a value between 1 and 10.
syn_tpd<n> Directive (D)	Specifies timing propagation for combinational delay through the black box. The *n* indicates a value between 1 and 10.
syn_tristate Directive (D)	Specifies that a black-box pin is a tristate pin.
syn_tsu<n> Directive (D)	Specifies the timing setup delay for input pins, relative to the clock. The *n* indicates a value between 1 and 10.
syn_useenables Attribute	Generates clock enable pins for registers.
translate_off/translate_on Directive (D)	Specifies sections of code to exclude from synthesis, such as simulation-specific code.

(D) indicates directives; all others are attributes.

APPENDIX C

Example Code

This appendix contains the code samples that are referenced by the corresponding chapter.

Example - Constant function

```
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
```

Back

Example - Constant math function counter

```
module top
#( parameter COUNT = 256 )
//Input
( input clk,
  input rst,
//Output
//Function used to compute width based on COUNT value of counter:
  output [$clog2(COUNT)-1:0] dout );
reg[$clog2(COUNT)-1:0]count;
always@(posedge clk)
begin
  if(rst)
     count = ‘b0;
  else
     count = count + 1’b1;
  end
assign dout = count;
endmodule
```

Back

Example - Constant math function RAM

```
module top
#
( parameter DEPTH = 256,
  parameter WIDTH = 16 )
(
//Input
  input clk,
  input we,
  input rst,
//Function used to compute width of address based on depth of RAM:
  input [$clog2(DEPTH)-1:0] addr,
  input [WIDTH-1:0] din,
//Output
  output reg[WIDTH-1:0] dout );
reg[WIDTH-1:0] mem[(DEPTH-1):0];
always @ (posedge clk)
  if (rst == 1)
     dout = 0;
  else
     dout = mem[addr];
always @(posedge clk)
  if (we) mem[addr] = din;
endmodule
```

Back

Example – RAM Inferencing

```
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
```

Back

Example - RAM with registered output

```
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] q;
reg [7:0] mem [127:0];
always @(posedge clk) begin
   q <= mem [a];
   if(we)
      mem[a] <= d;
   end
endmodule
```

Back

Example - RAM registered read address

```
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
```

Back

Example - Asynch FSM with continuous assignment

```
module async1 (out, g, d);
output out;
input g, d;
assign out = g & d | !g & out | d &out;
endmodule
```

Back

Example - Asynch FSM with 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
```

Back

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
```

Back

Example – Downward Read Cross-Module Reference

```
module top
( input a,
  input b,
  output c,
  output d
);
sub inst1 (.a(a), .b(b), .c(c) );
assign d = inst1.a;
```

```
endmodule
module sub
( input a,
  input b,
  output c
);
assign c = a  & b;
endmodule
```

Back

Example – Downward Write Cross-Module Reference

```
module top
( input a,
  input b,
  output c,
  output d
);
sub inst1 (.a(a), .b(b), .c(c), .d(d) );
assign top.inst1.d = a;
endmodule
module sub
( input a,
  input b,
  output c,
  output d
);

assign c = a & b;
endmodule
```

Back

Example – Upward Read Cross-Module Reference

```
module top
( input a,
  input b,
  output c,
  output d
);
```

```
sub inst1 (.a(a), .b(b), .c(c), .d(d) );
endmodule
module sub
( input a,
  input b,
  output c,
  output d
);
assign c = a & b;
assign d = top.a;
endmodule
```

Back

APPENDIX C

Example Code

This appendix contains the code samples that are referenced by the corresponding chapter.

Example - Initializing Unpacked Array to Default Value

```
parameter WIDTH = 2;
typedef reg [WIDTH-1:0] [WIDTH-1:0] MyReg;
module top (
   input logic Clk,
   input logic Rst,
   input MyReg DinMyReg,
   output MyReg DoutMyReg );
MyReg RegMyReg;

always@(posedge Clk, posedge Rst) begin
   if(Rst) begin
      RegMyReg <= '{default:0};
      DoutMyReg <= '{default:0};
   end
   else begin
      RegMyReg <= DinMyReg;
      DoutMyReg <= RegMyReg;
   end
end
endmodule
```

Back

Example - Initializing Unpacked Array Under Reset Condition

```
parameter WIDTH = 2;
typedef reg [WIDTH-1:0] [WIDTH-1:0] MyReg;
module top (
   input logic Clk,
   input logic Rst,
   input MyReg DinMyReg,
   output MyReg DoutMyReg );
MyReg RegMyReg;

always@(posedge Clk, posedge Rst) begin
   if(Rst) begin
      RegMyReg <= '{2'd0, 2'd0};
      DoutMyReg <= '{2'd0, 2'd0};
   end
   else begin
      RegMyReg <= DinMyReg;
      DoutMyReg <= RegMyReg;
   end
end
endmodule
```

Back

Example - Aggregate Assignment in Compilation Unit

```
// Start of compilation unit
parameter WIDTH = 2;
   typedef struct packed {
      int r;
      longint g;
      byte b;
      int rr;
      longint gg;
      byte bb;
   } MyStruct [WIDTH-1:0];
const MyStruct VarMyStruct = '{int:1,longint:10,byte:8'h0B} ;
const MyStruct ConstMyStruct =
   '{int:1,longint:$bits(VarMyStruct[0].r),byte:8'hAB} ;

module top (
   input logic Clk,
   input logic Rst,
   input MyStruct DinMyStruct,
   output MyStruct DoutMyStruct );
MyStruct StructMyStruct;

always@(posedge Clk, posedge Rst)
begin
   if(Rst) begin
      StructMyStruct <= VarMyStruct;
      DoutMyStruct <= ConstMyStruct;
   end
   else begin
      StructMyStruct <= DinMyStruct;
      DoutMyStruct <= StructMyStruct;
   end
end
endmodule
```

Back

Example - Aggregate Assignment in Package

```
package MyPkg;
parameter WIDTH = 2;
   typedef struct packed {
      int r;
      longint g;
      byte b;
      int rr;
      longint gg;
      byte bb;
   } MyStruct [WIDTH-1:0];
const MyStruct VarMyStruct = ‘{int:1,longint:10,byte:8’h0B} ;
const MyStruct ConstMyStruct =
   ‘{int:1,longint:$bits(VarMyStruct[0].r),byte:8’hAB} ;
endpackage : MyPkg
import MyPkg::*;

module top (
   input logic Clk,
   input logic Rst,
   input MyStruct DinMyStruct,
   output MyStruct DoutMyStruct );
MyStruct StructMyStruct;

always@(posedge Clk, posedge Rst) begin
   if(Rst) begin
      StructMyStruct <= VarMyStruct;
      DoutMyStruct <= ConstMyStruct;
   end
   else begin
      StructMyStruct <= DinMyStruct;
      DoutMyStruct <= StructMyStruct;
   end
end
endmodule
```

Back

Example - Initializing Specific Data Type

```
   parameter WIDTH = 2;
   typedef struct packed {
      byte r;
      byte g;
      byte b; }
   MyStruct [WIDTH-1:0];
   module top (
      input logic Clk,
      input logic Rst,
      input MyStruct DinMyStruct,
      output MyStruct DoutMyStruct );
   MyStruct StructMyStruct;

   always@(posedge Clk, posedge Rst) begin
      if(Rst) begin
         StructMyStruct <= ‘{byte:0,byte:0};
         DoutMyStruct <= ‘{byte:0,byte:0};
      end
      else begin
         StructMyStruct <= DinMyStruct;
         DoutMyStruct <= StructMyStruct;
      end
```

```
end
endmodule
```

Back

Example - Including Block Name with end Keyword

```
module src (in1,in2,out1,out2);
input in1,in2;
output reg out1,out2;
reg a,b;
always@(in1,in2)
   begin : foo_in
      a = in1 & in2;
      b = in2 | in1;
   end : foo_in
always@(a,b)
   begin : foo_value
      out1 = a;
      out2 = b;
   end : foo_value
endmodule
```

Back

Example - always_comb Block

```
module test01 (a, b, out1);
input a,b;
output out1;
reg out1;
always_comb
begin
   out1 = a & b;
end
endmodule
```

Back

Example - always_ff Block

```
module Test01 (a,b,clk,out1);
input a,b,clk;
output out1;
reg out1;
always_ff@(posedge clk)
   out1 <= a & b;
endmodule
```

