Synopsys FPGA Synthesis
Synplify Pro for Microsemi Edition
Reference

January 2014

SYNOPSYS®

Copyright Notice and Proprietary Information

Right to Copy Documentation

Destination Control Statement

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, Total-Recall, TSUPREM-4, VCSi, VHDL Compiler, VMC, and Worksheet Buffer are trademarks of Synopsys, Inc.

Service Marks (sm)

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

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

ARM and AMBA are registered trademarks of ARM Limited.

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

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

Printed in the U.S.A
January 2014

Contents

Chapter 1: Product Overview

Chapter 2: User Interface Overview

Chapter 3: HDL Analyst Tool

Chapter 4: Constraints

Chapter 5: SCOPE Constraints Editor

Chapter 6: Constraint Syntax

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 12: RAM and ROM Inference

Chapter 13: IP and Encryption Tools

Chapter 14: Scripts

Appendix A: Designing with Microsemi

Chapter 1

Product Overview

This document is part of a set that includes reference and procedural infor-mation 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, on page 20
- Overview of the Synthesis Tools, on page 24
- Starting the Synthesis Tool, on page 29
- Logic Synthesis Overview, on page 31
- Getting Help, on page 35

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

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 and in the *User Guide*.

- The Text Editor window, with a language-sensitive editor for writing and editing HDL code. See .

- The SCOPE® (Synthesis Constraint Optimization Environment®) tool, which provides a spreadsheet-like interface for managing timing constraints and design attributes. See .

- FSM Compiler, a symbolic compiler that performs advanced finite state machine (FSM) optimizations. See .

- Integration with the Identify RTL Debugger.

- FSM Explorer, which tries different state machine optimizations before picking the best implementation. See .

- The FSM Viewer, for viewing state transitions in detail. See .

- The Tcl window, a command line interface for running TCL scripts. See .

- 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 P&R Automatically after Synthesis, on page 492 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 38).

- 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.

Project view

Project Tree view

Status

Implementation Results view

Menus

Toolbars

Buttons

Tabs to access main views

Tabs to access Tcl Script and Messages

Output Window

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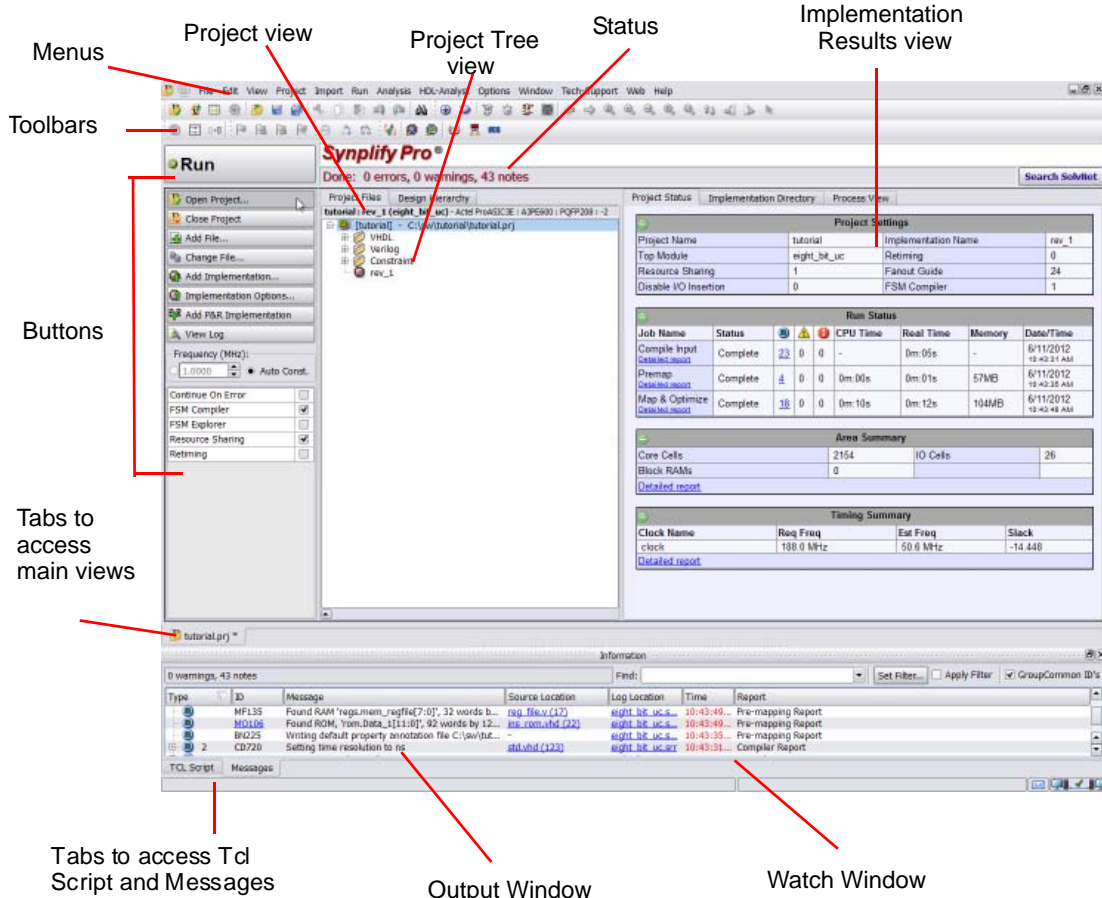
The following table shows where you can find information about different parts of the GUI, some of which are not shown in the above figure. For more information, see the *User Guide*.

For information about...	See...
Project window	The Project View, on page 38
RTL view	RTL View, on page 61
Technology view	Technology View, on page 62
Text Editor view	Text Editor View, on page 68
FSM Viewer window	FSM Viewer Window, on page 67
Tcl window	Tcl Script Window, on page 56
Watch Window	Watch Window, on page 52
SCOPE spreadsheet	SCOPE User Interface, on page 164
Other views and windows	The Project View, on page 38
Menu commands and their dialog boxes	Chapter 4, *User Interface Commands*
Toolbars	Toolbars, on page 85
Buttons	Buttons and Options, on page 101
Context-sensitive popup menus and their dialog boxes	Popup Menus, on page 317
Online help	Use the F1 keyboard shortcut or click the Help button in a dialog box. See Help Menu, on page 313, for more information.

Projects and Implementations

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

An *implementation* is one version (also called a revision) of a project, run with certain parameter or option settings. You can synthesize again, with a different set of options, to get a different implementation. In the Project view, an implementation is shown in the folder of its project; the active implementation 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.

Starting the Synthesis Tool

Before you can start the synthesis tool, you must install it and set up the software license appropriately. You can then start the tool interactively or in batch mode. 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 in Batch Mode

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 synplify_pro.

For complete syntax details, refer to *synplify_pro,* on page 91 in the *Command Reference*.

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

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 in two phases, making it as small as possible to improving circuit performance. These optimizations are technology independent. The final result is an srs database, which can be graphically represented in the RTL schematic view.

The following figure summarizes the stages of the standard compiler flow:
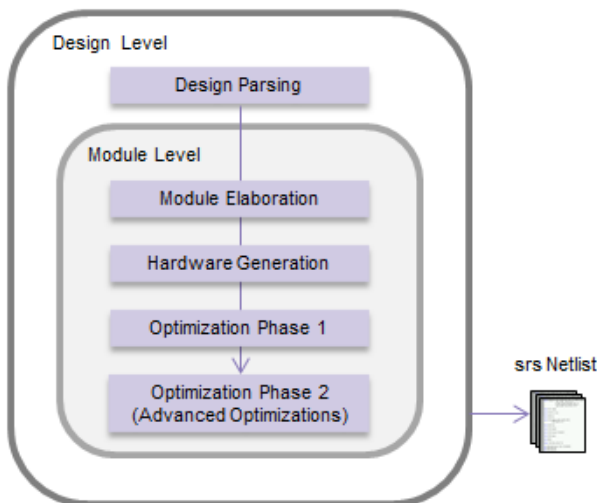
You can also run the compiler incrementally.

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 , 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

SYNOPSYS®
Accelerating Innovation

CHAPTER 2

User Interface Overview

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

This chapter describes the following aspects of the graphical user interface (GUI):

- The Project View, on page 38
- The Project Results View, on page 42
- Other Windows and Views, on page 51
- FSM Compiler, on page 76
- FSM Explorer, on page 78
- Using the Mouse, on page 78
- User Interface Preferences, on page 83
- Toolbars, on page 85
- Keyboard Shortcuts, on page 93
- Buttons and Options, on page 101

The Project View

The Project View is the main interface to the tool. The Project view consists of a Project Management View on the left and a Project Results View on the right. See Multiple Pane Project View, on page 38 for an overview:

Multiple Pane Project View

The Project Management view is on the left side of the window, and is used to create or open projects, create new implementations, set device options, and initiate design synthesis. You can use it to manage and synthesize hierarchical designs. The Project Results view is on the right.

The following figure shows the main parts of the interface. Additional detail about the project view is described here:

- Project Management View, on page 39
- The Project Results View

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 218).
Buttons and options	Allow immediate access to some of the more common commands. See Buttons and Options, on page 101 for details.
Project Management view	Lists the projects and implementations, and their associated HDL source files and constraint files. See Projects and Implementations, on page 28 for details.
Implementation Results view	Lists the result of the synthesis runs for the implementations of your design.

Project Management View
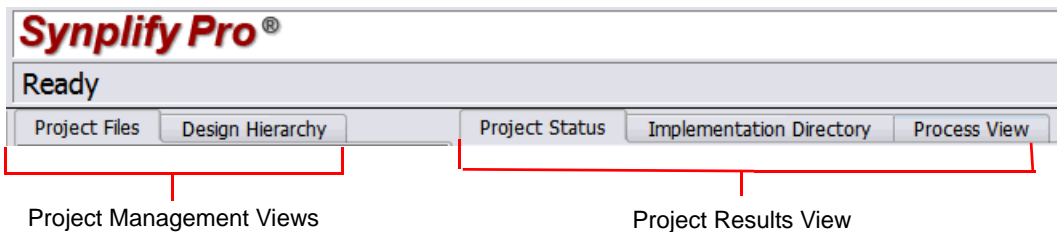
Project Management Views   Project Results View

The Project Management view is on the left side of the window, and is used to create or open projects, create new implementations, set device options, and initiate design synthesis. The graphical user interface (GUI) lets you manage hierarchical designs that can be synthesized independently and imported back to the top-level project in a team design flow. The following figure shows the Project view as it appears in the interface.

The following figure shows the Project view as it appears in the Synplify Pro interface.
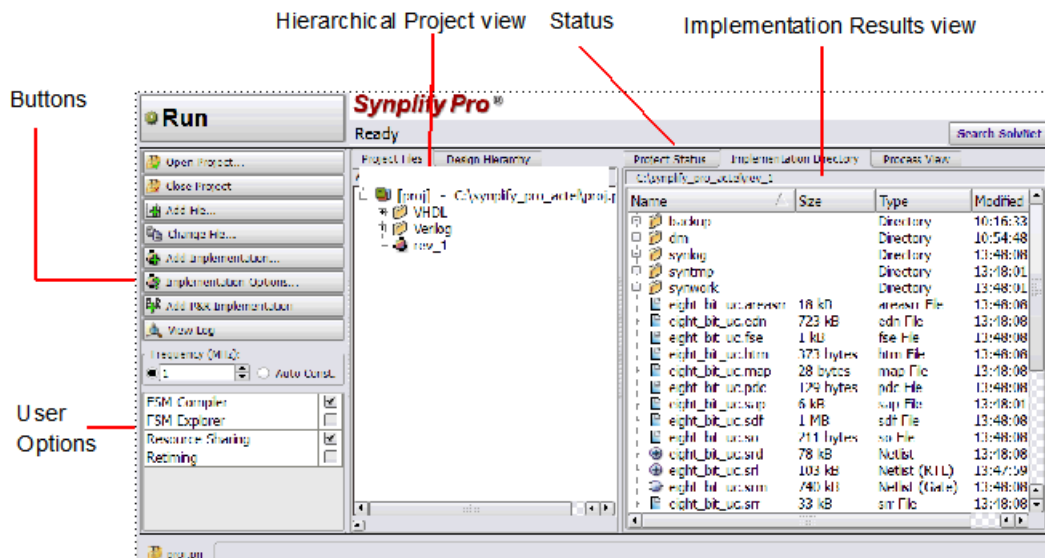
The Synplify Pro tool provides hierarchical management support for large designs. The tool lets you manage hierarchical projects in a team design flow, where you have independent hierarchical subprojects.

The Project view contains two tabs with different views of the design that help you manage hierarchical projects:

- Project Files Tab
- Design Hierarchy Tab

Both tabs in this view have right-click popup menu commands for managing design files and hierarchy. For descriptions of these commands, see Project Management Commands, on page 323.

Project Files Tab

The Project Files view displays the top-level design and any sub-projects that can be synthesized.
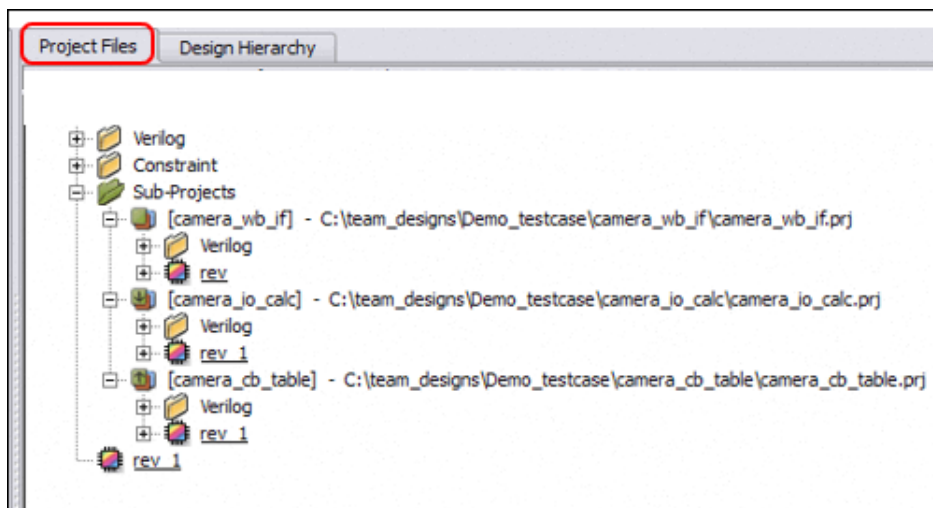
Design Hierarchy Tab

The Design Hierarchy view displays the instance block and design block hierarchy for a design.

The colors for the block icons represent the following:

Icon Description	Designates the following...
White rectangle surrounding a b	Black box
Yellow	Design block (subproject)
Yellow with a T inside	Top-level design
Green with a P inside	Design block that has been exported as a sub-project

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 Tab
- Implementation Directory
- Process View

Project Status Tab

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 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

For details about how to access synthesis results, see Accessing Specific Reports Quickly, on page 179.

Synplify Pro for Microsemi Edition Reference Manual
42   January 2014

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 Settings is populated with the project settings from the run_options.txt file after a synthesis run. This section displays information, like the following:

- 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 about the synthesis run:

- Job name - Jobs include Compiler Input, Premap, and Map & Optimize. The job might have a Detailed Report link. When you click this link, it takes you to the corresponding report in the log file.

- **Status** - Reports whether the job is running or completed.

- Notes, warnings, and errors – These columns are headed by the respective icons and display the number of messages. The messages themselves are displayed in the Messages tab, beside the TCL Script tab. Links are available to the error message and the log location.

- 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 the Detailed Report link when applicable, to go to the log file and information about the selected report. These reports are written to the synlog folder for the active implementation.

Area Summary

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

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.
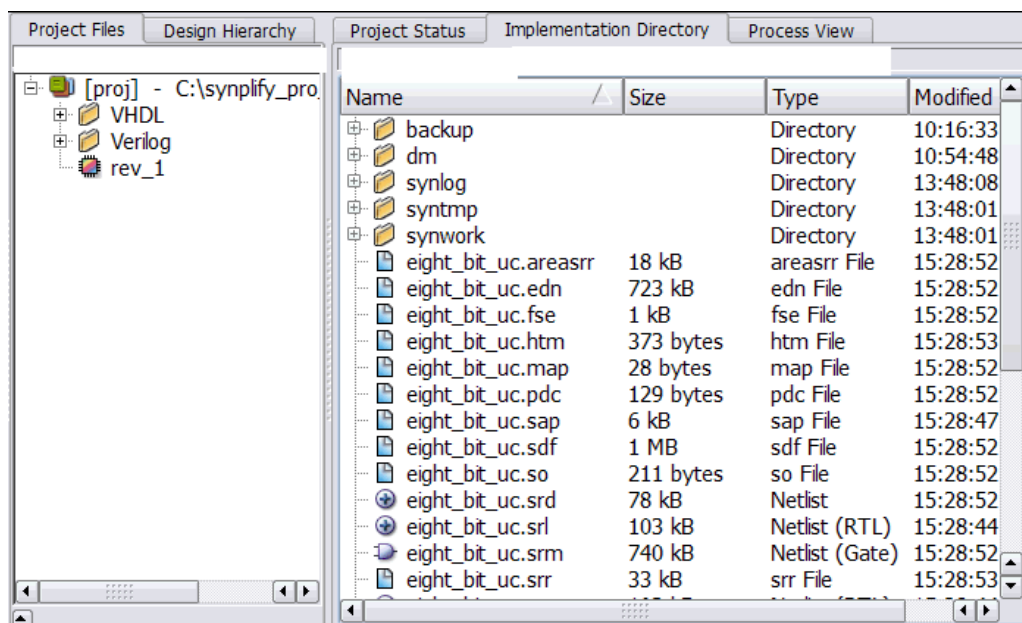
Process View

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 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 52

- Watch Window, on page 52

- Tcl Script and Messages Windows, on page 55

- Tcl Script Window, on page 56

- Message Viewer, on page 56

- Output Windows (Tcl Script and Watch Windows), on page 60

- RTL View, on page 61

- Technology View, on page 62

- Hierarchy Browser, on page 65

- FSM Viewer Window, on page 67

- Text Editor View, on page 68

- Context Help Editor Window, on page 71

- Interactive Attribute Examples, on page 73

- Search SolvNet, on page 75

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

Project view	The Project View, on page 38
SCOPE	SCOPE Tabs, on page 165

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 257) 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.
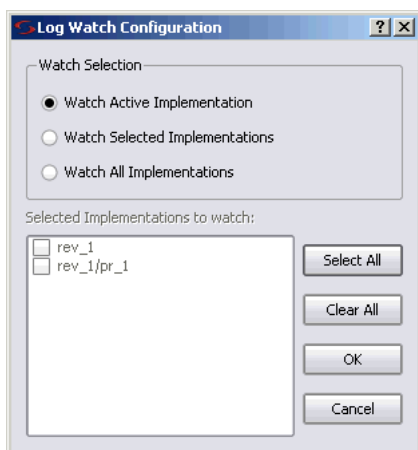
In the Watch window, indicate which implementations to watch under Watch Selection. The selected implementation(s) will display in the Watch window.

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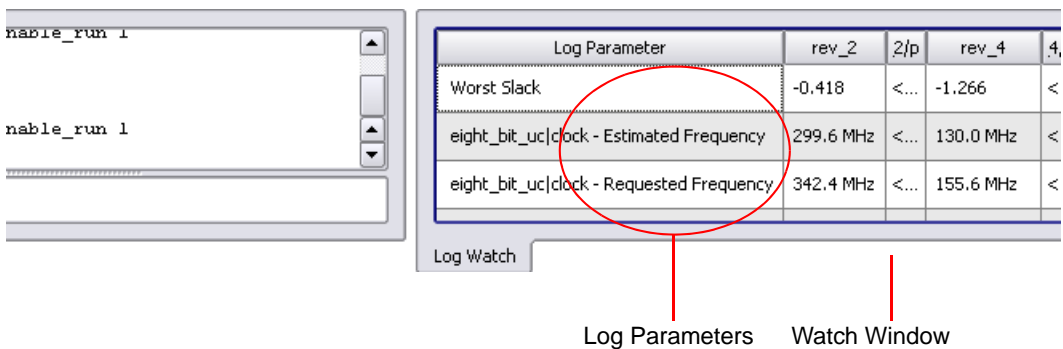
To choose log parameters from a pull-down menu, click in the Log Parameter section of the window. Click the pull-down arrow that appears to display the parameter list choices:

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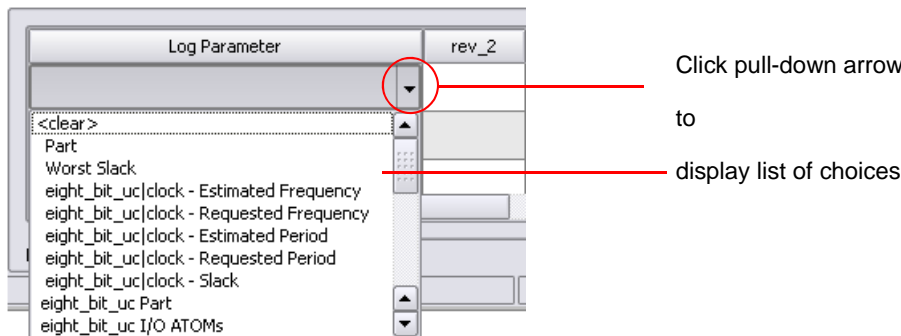
The Watch window creates an entry for each implementation of a project:

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.
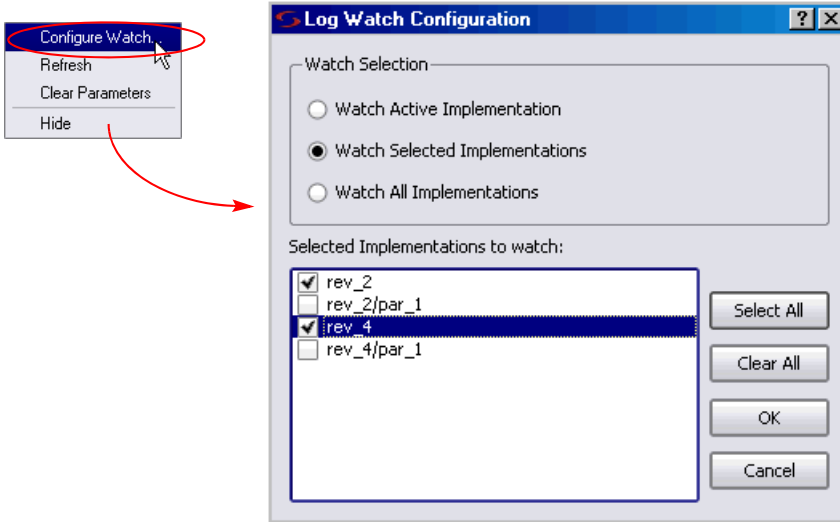
Tcl Script and Messages Windows

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.