Back

Example - always_latch Block

```
module Test01 (a,b,clk,out1);
input a,b,clk;
output out1;
reg out1;

always_latch
begin
   if(clk)
   begin
      out1 <= a & b;
   end
end
endmodule
```

Back

Example - Local Variable in Unnamed Block

```
module test(in1,out1);
input [2:0] in1;
output [2:0] out1;
integer i;
wire [2:0] count;
reg [2:0] temp;
assign count = in1;
```

```
always @ (count)
begin   // unnamed block
   integer i; //local variable
   for (i=0; i < 3; i = i+1)
      begin : foo
      temp = count + 1;
   end
end
assign out1 = temp;
endmodule
```

Back

Example - Compilation Unit Access

```
//$unit_4_state begin
logic foo_logic = 1'b1;
//$unit_4_state ends
module test (
input logic data1,
input clk,
output logic out1,out1_local );
//local variables
logic foo_logic = 1'b0;
///////
always @(posedge clk)
begin
   out1 <= data1 ^ $unit::foo_logic; //Referencing
      //the compilation unit value.
   out1_local <= data1 ^ foo_logic; //Referencing the
      //local variable.
end
endmodule
```

Back

Example - Compilation Unit Constant Declaration

```
//$unit begin
const bit foo_bit = "11";
const byte foo_byte = 8'b00101011;
//$unit ends
```

```
module test (clk, data1, data2, out1, out2);
input clk;
input bit data1;
input byte data2;
output bit out1;
output byte out2;

always @(posedge clk)
begin
   out1 <= data1 | foo_bit;
   out2 <= data2 & foo_byte;
end
endmodule
```

Back

Example - Compilation Unit Function Declaration

```
parameter fact = 2;
function automatic [63:0] factorial;
input [31:0] n;
   if (n==1)
      return (1);
   else
      return (n * factorial(n-1));
endfunction

module src (input [1:0] a, input [1:0] b, output logic [2:0] out);
always_comb
begin
   out = a + b + factorial(fact);
end
endmodule
```

Back

Example - Compilation Unit Net Declarations

```
//$unit
wire foo = 1'b1;
//End of $unit
module test (
input data,
output dout );
assign dout = data * foo;
endmodule
```

Back

Example - Compilation Unit Scope Resolution

```
//$unit begins
parameter width = 4;
//$unit ends
module test (data,clk,dout);
parameter width = 8; // local parameter
input logic [width-1:0] data;
input clk;
output logic [width-1:0] dout;

always @(posedge clk)
begin
   dout <= data;
end
endmodule
```

Back

Example - Compilation Unit Task Declaration

```
parameter FACT_OP = 2;
task automatic factorial(input integer operand,
   output [1:0] out1);
integer nFuncCall = 0;
begin
   if (operand == 0)
   begin
      out1 = 1;
   end
   else
   begin
      nFuncCall++;
      factorial((operand-1), out1);
      out1 = out1 * operand;
   end
end
endtask

module src (input [1:0] a, input [1:0] b, output logic [2:0] out);
logic [1:0] out_tmp;
always_comb
factorial(FACT_OP,out_tmp);
assign out = a + b + out_tmp;
endmodule
```

Back

Example - Compilation Unit User-defined Datatype Declaration

```
//$unit begins
typedef struct packed {
int a;
int b;} my_struct;
//End of $unit
module test (p,q,r);
input my_struct p;
input my_struct q;
output int r;
assign r = p.a * q.b;
endmodule
```

Back

Example - Compilation Unit Variable Declaration

```
//$unit begins
logic foo_logic = 1'b1;
//$unit ends
module test (
input logic data1,
input clk,
output logic out1 );

always @(posedge clk)
begin
   out1 <= data1 ^ foo_logic;
```

```
end
endmodule
```

Back

Example - Multi-dimensional Packed Array with Whole Assignment

```
module test (
input [1:0] [1:0] sel,
input [1:0] [1:0] data1,
input [1:0] [1:0] data2,
input [1:0] [1:0] data3,
output reg [1:0] [1:0] out1,
output reg [1:0] [1:0] out2,
output reg [1:0] [1:0] out3 );

always @(sel,data1,data2)
   begin
      out1 = (sel[1]==11)? data1 : {11,11};
      out2 = (sel[1]==2'b11)? data2 : {11,10};
      out3 = data3;
   end
endmodule
```

Back

Example - Multi-dimensional Packed Array with Partial Assignment

```
module test (
input [7:0] datain,
input [1:0][2:0][3:0] datain2,
output [1:0][1:0][1:0] array_out,
output [23:0] array_out2,
output [3:0] array_out2_first_element,
   array_out2_second_element, array_out2_zero_element,
output [1:0] array_out2_first_element_partial_slice );
assign array_out = datain;
assign array_out2 = datain2;
assign array_out2_zero_element = datain2[1][0];
assign array_out2_first_element = datain2[1][1];
assign array_out2_second_element = datain2[1][2];
assign array_out2_first_element_partial_slice =
   datain2[0][0][3-:2];
endmodule
```

Back

Example - Multi-dimensional Packed Array with Arithmetic Ops

```
module test (
input signed [1:0][2:0] a, b,
output signed [1:0] [2:0] c, c_bar, c_mult, c_div, c_per,
output signed [1:0][2:0] d, d_bar, d_mult, d_div, d_per,
output signed e, e_bar, e_mult, e_div, e_per );
assign c = a + b;
assign d = a[1] + b[1];
assign e = a[1][2] + a[1][1] + a[1][0] + b[1][2] + b[1][1]
     + b[1][0];
assign c_bar = a - b;
assign d_bar = a[1] - b[1];
assign e_bar = a[1][2] - a[1][1] - a[1][0] - b[1][2]
     - b[1][1] - b[1][0];
assign c_mult = a * b;
assign d_mult = a[1] * b[1];
assign e_mult = a[1][2] * a[1][1] * a[1][0] * b[1][2] *
     b[1][1] * b[1][0];
assign c_div = a / b;
assign d_div = a[1] / b[1];
assign e_div = a[1][2] / b[1][1];
assign c_per = a % b;
assign d_per = a[1] % b[1];
assign e_per = a[1][2] % b[1][1];
endmodule
```

Back

Example - Packed/Unpacked Array with Partial Assignment

```
module test (
input [1:0] sel [1:0],
input [63:0] data [3:0],
input [63:0] data2 [3:0],
output reg [15:0] out1 [3:0],
output reg [15:0] out2 [3:0]);

always @(sel, data)
   begin
      out1 = (sel[1]==2'b00)? data[3][63-:16] :
         ((sel[1]==2'b01)? data[2][47-:16] :
         ((sel[0]==2'b10)? data[1][(63-32)-:16] :
            data[0][(63-48)-:16] ) );
      out2[3][15-:16] = data2[3][63-:16];
      out2[2][15-:16] = data2[3][47-:16];
      out2[1][15-:16] = data2[3][(63-32)-:16];
      out2[0][15-:8] = data2[3][(63-48)-:8];
   end
endmodule
```

Back

Example - Multi-dimensional Array of Packed Structures Using Anonymous Type

```
module mda_str (
input struct packed {
logic [47:0] dest_addr;
logic [47:0] src_addr;
logic [7:0] type_len;
logic [63:0] data;
logic [2:0] crc;
} [1:0][3:0] str_pkt_in,
input sel1,
input [1:0] sel2,
output struct packed {
logic [47:0] dest_addr;
logic [47:0] src_addr;
logic [7:0] type_len;
logic [63:0] data;
logic [2:0] crc;
} str_pkt_out
);
always_comb
begin
str_pkt_out = str_pkt_in[sel1][sel2];
end
```

```
endmodule
```

Back

Example - Multi-dimensional Array of Packed and Unpacked Structures Using typedef

```
typedef struct {
byte r;
byte g;
byte b;
} [2:0] struct_im_t [0:1];
module mda_str (
input struct_im_t a,
input struct_im_t b,
output struct_im_t c,
input [7:0] alpha,
input [7:0] beta
);
typedef struct {
shortint r;
shortint g;
shortint b;
} [2:0] struct_im_r_t [0:1];
struct_im_r_t temp;
integer i,j;

always_comb
begin
   for(i=0;i<2;i=i+1)
   for(j=0;j<3;j=j+1)
   begin
      temp[i][j].r = a[i][j].r * alpha + b[i][j].r * beta;
      temp[i][j].g = a[i][j].g * alpha + b[i][j].g * beta;
      temp[i][j].b = a[i][j].b * alpha + b[i][j].b * beta;
      c[i][j].r = temp[i][j].r[15:8];
      c[i][j].g = temp[i][j].g[15:8];
      c[i][j].b = temp[i][j].b[15:8];
   end
end
endmodule
```

Back

Example - Multi-dimensional Array of Unpacked Structures Using typedef

```
typedef struct {
byte r;
byte g;
byte b;
} struct_im_t [2:0][1:0];
module mda_str (
input struct_im_t a,
input struct_im_t b,
output struct_im_t c,
input [7:0] alpha,
input [7:0] beta
);
typedef struct {
shortint r;
shortint g;
shortint b;
} struct_im_r_t [2:0][1:0];
struct_im_r_t temp;
integer i,j;

always_comb
begin
   for(i=0;i<3;i=i+1)
   for(j=0;j<2;j=j+1)
   begin
      temp[i][j].r = a[i][j].r * alpha + b[i][j].r * beta;
      temp[i][j].g = a[i][j].g * alpha + b[i][j].g * beta;
      temp[i][j].b = a[i][j].b * alpha + b[i][j].b * beta;
      c[i][j].r = temp[i][j].r[15:8];
      c[i][j].g = temp[i][j].g[15:8];
      c[i][j].b = temp[i][j].b[15:8];
   end
end
endmodule
```

Back

Example - Multi-dimensional Array of Packed Structures Using typedef

```
typedef struct packed {
logic [47:0] dest_addr;
logic [47:0] src_addr;
logic [7:0] type_len;
logic [63:0] data;
logic [3:0] crc;
} [1:0][1:0] str_pkt_mp_t;
typedef struct packed {
logic [47:0] dest_addr;
logic [47:0] src_addr;
logic [7:0] type_len;
logic [63:0] data;
logic [3:0] crc;
} str_pkt_t;
module mda_str (
input str_pkt_mp_t pkt_mp_in,
input sel1,
input sel2,
output str_pkt_t pkt_out
);
always_comb
begin
   pkt_out = pkt_mp_in[sel1][sel2];
end
endmodule
```

Back

Example - Array Querying Function with Data Type as Input

```
//Data type
typedef bit [1:2][4:1]bit_dt[3:2][4:1];
module top
(
//Output
output byte q1_left,
output byte q1_low );
assign q1_left = $left(bit_dt);
assign q1_low = $low(bit_dt);
```

```
endmodule
```

Back

Example - Array Querying Function $dimensions and $unpacked_dimensions Used on a Mixed Array

```
module top
(
//Input
input bit [1:2][4:1]d1[3:2][4:1],
//Output
output byte q1_dimensions,
output byte q1_unpacked_dimensions );
assign q1_dimensions = $dimensions(d1);
assign q1_unpacked_dimensions = $unpacked_dimensions(d1);
endmodule
```

Back

Example - Array Querying Function $left and $right Used on Packed 2D-data Type

```
module top
(
//Input
input logic[1:0][3:1]d1,
//Output
output byte q1_left,
output byte q1_right,
output byte q1_leftdimension
);
assign q1_left = $left(d1);
assign q1_right = $right(d1);
assign q1_leftdimension =$left(d1,2); // Dimension expression
  // returns value of the second dimension[3:1]
endmodule
```

Back

Example - Array Querying Function $low and $high Used on Unacked 3D-data Type

```
module top
(
//Input
input logic d1[2:1][1:5][4:8],
//Output
output byte q1_low,
output byte q1_high,
output byte q1_lowdimension
);
assign q1_low = $low(d1);
assign q1_high = $high(d1);
assign q1_lowdimension = $low(d1,3); // Dimension expression
  // returns value for the third dimension (i.e.,[4:8])
endmodule
```

Back

Example - Array Querying Function $size and $increment Used on a Mixed Array

```
module top
(
//Input
input byte d1[4:1],
//Output
output byte q1_size,
output byte q1_increment );
assign q1_size = $size(d1);
assign q1_increment = $increment(d1);
endmodule
```

Back

Example - Instantiating an interface Construct

```
 //TECHPUBS The following example defines, accesses, and
instantiates an interface construct.
 interface intf(input a, input b); //define the interface
 logic a1, b1;
 assign a1 = a;
 assign b1 = b;
 modport write (input a1, input b1); //define the modport
 endinterface

 module leaf(intf.write foo, output logic q); //access the intf
interface
 assign q = foo.a1 + foo.b1;
 endmodule

 module top(input a, input b, output q);
 intf inst_intf (a,b); //instantiate the intf interface
 leaf leaf_inst (inst_intf.write,q);
 endmodule
```

Back

Example - Type Casting of Aggregate Data Types

```
//Data type
typedef logic Logic_3D_dt [15:0][1:0][1:0];
typedef logic Logic_1D_dt [64:1];
module top (
//Inputs
input Logic_3D_dt Logic_3D,
//Outputs
output longint arith
);
//Constant delcaration
const Logic_1D_dt Logic_1DConst = '{default:1'b1};
//Arithmetic Operation
assign arith = longint'(Logic_3D) + longint'(Logic_1DConst);
endmodule
```

Back

Example - Bit-stream Casting

```
typedef struct {
bit start_bit = 0;
byte data_bits;
bit stop_bit = 1; }
```

```
uart_format_dt;
typedef logic tx_format_dt[9:0] ;
module top (
//Inputs
input byte data,
//Outputs
output tx_format_dt tx_data
);
uart_format_dt uart_data;
assign uart_data.data_bits = data;
assign tx_data = tx_format_dt’(uart_data);
endmodule
```

Back

Example - Sign Casting

```
module top (
//Inputs
input integer Integer,
input shortint Shortint,
//Outputs
output longint arith
);
//Arithmetic operation
assign arith = unsigned’(Integer) * unsigned’(Shortint);
endmodule
```

Back

Example - Size Casting

```
module top (
//Inputs
input longint Longint,
input byte Byte,
//Outputs
output shortint arith1
);
//Arithmetic operation
assign arith1 = 10’(Byte + Longint);
```

```
 endmodule
```

Back

Example - Type Casting of Singular Data Types

```
 typedef logic [31:0] unsigned_32bits;
 typedef logic [15:0] unsigned_16bits;
 module top (
 //Inputs
 input integer Integer,
 input shortint Shortint,
 //Outputs
 output longint arith
 );
 //Arithmetic operation
 assign arith = unsigned_32bits’(Integer) *
unsigned_16bits’(Shortint) ;
 endmodule
```

Back

Example - Basic Packed Union (arithmetic operation)

```
typedef union packed
{
  logic [3:0][0:3]u1;
  shortint u2;
  bit signed [1:2][8:1]u3;
}union_dt; // Union data type

module top
  (input union_dt d1,
   input union_dt d2,
   output union_dt q1,
   output union_dt q2
  );
assign q1.u2 = d1.u1 + d2.u2;
assign q2.u1 = d1.u2 - d1.u1[2][1];
endmodule
```

Back

Example - Array of Packed Union

```
typedef int int_dt;
typedef union packed
{
  int_dt u1;
  bit [0:3][1:8]u2;
}union_dt;
module top
  (input union_dt [1:0] d1, //Array of union
   input union_dt [1:0] d2, //Array of union
   output union_dt q1,
   output union_dt q2
  );
assign q1.u1 = d1[1].u1 ^ d2[0].u1;
assign q2.u2 = ~(d1[0].u2 | d2[1].u1);
endmodule
```

Back

Example - Basic Packed Union (logical operation)

```
typedef int unsigned UnsignInt_dt;
typedef union packed
{
  int u1;
  UnsignInt_dt u2;
}union_dt; //Union data type

module top
  (input union_dt d1,
input union_dt d2,
output union_dt q1,
output union_dt q2
);
assign q1.u1 = d1.u1 ^ d2.u1;
assign q2.u2 = d1.u2 | d2.u1;
endmodule
```