Messages panel displays errors, warnings, and notes

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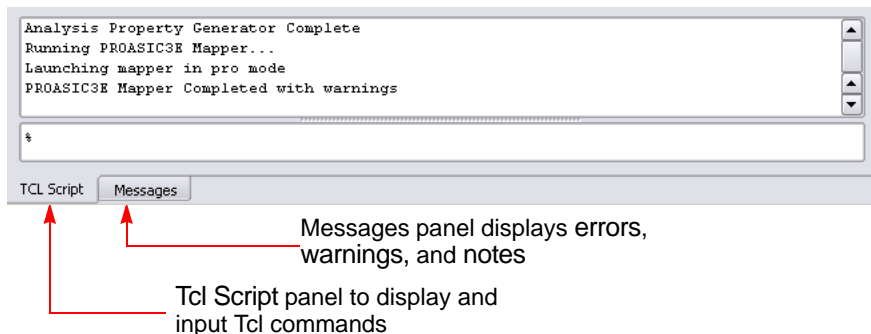
You can float the Tcl windows by clicking on a window edge while holding the Ctrl or Shift key. You can then drag the window to float it anywhere on the screen or dock it at an edge of the application window. Double-click in the banner to toggle between docked and floating.

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 56 and Message Viewer, on page 56.

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

- Synthesis Commands, on page 94, for information about the Tcl synthesis commands.

- Generating a Job Script, on page 465 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.
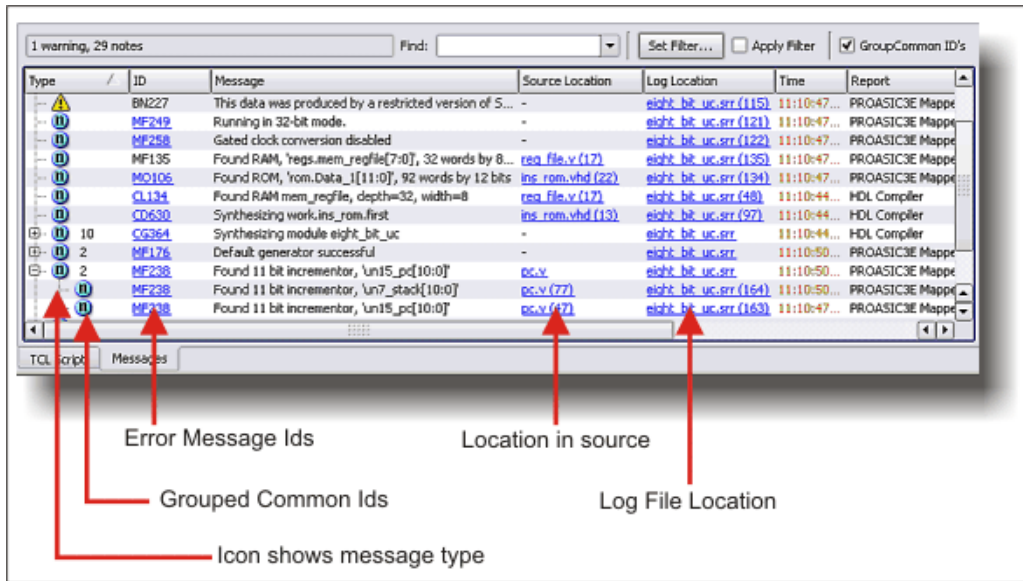
Interactive tasks in the Messages panel include:

- 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 189 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 59.
Apply Filter	Enable/disable the last saved filter.

Item	Description
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.
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 that the error, warning, or note message was recorded in the log file for the various stages of synthesis (for example: compiler, premap, and map). If you rerun synthesis, only new messages generate a new timestamp for this session. **Note:** Once synthesis has run to completion, all the srr files for the different stages of synthesis are merged into one unified srr file. If you exit the GUI, these timestamps remain the same when you re-open the same project in the GUI again.
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 189 in the *User Guide*.

	Enable	Type	ID	Message	Source Location	Log Location	Time
1	✔	Warning	FX107				
2	✔	Note	CD233				
3	✔	Note	CD630				
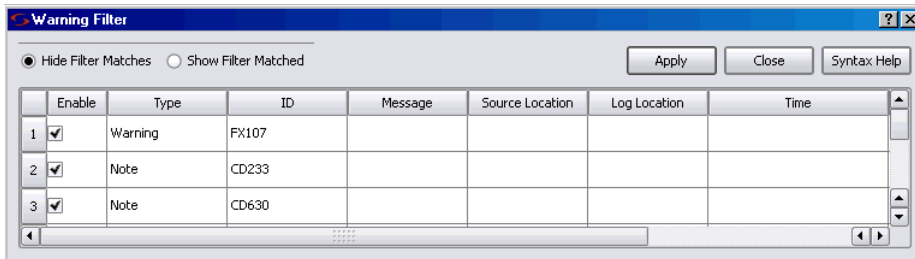
Item	Description
Hide Filter Matches	Hides matched criteria in the Messages Panel.
Show Filter Matches	Shows matched criteria in the Messages Panel.
Syntax Help	Gives quick syntax descriptions.
Apply	Applies the filter criteria to the Messages Panel report, without closing the window.
Type, ID, Message, Source Location, Log Location, Time, Report	Log file report criteria to use when filtering.

The following is a filtering example.
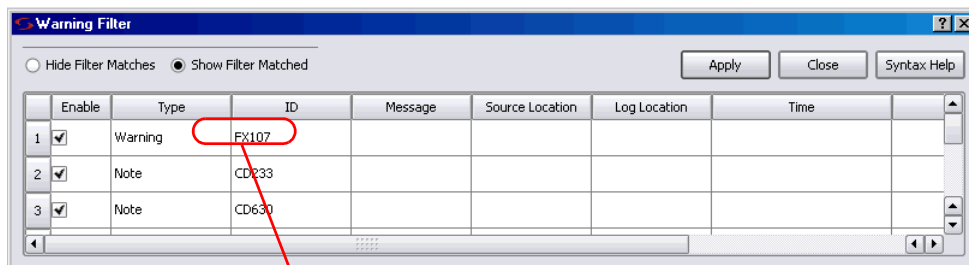
Show Filter Matches
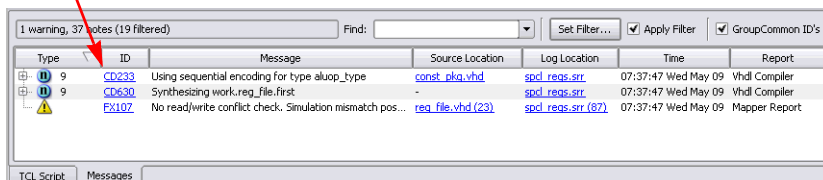
Hide Filter Matches
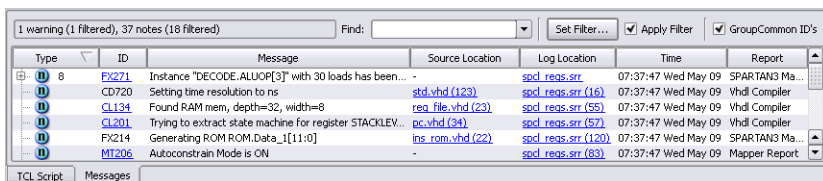
Output Windows (Tcl Script and Watch Windows)

The Output windows are the Tcl Script and Log Watch windows. To display or hide them, use View->Output Windows from the main menu. Refer to Watch Window, on page 52 and Tcl Script and Messages Windows, on page 55 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 62.

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 65. 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.
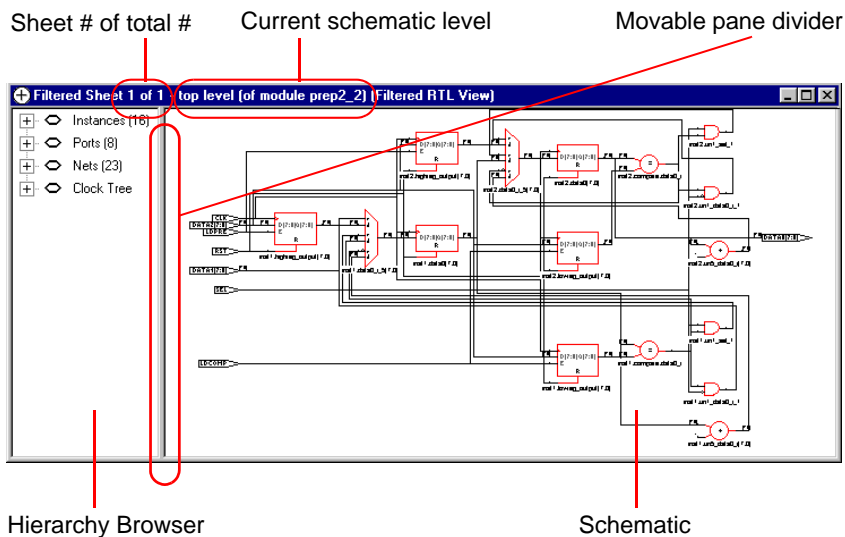
The design in the RTL schematic can be hierarchical or flattened. Further, the view can consist of the entire design or part of it. Different commands apply, depending on the kind of RTL view.

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

For information about...	See...
Hierarchy Browser	Hierarchy Browser, on page 65
Procedures for RTL view operations like crossprobing, searching, pushing/popping, filtering, flattening, etc.	Working in the Schematic Views, on page 204 of the *User Guide.*
Explanations or descriptions of features like object display, filtering, flattening, etc.	HDL Analyst Tool, on page 105
Commands for RTL view operations like filtering, flattening, etc.	Accessing HDL Analyst Commands, on page 107 HDL Analyst Menu, on page 272
Viewing commands like zooming, panning, etc.	View Menu: RTL and Technology Views Commands, on page 171
History commands: Back and Forward	View Menu: RTL and Technology Views Commands, on page 171
Search command	Find Command (HDL Analyst), on page 163

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 61.

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 65. 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.

Sheet # of total #   Current schematic level   Movable pane divider

Hierarchy Browser   Schematic
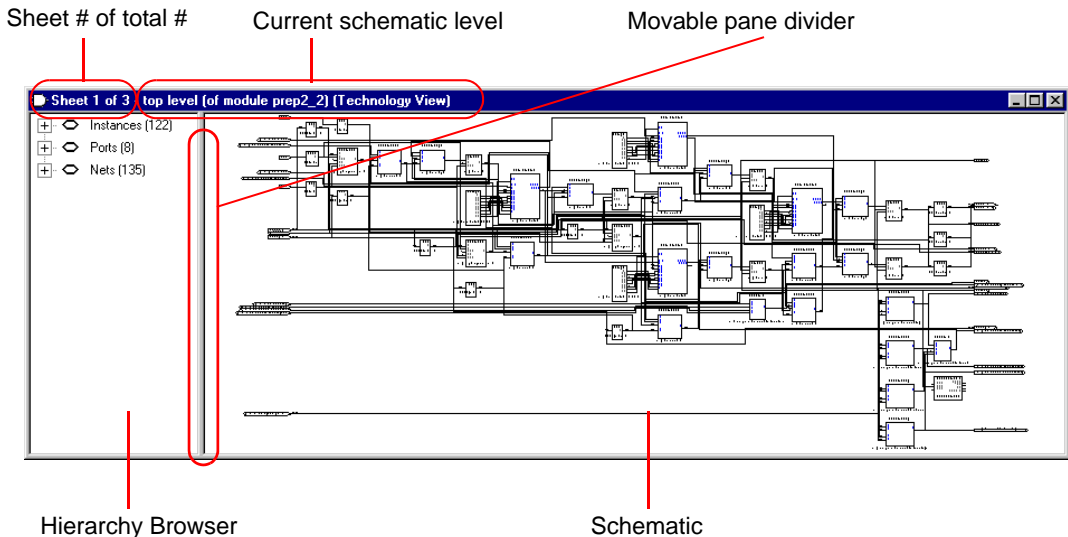
The schematic design can be hierarchical or flattened. Further, the view can consist of the entire design or a part of it. Different commands apply, depending on the kind of view. In addition to all the features available in RTL views, Technology views have two additional features: critical path filtering 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 65
Procedures for Technology view operations like crossprobing, searching, pushing/popping, filtering, flattening, etc.	Working in the Schematic Views, on page 204 of the *User Guide*
Explanations or descriptions of features like object display, filtering, flattening, etc.	HDL Analyst Tool, on page 105
Commands for Technology view operations like filtering, flattening, etc.	Accessing HDL Analyst Commands, on page 107 HDL Analyst Menu, on page 272
Viewing commands like zooming, panning, etc.	View Menu: RTL and Technology Views Commands, on page 171
History commands: Back and Forward	View Menu: RTL and Technology Views Commands, on page 171
Search command	Find Command (HDL Analyst), on page 163

Hierarchy Browser

The Hierarchy Browser is the left pane in the RTL and Technology views. (See RTL View, on page 61 and Technology View, on page 62.) 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 66 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 129 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	⊐	AND gate
⊐	Output port	⊐	NAND gate
⬭	Bidirectional port	D	OR gate
⊸	Net	D	NOR gate
⊡	Other primitive instance	⊅	XOR gate
⬛	Hierarchical instance	⊅	XNOR gate
⊡	Technology-specific primitive or inferred ROM	⊕	Adder
⊡	Register or inferred state machine	⊛	Multiplier
⊅	Multiplexer	⊜	Equal comparator
⊼	Tristate	⊙	Less-than comparator
⊳	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
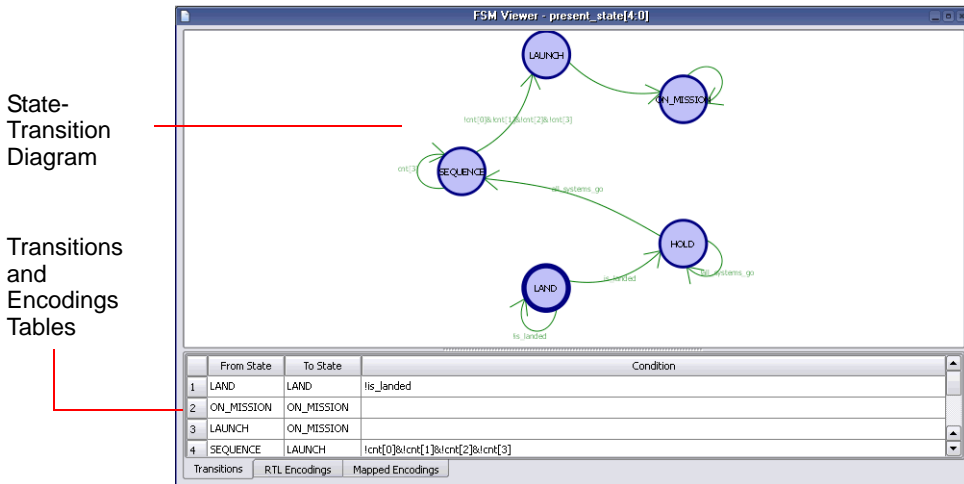
For the FSM Viewer to display state machine names for a Verilog design, you must use the Verilog parameter keyword. If you specify state machine names using the define keyword, the FSM Viewer displays the binary values for the state machines, rather than their names.

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.
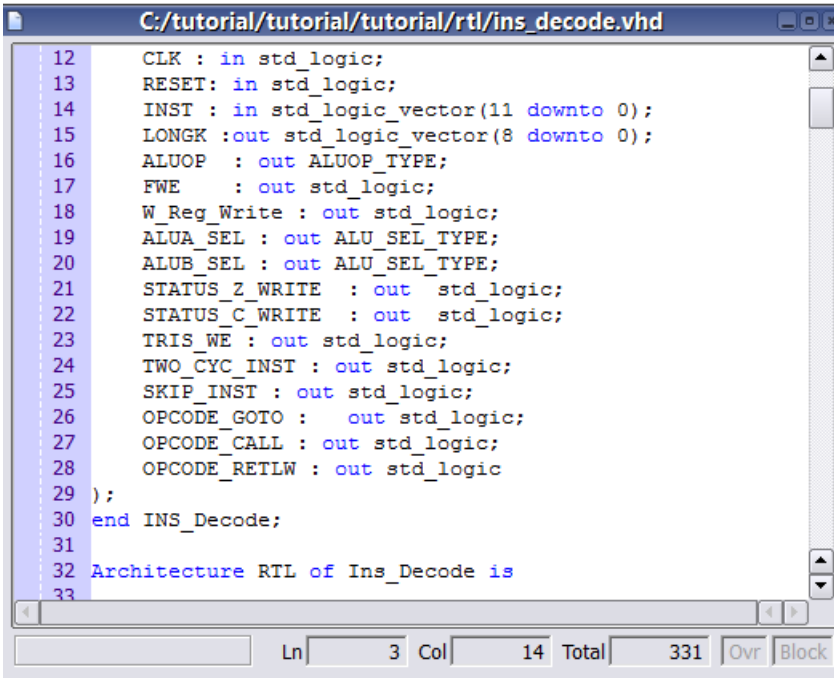
To accomplish this...	Do this...
Open the FSM Viewer	Run the FSM Compiler or the FSM Explorer. Use the push/pop mode in the RTL view to push down into the FSM and open the FSM Viewer window.
Hide/display the table	Use the FSM icons.
Filter selected states and their transitions	Select the states. Right-click 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 126, for information on the operation of pushing into a state machine.

- FSM Viewer Toolbar, on page 89, for information on the FSM icons.

- See Using the FSM Viewer, on page 263 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 36 in the *User Guide*.

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 ([icon]) on the Edit toolbar. The line number is highlighted to indicate that there is a bookmark at the beginning of the line.
Deleting a bookmark	Click the line with the bookmark. Choose Edit ->Toggle Bookmark, type Ctrl-F2, or click the Toggle Bookmark icon ([icon]) on the Edit toolbar.
Deleting all bookmarks	Choose Edit ->Delete all Bookmarks, type Ctrl-Shift-F2, or click the Clear All Bookmarks icon ([icon]) on the Edit toolbar.
Editing columns	Press and hold Alt, then drag the mouse down a column of text to select it.

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

See also:

- Editor Options Command, on page 293, for information on setting Text Editor preferences.

- File Menu, on page 154, for information on printing setup operations.

- Edit Menu Commands for the Text Editor, on page 160, for information on Text Editor editing commands.

- Text Editor Popup Menu, on page 319, for information on the Text Editor popup menu.

- Text Editor Toolbar, on page 90, for information on bookmark icons of the Edit toolbar.

- Keyboard Shortcuts, on page 93, for information on keyboard shortcuts that can be used in the Text Editor.

Context Help Editor Window

Use the Context Help button to copy Verilog or SystemVerilog constructs into your source file or Tcl constraint commands into your Tcl file. When you load a Verilog/SystemVerilog file or Tcl file into the UI, the Context Help button displays at the bottom of the window. Click this button to display the Context Help Editor.
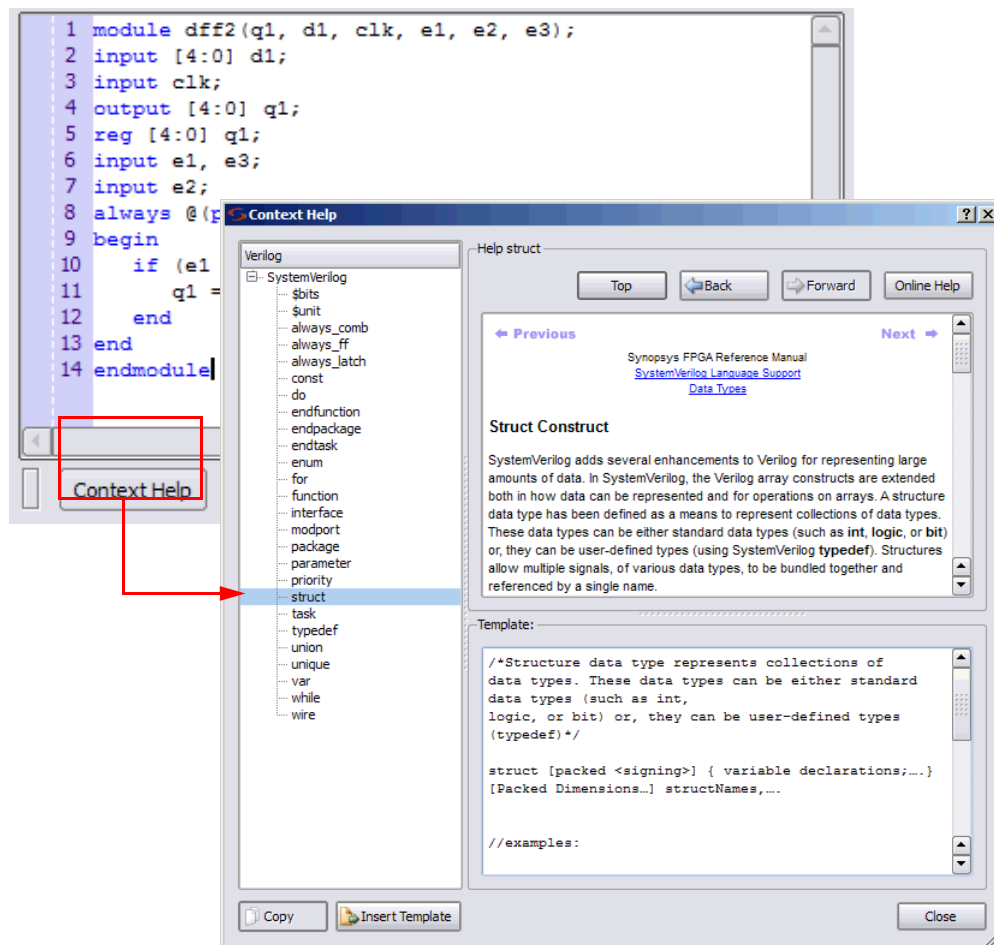
When you select a construct in the left-side of the window, the online help description for the construct is displayed. If the selected construct has this feature enabled, the online help topic is displayed on the top of the window and a generic code or command template for that construct is displayed at the bottom. The Insert Template button is also enabled. When you click the Insert Template button, the code or command shown in the template window is inserted into your file at the location of the cursor. This allows you to easily insert the code or constraint command 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 or constraint command 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.
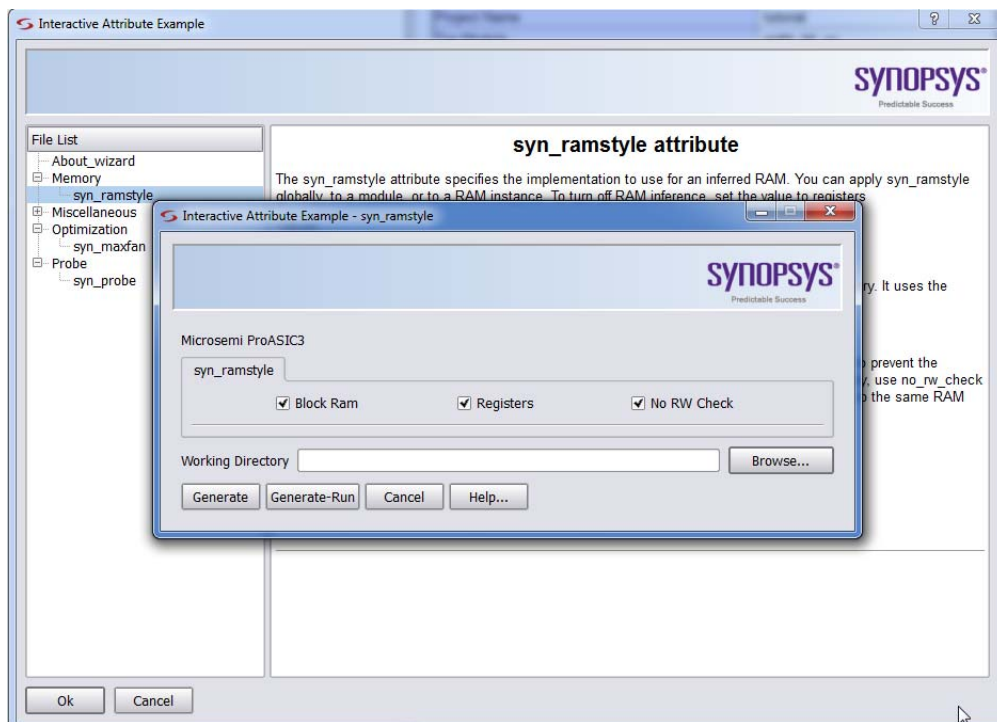
3. Double-click an attribute to start the wizard.