Back

Example - Nested Packed Union

```
typedef union packed
{
  byte u1;
  bit[1:0][4:1]u2;
  union packed
  {
     logic[8:1]nu1;
     byte unsigned nu2;
  }NstUnion; //Nested Union
}NstUnion_dt;

module top
  (input NstUnion_dt d1,
   input NstUnion_dt d2,
   output NstUnion_dt q1,
   output NstUnion_dt q2
  );
assign q1 = d1.NstUnion.nu1 & d2.u2[1];
assign q2.u1 = d2.NstUnion.nu2 |~ d1.u1;
endmodule
```

Back

Example - State-machine Design

```
module enum_type_check (clk, rst, same, statemachine1_is_five,
   statemachine2_is_six, statemachine1, statemachine2, both);
input clk, rst;
output reg same, statemachine1_is_five, statemachine2_is_six;
output int statemachine1, statemachine2, both;
enum {a[0:3] = 4} my,my1;

always@(posedge clk or posedge rst)
begin
if (rst)
   begin
      my <= a0;
   end
else
   case(my)
```

```
            a0 :begin
               my <= a1;
            end
            a1 :begin
               my <= a2;
            end
            a2 :begin
               my <= a3;
            end
            a3 :begin
               my <= a0;
            end
         endcase
      end

      always@(posedge clk or posedge rst)
      begin
      if (rst)
         begin
            my1 <= a0;
         end
      else
         case(my1)
            a0 :begin
               my1 <= a3;
            end
            a1 :begin
               my1 <= a0;
            end
            a2 :begin
               my1 <= a1;
            end
            a3 :begin
               my1 <= a2;
            end
         endcase
      end

      always@(posedge clk)
      begin
      statemachine1 <= my;
      statemachine2 <= my1;
```

```
both <= my + my1;
if (my == my1)
   same <= 1'b1;
else
   same <= 0;
if (my == 5)
   statemachine1_is_five <= 1'b1;
else
   statemachine1_is_five <= 1'b0;
if (my1 == 6)
   statemachine2_is_six <= 1'b1;
else
   statemachine2_is_six <= 1'b0;
end
endmodule
```

Back

Example – Type Parameter of Language-Defined Data Type

```
//Compilation Unit
module top
#(
  parameter type PTYPE = shortint,
  parameter type PTYPE1 = logic[3:2][4:1] //parameter is of
     //2D logic type
)
(
//Input Ports
  input PTYPE din1_def,
  input PTYPE1 din1_oride,
//Output Ports
  output PTYPE dout1_def,
  output PTYPE1 dout1_oride
);
sub u1_def //Default data type
(
  .din1(din1_def),
  .dout1(dout1_def)
);
sub #
(
  .PTYPE(PTYPE1) //Parameter type is override by 2D Logic
)
u2_oride
(
  .din1(din1_oride),
  .dout1(dout1_oride)
);
endmodule
//Sub Module
module sub
#(
  parameter type PTYPE = shortint //parameter is of shortint type
)
(
//Input Ports
  input PTYPE din1,
//Output Ports
  output PTYPE dout1
);
always_comb
begin
  dout1 = din1 ;
end
endmodule
```

Back

Example – Type Local Parameter

```
//Compilation Unit
module sub
#(
parameter type PTYPE1 = shortint, //Parameter is of shortint type
parameter type PTYPE2 = longint //Parameter is of longint type
)
(
//Input Ports
  input PTYPE1 din1,
//Output Ports
  output PTYPE2 dout1
);
//Localparam type definitation
```

```
localparam type SHORTINT_LPARAM = PTYPE1;
SHORTINT_LPARAM sig1;
assign sig1 = din1;
assign dout1 = din1 * sig1;
endmodule
```

Back

Example – Type Parameter of User-Defined Data Type

```
//Compilation Unit
typedef logic [0:7]Logic_1DUnpack[2:1];
typedef struct {
  byte R;
  int B;
  logic[0:7]G;
} Struct_dt;
module top
#(
  parameter type PTYPE = Logic_1DUnpack,
  parameter type PTYPE1 = Struct_dt
)
(
//Input Ports
  input PTYPE1    din1_oride,
//Output Ports
  output PTYPE1    dout1_oride
);
sub #
(
  .PTYPE(PTYPE1) //Parameter type is override by a structure type
)
u2_oride
(
  .din1(din1_oride),
  .dout1(dout1_oride)
);
endmodule
//Sub Module
module sub
#(
  parameter type PTYPE = Logic_1DUnpack // Parameter 1D
```

```
     // logic Unpacked data type
)
(
//Input Ports
  input PTYPE din1,
//Output Ports
  output PTYPE dout1
);
always_comb
begin
  dout1.R = din1.R;
  dout1.B = din1.B ;
  dout1.G = din1.G ;
end
endmodule
```

Back

Example – Parameter of Type enum

```
typedef enum {s1,s2,s3=24,s4=15,s5} enum_dt;
module sub
#(parameter enum_dt ParamEnum = s4)
   (input clk,
    input rst,
    input enum_dt d1,
    output enum_dt q1 );

always_ff@(posedge clk or posedge clk)
begin
if(rst)
   begin
      q1 <= ParamEnum;
   end
else
   begin
      q1 <= d1;
   end
end
endmodule
```

Back

Example – Parameter of Type longint Unpacked Array

```
module sub
#(parameter longint ParamMyLongint [0:1] ='{64'd1124,64'd1785})
   (input clk,
    input rst,
    input longint d1 [0:1],
    output longint q1 [0:1] );

always_ff@(posedge clk or posedge clk)
begin
if(rst)
   begin
      q1 <= ParamMyLongint;
   end
else
   begin
      q1 <= d1;
   end
end
endmodule
```

Back

Example – Parameter of Type longint

```
module sub
#(parameter longint ParamLongint = 64'd25)
   (input clk,
    input rst,
    input longint d1 ,
    output longint q1 );

always_ff@(posedge clk or posedge clk)
begin
if(rst)
   begin
      q1 <= ParamLongint;
   end
else
   begin
      q1 <= d1;
   end
end
endmodule
```

Back

Example – Parameter of Type structure

```
typedef byte unsigned Byte_dt;
typedef struct packed
   {shortint R;
    logic signed [4:3] G;
    bit [15:0] B;
    Byte_dt Y;
   }Struct_dt;

module sub
#(parameter Struct_dt ParamStruct ='{16'd128,2'd2,12'd24,8'd123})
   (
   //Input
    input clk,
    input rst,
    input Struct_dt d1,
   //Output
    output Struct_dt q1 );

always_ff@(posedge clk or posedge rst)
begin
if(rst)
   begin
      q1 <= ParamStruct;
   end
else
   begin
      q1 <= d1 ;
   end
end
endmodule
```

Back

Example - Simple typedef Variable Assignment

```
module src (in1,in2,out1,out2);
input in1,in2;
output reg out1,out2;
typedef int foo;
foo a,b;

assign a = in1; assign b = in2;
always@(a,b)
   begin
      out1 = a;
      out2 = b;
   end
endmodule
```

Back

Example - Using Multiple typedef Assignments

```
module src (in1,in2,in3,in4,out1,out2,out3);
input [3:0] in1,in2;
input in3,in4;
output reg [3:0] out1;output reg out2,out3;
typedef bit signed [3:0] foo1;
typedef byte signed foo2;
typedef int foo3;
struct {
   foo1 a;
   foo2 b;
   foo3 c;
   } foo;

always@(in1,in2,in3,in4)
   begin
      foo.a = in1 & in2;
      foo.b = in3 | in4;
      foo.c = in3 ^ in4;
   end

always@(foo.a,foo.b,foo.c)
   begin
```

```
          out1 = foo.a;
          out2 = foo.b;
          out3 = foo.c;
       end
    endmodule
```

Back

Example - Extern Module Instantiation

```
extern module top (input logic clock, load,
  input reset, input logic [3:0] d,
  output logic [3:0] cnt);
module top ( .* );
always @(posedge clock or posedge reset)
  begin
  if (reset)
     cnt <= 0;
  else if (load)
     cnt <= d;
  else cnt <= cnt + 1;
  end
endmodule
```

Back

Example - Extern Module Reference

```
extern module counter (clock, load, reset, d, cnt);
module top (clock, load, reset, d, cnt);
input logic clock, load;
input reset;
input logic [3:0] d;
output logic [3:0] cnt;
counter cnt1 (.clock(clock), .load(load), .reset(reset),
  .d(d), .cnt(cnt) );
endmodule
module counter (clock, load, reset, d, cnt );
input logic clock, load;
input reset;
input logic [3:0] d;
output logic [3:0] cnt;
```

```
always @(posedge clock or posedge reset)
begin
  if (reset)
     cnt <= 0;
  else if (load)
     cnt <= d;
  else cnt <= cnt + 1;
end
endmodule
```

Back

Example - $bits System Function

```
module top (input logic Clk,
   input logic Rst,
   input logic [7:0] LogicIn,
   output logic [$bits(LogicIn)-1:0] LogicOut,
   output logic [7:0] LogicConstSize );
logic [7:0] logic_const = 8’d0;

always@(posedge Clk, posedge Rst) begin
   if(Rst) begin
      LogicConstSize <= ‘d0;
      LogicOut <= logic_const;
   end
   else begin
      LogicConstSize <= $bits(logic_const);
      LogicOut <= $bits(LogicIn)-1 ^ LogicIn;
   end
end
endmodule
```

Back

Example - $bits System Function within a Function

```
module top (input logic Clk,
   input logic Rst,
   input logic [7:0] LogicIn,
   output logic [$bits(LogicIn)-1:0] LogicOut,
   output logic [7:0] LogicSize  );
```

```
function logic [$bits(LogicIn)-1:0]
   incr_logic (logic [7:0] a);
incr_logic = a + 1;
endfunction

always@(posedge Clk, posedge Rst) begin
   if(Rst) begin
      LogicSize <= 'd0;
      LogicOut <= 'd0;
   end
   else begin
      LogicSize <= $bits(LogicIn);
      LogicOut <= incr_logic(LogicIn);
   end
end
endmodule
```

Back

Example – Accessing Variables Declared in a generate-case

```
module test #(
  parameter mod_sel = 1,
  mod_sel2 = 3 )
     (input [7:0] a1,
      input [7:0] b1,
      output [7:0] c1,
      input [1:0][3:1] a2,
      input [1:0][3:1] b2,
      output [1:0][3:1] c2 );
typedef logic [7:0] my_logic1_t;
typedef logic [1:0][3:1] my_logic2_t;
generate
case(mod_sel)
  0:
  begin:u1
     my_logic1_t c1;
     assign c1 = a1 + b1;
  end
  1:
  begin:u1
     my_logic2_t c2;
     assign c2 = a2 + b2;
  end
  default:
  begin:def
     my_logic1_t c1;
     assign c1 = a1 + b1;
  end
endcase
endgenerate
generate
case(mod_sel2)
  0:
  begin:u2
     my_logic1_t c1;
     assign c1 = a1 + b1;
  end
  1:
  begin:u2
     my_logic2_t c2;
     assign c2 = a2 + b2;
  end
  default:
  begin:def2
     my_logic1_t c1;
     assign c1 = a1 * b1;
  end
endcase
endgenerate
assign c2 = u1.c2;
assign c1 = def2.c1;
endmodule
```

Back

Example – Shift Register Using generate-for

```
module sh_r #(
  parameter width = 8,
  pipe_num = 3 )
     (input clk,
      input[width-1:0]din,
      output[width-1:0] dout );
```

```
genvar i;
generate
  for(i=0;i<pipe_num;i=i+1)
  begin:u
  reg [width-1:0] sh_r;
     if(i==0)
     begin
        always @ (posedge clk)
           sh_r <= din;
     end
     else
     begin
        always @ (posedge clk)
           sh_r <= u[i-1].sh_r;
     end
  end
endgenerate
assign dout = u[pipe_num-1].sh_r;
endmodule
```

Back

Example Accessing Variables Declared in a generate-if

```
module test #(
  parameter width = 8,
     sel = 0 )
        (input [width-1:0] a,
         input [width-1:0] b,
         input clk,
         output [(2*width)-1:0] c,
         output bit_acc,
         output [width-3:0] prt_sel );
  genvar i;
  reg [width-1:0] t_r;
generate
  if(sel == 0)
  begin:u
     wire [width-1:0] c;
     wire [width-1:0] t;
     assign {c,t} = {~t_r,a|b};
  end
  else
  begin:u
     wire [width-1:0] c;
     wire [width-1:0] t;
     assign {c,t} = {~t_r,a^b};
  end
endgenerate
always @ (posedge clk)
begin
     t_r <= u.t;
  end
assign c = u.c;
assign bit_acc = u.t[0];
  assign prt_sel = u.t[width-1:2];
endmodule
```

Back

Example - Do-while with case Statement

```
module src (out, a, b, c, d, sel);
output [3:0] out;
input [3:0] a, b, c, d;
input [3:0] sel;
reg [3:0] out;
integer i;

always @ (a or b or c or d or sel)
   begin
   i=0;
   out = 3’b000;
      do
         begin
            case (sel)
               4’b0001: out[i] = a[i];
               4’b0010: out[i] = b[i];
               4’b0100: out[i] = c[i];
               4’b1000: out[i] = d[i];
               default: out = ‘bx;
            endcase
         i= i+1;
         end
      while (i < 4);
   end
endmodule
```

Back

Example - Do-while with if-else Statement

```
module src (out, a, b, c, d, sel);
output [3:0] out;
input [3:0] a, b, c, d;
input [3:0] sel;
reg [3:0] out;
integer i;

always @ (a or b or c or d or sel)
   begin
   i=0;
   out = 4'b0000;
      do
         begin
            if(sel == 4'b0001) out[i] = a[i];
            else if(sel == 4'b0010) out[i] = b[i];
            else if(sel == 4'b0100) out[i] = c[i];
            else if(sel == 4'b1000) out[i] = d[i];
            else out = 'bx;
         i= i+1;
         end
      while (i < 4);
   end
endmodule
```

Back

Example - Simple do-while Loop

```
module src (in1,in2,out);
input [7:0] in1,in2;
output reg [7:0] out;
integer i;
```

```
always @ (in1,in2)
begin
   i = 0;
   do
      begin
         out[i] = in1[i] + in2[i];
         i = i+1;
      end
   while (i < 8 );
end
endmodule
```

Back

Example - Simple for Loop

```
module simpleloop (output reg [7:0]y, input [7:0]i, input clock);
always@(posedge clock)
begin : loop
    for (int count=0; count < 8; count=count+1) // SV code
    y[count]=i[count];
end
endmodule
```

Back

Example - For Loop with Two Variables

```
module twovarinloop (in1, in2, out1, out2);
parameter p1 = 3;
input [3:0] in1;
input [3:0] in2;
output [3:0] out1;
output [3:0] out2;
reg [3:0] out1;
reg [3:0] out2;

always @*
   begin
      for (int i = 0, int j = 0; i <= p1; i++)
      begin
         out1[i] = in1[i];
```

```
              out2[j] = in2[j];
              j++;
           end
        end
     endmodule
```

Back

Example - Inside operator with array of parameter at LHS operator

```
module top
(
//Input
input byte din1,
//Output
output logic dout
);

parameter byte param1[1:0] = '{8'd12,8'd111};
assign dout = (din1) inside {param1,121,-16};
endmodule
```

Back

Example - Inside operator with dynamic input at LHS operator

```
module top
(
//Input
input byte din,
//Output
output logic dout
);

assign dout = din inside {8'd2, -8'd3, 8'd5};
endmodule
```

Back

Example - Inside operator with dynamic input at LHS and RHS operators

```
module top
(
//Input
input byte din1,
input byte din2,
//Output
output logic dout
);

assign dout = (din1) inside {din2,105,-121,-116};
endmodule
```

Back

Example - Inside operator with expression at LHS operator

```
module top
(
//Input
input byte din1,
input byte din2,
//Output
output logic dout
);

assign dout = (din1 | din2) inside {14,17,2,20};
endmodule
```

Back

Example - Constant Declarations

```
package my_pack;
const logic foo_logic = 1'b1;
endpackage
import my_pack::*;

module test (
input logic inp,
input clk,
output logic out );

always @(posedge clk)
begin
   out <= inp ^ foo_logic;
end
endmodule
```