4. Specify the Working Directory location to write your project.

5. Click Generate to generate a project for your attribute.

   A project will be created with an implementation for each attribute value selected.

6. Click Generate Run to run synthesis for all the implementations. When synthesis completes:

   – The Technology view opens to show how the selected attribute impacts synthesis.

   – You can compare resource utilization and timing information between implementations in the Log Watch window.

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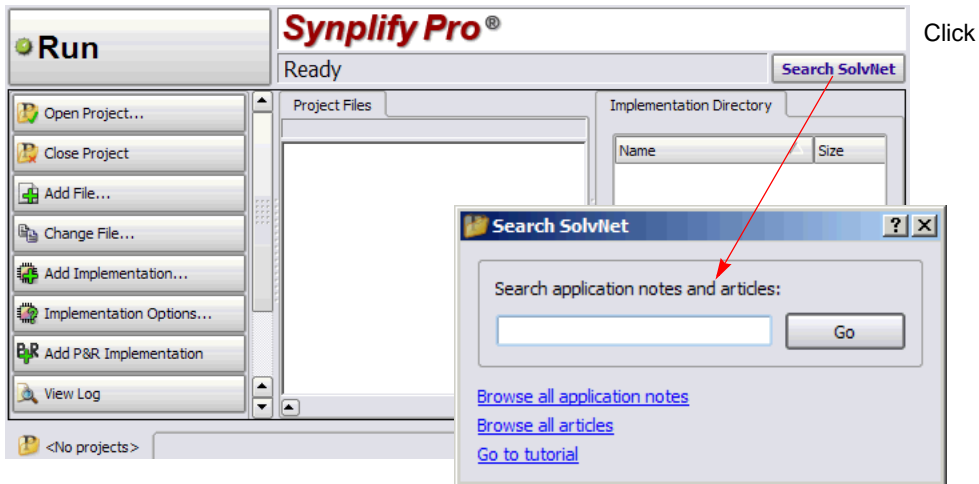
You can search the SolvNet database for Articles and Application Notes using the following methods:

- 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.

- Click the Go to tutorial link—takes you to the tutorial page for the Synopsys FPGA product you are using (same as Help->Tutorial).

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 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 67.

See also:

- Log File, on page 257, for information on the log file.

- RTL and Technology Views Popup Menus, on page 337, 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 348* 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 353 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 79

- Using the Mouse Buttons, on page 81

- Using the Mouse Wheel, on page 83

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.
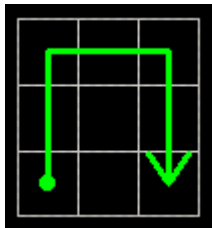
Redo Last Operation   Back to Previous View
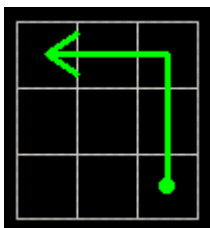
The Mouse Stroke Tutor

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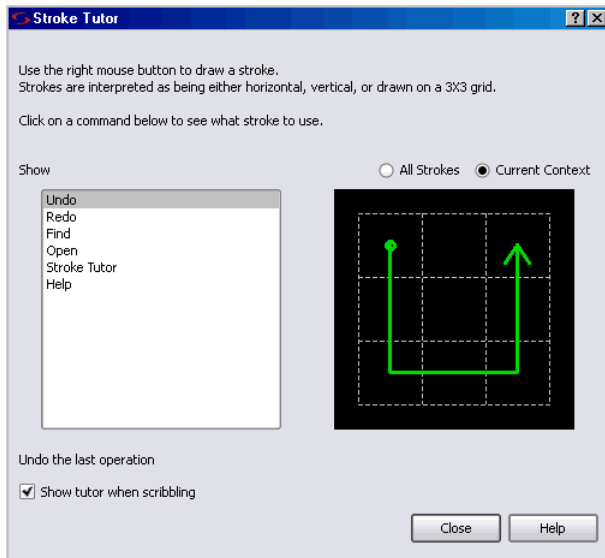
The tutor displays the available strokes along with a description and a diagram of the stroke. You can draw strokes while the tutor is displayed.

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 61, and Technology View, on page 62, 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.

- Analyst Toolbar — Manipulation of RTL and Technology views.

- FSM Viewer Toolbar — Display of finite state machine (FSM) information.

- Text Editor Toolbar — Text Editor bookmark commands.

- Tools Toolbar — Opens supporting tools.

You can enable or disable the display of individual toolbars – see Toolbar Command, on page 173.

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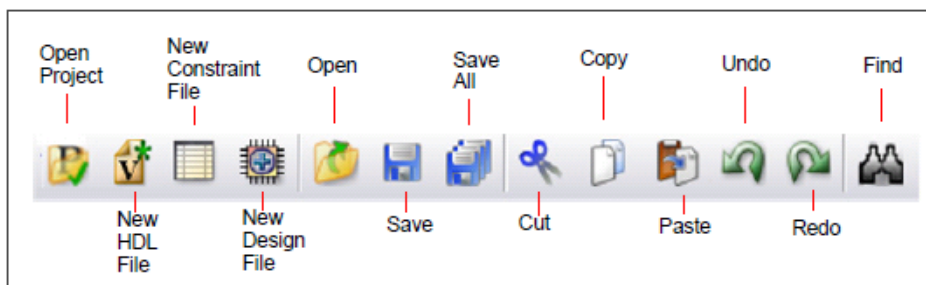
The following table describes the default Project icons. Each is equivalent to a File or Edit menu command; for more information, see the following:

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.

Icon	Description
Save All	Saves all files associated with the current design. Same as File ->Save All.
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.

Analyst Toolbar

The Analyst toolbar becomes active after a design has been compiled. The toolbar provides the following icons, by default:
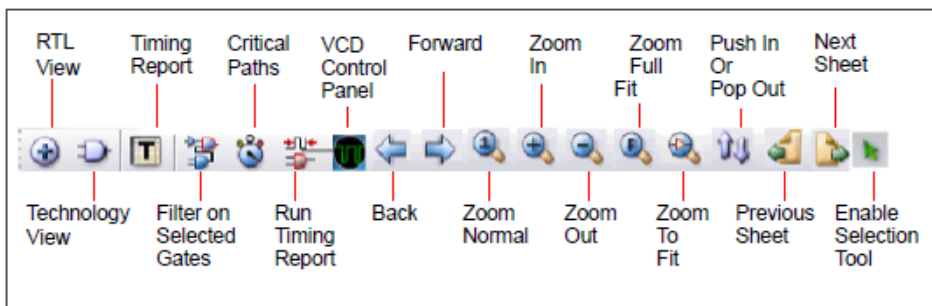
The following table describes the default Analyst icons. Each is equivalent to an HDL Analyst menu command – see HDL Analyst Menu, on page 272, for more information.

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.
Filter Schematic	Filters your entire design to show only the selected objects. The result is a *filtered* schematic.
	Same as HDL Analyst ->Filter Schematic.
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.
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 260.
	Only available for certain device technologies. (Not available in Synplify.)
	Same as Analysis ->Timing Analyst.
VCD Panel	When enabled, displays the VCS-Analyst Simulation Panel in the HDL Analyst Technology view.
⇐ Back	Goes backward in the history of displayed sheets of the current HDL Analyst view.
	Same as View ->Back.

Icon	Description
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	Zooms the view in or out. Buttons stay active until deselected.
Zoom Out	Same as View ->Zoom In or View ->Zoom Out.
Zoom Full	Zoom that reduces the active view to display the entire design.
	Same as View ->Full View.[b]
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.
Select Tool	Switches from zoom to the selection tool.

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.

Toggle FSM Table ——— Filter by outputs

Unfilter FSM

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 170, 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 85.) The Edit toolbar provides the following icons, by default:

Toggle Bookmark   Previous Bookmark

Next Bookmark   Clear All Bookmarks
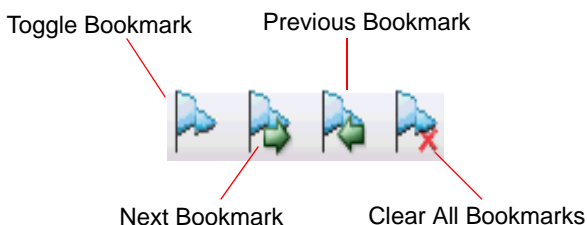
The following table describes the default Edit icons. Each is available in the Text Editor, and each is equivalent to an Edit menu command there – see Edit Menu Commands for the Text Editor, on page 160, for more information.

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.

Tools Toolbar

The Tools Toolbar opens supporting tools.

Icon	Description
Constraint Check	Checks the syntax and applicability of the timing constraints in the constraint 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 Tools, on page 493 of the User Guide.
Launch Identify Debugger	Launches the Synopsys Identify Debugger product. For more information, see Working with the Identify Tools, on page 493 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 221.
Launch SystemDesigner	Not applicable for Microsemi technologies.
VCS Simulator	Configures and launches the VCS simulator.

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 275).
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 273).
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 163.)
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 273).

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 273).
Ctrl-Enter (Return)	In the FSM Viewer, hides all but the selected state(s).
	Same as View->Filter->Selected (see View Menu, on page 170).
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 160).
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 160).
	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 171).
Ctrl-h	In the Text Editor, replaces text. Same as Edit->Replace (see Edit Menu Commands for the Text Editor, on page 160).
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 273).
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 273).

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 170).
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 273).
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 273).
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 170).
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 160). 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 172).
Ctrl-v	Pastes the last object copied or cut. Same as Edit ->Paste.

96   Synplify Pro for Microsemi Edition Reference Manual
January 2014

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 171).
	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 171).
	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 160).
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 279).
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 272).
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 272).
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 160).

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 272).
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 272).
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 171).
	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 160).
F4	In the Project view, adds a file to the project. Same as Project->Add Source File (see Build Project Command, on page 158).
	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 171).
F5	Displays the next source file error.
	Same as Run->Next Error/Warning (see Run Menu, on page 213).
F7	Compiles your design, without mapping it.
	Same as Run->Compile Only (see Run Menu, on page 213).
F8	Synthesizes (compiles and maps) your design.
	Same as Run->Synthesize (see Run Menu, on page 213).

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 213).
	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 171).
F11	Toggles zooming in.
	Same as View->Zoom In (see View Menu: RTL and Technology Views Commands, on page 171).
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 275.
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 272).
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 272).
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 272).
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).
Shift-F5	Displays the previous source file error.
	Same as Run->Previous Error/Warning (see Run Menu, on page 213).
Shift-F7	Checks source file syntax.
	Same as Run->Syntax Check (see Run Menu, on page 213).

Keyboard Shortcut	Description
Shift-F8	Checks synthesis. Same as Run->Synthesis Check (see Run Menu, on page 213).
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 213).
Shift-F11	Toggles zooming out. Same as View->Zoom Out (see View Menu, on page 170).
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 281.

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.
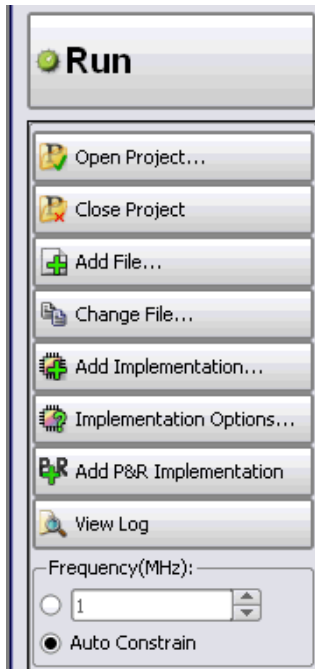
The following table describes the Project View buttons and options.

Button/Option	Action
Open Project...	Opens a new or existing project.
	Same as File->Open Project (see Open Project Command, on page 158).
Close Project	Closes the current project.
	Same as File->Close Project (see Run Menu, on page 213).
Add File...	Adds a source file to the project.
	Same as Project->Add Source File (see Build Project Command, on page 158).

Button/Option	Action
Change File...	Replaces one source file with another.
	Same as Project ->Change File (see Change File Command, on page 181).
Add Implementation	Creates a new implementation.
	Same as Project ->New Implementation.
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 191).
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 333 for a description of the dialog box, and Running P&R Automatically after Synthesis, on page 492 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 170).
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 287 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 76 and Optimizing State Machines, on page 347 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 78 and Running the FSM Compiler, on page 349 in the *User Guide*.
Resource Sharing	When enabled, makes the compiler use resource sharing techniques. This option does not affect resource sharing by the mapper.
	The option is the same as the Resource Sharing option on the Options panel of the Implementation Options dialog box. See Sharing Resources, on page 345 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, on page 37 for syntax details.
Run	Runs synthesis (compilation and mapping).
	Same as the Run->Synthesize command (see Run Menu, on page 213).

C H A P T E R 3

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.

- HDL Analyst Views and Commands, on page 106
- Schematic Objects and Their Display, on page 108
- Basic Operations on Schematic Objects, on page 117
- Multiple-sheet Schematics, on page 123
- Exploring Design Hierarchy, on page 126
- Filtering and Flattening Schematics, on page 133
- Timing Information and Critical Paths, on page 139

For additional information, see the following:

- Descriptions of the HDL Analyst commands in Chapter 4, *User Interface Commands*:
- Chapter 13, *Optimizing Processes for Productivity* 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 61 and Technology View, on page 62 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 106
- Accessing HDL Analyst Commands, on page 107

Filtered and Unfiltered Schematic Views

HDL Analyst views (RTL View, on page 61 and Technology View, on page 62) 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
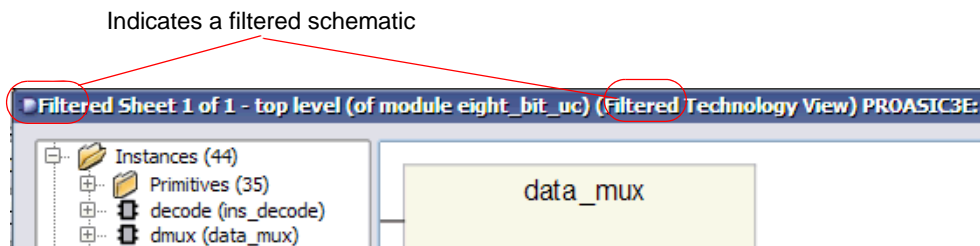
Filtering commands affect only the displayed schematic, not the under-lying design. See the following topics:

- For a detailed description of filtering, see Filtering and Flattening Schematics, on page 133.

- For procedures on using filtering, see Filtering Schematics, on page 251 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 272

- View Menu, on page 170

- RTL and Technology Views Popup Menus, on page 337

- Analyst Toolbar, on page 87

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 108
- Sheet Connectors, on page 109
- Primitive and Hierarchical Instances, on page 110
- Hidden Hierarchical Instances, on page 113
- Transparent and Opaque Display of Hierarchical Instances, on page 111
- Schematic Display, on page 113

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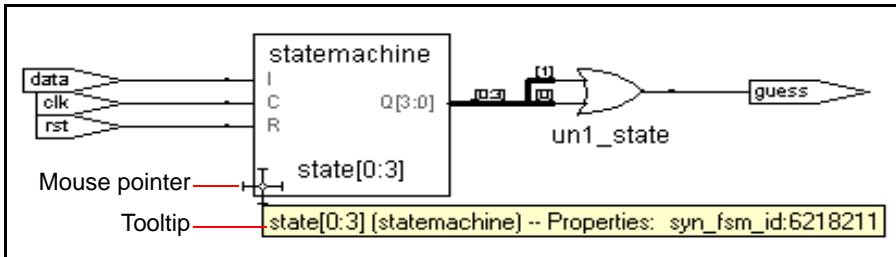
To disable tooltip display, select View -> Toolbars and disable the Show
Tooltips option. Do this if you want to reduce clutter.

See also

- Pin and Pin Name Display for Opaque Objects, on page 115

- HDL Analyst Options Command, on page 297

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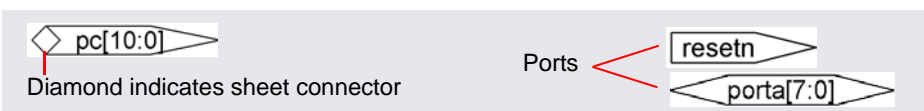
If you enable the Show Sheet Connector Index option in the (Options->HDL Analyst
Options), the empty diamond becomes a hexagon with a list of the connected
sheets. You go to a connecting sheet by right-clicking a sheet connector and
choosing the sheet number from the popup menu. The menu has as many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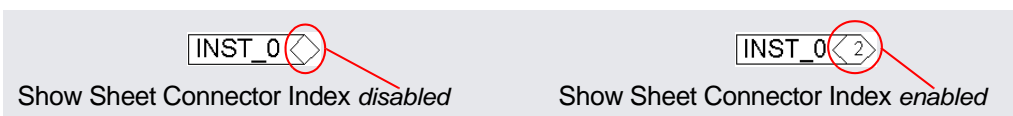
See also

- Multiple-sheet Schematics, on page 123
- HDL Analyst Options Command, on page 297
- RTL and Technology Views Popup Menus, on page 337

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 66). 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 111 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 113 for more information.

Transparent and Opaque Display of Hierarchical Instances

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

- Opaque instances are pale yellow boxes, and do not display their internal hierarchy. This is the default display.
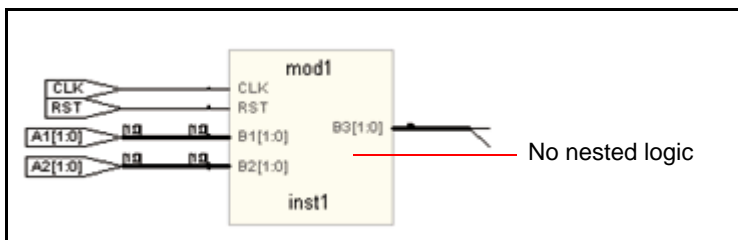
- Transparent instances display some or all their lower-level hierarchy nested inside a hollow box with a pale yellow border. Transparent instances are only displayed in filtered schematics, and are a result of certain commands. See Looking Inside Hierarchical Instances, on page 131 for information about commands that generate transparent instances.

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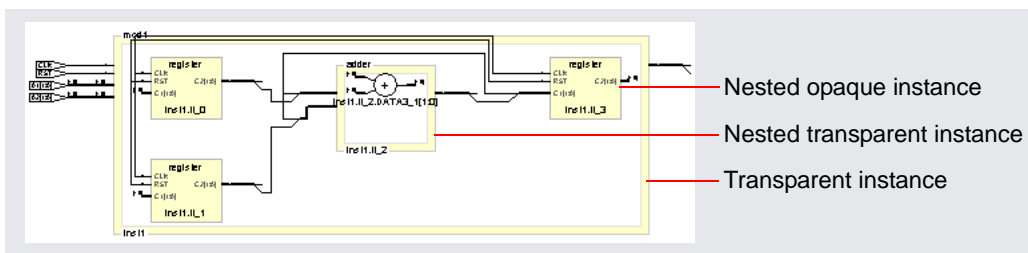
See also

- Looking Inside Hierarchical Instances, on page 131
- Multiple Sheets for Transparent Instance Details, on page 125
- Filtered and Unfiltered Schematic Views, on page 106

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 337). 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.
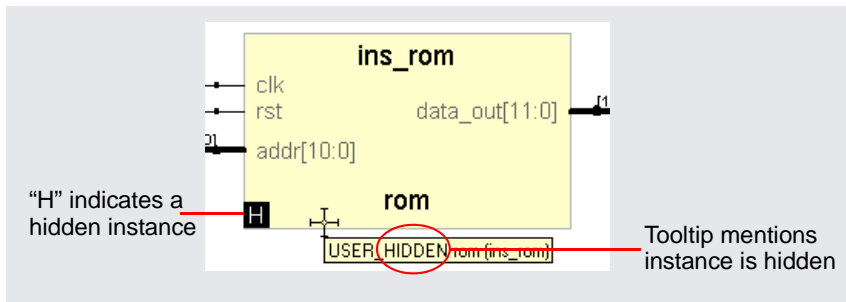
Schematic Display

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 297 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 115 and Pin and Pin Name Display for Transparent Objects, on page 115 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.
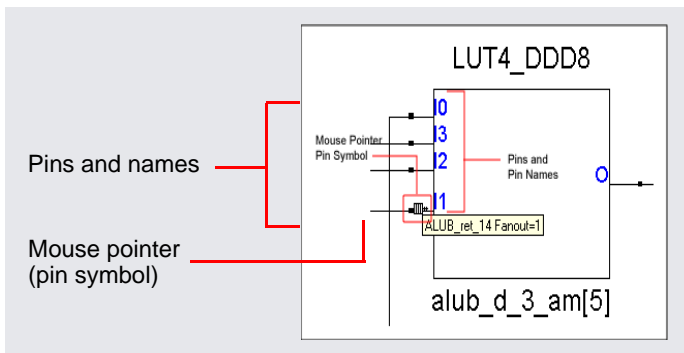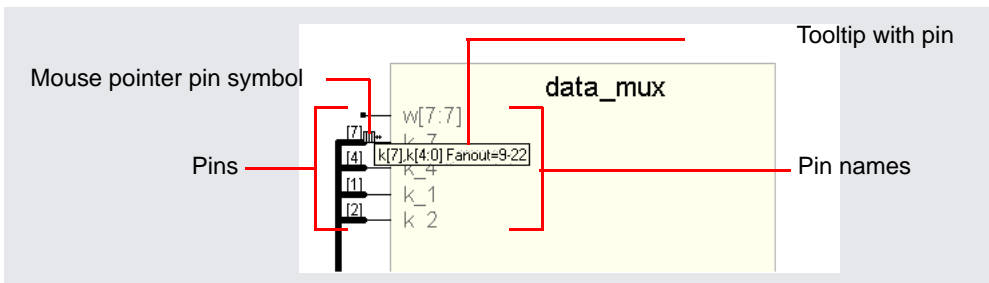
Pin and Pin Name Display for Transparent Objects

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.
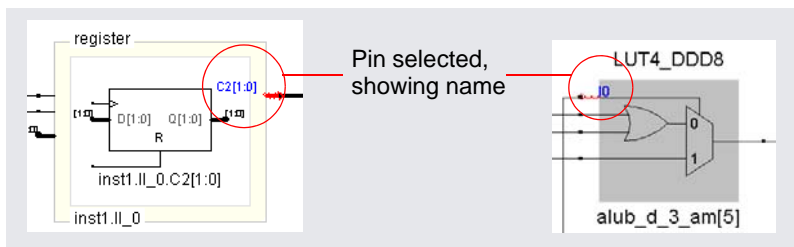
See also:

- HDL Analyst Options Command, on page 297
- Controlling the Amount of Logic on a Sheet, on page 123
- Analyzing Timing in Schematic Views, on page 270 in the *User Guide*

Basic Operations on Schematic Objects

Basic operations on schematic objects include the following:

- Finding Schematic Objects, on page 117

- Selecting and Unselecting Schematic Objects, on page 119

- Crossprobing Objects, on page 120

- Dragging and Dropping Objects, on page 122

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

- Filtering and Flattening Schematics, on page 133

- Timing Information and Critical Paths, on page 139

- Multiple-sheet Schematics, on page 123

- Exploring Design Hierarchy, on page 126

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 226 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 226 in the *User Guide* for details.