Back

Example - Direct Reference Using Scope Resolution Operator (::)

```
package mypack;
logic foo_logic = 1'b1;
endpackage
module test (
input logic data1,
input clk,
output logic out1 );

always @(posedge clk)
begin
   out1 <= data1 ^ mypack::foo_logic;
end
endmodule
```

Back

Example - Function Declarations

```
package automatic_func;
parameter fact = 2;
function automatic [63:0] factorial;
input [31:0] n;
if (n==1)
   return (1);
else
   return (n * factorial(n-1));
endfunction
endpackage
```

```
import automatic_func::*;
module src (input [1:0] a, input [1:0] b,
   output logic [2:0] out );
always_comb
begin
   out = a + b + factorial(fact);
end
endmodule
```

Back

Example - Importing Specific Package Items

```
package mypack;
logic foo_logic = 1'b1;
endpackage
module test (
input logic data1,
input clk,
output logic out1 );
import mypack::foo_logic;

always @(posedge clk)
begin
   out1 <= data1 ^ foo_logic;
end
endmodule
```

Back

Example - Import Statements from Other Packages

```
package param;
parameter fact = 2;
endpackage
package automatic_func;
import param::*;
function automatic [63:0] factorial;
input [31:0] n;
   if (n==1)
      return (1);
   else
```

```
      return (n * factorial(n-1));
endfunction
endpackage

import automatic_func::*;
import param::*;
module src (input [1:0] a, input [1:0] b,
   output logic [2:0] out );
always_comb
begin
   out = a + b + factorial(fact);
end
endmodule
```

Back

Example - Parameter Declarations

```
package mypack;
parameter a_width = 4;
parameter b_width = 4;
localparam product_width = a_width+b_width;
endpackage
import mypack::*;

module test (
input [a_width-1:0] a,
input [b_width-1:0] b,
output [product_width-1:0] c );
assign c = a * b;
endmodule
```

Back

Example - Scope Resolution

```
 //local parameter overrides package parameter value (dout <=
data[7:0];)
 package mypack;
 parameter width = 4;
 endpackage
```

```
import mypack::*;
module test (data,clk,dout);
parameter width = 8; // local parameter
input logic [width-1:0] data;
input clk;
output logic [width-1:0] dout;

always @(posedge clk)
begin
   dout <= data;
end
endmodule
```

Back

Example - Task Declarations

```
package mypack;
parameter FACT_OP = 2;
   task automatic factorial(input integer operand,
      output [1:0] out1);
   integer nFuncCall = 0;
   begin
   if (operand == 0)
      begin
         out1 = 1;
      end
   else
      begin
         nFuncCall++;
         factorial((operand-1), out1);
         out1 = out1 * operand;
      end
   end
   endtask
endpackage
import mypack::*;

module src (input [1:0] a, input [1:0] b,
   output logic [2:0] out );
logic [1:0] out_tmp;
```

```
always_comb
factorial(FACT_OP,out_tmp);
assign out = a + b + out_tmp;
endmodule
```

Back

Example - User-defined Data Types (typedef)

```
package mypack;
typedef struct packed {
   int a;
   } my_struct;
endpackage
import mypack::my_struct;

module test (inp1,inp2,out);
input my_struct inp1;
input my_struct inp2;
output int out;
assign out = inp1.a * inp2.a;
endmodule
```

Back

Example - Wildcard (*) Import Package Items

```
package mypack;
logic foo_logic = 1'b1;
endpackage
module test (
input logic data1,
input clk,
output logic out1 );
import mypack::*;

always @(posedge clk)
begin
   out1 <= data1 ^ foo_logic;
end
```

```
endmodule
```

Back

Example – Packed type inputs/outputs with LHS operator

```
module streaming
(
input byte a,
output byte str_rev,
output byte str
);

assign {>>{str}} = a;
assign {<<{str_rev}} = a;

endmodule
```

Back

Example – Packed type inputs/outputs with RHS operator

```
module streaming
(
input longint a,
output longint str_rev,
output longint str
);

assign str_rev = {<< {a}};
assign str = {>> {a}};

endmodule
```

Back

Example – Slice-size streaming with LHS slice operation

```
module streaming
(
input logic a[1:8],
output logic signed [1:4] str_rev[1:2],
output logic signed [1:4] str[1:2]
);

assign {>>4{str}} = a;
assign {<<4{str_rev}} = a;

endmodule
```

Back

Example – Slice-size streaming with RHS operator

```
typedef shortint shortint_dt [2:1];
typedef byte byte_dt [1:2][3:2];
typedef struct {
  logic [3:0] a [2:1];
  byte b;
  shortint c[4:2]; }
struct_dt;
module streaming (
  input shortint_dt a,
  input  byte_dt b,
  output struct_dt c_pack,
  output struct_dt c_unpack );
assign c_pack = {<< 5 {a}};
assign c_unpack = {<< 2 {b}};
endmodule
```

Back

Example – Unpacked type inputs/outputs with RHS operator

```
typedef logic [5:0]my_dt [1:0];

module streaming
(
input logic [5:0] a[1:0], //same layout - size same as the output
input logic [3:0] b[2:0], //different layout - same size as output
input logic [2:0]c[1:0], //different layout and size
output my_dt str_rev1,
output my_dt str_rev_difflay,
output my_dt str_rev_less
);

assign str_rev1 = {<<{a}};
assign str_rev_difflay = {<< {b}};
assign str_rev_less = {<< {c,2'b11}};

endmodule
```

Back

Example - Priority case

```
module src (out, a, b, c, d, sel);
output out;
input a, b, c, d;
input [3:0] sel;reg out;

always @ (a,b,c,d,sel)
   begin
      priority case (sel)
         4'b0000: out = c;
         4'b0001: out = b;
         4'b0100: out = d;
         4'b1000: out = a;
      endcase
   end
endmodule
```

Back

Example - Unique case

```
module src (out, a, b, c, d, sel);
output out;
input a, b, c, d;
input [3:0] sel;
reg out;

always @ (a,b,c,d,sel)
   begin
      unique case (sel)
         4'b0001: out = c;
```

```
                  4'b0010: out = b;
                  4'b0100: out = d;
                  4'b1000: out = a;
               endcase
            end
         endmodule
```

Back

APPENDIX C

Example Code

This appendix contains the code samples that are referenced by the corresponding chapter.

Example - Direct Instantiation Using Configuration Declaration

```
--Entity to be instantiated using the configuration
library ieee;
use ieee.std_logic_1164.all;
entity module0 is
  generic (SIZE : integer := 10);
  port (l : in std_logic_vector(SIZE-1 downto 0);
     m : in std_logic_vector(SIZE-1 downto 0);
     out1 : out std_logic_vector(SIZE-1 downto 0) );
end entity module0;
architecture behv of module0 is
begin
  out1 <= l xor m;
end behv;
-- Configuration for the entity module0
configuration conf_sub of module0 is
  for behv
  end for;
end conf_sub;
-- Module in which the entity module0 is instantiated
-- using the configuration
library ieee;
use ieee.std_logic_1164.all;
entity top is
```

```
    port (in0 : in std_logic_vector(31 downto 0);
       in1 : in std_logic_vector(31 downto 0);
       out1 : out std_logic_vector(31 downto 0) );
  end entity top;
  architecture behv of top is
  begin
  U0: configuration conf_sub
    generic map (SIZE => 32)
    port map (l => in0,
       m => in1,
       out1 => out1 );
  end behv;
```

Back

Example - Direct Instantiation Using Configuration Declaration

```
--Entity to be instantiated using the configuration
library ieee;
use ieee.std_logic_1164.all;
entity module0 is
  generic (SIZE : integer := 10);
  port (l : in std_logic_vector(SIZE-1 downto 0);
     m : in std_logic_vector(SIZE-1 downto 0);
     out1 : out std_logic_vector(SIZE-1 downto 0) );
end entity module0;
architecture behv of module0 is
begin
  out1 <= l xor m;
end behv;
-- Configuration for the entity module0
configuration conf_sub of module0 is
  for behv
  end for;
end conf_sub;
-- Module in which the entity module0 is instantiated
-- using the configuration
library ieee;
use ieee.std_logic_1164.all;
entity top is
  port (in0 : in std_logic_vector(31 downto 0);
     in1 : in std_logic_vector(31 downto 0);
```

Copyright © 2012 Synopsys, Inc.

```
     out1 : out std_logic_vector(31 downto 0) );
end entity top;
architecture behv of top is
begin
U0: configuration conf_sub
  generic map (SIZE => 32)
  port map (l => in0,
     m => in1,
     out1 => out1 );
end behv;
```

Back

APPENDIX C

Example Code

This appendix contains the code samples that are referenced by the corresponding chapter.