- Edit -> Find command

  The Edit -> Find command is described in Find Command (HDL Analyst), on page 163. 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 228 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 133 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 211 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 117

- HDL Analyst Menu, on page 272

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 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 109) mark logic connections from one sheet to the next.

For more information, see

- Controlling the Amount of Logic on a Sheet, on page 123
- Navigating Among Schematic Sheets, on page 123
- Multiple Sheets for Transparent Instance Details, on page 125

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 297
- Setting Schematic View Preferences, on page 215 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
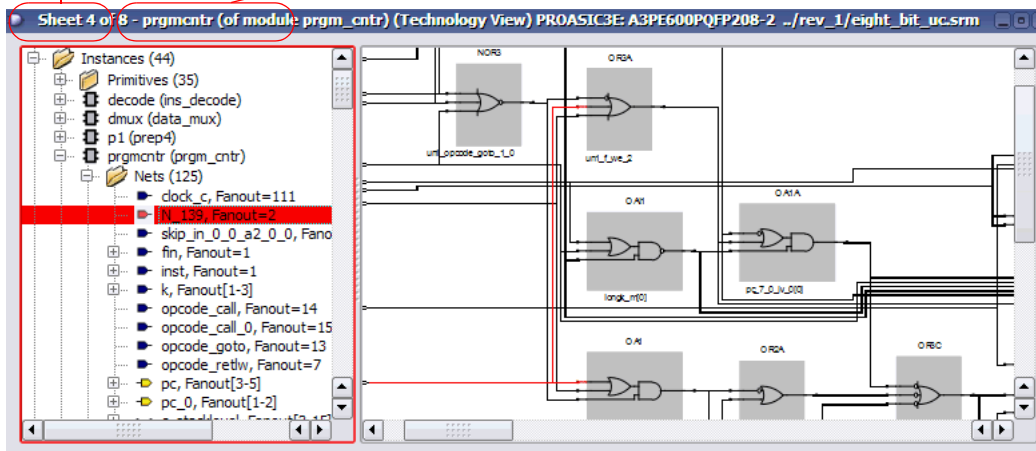
Navigating Among Sheets

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 213 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 133
- View Menu: RTL and Technology Views Commands, on page 171
- Pushing and Popping Hierarchical Levels, on page 126

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 123.

- 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 123
- View Menu: RTL and Technology Views Commands, on page 171

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

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 129 and Looking Inside Hierarchical Instances, on page 131 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 67, 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 127 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.
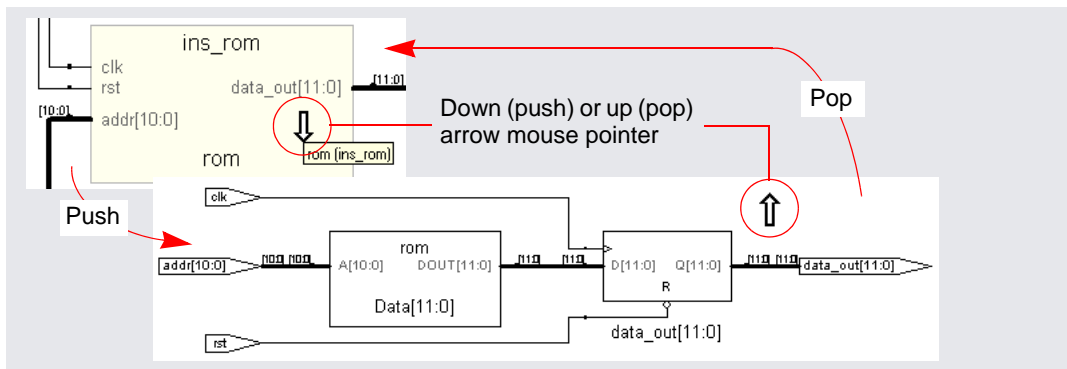
Once Push/Pop mode is enabled, you push or pop as follows:

- 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.ll_3 descends two levels.
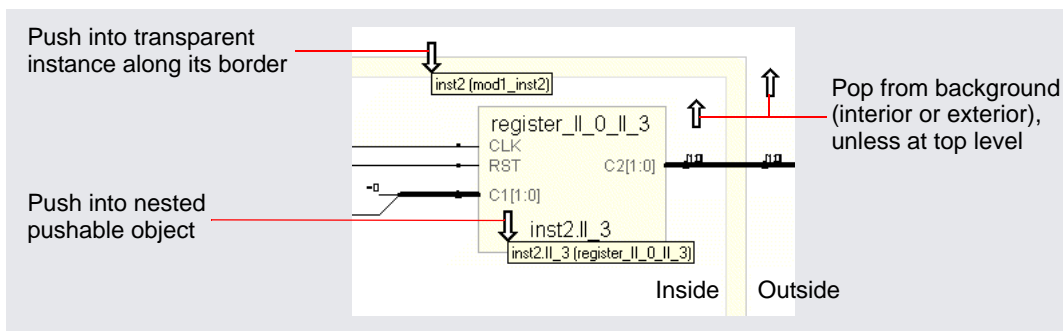
The following arrow mouse pointers indicate status in Push/Pop mode. For other indicators, see Pushable Schematic Objects, on page 126.

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 113
- Transparent and Opaque Display of Hierarchical Instances, on page 111
- Using Mouse Strokes, on page 79
- Navigating With a Hierarchy Browser, on page 129

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 126 and Looking Inside Hierarchical Instances, on page 131 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.
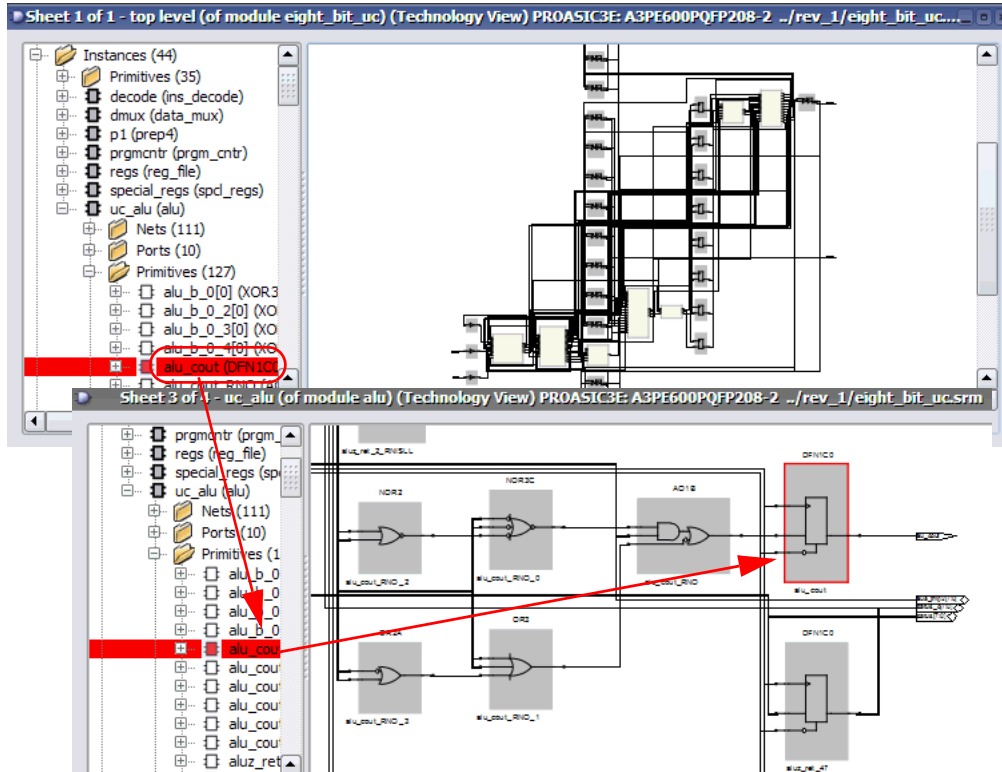
Explore the browser hierarchy by expanding or collapsing the categories in the browser. You can also use the arrow keys (left, right, up, down) to move up and down the hierarchy and select objects. To select more than one object, press Ctrl and select the objects in the browser. To select a range of 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 120

- Pushing and Popping Hierarchical Levels, on page 126

- Hierarchy Browser Popup Menu Commands, on page 337

Looking Inside Hierarchical Instances

An alternative method of viewing design hierarchy is to examine transparent hierarchical instances (see and 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
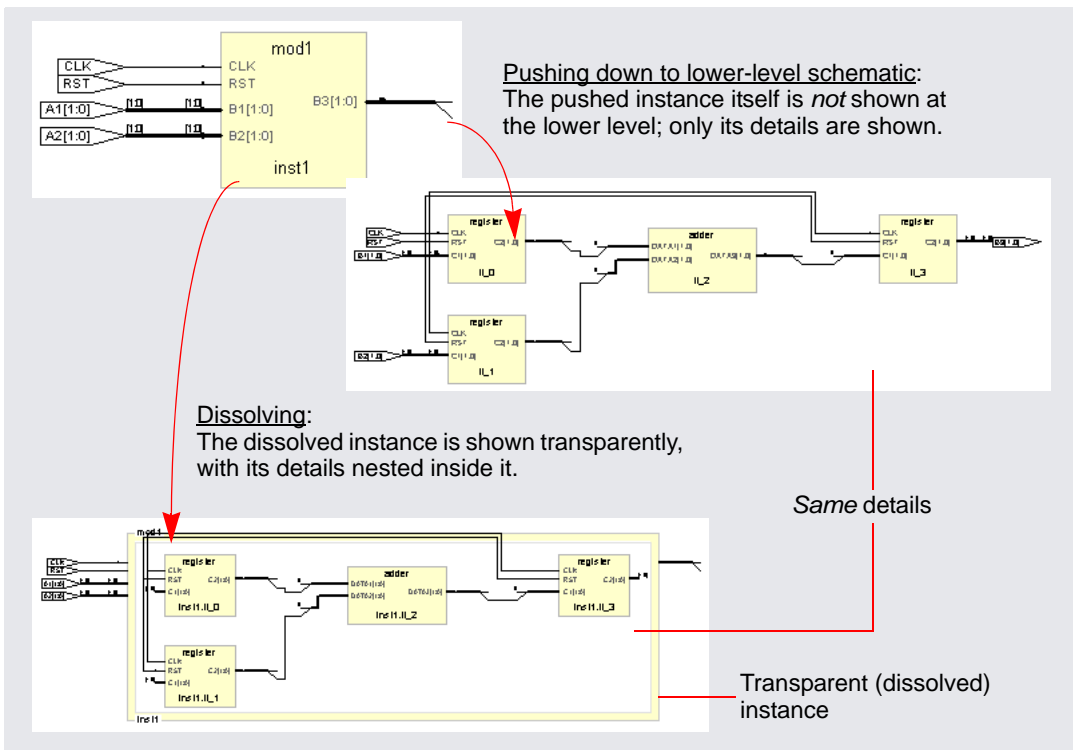
You cannot control the display of transparent instances directly. However, you can perform the following operations, which result in the display of transparent instances:

- 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 133 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 126

- Navigating With a Hierarchy Browser, on page 129

- HDL Analyst Command, on page 273

- Transparent and Opaque Display of Hierarchical Instances, on page 111

- Hidden Hierarchical Instances, on page 113

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 133
- Combined Filtering Operations, on page 134
- Returning to The Unfiltered Schematic, on page 135
- Commands That Flatten Schematics, on page 135
- Selective Flattening, on page 136
- Filtering Compared to Flattening, on page 137

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 251* in the *User Guide.*

See also:

- Filtered and Unfiltered Schematic Views, on page 106

- HDL Analyst Menu, on page 272 and RTL and Technology Views Popup Menus, on page 337

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 275 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 273 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 273 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 279 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 117 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 337
- View Menu: RTL and Technology Views Commands, on page 171

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.	

Command	Unfiltered Schematic	Filtered Schematic
Technology-> Flattened View	Creates a new, unfiltered Technology schematic of the entire design, flattened to the level of technology cells.	
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 136).

See also:

- Filtering Compared to Flattening, on page 137
- Selective Flattening, on page 136

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 337 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 337

- Selective Flattening, on page 136

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 139
- Critical Paths and the Slack Margin Parameter, on page 140
- Examining Critical Path Schematics, on page 141

See the following for more information about timing and result analysis:

- Watch Window, on page 52
- Log File, on page 257
- Chapter 13, *Optimizing Processes for Productivity* 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 197). 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 260

- Timing Reports, on page 263

- Log File, on page 257

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 272
- HDL Analyst Command, on page 273
- Handling Negative Slack, on page 276 of the *User Guide*
- Analyzing Timing in Schematic Views, on page 270 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 135
- Filtering and Flattening Schematics, on page 133

CHAPTER 4

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 147 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 145	Overview of constraint files
Timing Constraints, on page 147	Overview of timing constraint definitions and FDC file generation.
SCOPE Constraints Editor, on page 163	Information about automatic generation of timing and design constraints.
Chapter 6, *Constraint Syntax*	Timing constraint syntax
Chapter 6, *Constraint Syntax*	Design 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 147. The figure also indicates the
specific processes controlled by attributes and directives.
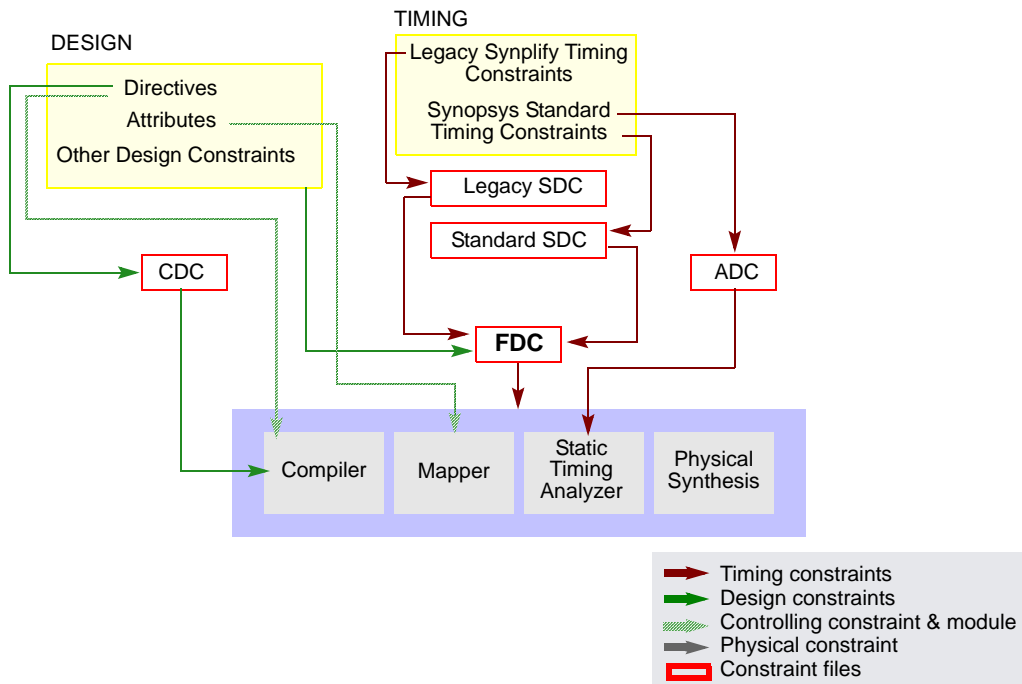
The table is a summary of the various kinds of constraint files.

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

The synthesis tools have supported different timing formats in the past, and this section describes some of the details of standardization:

Legacy SDC and Synopsys Standard SDC

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. This 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, which only contained timing constraints. Non-timing constraints were 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-2012.09 release, Synopsys standard timing constraint format has replaced the legacy-style constraint format, and a new FDC (FPGA design constraint) file consolidates both timing and design formats. As a result of these updates, there are some changes in the use model:

- Timing constraints in the legacy format are converted and included in an FDC file, which includes both timing and non-timing constraints. The file uses the Synopsys standard syntax for timing constraints (create_clock, set_multicycle_path...). The syntax for non-timing design constraints is 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.

FDC File Generation

The following figure is a simplified summary of constraint-file handling and the generation of fdc.

It is not required that you convert Synopsys standard sdc constraints as the figure implies, because they are already in the correct format. You could have a design with mixed constraints, with separate Synopsys standard sdc and fdc files. The disadvantage to keeping them in the standard sdc format is that you cannot view or edit the constraints through the SCOPE interface.
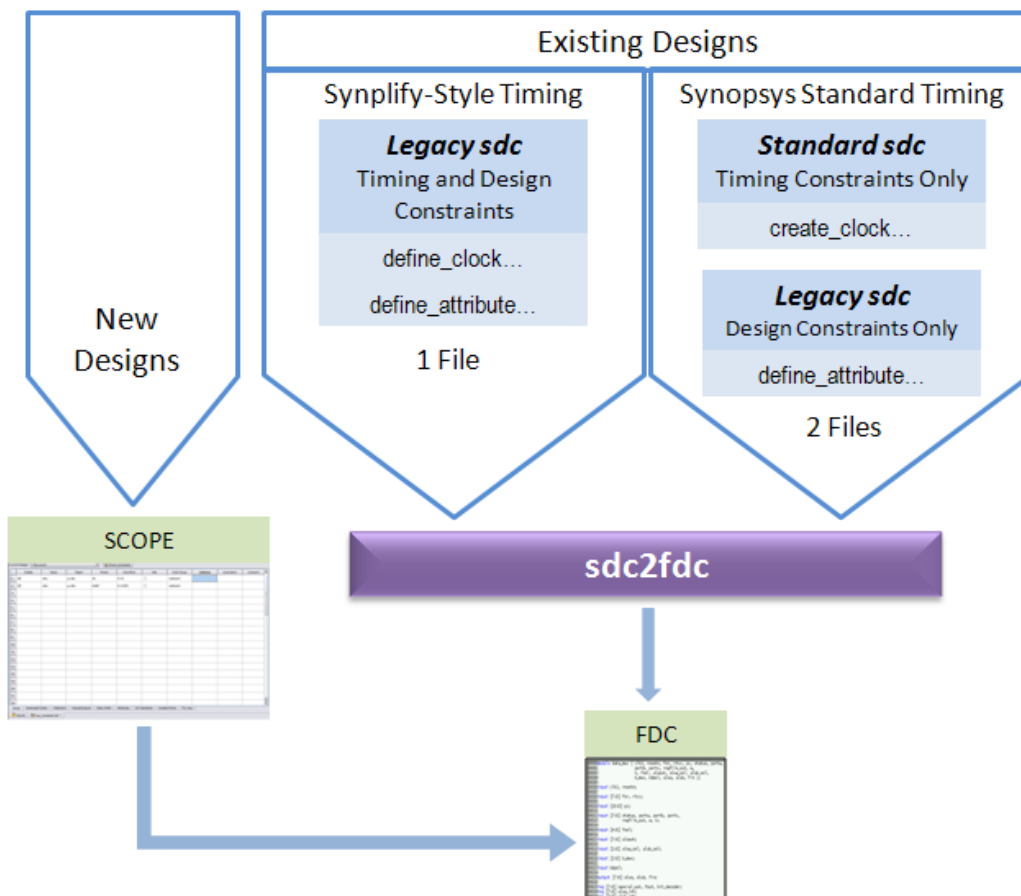
Timing Constraint Precedence in Mixed Constraint Designs

Your design could include timing constraints in a Synopsys standard sdc file and others in an fdc file. With mixed timing constraints in the same design, the following order of precedence applies:

- The tool reads the file order listed in the project file and any conflicting constraint overwrites a previous constraint.

With the legacy timing constraints, it is strongly recommended that you convert them to the fdc format. However, even if you retain the old format in an existing design, they must be used alone and cannot be mixed in the same design as fdc or Synopsys standard timing sdc constraints. Specifically, do not specify timing constraints using mixed formats. For example, do not define clocks with define_clock and create_clock together in the same constraint file or multiple SDC/FDC files.

For the list of FPGA timing constraints (FDC) and their syntax, see FPGA Timing Constraints, on page 212.

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 152 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 150 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 5, *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 188. 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 Using Auto Constraints, on page 287 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 206 to determine the constraint that takes precedence.

See Also

To specify the correct syntax for the timing and design commands, see:

- Chapter 6, *Constraint Syntax*
- *Attribute Reference Manual*

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 150l) 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 210 or Design Constraints, on page 242.

See Also

To use the current SCOPE editor, see:

- Chapter 5, *SCOPE Constraints Editor*
- Chapter 5, *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, on page 64

In addition to the information above, refer to Converting SDC to FDC, on page 150 for information about using this utility.

sdc2fdc Conversion

The sdc2fdc Tcl shell command translates legacy FPGA timing constraints to Synopsys FPGA timing constraints. This command scans the input SDC files and attempts to convert constraints for the implementation.

For details, see the following:

- Troubleshooting Conversion Error Messages, on page 154
- sdc2fdc FPGA Design Constraint (FDC) File, on page 156
- sdc2fdc, on page 64 in the *Command Reference* manual (syntax)

Troubleshooting Conversion Error Messages

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 to this effect displays in the Tcl shell window.

sdc2fdc FPGA Design Constraint (FDC) File

The FDC constraint file generated after running sdc2fdc contains translated legacy FPGA timing constraints (SDC), which are now in the FDC format. This file is divided into two sections:

First	Contains the following: • Valid FPGA design constraints (e.g. define_scope_collection and define_attribute) • Legacy timing constraints that were not translated because they were specified with -disable.
Second	Contains the legacy timing constraints that were translated.

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}]
```

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 272 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 272).
_cck_fdc_rpt	Lists the wildcard expansion results of running the constraint checker for collections with the get_* and all_* object query commands using the check_fdc_query Tcl command. See check_fdc_query, on page 28 for more information.

File	Description
`_scck.rpt`	Lists the results of running the constraint checker for collections with the get_* and all_* object query commands.
`.ta`	Reports timing analysis results (see Generating Custom Timing Reports with STA, on page 277).
`.srr` or `.htm`	Reports post-synthesis timing results as part of the text or HTML log file (see Timing Reports, on page 263 and Log File, on page 257).

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 140 and find, on page 98.
Collections	define_collection, c_union…	Create, copy, evaluate, traverse, and filter collections. See Using Collections, on page 140 and synhooks File Syntax, on page 668 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 287 in the *User Guide.*

CHAPTER 5

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 144 for definitions of synthesis constraints.

Topics in this section include:

- SCOPE User Interface, on page 164

- SCOPE Tabs, on page 165

- Industry I/O Standards, on page 190

- Delay Path Timing Exceptions, on page 193

- Specifying From, To, and Through Points, on page 199

- Conflict Resolution for Timing Exceptions, on page 206

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.
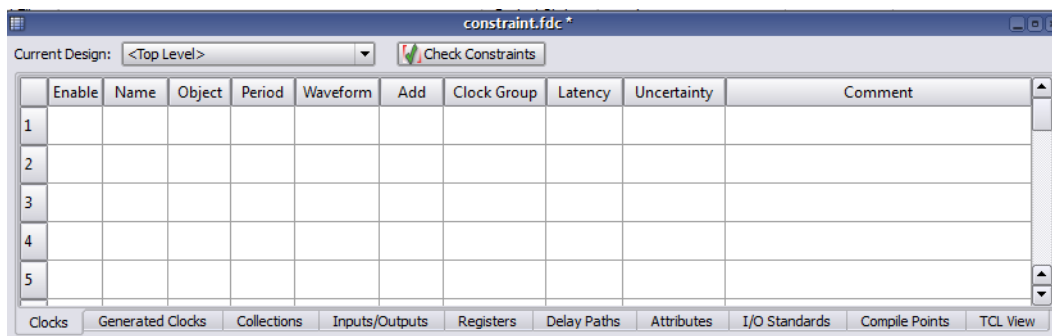
From this editor, you specify timing constraints for clocks, ports, and nets as well as design constraints such as attributes, collections, and compile points. However, you cannot set black-box constraints from the SCOPE window.

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 Using the SCOPE Editor, on page 110 in the *User Guide*.

SCOPE Tabs

Here is a summary of the constraints created through the SCOPE editor:

SCOPE Panel	See...
Clocks	Clocks, on page 165
Generated Clocks	Generated Clocks, on page 170
Collections	Collections, on page 173
Inputs/Outputs	Inputs/Outputs, on page 175
Registers	Registers, on page 178
Delay Paths	Delay Paths, on page 179
Attributes	Attributes, on page 182
I/O Standards	I/O Standards, on page 183
Compile Points	Compile Points, on page 185
TCL View	TCL View, on page 188

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 Naming Rule Syntax Commands, on page 241 for details.

Clocks

You use the Clocks panel of the SCOPE spreadsheet to define a signal as a clock.

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 Delay	Specifies whether to add this delay 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 clocks with the same source must have different names.

Synplify Pro for Microsemi Edition Reference Manual
166   January 2014

Field	Description
Clock Group	Assigns clocks to asynchronous clock groups. 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 167.
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

Clock grouping is associative; two clocks can be asynchronous to each other but both can be synchronous with a third clock.
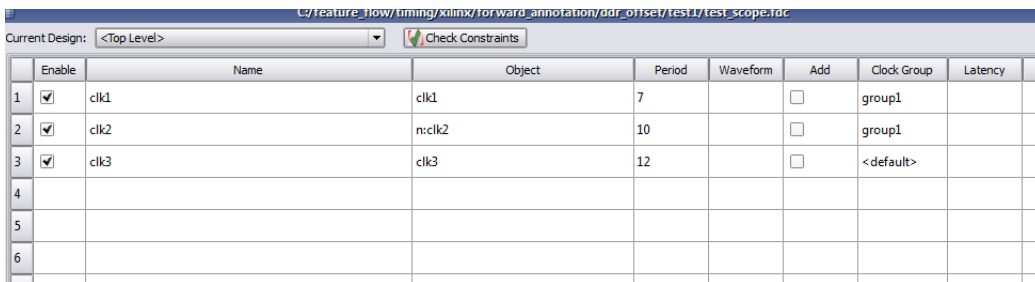
The SCOPE GUI prompts you for a clock group for each clock that you define. By default, the tool assigns all clocks to the default clock group. When you add a name that differs from the default clock group name, the clock is assigned its own clock group and is asynchronous to the default clock group 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. You want clk1 and clk2 to be in the same clock group—synchronous to each other but asynchronous to clk3. You can do this by adding a name in the Clock Group column, as shown below:

	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									

Current Design: <Top Level>   Check Constraints

C:/feature_flow/timing/xilinx/forward_annotation/ddr_offset/test1/test_scope.fdc

This definition assigns clk1 and clk2 to clock group group1, synchronous to each other and asynchronous to clk3. This is the equivalent command that appears in the text editor window:

```
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. 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}
```

You want to define clka and clkb as asynchronous to each other and clka and clkb as 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 table shows legacy clock definitions and their translated FDC equivalents:

Legacy Definition	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

FDC Definition	###==== 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 the following sections:

- create_clock, on page 213
- set_clock_groups, on page 219
- set_clock_latency, on page 222
- set_clock_uncertainty, on page 225

For information about other SCOPE panels, see SCOPE Tabs, on page 165.

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 215.

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 pin, which is either a master clock source pin or a fanout pin of the master clock driving the generated clock definition pin. The clock waveform at the master pin is used for deriving the generated clock waveform.
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.

Field	Description
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.
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 .

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 SCOPE Collections, on page 141 of the *User Guide*.

	Enable	Name	Command	Comment
1				
2				
3				

Collections

Field	Description
Name	Enter the collection name.
Command	Select a collection creation command from the drop-down menu. See Collection Commands, on page 117 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
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 165.

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.

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.

Field	Description
Clock Fall	Specifies that the delay relative to the falling edge of the clock. For examples, see Input Delays, on page 176 and Output Delays, on page 176.
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 176 and Output Delays, on page 176.
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.

Priority of Multiple I/O Constraints

You can specify multiple input and output delays 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
```

See Also

For equivalent Tcl syntax, see:

- set_input_delay, on page 229

- set_output_delay, on page 237

For information about all SCOPE panels, see SCOPE Tabs, on page 165.

Registers

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 set_reg_input_delay, on page 239, and set_reg_output_delay, on page 240 for the equivalent Tcl commands.
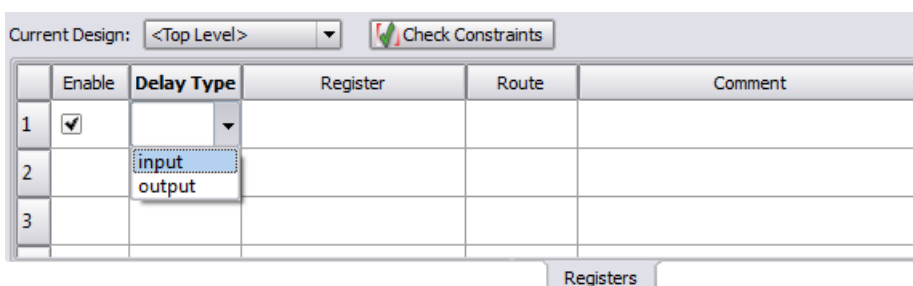
The Registers SCOPE panel includes the following fields:

Field	Description
Enabled	(Required) Turn this on to enable the constraint.
Delay Type	(Required) Specifies whether the delay is an input or output delay.
Register	(Required) Specifies the name of the register. If you have initialized a compiled design, you can choose from the pull-down list.
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.
Comment	Lets you enter comments that are included in the constraints file.

Delay Paths

Use the Delay Paths panel to define the timing exceptions.

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 • Naming Objects, on page 456
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 • Naming Objects, on page 456
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 • Naming Objects, on page 456
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 234

  - set_false_path, on page 227

  - set_max_delay, on page 231

  - reset_path, on page 217

- For more information on timing exception constraints and how the tool resolves conflicts, see:

  - Delay Path Timing Exceptions, on page 193

  - Conflict Resolution for Timing Exceptions, on page 206

- For information about all SCOPE panels, see SCOPE Tabs, on page 165.

Attributes

You can assign attributes directly in the editor.

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 4.

- For information about all SCOPE panels, see .

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							

I/O Standards

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 190 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

- The Tcl equivalent of this constraint is define_io_standard.

- For information about all SCOPE panels, see SCOPE Tabs, on page 165.

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.

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 must specify a view module, with a v: prefix 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 365 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.
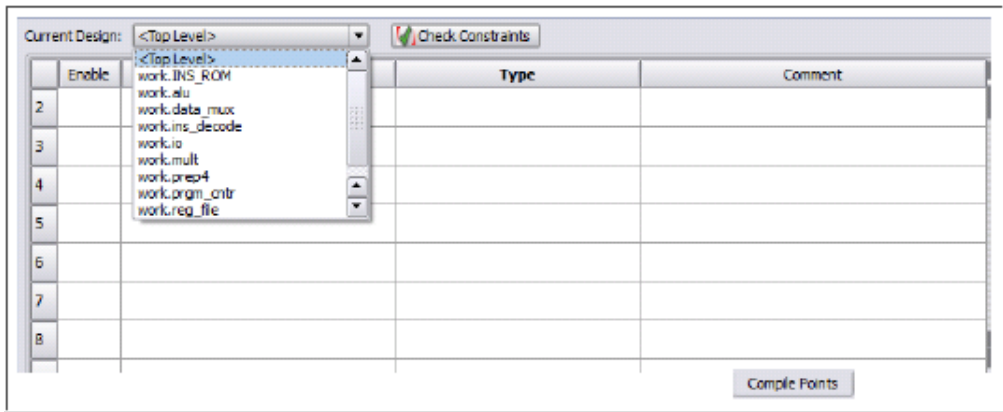
See Also

- 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 378 in the *User Guide*.

- For information about all SCOPE panels, see SCOPE Tabs, on page 165.

TCL View

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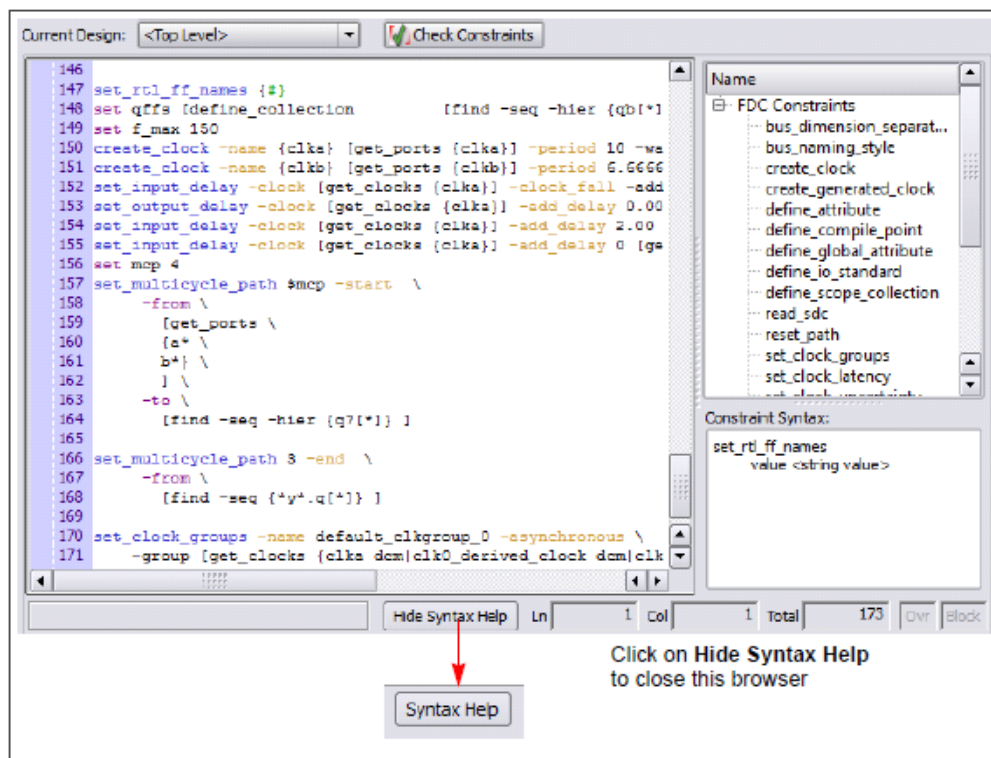
This text editor provides the following capabilities:

- Uses dynamic keyword expansion and tool tips for commands that
    - Automatically completes the command from a popup list
    - Displays complete command syntax as a tool tip
    - Displays parameter options for the command from a popup list
    - Includes a keyword command syntax help
- Checks command syntax and uses color indicators that
    - 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 120.

See Also

- For Tcl timing constraint syntax, see FPGA Timing Constraints, on page 212.

- For Tcl design constraint syntax, see Chapter 6, *Constraint Syntax*.

- You can also use the SCOPE editor to set attributes. See How Attributes and Directives are Specified, on page 4 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 190.

For vendor-specific I/O standards, see Microsemi I/O Standards, on page 677.

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 193
- False Paths, on page 196

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 set_multicycle_path, on page 234. This section describes the following:

- Multi-cycle Path with Different Start and End Clocks, on page 193
- Multicycle Path Examples, on page 194

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									

Delay Paths

Multicycle Path Examples

Multicycle Path Example 1

If you apply a multicycle 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.
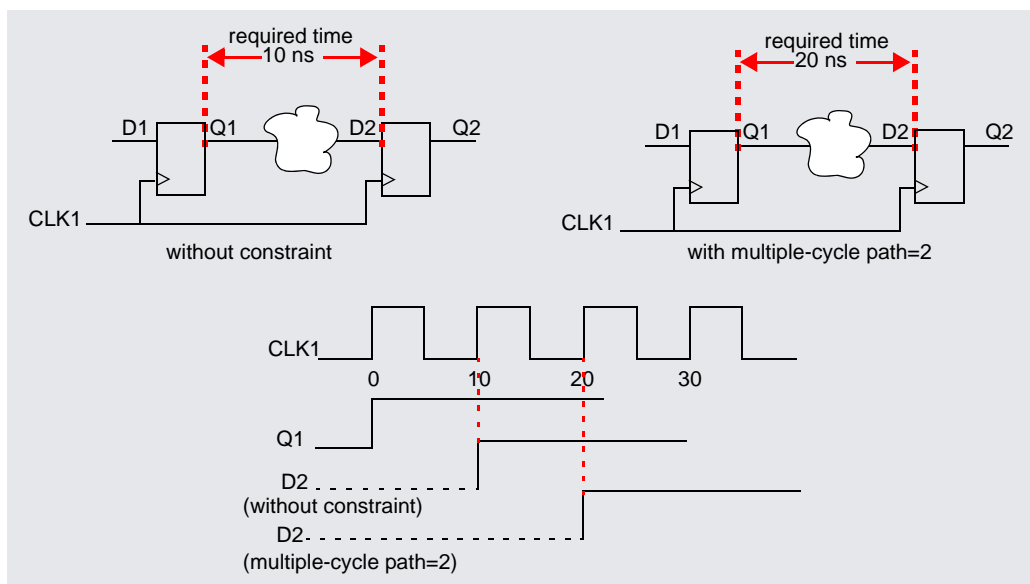
Multicycle Path Example 2

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).
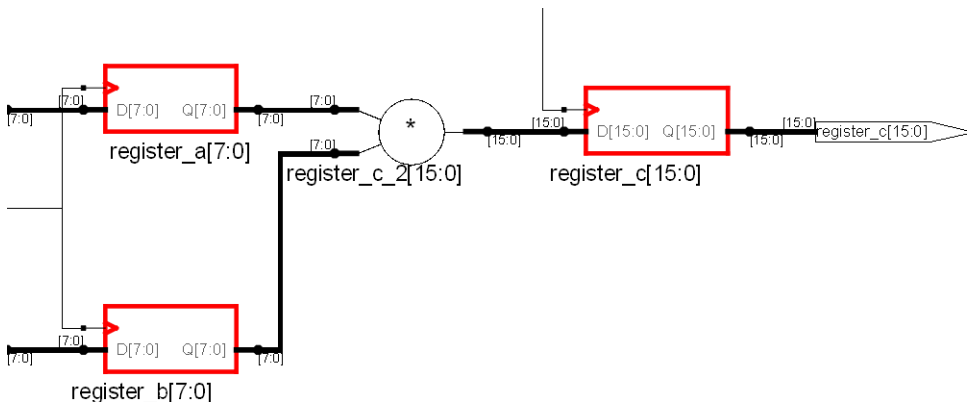
The design frequency goal is 50 MHz (20 ns) and the multiply function takes 35 ns, but it is given 2 clock cycles. After optimization, this 35 ns path is normally reported as a timing violation because it is more than the 20 ns clock-cycle timing goal. To avoid reporting the paths as timing violations, 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 set_false_path, on page 227.

This section describes the following:

- Types of False Paths, on page 196
- Priority of False Path Constraints, on page 197

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.
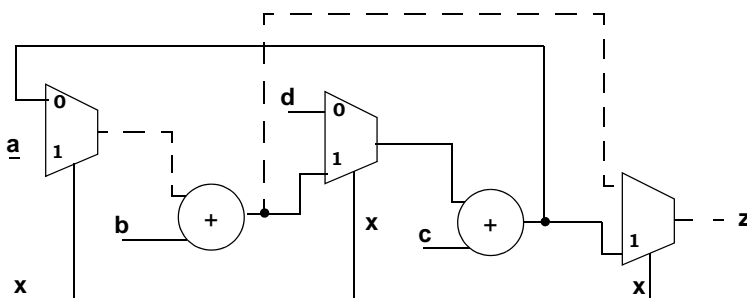
- Code-introduced false paths

  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 GUI, 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 GUI).

    - 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 multicycle paths, false paths, and max delay paths constraints.

-

-

-

-

-

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 GUI 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 syn_black_box, on page 37) 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 204 for more information.	Clocks. See Clocks as From/To Points, on page 204 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 GUI (see Product of Sums Interface, on page 202). 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_max_delay
-through {regs_mem[2], prgcntr.pc[7], dmux.alub[0]} 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_max_delay
-through regs_mem[2]
-through prgcntr.pc[7]
-through dmux.alub[0] 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 {A₁ A₂...Aₙ} -through {B₁ B₂ B₃}
```

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 GUI to format -through points for nets with multicycle path, false path, and max delay path constraints in the Product of Sums (POS) interface of the SCOPE editor. 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 126 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
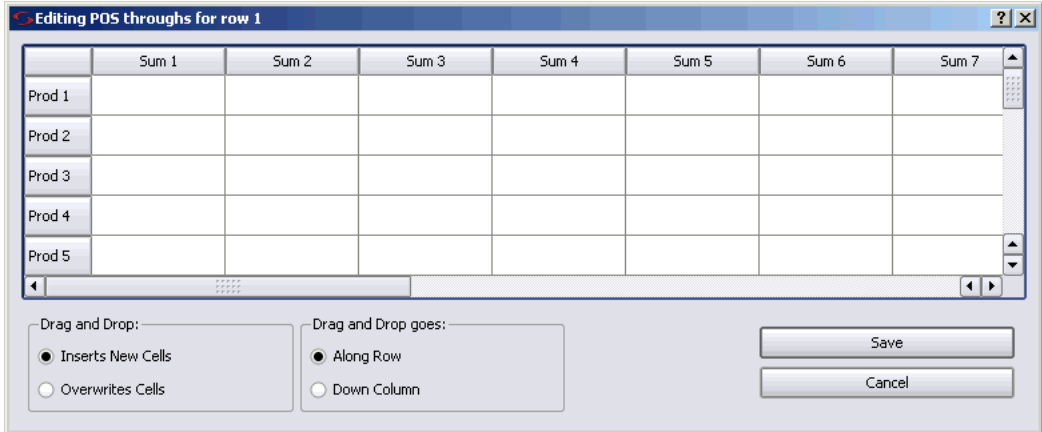
Field	Description
Prod 1, 2, etc.	Type the first net name in a cell in a Prod row, or drag the net from a HDL Analyst view into the cell. Repeat this step along the same row, adding other nets in the Sum columns. The nets in each row form an OR list.
Sum 1, 2, etc.	Type the first net name in the first 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:

set_*timing_exception* **-from** | **-to { c:***clock_name* [**:***edge*] **}**

where

- *timing_exception* is one of the following constraint types: multicyclepath, false-path, or maxdelay

- **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_max_delay -to {c:clk1:f} 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_max_delay -from {c:clk1:r} -through (n:MYINST.mybus2[9]) .2
```

Conflict Resolution for Timing Exceptions

The term *timing exceptions* refers to the false path, max 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 207.
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 208.
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 209.
4	Identical timing constraints, except constraint values differ.	Tightest, or most constricting constraint.	Identical Constraints with Different Values, on page 209.

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 177 and Priority of False Path Constraints, on page 197.

Conflicting Timing Exceptions

The first (and highest) level of resolution occurs when timing exceptions—false paths, max path delay, or multicycle 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:
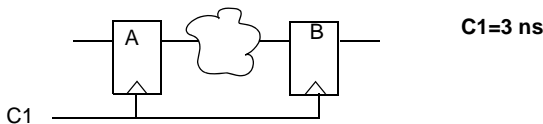

```
set_false_path -from {c:C1:r}
set_max_delay -from {i:A} -to {i:B} 10
set_multicycle_path -from {i:A} -to {i:B} 2
```

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.
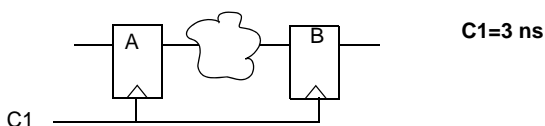

```
set_multicycle_path -from {i:A} 3
set_multicycle_path -to {i:B} 2
```

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 197 and Priority of Multiple I/O Constraints, on page 177.

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 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												

Clocks | Clock to Clock | Collections | Inputs/Outputs | Registers | Delay Paths | Attributes | I/O Standards | Compile Points | Other
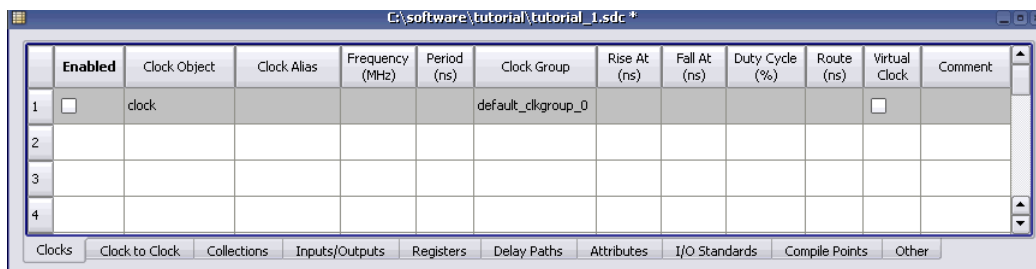
The details of the legacy SCOPE interface and constraint syntax are no longer documented here. Refer to the Solvnet article on legacy constraints for details.

CHAPTER 6

Constraint Syntax

The following describe Tcl equivalents for the timing and design constraints you specify in the SCOPE editor or in a constraint file.

- FPGA Timing Constraints, on page 212
- Design Constraints, on page 242

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 110 in the *User Guide*.

For specific information on individual constraint options and arguments, see the *Synthesis Commands* PDF document at https://solvnet.synopsys.com/dow_retrieve/G-2012.06/manpages/ni/syn2.pdf.

For information on which options and arguments are supported, see the *SDC Standard for FPGA Synthesis* document on SolvNet.

For general information on the Design Constraints Format, see the *Using the Synopsys Design Constraints Format Application Note* on SolvNet.

The remainder of this section describes the constraint file syntax for the following FPGA timing constraints in the FPGA synthesis tools.

- 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

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 pin, which is either a master clock source pin or a fanout pin of the master clock driving the generated clock definition pin. The clock waveform at the master pin is used for deriving the generated clock waveform.

-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}]
```

This set_clock_groups constraint specifies that clock clk1, clk2, and clk3 are synchronous to one another, but asynchronous to all other clocks in the design.

```
set_clock_groups -asynchronous -group {clk1 clk2 clk3}
```

This set_clock_groups constraint specifies that clk1 and clk2 are synchronous to one another, but are asynchronous to clk3 and clk4. However, clk3 and clk4 are synchronous to each other.

```
set_clock_groups -asynchronous -group {clk1 clk2}
    -group {clk3 clk4}
```

This set_clock_groups constraint specifies that clk4 is asynchronous to all other clocks in the design.