Example - Context declaration

```
context zcontext is
  library ieee;
  use ieee.std_logic_1164.all;
end context zcontext;

context work.zcontext;
use ieee.numeric_std.all;

entity myTopDesign is
  port (in1: in std_logic_vector(1 downto 0);
     out1: out std_logic_vector(1 downto 0) );
end myTopDesign;

architecture myarch2 of myTopDesign is
begin
  out1 <= in1;
end myarch2;
```

Back

Example - Unconstrained element types

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
package myTypes is
  type memUnc is array (natural range <>) of std_logic_vector;
  function summation(varx: memUnc) return std_logic_vector;
end package myTypes;
package body myTypes is
  function summation(varx: memUnc) return std_logic_vector is
     variable sum: varx'element;
  begin
     sum := (others => '0');
        for I in 0 to varx'length - 1 loop
           sum := sum + varx(I);
        end loop;
     return sum;
  end function summation;
end package body myTypes;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use work.myTypes.all;
entity sum is
  port (in1: memUnc(0 to 2)(3 downto 0);
        out1: out std_logic_vector(3 downto 0) );
end sum;
architecture uncbehv of sum is
begin
  out1 <= summation(in1);
end uncbehv;
```

Back

Example - Unconstrained elements within nested arrays

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
package myTypes is
  type t1 is array (0 to 1) of std_logic_vector;
  type memUnc is array (natural range <>) of t1;
  function doSum(varx: memUnc) return std_logic_vector;
end package myTypes;
package body myTypes is
  function addVector(vec: t1) return std_logic_vector is
     variable vecres: vec’element := (others => ‘0’);
  begin
     for I in vec’Range loop
        vecres := vecres + vec(I);
     end loop;
     return vecres;
  end function addVector;
  function doSum(varx: memUnc) return std_logic_vector is
     variable sumres: varx’element’element;
  begin
     if (varx’length = 1) then
        return addVector(varx(varx’low));
     end if;
     if (varx’Ascending) then
        sumres := addVector(varx(varx’high)) +
           doSum(varx(varx’low to varx’high-1));
     else
        sumres := addVector(varx(varx’low)) +
           doSum(varx(varx’high downto varx’low+1));
     end if;
     return sumres;
  end function doSum;
end package body myTypes;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use work.myTypes.all;
entity uncfunc is
  port (in1: in memUnc(1 downto 0)(open)(0 to 3);
        in2: in memUnc(0 to 2)(open)(5 downto 0);
        in3: in memUnc(3 downto 0)(open)(2 downto 0);
        out1: out std_logic_vector(5 downto 0);
        out2: out std_logic_vector(0 to 3);
        out3: out std_logic_vector(2 downto 0) );
end uncfunc;
architecture uncbehv of uncfunc is
begin
```

```
    out1 <= doSum(in2);
    out2 <= doSum(in1);
    out3 <= doSum(in3);
  end uncbehv;
```

Back

Example - Unconstrained record elements

```
library ieee;
use ieee.std_logic_1164.all;
entity unctest is
  port (in1: in std_logic_vector (2 downto 0);
        in2: in std_logic_vector (3 downto 0);
        out1: out std_logic_vector(2 downto 0) );
end unctest;
architecture uncbehv of unctest is
  type zRec is record
     f1: std_logic_vector;
     f2: std_logic_vector;
  end record zRec;
subtype zCnstrRec is zRec(f1(open), f2(3 downto 0));
subtype zCnstrRec2 is zCnstrRec(f1(2 downto 0), f2(open));
signal mem: zCnstrRec2;
begin
  mem.f1 <= in1;
  mem.f2 <= in2;
  out1 <= mem.f1 and mem.f2(2 downto 0);
end uncbehv;
```

Back

Example - all keyword

```
entity mycomp is
  port (a, c: in bit; b: out bit);
end mycomp;

architecture myarch of mycomp is
begin
  process (all)
  begin
```

```
        b <= not a or c;
     end process;
   end myarch;
```

Back

Example 1: VHDL 2008 Style Conditional Operator

```
entity condOpTest is
port (
sel, in1, in2: in bit;
res: out bit
);
end condOpTest;
architecture rtlArch of condOpTest is
begin
process(in1,in2,sel)
begin
if sel then
res <= in2;
else
res <= in1;
end if;
end process;
end rtlArch;
```

Back

Example 2: VHDL 1993 Style Conditional Operator

```
entity condOpTest is
port (
sel, in1, in2: in bit;
res: out bit
);
end condOpTest;
architecture rtlArch of condOpTest is
begin
process(in1,in2,sel)
begin
if sel = ‘1’ then
```

```
res <= in2;
else
res <= in1;
end if;
end process;
end rtlArch;
```

Back

Example 1: Logical Operators

```
entity reductionOpTest is
port (
invec: in bit_vector(2 downto 0);
nandout, xorout, xnorout, norout, orout, andout: out bit
);
end reductionOpTest;

architecture rtlArch of reductionOpTest is
begin
nandout <= nand invec;
xorout <= xor invec;
xnorout <= xnor invec;
norout <= nor invec;
orout <= or invec;
andout <= and invec;
end rtlArch;
```

Back

Example: Relational Operator

```
entity relOpTest is
port (
in1, in2: in bit;
res_eq, res_lteq: out bit
);
end relOpTest;
architecture rtlArch of relOpTest is
begin
res_eq <= in1 ?= in2;
```

```
res_lteq <= in1 ?<= in2;
end rtlArch;
```

Back

Example - Including generics in packages

```
-- Generic Package Declaration
package myTypesGeneric is
  generic
  (width: integer := 7; testVal: bit_vector(3 downto 0) := "0011";
   dfltVal: bit_vector(3 downto 0) := "1110"
  );
  subtype nvector is bit_vector(width-1 downto 0);
  constant resetVal: bit_vector(3 downto 0) := dfltVal;
  constant myVal: bit_vector(3 downto 0) := testVal;
end package myTypesGeneric;

-- Package instantiation
package myTypes is new work.myTypesGeneric
  generic map
  (width => 4, dfltVal => "0110"
  );

library IEEE;
package my_fixed_pkg is new IEEE.fixed_generic_pkg
  generic map
  (fixed_round_style    => IEEE.fixed_float_types.fixed_round,
   fixed_overflow_style => IEEE.fixed_float_types.fixed_saturate,
   fixed_guard_bits     => 3,
   no_warning           => false
  );

use work.myTypes.all;
use work.my_fixed_pkg.all;

entity myTopDesign is
  port  (in1: in nvector; out1: out nvector;
     insf: in sfixed(3 downto 0);
     outsf: out sfixed(3 downto 0);
     out2, out3, out4: out bit_vector(3 downto 0)
     );
```

```
end myTopDesign;

architecture myarch2 of myTopDesign is
begin
  out1 <= in1;
  out2 <= resetVal;
  out3 <= myVal;
  outsf <= insf;
end myarch2;
```

Back

Example: Minimum Maximum Predefined Functions

```
entity minmaxTest is
port (ary1, ary2: in bit_vector(3 downto 0);
minout, maxout: out bit_vector(3 downto 0);
unaryres: out bit
);
end minmaxTest;
architecture rtlArch of minmaxTest is
begin
maxout <= maximum(ary1, ary2);
minout <= minimum(ary1, ary2);
unaryres <= maximum(ary1);
end rtlArch;
```

Back

Example - Case-generate statement with alternatives

```
entity myTopDesign is
  generic (instSel: bit_vector(1 downto 0) := “10”);
  port (in1, in2, in3: in bit; out1: out bit);
end myTopDesign;
architecture myarch2 of myTopDesign is
component mycomp
  port (a: in bit; b: out bit);
end component;
begin
```

```
a1: case instSel generate
  when “00” =>
     inst1: component mycomp port map (in1,out1);
  when “01” =>
     inst1: component mycomp port map (in2,out1);
  when others =>
     inst1: component mycomp port map (in3,out1);
  end generate;
end myarch2;
```