```
set_clock_groups -asynchronous -group {clk4}
```

Note: The -group option contains a list of clocks that are separated by a space. Do not add a comma after the clock name.

For example, if you specify the following:

```
set_clock_groups -asynchronous -group {clk1, clk2}
```

The tool generates a warning that clk1 cannot be found, because the comma is treated as part of the clock name. This is true for all constraints that contain lists.

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***}**
>   [**-disable**]

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. For information about the equivalent SCOPE spreadsheet interface, see Registers, on page 178.

Use this constraint instead of the legacy constraint, define_reg_input_delay.

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.

Description

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. For information about the equivalent SCOPE spreadsheet interface, see .

Use this constraint instead of the legacy constraint, define_reg_output_delay.

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*

Design Constraints

This section describes the constraint file syntax for the following non-timing design constraints:

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 185. (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 365 for more information.

Refer to Methods for Creating Constraints, on page 152 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 Methods for Creating Constraints, on page 152 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 183 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 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 280 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 sdc2fdc, on page 64.
`.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 297.
		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 165.
.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 251, 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 45 of the *User Guide.*
.vhd	Source files (VHDL)	Design source files in VHDL format. See VHDL, on page 250 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 45 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 251, 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 45 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 15, *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 Synplicity 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 252 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 272 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 198 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 197).

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 Tab, on page 42.
`.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, on page 98 and Using the Tcl Find Command to Define Collections, on page 134 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 183 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.
`.srd`	Intermediate mapping files	Used to save mapping information between synthesis runs. You do not need to use these files.
`.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.

Extension	File	Description
`.srr`	Synthesis log file	Provides information on the synthesis run, as well as area and timing reports. See Log File, on page 257, 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 170.
`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 170.
`.ta`	Customized Timing Report	Contains the custom timing information that you specify through Analysis->Timing Analyst. See Analysis Menu, on page 260, for more information.
`_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 260 for more information.

Extension	File	Description
`.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 134 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 197.

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 197.
		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 101) 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 Messages tab in the Tcl window.

For further details about different sections 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 259
Buffers added to clocks in certain supported technologies.	Timing Reports, on page 263
Buffers added to nets.	Net Buffering Report, on page 260
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 263
Compile point remapping.	Compile Point Information, on page 260
Resources used by synthesis mapping.	Resource Usage Report, on page 261
Design changes made as a result of retiming.	Retiming Report, on page 261

Compiler Report

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

- Project information: 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 262 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, RTL optimizations, and informational or warning messages. See Errors, Warnings, Notes, and Messages, on page 262 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 378 of 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 263, 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 260). Their format is the same as the timing report.

Resource Usage Report

A resource usage report is added to the log file each time you compile or synthesize. The format of the 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

See Checking Resource Usage, on page 187 in the *User Guide* for a brief procedure on using the report to check for overutilization.

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 OF 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 270 for details on this report.

The timing reports in the srr/htm and ta files have the following sections:

- Timing Report Header, on page 264
- Performance Summary, on page 264
- Clock Relationships, on page 266
- Interface Information, on page 268
- Detailed Clock Report, on page 268
- Asynchronous Clock Report, on page 270

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.

```
00055
00056 ##### START TIMING REPORT #####
00057 # Timing Report written on Fri Sep 06 13:38:15 2002
00058 #
00059
00060
00061 Top view:          mod2
00062 Paths requested:       5
00063 Constraint File(s):
00064 @N| This timing report estimates place and route data. Please look
00065 @N| Clock constraints cover all FF-to-FF, FF-to-output, input-to-FF
00066
```

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 , 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). 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.

Performance Summary Column	Description
Requested/Estimated Frequency	Target frequency goal /estimated value after synthesis. See Cross-Clock Path Timing Analysis, on page 267 for information on how cross-clock path slack is reported.
Requested/Estimated Period	Target clock period/estimated value after synthesis.
Slack	Difference between estimated and requested period. See Cross-Clock Path Timing Analysis, on page 267 for information on how cross-clock path slack is reported.
Clock Type	The type of clock: inferred, declared, derived or system. For more information, see Clock Types, on page 265.
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 267 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 167. For tips on using clock groups, see Defining Other Clock Requirements, on page 164 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    -
=====================================================================================================
 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 (
```

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: define_input_delay, or default value of 0.
	Output ports: path delay (including clock-to-out delay of source register).
	For purely combinational paths, the propagation delay is calculated from the driving input port.
Required Time	Input ports: clock period – (path delay + setup time of receiving register + define_reg_input_delay value).
	Output ports: clock period – define_output_delay. Default value of define_output_delay is 0.
Slack	Required Time – Arrival Time

Detailed Clock Report

Each clock reported in the performance summary also has a detailed clock report section in the timing report. The clock reports are listed in order of negative slack.

General Critical Path Information

This section contains general information about the most critical paths in the design.

Clock Information	Description
N most critical start points	Start points can be input ports or registers. If the start point is a register, you see the starting pin in the report. To change the number of start points reported, choose Project -> Implementation Options, and set the number on the Timing Report panel.
N most critical end points	End points can be output ports or registers. If the end point is a register, you see the ending pin in the report. To change the number of end points reported, select Project -> Implementation Options, and set the number on the Timing Report panel.
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 264) 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

Critical path information	Description
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.

	A	B	C	D	E	F	G	H	I
	async_clk.rpt.csv								
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 Tcl Syntax Guidelines for Constraint Files, on page 52 in the *User Guide*.

This section describes the following topics:

- Reporting Details, on page 272
- Inapplicable Constraints, on page 273
- Applicable Constraints With Warnings, on page 274
- Sample Constraint Check Report, on page 275

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 273 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 274 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
create_generated_clock	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
set_reg_input_delay set_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_uncertainty	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.
set_reg_input_delay set_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 1138R, built Jun 7 2013
# Copyright (C) 1994-2013, Synopsys, Inc.

# Written on Fri Jun 7 09:42:22 2013
##### DESIGN INFO ###################################################

Top View:               "decode_top"
Constraint File(s):     "C:\timing_88\FPGA_decode_top.sdc"
##### 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

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, tri0, tri1	Net types: trireg, triand, trior, wand, wor, charge strength
Register types: • reg, integer, time (64-bit reg) • arrays of reg	Register types: real
Gate primitive, module, and macromodule 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 +, -, *, /, %, <, >, <=, >=, ==, !=, ===, !==, &&, ||, !, ~, &, ~&, |, ~|, ^~, ~^, ^, <<, >>, ?:, { }, {{ }}, rol, ror (See Operators, on page 290 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

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)

Data Types

Verilog data types can be categorized into the following general types:

- Net Data Types, on page 284
- Register Data Types, on page 284
- Miscellaneous Data Types, on page 285

Net Data Types

Net data types are used to model physical connections. The following net types are supported:

wire	Connects elements; used with nets driven by a single gate or continuous assignment
tri	Connects elements; used when a net includes more than one driver
tri0	Models resistive pulldown device (its value is 0 when no driver is present)
tri1	Models resistive pullup device (its value is 1 when no driver is present)

While the Synopsys FPGA Verilog compiler allows the use of tri0 and tri1 nets, these nets are treated as wire net types during synthesis, and any variable declared as a tri0 or tri1 net type behaves as a wire net type. A warning is issued in the log file alerting you that a tri0 or tri1 variable is being treated as a wire net type and that a simulation mismatch is possible.

Register Data Types

The supported register data types are outlined in the following table:

reg	A 1-bit wide data type; when more than one bit is required, a range declaration is included
integer	A 32-bit wide data type that cannot include a range declaration
time	A 64-bit wide data type that stores simulation time as an unsigned number; a range declaration is not allowed

Miscellaneous Data Types

The following data types are also supported:

parameter	Specifies a constant value for a variable (see Creating a Scalable Module, on page 333)
localparam	A local constant parameter (see Localparams, on page 309)
genvar	A Verilog 2001 temporary variable used for index control within a generate loop (see Generate Statement, on page 311)

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 precede any inputs. For tristate gates, outputs come first, then inputs, and then enable. The following tables list the supported 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

Port Definitions

Port signals are defined as input, output, or bidirectional and are referred to as the port list for the module. The three signal declarations are input, output, and inout as described in the following table.

input	An input signal to the module
output	An output signal from the module
inout	A bidirection signal to/from the module

Statements

Statement types include loop statements, case statements, and conditional statements as described in the ensuing subsections.

loop Statements

Loop statements are used to modify blocks of procedural statements. The loop statements include for, repeat, while, and forever as described in the following table:

for	Continues to execute a given statement until the expression becomes true; the first assignment is executed initially and then the expression is evaluated repeatly
repeat	Executes a given statement a fixed number of times; the number of executions is defined by the expression following the repeat keyword.

while	Executes a given statement until the expression becomes true
forever	Continuously repeats the ensuing statement

case Statements

Case statements select one statement from a list of statements based on the value of the case expression. A case statement is introduced with a case, casex, or casez keyword and is terminated with an endcase statement. A case statement can include a default condition that is taken when none of the case select expressions is valid.

case	allow branching on multiple conditional expressions based on case statement matching
casex	allows branching of multiple conditional expression matching where any 'x' (unknown) or 'z' value appearing in the case expression is treated as a don't care
casez	allows branching of multiple conditional expression matching where any 'z' (high impedance) value appearing in the case expression is treated as a don't care
endcase	terminates a case, casex, or casez statement
default	assigns a case expression to a default condition when there are no other matching conditions

Conditional Statements

Conditional statements are used to determine which statement is to be executed based on a conditional expression. The conditional statements include if, else, and else if. The simplified syntax for these conditional statements is either:

    if (*conditionalExpression***)**
        *statement1***;**
    else
        *statement2***;**

or

if (*conditionalExpression***)**
    *statement1***;**
else if (*conditionalExpression***);**
    *statement2***;**
else
    *statement3***;**

The if statement can be used in one of two ways:

- as a single "if-else" statement shown in the first simplified syntax

- as a multiple "if-else-if" statement shown in the second simplified syntax

In the first syntax, when *conditionalExpression* evaluates true, *statement1* is executed, and when *conditionalExpression* evaluates false, *statement2* is executed.

In the second syntax, when *conditionalExpression* evaluates true, *statement1* is executed as in the first syntax example. However, when *conditionalExpression* evaluates false, the second conditional expression (else if) is evaluated and, depending on the result, either *statement2* or *statement3* is executed.

Blocks

Blocks delimit a set of statements. The block is typically introduced by a keyword that identifies the start of the block, and is terminated by an end keyword that identifies the end of the block.

module/endmodule Block

The module/endmodule block is the basic compilation unit in Verilog. Modules are introduced with the module (or macromodule) keyword and are terminated by the endmodule keyword. For more information, see Verilog Module Template, on page 332. The following example shows the basic module syntax.

```
module add (out, in1, in2);output out;
input in1, in2;
assign out = in1 & in2;
endmodule
```

begin/end Block

A begin/end block provides a method of grouping multiple statements into a always block. The statements within the this block are executed in the order listed. When a timing control statement is included within the block, execution of the next statement is delayed until after the timing delay. The following example illustrates a begin/end block:

```
module tmp (in1, in2, out1, out2);
input in1, in2;
output out1, out2;
reg out1, out2;

always@(in1, in2)
begin
   out1 =(in1 & in2);
   out2 =(in1 | in2);
end
endmodule
```

fork/join Block

A fork/join block provides a method of grouping multiple statements into a an always block. The statements within this block are executed simultaneously. With parallel blocks, because all statements are executed at the same time, mutually dependent statements are not allowed. The following example illustrates a fork/join block:

```
module tmp (in1, in2, out1, out2);
input in1, in2;
output out1, out2;
reg out1, out2;

always@(in1, in2)
fork
   out1 =(in1 & in2);
   out2 =(in1 | in2);
join
endmodulefork, join
```

generate/endgenerate Block

A generate block is created using one of the generate-loop, generate-conditional, or generate-case format. The block is introduced with the keyword generate and terminated with the keyword endgenerate. For more information, see Generate Statement, on page 311.

Operators

Arithmetic Operators

Arithmetic operators can be used with all data types.

Symbol	Usage	Function
+	a + b	a plus b
-	a - b	a minus b
*	a * b	a multiplied by b
/	a / b	a divided by b
%	a % b	a modulo b

The / and % operators are supported for compile-time constants and constant powers of two. For the modulus operator (%), the result takes the sign of the first operand.

Relational Operators

Relational operators compare expressions. The value returned by a relational operator is 1 if the expression evaluates true or 0 if the expression evaluates false.

Symbol	Usage	Function
<	a < b	a is less than b

>	a > b	a is greater than b
<=	a <= b	a is less than or equal to b
=>	a => b	a equal to or greater than b

Equality Operators

The equality operators compare expressions. When a comparison fails, the result is 0, otherwise it is 1. When both operands of a logical equality (==) or logical inequality (!=) contain an unknown value (x) or high-impedance (z) value, the result of the comparison is unknown (x); otherwise the result is either true or false.

When an operands of case equality (===) or case inequality (!==) contains an unknown value (x) or high-impedance (z) value, the result is calculated bit-by-bit.

Symbol	Usage	Function
==	m == n	m is equal to n
!=	m != n	m is not equal to n
===	m === n	m is identical to n
!==	m !== n	m is not identical to n

When an equality (==) or inequality (!=) operator includes unknown bits (for example, A==4'b10x1 or A!=4'b111z), the Synopsys 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

Logical Operators

Logical operators connect expressions. The result a logical operation is 0 if false, 1 if true, or x (unknown) if ambiguous. The negation operator (!) changes a nonzero or true value of the operand to 0 or a zero or false value to 1; an ambiguous value results in x (unknown) value.

Symbol	Usage	Function
&&	a && b	a and b
||	a || b	a or b
!	!a	not a

Bitwise Operators

Bitwise operators are described in the following table:

Symbol	Usage	Function
~	~m	Invert each bit
&	m & n	AND each bit
|	m | n	OR each bit
^	m ^ n	Exclusive OR each bit
~^, ^~	m ~^ n m ^~ n	Exclusive NOR each bit

Unary Reduction Operators

Unary reduction operators are described in the following table:

Symbol	Usage	Function
&	&m	AND all bits
~&	~&m	NAND all bits
|	|m	OR all bits
~|	~|m	NOR all bits
^	^m	Exclusive OR all bits
^~, ~^	~^m ^~m	Exclusive NOR all bits

Shift/Rotate Operators

Shift/rotate operators are described in the following table:

Symbol	Usage	Function
<<	m << n	Shift left n times
>>	m >> n	Shift right n times
rol	m rol n	Rotate left n times
ror	m ror n	Rotate right n times

Rol and ror are the rotate left and rotate right operators respectively, and shift bits by the specified amount. Rol shifts each bit in the left operand by the number of bits specified in the right operand. Bits in the left-most positions are shifted to become the right-most bits. For example:

```
myreg <= myreg rol 2;
```

Ror shifts each bit in the right operand by the number of bits specified in the left operand. Bits in the right-most positions are shifted to become the left-most bits. For example:

```
myreg <= myreg ror 2;
```

Miscellaneous Operators

Miscellaneous operators are described in the following table:

Symbol	Usage	Function
? :	sel? m:n	If sel is true, select m
{ }	{m,n}	Concatenate m to n
{{ }}	{n{m}}	Replicate m n times

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 328 and Setting Verilog and VHDL Options, on page 82 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 that 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.

294   Synplify Pro for Microsemi Edition Reference Manual
                                                                                      January 2014

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 `elsif 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
```

Localparam

In Verilog 2001, localparam (constants that cannot be redefined or modified) follow the same parameter rules in regard to size and sign. Unlike parameter, localparam cannot be overidden by a defparam from another module.

Example:

```
parameter ONE = 1
localparam TWO=2*ONE
localparam [3:0] THREE=TWO+1;
localparam signed [31:0] FOUR=2*TWO;
```

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

The following limitations currently exist with cross-module referencing:

- Cross-module referencing through an array of instances is not supported.

- Cross-module referencing into generate blocks is not supported.

- In upward cross-module referencing, the reference must be an absolute path (an absolute path is always from the top-level module).

- Functions and tasks cannot be accessed 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, on page 319

- Library Support in Verilog, on page 320

- Constant Function Syntax Restrictions, on page 324

- Multi-dimensional Array Syntax Restrictions, on page 325

- Signed Multipliers in Verilog, on page 326

- Verilog Language Guidelines: always Blocks, on page 327

- Initial Values in Verilog, on page 328

- Cross-language Parameter Passing in Mixed HDL, on page 330

- Library Directory Specification for the Verilog Compiler, on page 331

General Synthesis Guidelines

Some general guidelines are presented here to help you synthesize your Verilog design. See Verilog Module Template, on page 332 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.
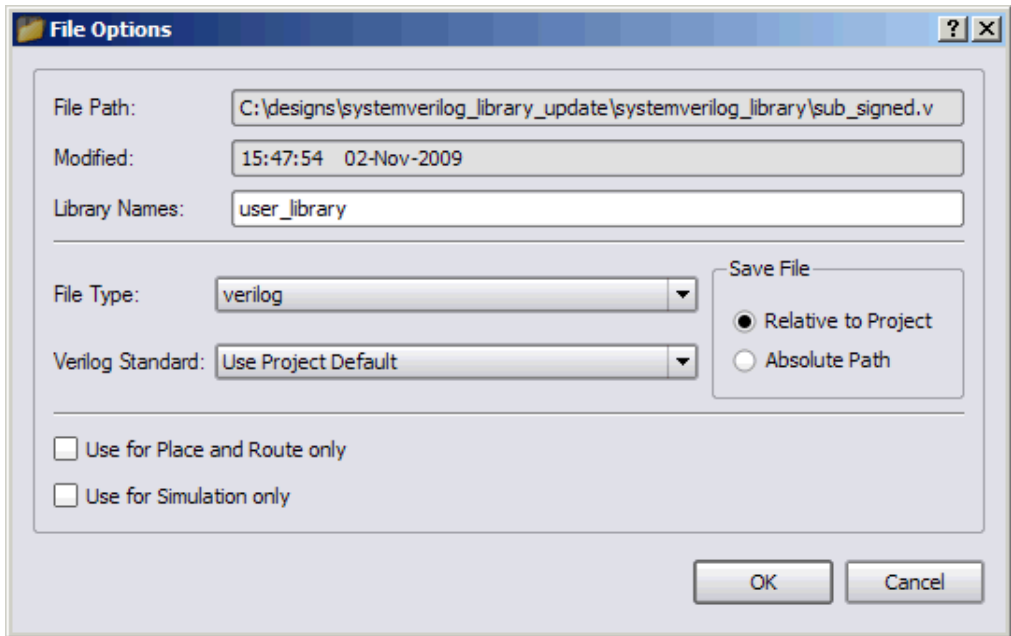
Searching for Verilog Design Units in Mixed-HDL Designs

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 *library-Name.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"
```
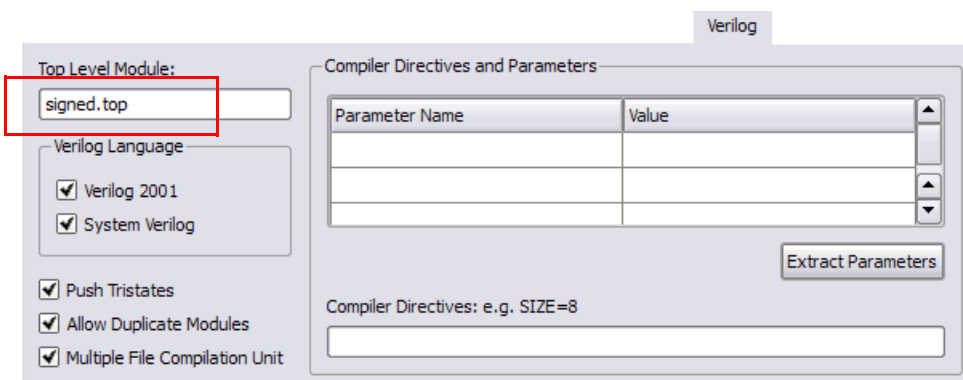

Limitations

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.

```verilog
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

```verilog
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

```verilog
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. For initial values for RAM, see Initial Values for RAMs, on page 590.

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
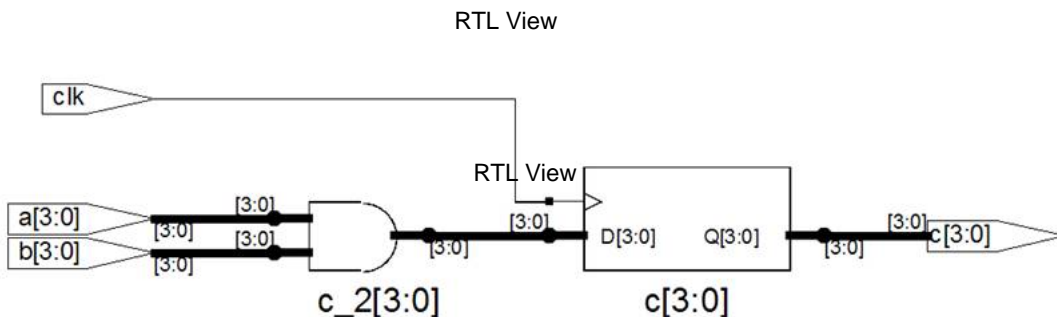
However, if the register is initialized to a value of 1, the compiler ensures that the initial value is used in synthesis.

```
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
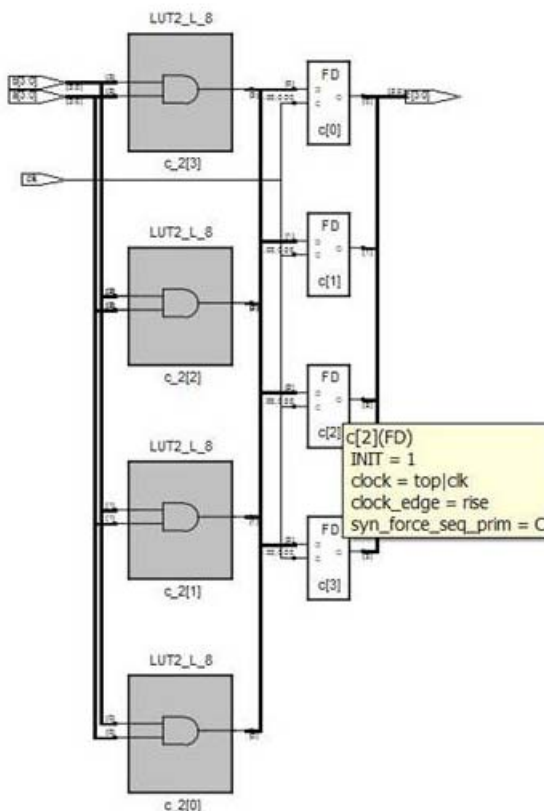
Cross-language Parameter Passing in Mixed HDL

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:

```verilog
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.

A disable statement can be included to terminate an active procedure within a module. As shown in the example, including a disable statement in the begin/end block prevents the out2 =(in1 | in2) expression from being executed.

```
always@(in1, in2)
begin : comb1
   out1 =(in1 & in2);
disable comb1;
   out2 =(in1 | in2);
endendmodule
```

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

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

You can instantiate a Verilog module for the VHDL entity adder and override its size parameter using the following statement highlighted in the Verilog code:

```
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
```

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 296.

```
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 *install-Directory*/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 343

- Level-sensitive Latches, on page 344

- Sets and Resets, on page 347

- SRL Inference, on page 351

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

When assigning signals in an always block:

- Explicitly declare, as a 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

```verilog
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

```verilog
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.
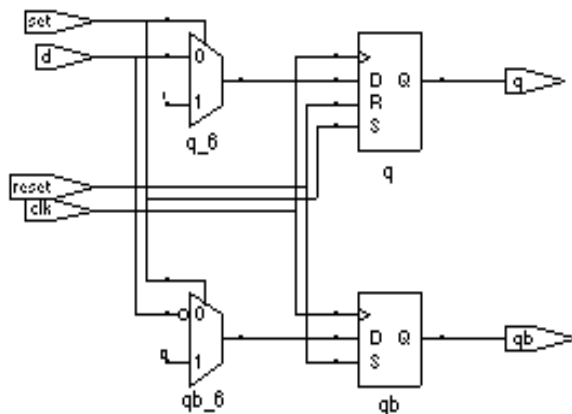
Event Control Syntax

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 assign-
ment 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 353

- State Values, on page 355

- Asynchronous State Machines, on page 356

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 356, 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 354.

- 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.

- 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_enum_encoding, on page 52 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, on page 43, 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

```
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

Instantiating Black Boxes in Verilog

Black boxes are modules with just the interface specified; internal information is ignored by the software. Black boxes can be used to directly instantiate:

- Technology-vendor primitives and macros (including I/Os).

- User-designed macros whose functionality was defined in a schematic editor, or another input source. (When the place-and-route tool can merge design netlists from different sources.)

Black boxes are specified with the syn_black_box directive. If the macro is an I/O, use black_box_pad_pin=1 on the external pad pin. The input, output, and delay through a black box is specified with special black box timing directives (see syn_black_box, on page 37).

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, on page 180 for information on the translate_off and translate_on directives.

If the black box has tristate outputs, you must define these outputs with a black_box_tri_pins directive (see black_box_tri_pins, on page 22).

For information on how to instantiate black boxes and technology-vendor I/Os, see *Defining Black Boxes for Synthesis, on page 294* 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 *after* the *object* declaration and *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 example1(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, on page 372

- Unsized Literals, on page 377

- Data Types, on page 377

- Arrays, on page 385

- Data Declarations, on page 389

- Operators and Expressions, on page 396

- Procedural Statements and Control Flow, on page 407

- Processes, on page 411

- Tasks and Functions, on page 416

- Hierarchy, on page 420

- Interface, on page 428

- System Tasks and System Functions, on page 436

- Generate Statement, on page 438

- Assertions, on page 444

- Keyword Support, on page 448

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 375 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 • Set Membership Case Inside 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, which represent 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 of the following:

- 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[*M*] = 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 388

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 *regis-
ters* 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* dimen-
sions. 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 expres-
sions 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. Assignment of a whole multi-dimensional array to another.

2. Access (reading) of an entire multi-dimensional array.

3. Assignment of an index (representing a complete dimension) of a multi-dimensional array to another.

4. Access (reading) of an index of a multi-dimensional array.

5. Assignment of a slice of a multi-dimensional array.

6. Access of a slice of a multi-dimensional array.

7. Access of a variable part-select of a multi-dimensional array.

In addition, wire declarations are supported for any packed or unpacked data type. This support includes multi-dimensional enum and struct data types in input port declarations (see Nets, on page 390 for more information).

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.

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.

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 437.

Data Declarations

There are several data declarations in SystemVerilog: *literals*, *parameters*, *constants*, *variables*, *nets*, and *attributes*. The following are described here:

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. When using wire with struct and union constructs,
each member of the construct must be a 4-state data type.

Syntax

wire *4stateDataType identifierName* **;**

Example – Logic Type Defined as a Wire Type

```
module top (
    input wire logic [1:0] din1,din2 , // logic defined as wire
    output logic [1:0] dout  );
    assign dout = din1 + din2;
endmodule
```

Example – struct Defined as a Wire Type

```
typedef struct { logic [4:1] a;
} MyStruct;

module top (
    input wire MyStruct [1:0] din [1:0] [1:0], // structure
        // defined as wire
    output wire MyStruct [1:0] dout [1:0] [1:0] ); // structure
        // defined as wire
assign dout = din;
endmodule
```

Restrictions

Using wire with a 2-state data type (for example, int or bit) results in the following error message:

CG1205 | Net data types must be 4-state values

A lexical restriction also applies to a net or port declaration in that the net type keyword wire cannot be followed by reg.

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

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).

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

Set Membership Case Inside Operator

With the case inside operator, a case expression is compared to each case item. Also, when using this operator, the case items can include an open range. The comparison returns TRUE when the case expression matches a case item, otherwise it returns FALSE.

Syntax

```
[unique|priority] case (caseExpression) inside
    (caseItem) : statement ;
    (caseItem) : statement ;

        .
        .
        .
    [default : statement ;]
endcase
```

In the above syntax, *caseItem* can be:

- a list of constants

- an open range

- a combination of a list of constants and an open range

The case inside operator supports the following optional modifiers:

- unique – each *caseItem* is unique and there are no overlapping *caseItems*. If there is an overlapping *caseItem*, a warning is issued.

- priority – the case statement is prioritized and all possible legal cases are covered by the case statement. If the *caseExpression* fails to match any of the *caseItems*, a warning is issued.

Example – Case Inside

```
module top# (
   parameter byte p1[2:1][4:1] = '{'{0,2,4,6},'{1,3,5,7}} )
//Input
(  input logic[4:1]sel,a,b,
//Output
   output logic[3:1] q );

always_comb begin
   case (sel) inside
      8,p1[1],10,12,14:q <= a;
      p1[2],9,11,13,15:q <= b;
   endcase
end
endmodule
```

Example – Unique Case Inside

```
module top# (
   parameter byte p1[2:1][4:1] = '{'{15,14,13,12},'{0,1,2,3}} )
//Input
(  input logic[4:1]sel1,sel2,
   input byte a,b,
//Output
   output byte q );

generate begin
   always@(*) begin
      unique case (sel1^sel2) inside
         p1 : q = a+b;
         [4:7],13,14,15 : q = a ^ b;
         [9:12],8 : q = a*b;
      endcase
   end
end
endgenerate
endmodule
```

Example – Priority Case Inside

```
typedef enum logic[4:1] {s[0:15]} EnumDt;

module top (
   input logic reset,
   input logic clock,
   input logic x,
   input logic[2:1] y,
   output logic[3:1] op );
EnumDt state;

always@(posedge reset or posedge clock)
begin
   if (reset == 1'b1)
   begin
      op <= 3'b000;
      state <= s0;
   end
   else
   begin
      priority case (state) inside
         [s0:s2],s12 : begin
            if (x == 1'b0  && y == 1'b0)
            begin
               state <= s3;
               op <= 3'b001;
            end
            else
            begin
               state <= s2;
               op <= 3'b000;
            end
         end
         [s3:s5] : begin
            if(x == 1'b1  && y== 1'b0)
            begin
               state <= s7;
               op <= 3'b010;
            end
            else
            begin
               state <= s9;
               op <= 3'b110;
            end
         end
         [s6:s8],s13 : begin
```

```
                    if(x == 1'b0  &&  y== 1'b1)
                    begin
                       state <= s11;
                       op <= 3'b011;
                    end
                    else if (x == 1'b0 && y == 1'b1)
                    begin
                       state <= s4;
                       op <= 3'b010;
                    end
                 end
                 [s9:s11] : begin
                    if(x == 1'b1  && y== 1'b1 )
                    begin
                       state <= s5;
                       op <= 3'b100;
                    end
                    else if (x == 1'b0 && y == 1'b1)
                    begin
                       state <= s0;
                       op <= 3'b111;
                    end
                 end
                 default : begin
                    state <= s1;
                    op <= 3'b111;
                 end
              endcase
           end
       end
     endmodule
```

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, below
- For Loops, on page 408
- Unnamed Blocks, on page 409
- Block Name on end Keyword, on page 409
- Unique and Priority Modifiers, on page 409

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:

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 .

- 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 411).

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 repre-
sent 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 416
- endtask /endfunction Names, on page 419

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

Synplify Pro for Microsemi Edition Reference Manual
416   January 2014

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 423

- Port Connection Constructs, on page 424

- Extern Module, on page 427

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.

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.

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]);
    ```

- 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 425. 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

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 429 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 437

$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

Limitations

The $bits system function is not supported under the following conditions:

- Passing an interface member as an argument to the $bits function is not supported. In the example

```
parameter logic[2:0] din = $bits(ff_if_0.din);
```

  interface instance ff_if_0.din is one of the ports of the modport. To avoid the limitation, use the actual value as the argument in place of the interface member.

- $bits cannot be used within a module instantiation:

```
module Top
   (output foo);
   Intf intf();
   Foo #(.PARAM($bits(intf.i))) Foo (.foo);
endmodule : Top
```

- $bits is not supported with params/localparams:

```
localparam int WIDTH = $bits(ramif.port0_out);
```

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 System-Verilog 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

SYNOPSYS®
Accelerating Innovation

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 450
- Unsupported VHDL Language Constructs, on page 451
- Partially-supported VHDL Language Constructs, on page 452
- Ignored VHDL Language Constructs, on page 452

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

- report

- initial values on inout ports

- assert on ports and signals

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
'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*] **;**

In the above syntax:

- 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 where 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

l**ibrary** *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 any files that instantiate 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 the creation of 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;
```

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) when-
ever 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, subtrac-
tors, 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 521

Simple Component Instantiation

In this method, a component is first declared either in the declaration region of the architecture, or in a package of (typically) component declarations, and then instantiated in the statement region of the architecture. By default, the synthesis process binds a named entity (and architecture) in the working library to all component instances that specify a component declaration with the same name.

Syntax

```
label : [component] declaration_name
    [generic map (actual_generic1, actual_generic2, ... ) ]
    [port map ( port1, port2, ... ) ] ;
```

The use of the reserved word component is optional in component instantiations.

Example: VHDL 1987

```
architecture struct of hier_add is
component add
   generic (size : natural);
   port (a : in bit_vector(3 downto 0);
         b : in bit_vector(3 downto 0);
         result : out bit_vector(3 downto 0) );
end component;

begin
-- Simple component instantiation
add1: add
   generic map(size => 4)
   port map(a => ain,
      b => bin,
      result => q);

-- Other code
```

Example: VHDL 1993

```
architecture struct of hier_add is
component add
   generic (size : natural);
   port (a : in bit_vector(3 downto 0);
         b : in bit_vector(3 downto 0);
         result : out bit_vector(3 downto 0) );
end component;

begin
-- Simple component instantiation
add1: component add -- Component keyword new in 1993
   generic map(size => 4)
   port map(a => ain,
      b => bin,
      result => q);

-- Other code
```

Note: If no entity is found in the working library named the same as the declared component, all instances of the declared component are mapped to a black box and the error message "Unbound component mapped to black box" is issued.

VHDL Selected Name Support

Selected Name Support (SNS) is provided in VHDL for constants, operators, and functions in library packages. SNS eliminates ambiguity in a design referencing elements with the same names, but that have unique functionality when the design uses the elements with the same name defined in multiple packages. By specifying the library, package, and specific element (constant, operator, or function), SNS designates the specific constant, operator, or function used. This section discusses all facets of SNS. Complete VHDL examples are included to assist you in understanding how to use SNS effectively.

Constants

SNS lets you designate the constant to use from multiple library packages. To incorporate a constant into a design, specify the library, package, and constant. Using this feature eliminates ambiguity when multiple library packages have identical names for constants and are used in an entity-architecture pair.

The following example has two library packages available to the design constants. Each library package has a constant defined by the name C1 and each has a different value. SNS is used to specify the constant (see work.PACKAGE.C1 in the constants example below).

```
    -- CONSTANTS PACKAGE1
    library IEEE;
    use IEEE.std_logic_1164.all;
    package PACKAGE1 is
        constant Cl: std_logic_vector := "10001010";
    end PACKAGE1;
```

```
-- CONSTANTS PACKAGE2
library IEEE;
use IEEE.std_logic_1164.all;
package PACKAGE2 is
   constant C1: std_logic_vector := "10110110";
end PACKAGE2;

-- CONSTANTS EXAMPLE
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity CONSTANTS is
   generic (num_bits : integer := 8) ;
      port (a,b: in std_logic_vector (num_bits -1 downto 0);
            out1, out2: out std_logic_vector (num_bits -1 downto 0)
            );
end CONSTANTS;

architecture RTL of CONSTANTS is
begin
   out1 <= a - work.PACKAGE1.Cl; -Example of specifying SNS
   out2 <= b - work.PACKAGE2.C1; -Example of specifying SNS
end RTL;
```

In the above design, outputs out1 and out2 use two C1 constants from two different packages. Although each output uses a constant named C1, the constants are not equivalent. For out1, the constant C1 is from PACKAGE1. For out2, the constant C1 is from PACKAGE2. For example:

```
out1 <= a - work.PACKAGE1.Cl; is equivalent to out1 <= a - "10001010";
```

whereas:

```
out2 <= b - work.PACKAGE2.Cl; is equivalent to out2 <= b - "10110110";
```

The constants have different values in different packages. SNS specifies the package and eliminates ambiguity within the design.

Functions and Operators

Functions and operators in VHDL library packages customarily have overlapping naming conventions. For example, the add operator in the IEEE standard library exists in both the std_logic_signed and std_logic_unsigned packages. Depending upon the add operator used, different values result. Defining only one of the IEEE library packages is a straightforward solution to eliminate ambiguity, but applying this solution is not always possible. A design requiring both std_logic_signed and std_logic_unsigned package elements must use SNS to eliminate ambiguity.

Functions

In the following example, multiple IEEE packages are declared in a 256x8 RAM design. Both std_logic_signed and std_logic_unsigned packages are included. In the RAM definition, the signal address_in is converted from type std_logic_vector to type integer using the CONV_INTEGER function, but which CONV_INTEGER function will be called? SNS determines the function to use. The RAM definition clearly declares the std_logic_unsigned package as the source for the CONV_INTEGER function.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;
use IEEE.numeric_std.all;

entity FUNCTIONS is
   port (address : in std_logic_vector(7 downto 0);
         data_in : in std_logic_vector(7 downto 0);
         data_out : out std_logic_vector(7 downto 0);
         we : in std_logic;
         clk : in std_logic );

end FUNCTIONS;

architecture RTL of FUNCTIONS is
type mem_type is array (255 downto 0) of
   std_logic_vector (7 downto 0);
signal mem: mem_type;
signal address_in: std_logic_vector(7 downto 0);
begin
data_out <= mem(IEEE.std_logic_unsigned.CONV_INTEGER(address_in));
   process (clk)
   begin
```

```
            if rising_edge(clk) then
                if (we = '1') then
                    mem(IEEE.std_logic_unsigned.CONV_INTEGER(address_in))
                        <= data_in;
                end if;
            address_in <= address;
            end if;
        end process;
    end RTL;
```

Operators

In this example, comparator functions from the IEEE std_logic_signed and std_logic_unsigned library packages are used. Depending upon the comparator called, a signed or an unsigned comparison results. In the assigned outputs below, the op1 and op2 functions show the valid SNS syntax for operator implementation.

```
    library IEEE;
    use IEEE.std_logic_1164.std_logic_vector;
    use IEEE.std_logic_signed.">";
    use IEEE.std_logic_unsigned.">";

    entity OPERATORS is
        port (in1 :std_logic_vector(1 to 4);
              in2 :std_logic_vector(1 to 4);
              in3 :std_logic_vector(1 to 4);
              in4 :std_logic_vector(1 to 4);
              op1,op2 :out boolean );
    end OPERATORS;

    architecture RTL of OPERATORS is
    begin
        process(in1,in2,in3,in4)
        begin

            --Example of specifying SNS
            op1 <= IEEE.std_logic_signed.">"(in1,in2);

            --Example of specifying SNS
            op2 <= IEEE.std_logic_unsigned.">"(in3,in4);
        end process;
    end RTL;
```

User-defined Function Support

SNS is not limited to predefined standard IEEE packages and packages supported by the synthesis tool; SNS also supports user-defined packages. You can create library packages that access constants, operators, and functions in the same manner as the packages supported by IEEE or the synthesis tool.

The following example incorporates two user-defined packages in the design. Each package includes a function named func. In PACKAGE1, func is an XOR gate, whereas in PACKAGE2, func is an AND gate. Depending on the package called, func results in either an XOR or an AND gate. The function call uses SNS to distinguish the function that is called.

```
-- USER DEFINED PACKAGE1
library IEEE;
use IEEE.std_logic_1164.all;
package PACKAGE1 is
    function func(a,b: in std_logic) return std_logic;
end PACKAGE1;

package body PACKAGE1 is
    function func(a,b: in std_logic) return std_logic is
begin
    return(a xor b);
end func;
end PACKAGE1;

-- USER DEFINED PACKAGE2
library IEEE;
use IEEE.std_logic_1164.all;

package PACKAGE2 is
    function func(a,b: in std_logic) return std_logic;
end PACKAGE2;

package body PACKAGE2 is
    function func(a,b: in std_logic) return std_logic is
begin
    return(a and b);
end func;
end PACKAGE2;

-- USER DEFINED FUNCTION EXAMPLE
library IEEE;
use IEEE.std_logic_1164.all;
```

```
entity USER_DEFINED_FUNCTION is
   port (in0: in std_logic;
         in1: in std_logic;
         out0: out std_logic;
         out1: out std_logic );
end USER_DEFINED_FUNCTION;

architecture RTL of USER_DEFINED_FUNCTION is
begin
   out0 <= work.PACKAGE1.func(in0, in1);
   out1 <= work.PACKAGE2.func(in0, in1);
end RTL;
```

Demand Loading

In the previous section, the user-defined function example successfully uses SNS to determine the func function to implement. However, neither PACKAGE1 nor PACKAGE2 was declared as a use package clause (for example, work.PACKAGE1.all;). How could func have been executed without a use package declaration? A feature of SNS is demand loading: this loads the necessary package without explicit use declarations. Demand loading lets you create designs using SNS without use package declarations, which supports all necessary constants, operators, and functions.

VHDL 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:

☐ (empty field)

☑ 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;
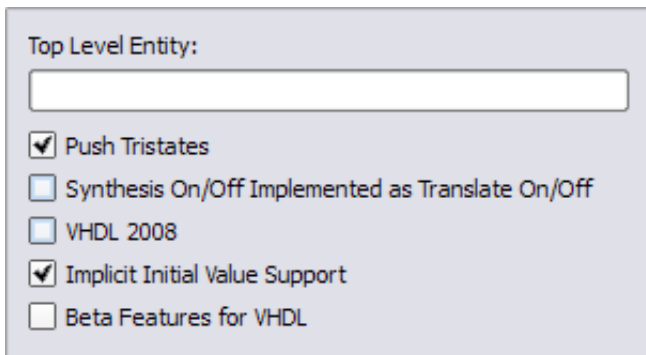
entity top is
   port (clk:in std_logic;
          a : in integer range 1 to 8;
          b : out integer range 1 to 8;
          d : out positive range 1 to 7 );
end entity top;
```

```
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;
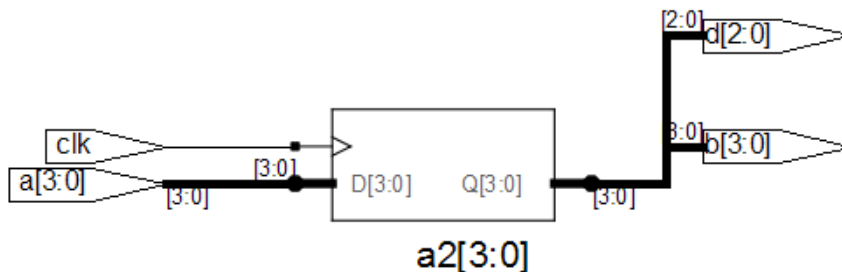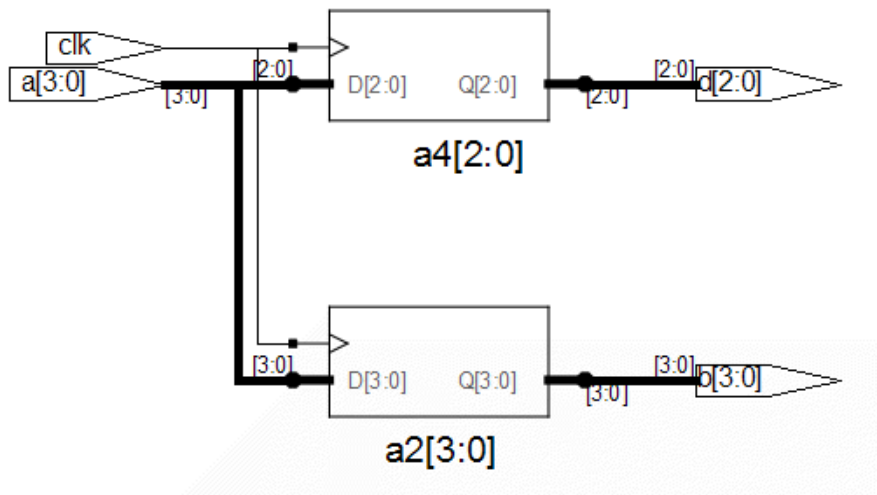
entity top is
   port (clk,set:in std_logic;
         a : in integer range -6 to 8;
         b : out integer range -6 to 8 );
end entity top;

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, on page 490
- VHDL Language Guidelines, on page 491
- Model Template, on page 492
- Constraint Files for VHDL Designs, on page 493
- Creating Flip-flops and Registers Using VHDL Processes, on page 494
- Clock Edges, on page 496
- Defining an Event Outside a Process, on page 497
- Using a WAIT Statement Inside a Process, on page 497
- Level-sensitive Latches Using Concurrent Signal Assignments, on page 498
- Level-sensitive Latches Using VHDL Processes, on page 499
- Signed mod Support for Constant Operands, on page 502
- Sets and Resets, on page 504

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 function-
ality 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) :=
          "1000000000000000000000000000000000";
          CONSTANT DENOMINATOR : unsigned(32 DOWNTO 0) :=
          "000000000000000000000000000000011";
          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

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 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 512.

- If a state machine defined in the code feeds sequential elements in a different clock domain, some encoding values can cause metastability. By default, the synthesis tools choose the optimal encoding value based on the number of states in the state machine. This can introduce additional decode logic that could cause metastability when it feeds sequential elements in a different clock domain. To prevent this instability, use syn_encoding = "original" to guide the synthesis tool for these cases.

- 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 syn_encoding Values, on page 43 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 syn_encoding, on page 43, 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 How Attributes and Directives are Specified, on page 4, 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 511.

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 immediately 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 521.

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 521
- Configuration Declaration, on page 525
- VHDL Configuration Statement Enhancement, on page 531

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 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;
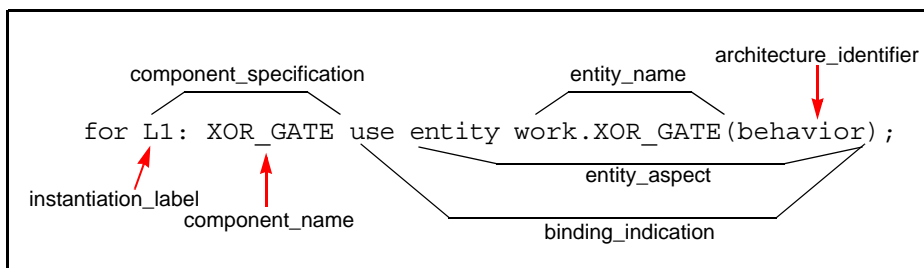
entity adder is
   port (a : in std_logic;
         b : in std_logic;
         cin : in std_logic;
         s : out std_logic;
         cout : out std_logic );
end adder;

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;
```
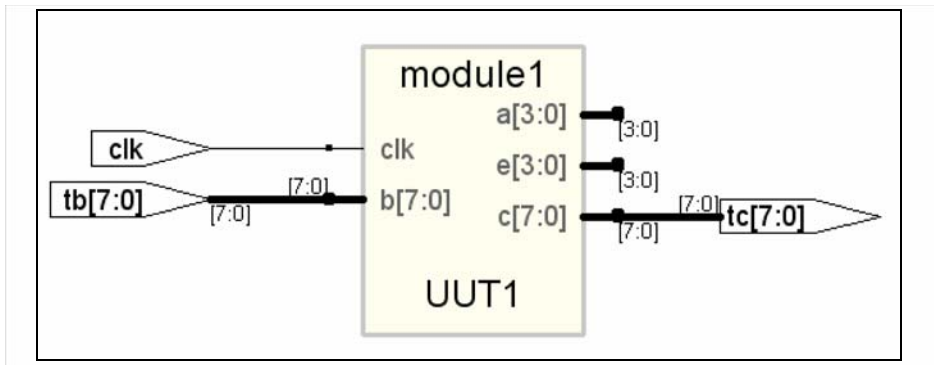
```
architecture top_a of top is
component myadder
   port (a : in std_logic;
         b : in std_logic;
         cin : in std_logic;
         s : out std_logic;
         cout : out std_logic);
end component;

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

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 521.

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 config-
uration declaration defines the adder architecture to use for each instantia-
tion. 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 decla-ration) is directly instantiated by writing a component instantiation state-ment 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

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 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 546
- Creating a Scalable Design Using VHDL Generics, on page 547
- Using a Scalable Architecture with VHDL Generics, on page 548
- Creating a Scalable Design Using Generate Statements, on page 550

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;
```

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 How Attributes and Directives are Specified, on page 4.