Back

Example - Case-generate statement with labels for configuration

```
entity myTopDesign is
generic (selval: bit_vector(1 downto 0) := “10”);
  port (in1, in2, in3: in bit; tstIn: in bit_vector(3 downto 0);
        out1: out bit);
end myTopDesign;
architecture myarch2 of myTopDesign is
  component mycomp
     port (a: in bit; b: out bit);
  end component;
begin
a1: case selval generate
  when spec1: “00” | “11”=> signal inRes: bit;
     begin
        inRes <= in1 and in3;
        inst1: component mycomp port map (inRes,out1);
     end;
  when spec2: “01” =>
     inst1: component mycomp port map (in1, out1);
  when spec3: others =>
     inst1: component mycomp port map (in3,out1);
  end generate;
end myarch2;
entity mycomp is
  port (a : in bit;
        b : out bit);
end mycomp;
architecture myarch of mycomp is
```

```
begin
  b <= not a;
end myarch;
architecture zarch of mycomp is
begin
  b <= ‘1’;
end zarch;
configuration myconfig of myTopDesign is
for myarch2
  for a1 (spec1)
     for inst1: mycomp use entity mycomp(myarch);
     end for;
  end for;
  for a1 (spec2)
     for inst1: mycomp use entity mycomp(zarch);
     end for;
  end for;
  for a1 (spec3)
     for inst1: mycomp use entity mycomp(myarch);
     end for;
  end for;
end for;
end configuration myconfig;
```

Back

Example - Else/elsif clauses in if-generate statements

```
entity myTopDesign is
  generic (genval: bit_vector(1 downto 0) := “01”);
  port (in1, in2, in3: in bit; out1: out bit);
end myTopDesign;

architecture myarch2 of myTopDesign is

component mycomp
  port (a: in bit;
     b: out bit );
end component;

begin
```

```
a1:
  if spec1: genval=”10” generate
     inst1: mycomp port map (in1,out1);
  elsif spec2: genval=”11” generate
     inst1: component mycomp port map (in2,out1);
  else spec3: generate
     inst1: component mycomp port map (in3,out1);
  end generate;
end myarch2;


library ieee;
use ieee.std_logic_1164.all;

entity mycomp is
  port ( a: in bit;
        b : out bit);
end entity mycomp;

architecture myarch1 of mycomp is
begin
  b <= ‘1’ xor a;
end myarch1;

architecture myarch2 of mycomp is
begin
  b <= ‘1’ xnor a;
end myarch2;

architecture myarch3 of mycomp is
signal temp : bit := ‘1’;
begin
  b <= temp xor not(a);
end myarch3;

configuration myconfig of myTopDesign is
  for myarch2
     for a1 (spec1)
        for inst1: mycomp
           use entity mycomp(myarch1);
        end for;
     end for;
```

```
        for a1 (spec2)
           for inst1: mycomp
              use entity mycomp(myarch2);
           end for;
        end for;
        for a1 (spec3)
           for inst1: mycomp
              use entity mycomp(myarch3);
           end for;
        end for;
     end for;
   end configuration myconfig;
```

Back

Example - Use of case? statement

```
library ieee;
use ieee.std_logic_1164.all;
entity myTopDesign is
  port (in1, in2: in bit;
        sel: in std_logic_vector(2 downto 0);
        out1: out bit );
end myTopDesign;
architecture myarch2 of myTopDesign is
begin
  process(all)
  begin
     a1: case? sel is
        when “1--” =>
           out1 <= in1;
        when “01-” =>
           out1 <= in2;
        when others =>
           out1 <= in1 xor in2;
     end case?;
  end process;
end myarch2;
```

Back

Example - Use of select? Statement

```
library ieee;
use ieee.std_logic_1164.all;
entity myTopDesign is
  port (in1, in2: in bit;
        sel: in std_logic_vector(2 downto 0);
        out1: out bit );
end myTopDesign;
architecture myarch2 of myTopDesign is
begin
  with sel select?
     out1 <=
     in1 when “1--”,
     in2 when “01-”,
     in1 xor in2 when others;
end myarch2;
```

Back

Example - extended character set

```
library ieee;
use ieee.std_logic_1164.all;
entity get_version is
  port ( ver : out string(16 downto 1));
end get_version;
architecture behv of get_version is
constant version : string (16 downto 1) := “version ©«ãëïõü»”;
-- Above string includes extended ASCII characters that
-- fall between 127-255
begin
  ver <= version;
end behv;
```

Back

APPENDIX C

Example Code

This appendix contains the code samples that are referenced by the corresponding chapter.

Example – Active Net and Constant GND Driving Output Net (Verilog)

```
module test(clk,data_in,data_out,radd,wradd,wr,rd);
input clk,wr,rd;
input data_in;
input [5:0]radd,wradd;
output data_out;
// component instantiation for shift register module
shrl srl_lut0 (
  .clk(clk),
  .sren(wr),
  .srin(data_in),
  .srout(data_out)
  );
// Instantiation for ram
dpram dpram_lut3 (
  .clk(clk),
  .data_in(data_in),
  .data_out(data_out),
  .radd(radd),
  .wradd(wradd),
  .wr(wr),
  .rd(rd)
  );
```

```
endmodule

module shrl (clk,sren,srin,srout);
input clk;
input sren;
input srin;
output srout;

parameter width = 32;
reg [width-1:0] sr;

always@(posedge clk)
begin
  if (sren == 1)
  begin
     sr <= {sr[width-2:0], srin};
  end
end
// Constant net driving

// the output net
assign srout = 1’b0;
endmodule

module dpram(clk,data_in,data_out,radd,wradd,wr,rd);
input clk,wr,rd;
input data_in;
input [5:0]radd,wradd;
output data_out;

reg dout;
reg [0:0]mem[63 :0];

always @ (posedge clk)
begin
  if(wr)
     mem[wradd] <= data_in;
end

always @ (posedge clk)
begin
  if(rd)
```

```
        dout <= mem[radd];
     end

   assign  data_out = dout;

   endmodule
```

Back

Example – Active Net and Constant VCC Driving Output Net (VHDL)

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

entity test is
port (
     clk,rst          : in  std_logic;
     sr_en            : in  std_logic;
     data             : in  std_logic;
     data_op          : out std_logic
  );
end entity test;

architecture rtl of test is
component shrl
  generic ( sr_length   : natural );
  port (
     clk          : in  std_logic;
     sr_en        : in  std_logic;
     sr_ip        : in  std_logic;
     sr_op        : out std_logic
  );
end component shrl;

component d_ff
  port (
     data, clk, rst : in std_logic;
     q : out std_logic
  );
end component d_ff;

begin
```

```
-- instantiation of shift register
shift_register : shrl
  generic map( sr_length => 64  )
  port map (
     clk         => clk,
     sr_en       => sr_en,
     sr_ip       => data,
     sr_op       => data_op  );

-- instantiation of flipflop
dff1 : d_ff
  port map
     data => data,
     clk => clk,
     rst => rst,
     q => data_op);

end rtl;
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity shrl is
  generic (   sr_length     : natural   );
  port (
     clk         : in  std_logic;
     sr_en       : in  std_logic;
     sr_ip       : in  std_logic;
     sr_op       : out std_logic
  );
end entity shrl;

architecture rtl of shrl is
signal sr_reg     : std_logic_vector(sr_length-1 downto 0) ;
begin
shreg_lut: process (clk)
begin
  if rising_edge(clk) then
     if sr_en = ‘1’ then
        sr_reg <= sr_reg(sr_length-2 downto 0) & sr_ip;
     end if;
  end if;
end process shreg_lut;

-- Constant net driving output net
sr_op <= ‘1’;

end architecture rtl;
library IEEE;
use IEEE.std_logic_1164.all;

entity d_ff is
  port
     data, clk, rst : in std_logic;
     q : out std_logic);
end d_ff;

architecture behav of d_ff is
begin
  FF1:process (clk) begin
     if (clk’event and clk = ‘1’) then
        if (rst = ‘1’) then
           q <= ‘0’;
        else q <= data;
        end if;
     end if;
  end process FF1;

end behav;
```

Back

Index