For information on how to instantiate black boxes and technology-vendor I/Os, see Defining Black Boxes for Synthesis, on page 294 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 syn_black_box, on page 37, 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 554
- Declaring Attributes, on page 555

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 556
- Sequential Logic Examples, on page 557

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:

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.

- Operators, on page 560

- Unconstrained Data Types, on page 563

- Unconstrained Record Elements, on page 565

- Predefined Functions, on page 566

- Packages, on page 568

- Generics in Packages, on page 571

- Context Declarations, on page 571

- Case-generate Statements, on page 572

- Else/elsif Clauses, on page 575

- Syntax Conventions, on page 576

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 characters other than 0 and 1 and string formats including signed/unsigned and string length

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

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.

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 (Implementa-tion Options dialog box) to enable use of these packages and the ?? operator.

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](#)

C H A P T E R   1 2

RAM and ROM Inference

This chapter provides guidelines and Verilog or VHDL examples for coding RAMs for synthesis. It covers the following topics:

Guidelines and Support for RAM Inference

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

Inference in Synthesis	Instantiation
Advantages	**Advantages**
Portable coding style	Most efficient use of the RAM primitives of a specific technology
Automatic timing-driven synthesis	
No additional tool dependencies	Supports all kinds of RAMs
Limitations	**Limitations**
Glue logic to implement the RAM might result in a sub-optimal implementation	Source code is not portable because it is technology-dependent
Can only infer synchronous RAMs	Limited or no access to timing and area data if the RAM is a black box
No support for address wrapping	
Pin name limitations means some pins are always active or inactive	Inter-tool access issues, if the RAM is a black box created with another tool

You must structure your source code correctly for the type of RAM you want to infer. The following table lists the supported technology-specific RAMs that can be generated by the synthesis tool.

RAM Type	Microsemi
Single Port	x
Dual Port	x
True Dual Port	x
Asymmetric	
Byte Enable	
Wide-Byte Write Enable	

The Block RAM Examples, on page 579 section contains code examples for
the various types of RAM that you can infer.

Block RAM Examples

The examples below show you how to define RAM in the RTL code so that the
synthesis tools can infer block RAM. See the following for details:

- Block RAM Mode Examples, on page 579

- Single-Port Block RAM Examples, on page 583

- Dual-Port Block RAM Examples, on page 586

- True Dual-Port RAM Example, on page 588

For details about inferring block RAM, see Automatic RAM Inference, on
page 308 in the *User Guide*.

Block RAM Mode Examples

The coding style supports the enable and reset pins of the block RAM
primitive. The tool supports different write mode operations for single-port
and dual-port RAM. This section contains examples of how to specify the
supported block RAM output modes:

- WRITE_FIRST Mode Example, on page 579

- READ_FIRST Mode Example, on page 581

- NO_CHANGE Mode Example, on page 582

WRITE_FIRST Mode Example

This example shows the WRITE_FIRST mode operation with active enable.

```
module v_rams_02a (clk, we, en, addr, di, dou);
input clk;
input we;
input en;
input [5:0] addr;
input [63:0] di;
output [63:0] dou;
reg [63:0] RAM [63:0];
reg [63:0] dou;

always @(posedge clk)
begin
if (en)
   begin
   if (we)
      begin
         RAM[addr] <= di;
         dou <= di;
      end
   else
      dou <= RAM[addr];
   end
end
endmodule

always @(posedge clk)
if (en & we) mem[addr] <= data_in;
endmodule
```

The following figure shows the RTL view of a WRITE_FIRST mode RAM
with output registered. The Technology view shows that the RAM is
mapped to a block RAM.

READ_FIRST Mode Example

The following piece of code is an example of READ_FIRST mode with both enable and reset, with reset taking precedence:

```
module ram_test(data_out, data_in, addr, clk, rst, en, we);
output [7:0]data_out;
input [7:0]data_in;
input [6:0]addr;
input clk, en, rst, we;
reg [7:0] mem [127:0] /* synthesis syn_ramstyle = "block_ram" */;
reg [7:0] data_out;
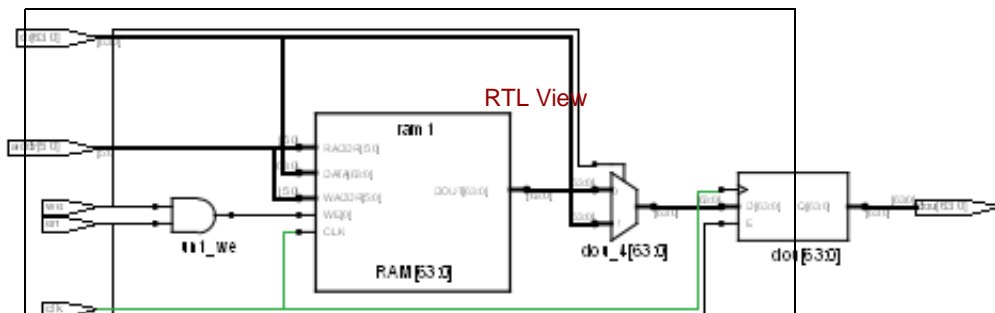
always@(posedge clk)
if(rst == 1)
   data_out <= 0;
else begin
   if(en) begin
      data_out <= mem[addr];
   end
end

always @(posedge clk)
if (en & we) mem[addr] <= data_in;
endmodule
```

The following figure shows the RTL view of a READ_FIRST RAM with inferred enable and reset, with reset taking precedence. The Technology view shows that the inferred RAM is mapped to a block RAM.

NO_CHANGE Mode Example

This NO_CHANGE mode example has neither enable nor reset. If you register the read address and the output address, the software infers block RAM.

```
module ram_test(data_out, data_in, addr, clk, we);
output [7:0]data_out;
input [7:0]data_in;
input [6:0]addr;
input clk,we;
reg [7:0] mem [127:0] /* synthesis syn_ramstyle = "block_ram" */;
reg [7:0] data_out;
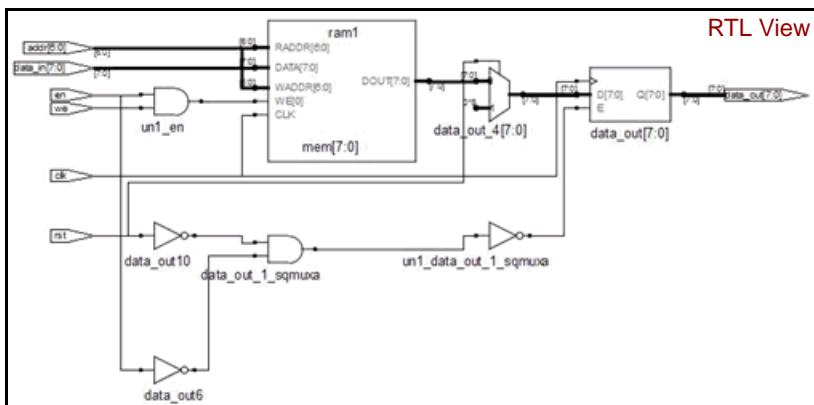
always@(posedge clk)
if(we == 1)
   data_out <= data_out;
else
   data_out <= mem[addr];

always @(posedge clk)
if (we) mem[addr] <= data_in;

endmodule
```

The next figure shows the RTL view of a NO_CHANGE RAM. The Technology view shows that the RAM is mapped to block RAM.

Single-Port Block RAM Examples

This section describes the coding style required to infer single-port block RAMs. For single-port RAM, the same address is used to index the write-to and read-from RAM. See the following examples:

Single-Port RAM with Read Address Registered Example

In these examples, the read address is registered, but the write address (which is the same as the read address) is not registered. There is one clock for the read address and the RAM.

Verilog Example: Read Address Registered

```verilog
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;
/* The array of an array register ("mem") from which the RAM is
inferred*/
reg [7:0] mem [127:0] ;
assign q = mem[read_add];

always @(posedge clk) begin
read_add <= a;
if(we)
   /* Register RAM Data */
   mem[a] <= d;
end

endmodule
```
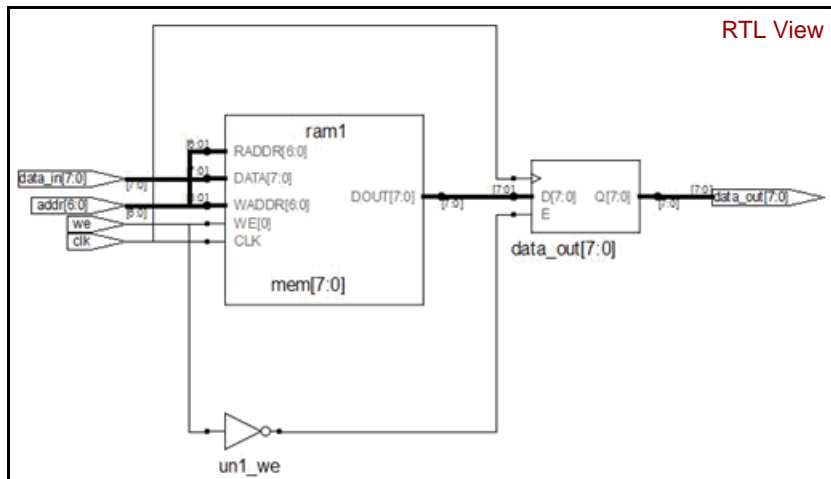
VHDL Example: READ Address Registered

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

entity ram_test is
   port (d : in std_logic_vector(7 downto 0);
         a : in std_logic_vector(6 downto 0);
         we : in std_logic;
         clk : in std_logic;
         q : out std_logic_vector(7 downto 0) );
end ram_test;

architecture rtl of ram_test is
type mem_type is array (127 downto 0) of
   std_logic_vector (7 downto 0);
signal mem: mem_type;
signal read_add : std_logic_vector(6 downto 0);
begin
   process (clk)
   begin
      if rising_edge(clk) then
         if (we = '1') then
            mem(conv_integer(a)) <= d;
         end if;
         read_add <= a;
      end if;
   end process;

q <= mem(conv_integer(read_add));
end rtl ;
```

Single-Port RAM with RAM Output Registered Examples

In this example, the RAM output is registered, but the read and write addresses are unregistered. The write address is the same as the read address. There is one clock for the RAM and the output.

Verilog Example: Data Output Registered

```verilog
module ram_test(q, a, d, we, clk);
output [7:0] q;
input [7:0] d;
input [6:0] a;
input clk, we;
/* The array of an array register ("mem") from which the RAM is
inferred */
reg [7:0] mem [127:0] ;
reg [7:0] q;

always @(posedge clk) begin
q = mem[a];
if(we)
   /* Register RAM Data */
   mem[a] <= d;
end

endmodule
```

VHDL Example: Data Output Registered

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

entity ram_test is
   port (d: in std_logic_vector(7 downto 0);
         a: in integer range 127 downto 0;
         we: in std_logic;
         clk: in std_logic;
      q: out std_logic_vector(7 downto 0) );
end ram_test;

architecture rtl of ram_test is
type mem_type is array (127 downto 0) of
   std_logic_vector (7 downto 0);
signal mem: mem_type;
begin
   process(clk)
   begin
      if (clk'event and clk='1') then
         q <= mem(a);
            if (we='1') then
```

```
                mem(a) <= d;
           end if;
      end if;
   end process;
end rtl;
```

Dual-Port Block RAM Examples

The following example or RTL code results in simple dual-port block RAMs
being implemented in supported technologies.

Verilog Example: Dual-Port RAM

This Verilog example has two read addresses, both of which are registered,
and one address for write (same as a read address), which is unregistered. It
has two outputs for the RAM, which are unregistered. There is one clock for
the RAM and the addresses.

```verilog
module dualportram ( q1,q2,a1,a2,d,we,clk1) ;
output [7:0]q1,q2;
input [7:0] d;
input [6:0]a1,a2;
input clk1,we;
wire [7:0] q1;
reg [6:0] read_addr1,read_addr2;
reg[7:0] mem [127:0] /* synthesis syn_ramstyle = "no_rw_check" */;
assign q1 = mem [read_addr1];
assign q2 = mem[read_addr2];

always @ ( posedge clk1) begin
read_addr1 <= a1;
read_addr2 <= a2;
if (we)
   mem[a2] <= d;
end

endmodule
```

VHDL Example: Dual-Port RAM

The following VHDL example is of READ_FIRST mode for a dual-port RAM:

```
Library IEEE ;
use IEEE.std_logic_1164.all ;
use IEEE.std_logic_arith.all ;
use IEEE.std_logic_unsigned.all ;

entity Dual_Port_ReadFirst is
   generic (data_width: integer :=4;
   address_width: integer :=10);

port (write_enable: in std_logic;
   write_clk, read_clk: in std_logic;
   data_in: in std_logic_vector (data_width-1 downto 0);
   data_out: out std_logic_vector (data_width-1 downto 0);
   write_address: in std_logic_vector (address_width-1 downto 0);
   read_address: in std_logic_vector (address_width-1 downto 0)
   );
end Dual_Port_ReadFirst;

architecture behavioral of Dual_Port_ReadFirst is
type memory is array (2**(address_width-1) downto 0) of
   std_logic_vector (data_width-1 downto 0);
signal mem : memory;

signal reg_write_address : std_logic_vector (address_width-1 downto 0);
signal reg_write_enable: std_logic;

attribute syn_ramstyle : string;
attribute syn_ramstyle of mem : signal is "block_ram";

begin
register_enable_and_write_address:
   process (write_clk,write_enable,write_address,data_in)
   begin
      if (rising_edge(write_clk)) then
         reg_write_address <= write_address;
         reg_write_enable <= write_enable;
      end if;
   end process;
```

```
write:
   process (read_clk,write_enable,write_address,data_in)
   begin
      if (rising_edge(write_clk)) then
         if (write_enable='1') then
            mem(conv_integer(write_address))<=data_in;
         end if;
      end if;
   end process;

read:
   process (read_clk,write_enable,read_address,write_address)
   begin
      if (rising_edge(read_clk)) then
         if (reg_write_enable='1') and (read_address =
            reg_write_address) then data_out <= "XXXX";
         else
            data_out<=mem(conv_integer(read_address));
         end if;
      end if;
   end process;

end behavioral;
```

True Dual-Port RAM Example

You must use a registered read address when you code true dual-port RAM. If you do not, you could get error messages. The following RTL example shows the recommended coding style for true dual-port block RAM. It is a Verilog example where the tool infers true dual-port RAM from a design with multiple writes:

```
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
```
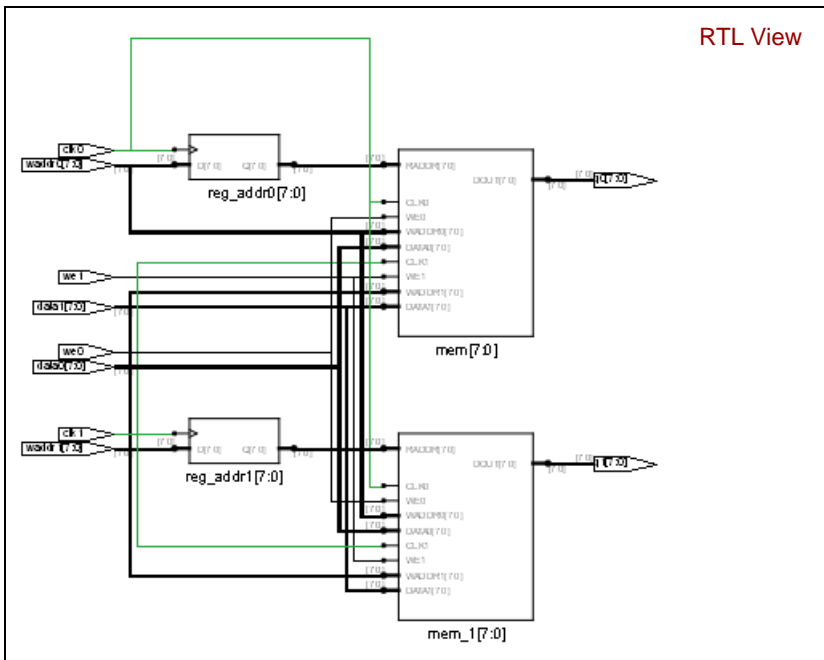
Limitations to RAM Inference

RAM inference is only supported for synchronous RAMs.

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 591 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 591 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 593), 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 593).

Binary File Format

The binary data file mem.ini that corresponds to the example in RAM Initialization Example, on page 590 looks like this:

```
11111111111111111111100110111    /* data for address 0 */
11111111111111111111101100111    /* data for address 1 */
11111111111111111111111000010
11111111111111111111100100001
11111111111111111111101110000
11111111111111111111011100110
... /* 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 590 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.

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 400 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.

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.

Verilog 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
```

VHDL 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
```

ROM Initialization with Generate Block

The software supports conditional ROM initialization with the generate block,
as shown in the following example:

```
generate
    if (INIT) begin
        initial
        begin
            $readmemb("init.hex",mem);
        end
    end
endgenerate
```

C H A P T E R 1 3

IP and Encryption Tools

This chapter describes the SYNCore IP functionality that is bundled with the synthesis tools, and supported IP encryption standards.

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 602
- FIFO Read and Write Operations, on page 603
- FIFO Ports, on page 605
- FIFO Parameters, on page 607
- FIFO Status Flags, on page 609
- FIFO Programmable Flags, on page 612

For further information, refer to the following:

- Specifying FIFOs with SYNCore, on page 394 of the *User Guide.*, for information about using the wizard to generate FIFOs
- Launch SYNCore Command, on page 221 and SYNCore FIFO Wizard, on page 223 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 603 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 604 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 609 and FIFO Programmable Flags, on page 612.

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.
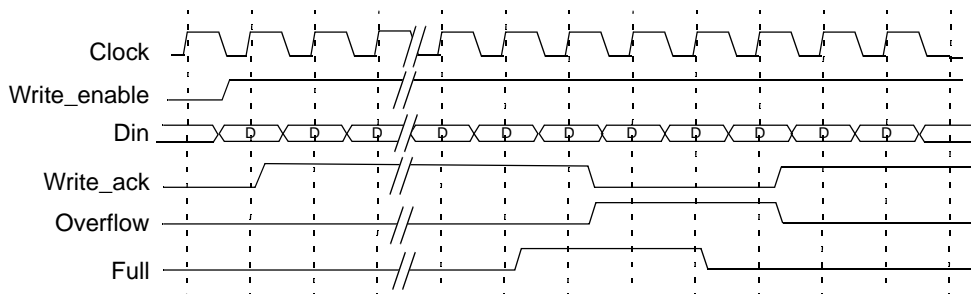
FIFO Read Operation

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 underflow of 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.
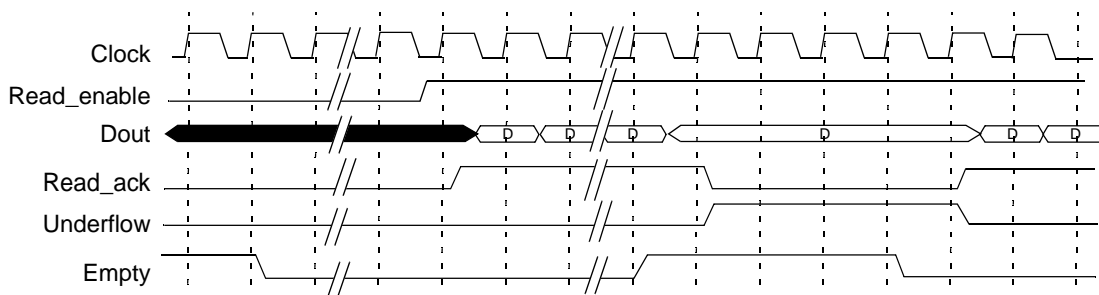
FIFO Ports

The following figure shows the FIFO ports.
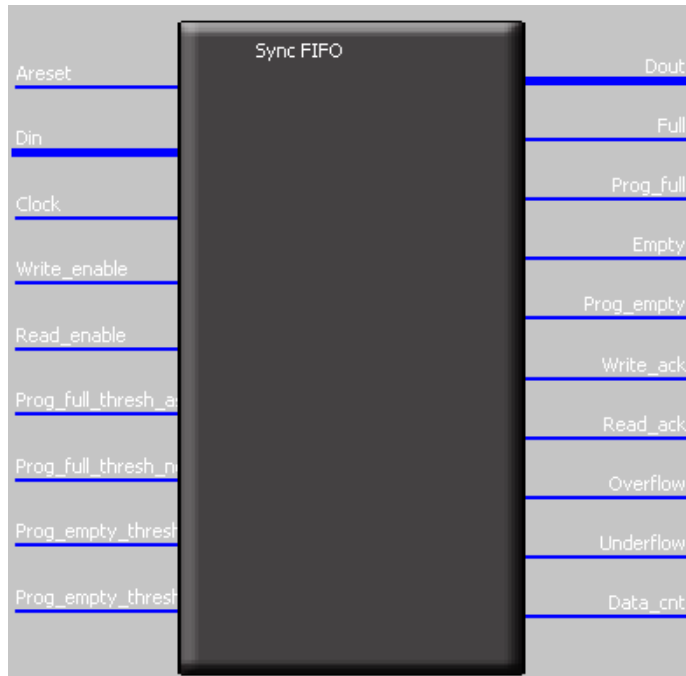
Port Name	Description
Almost_empty	Almost empty flag output (active high). Asserted when the FIFO is almost empty and only one more read can be performed. Can be active high or active low.
Almost_full	Almost full flag output (active high). Asserted when only one more write can be performed into the FIFO. 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 616 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 613 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

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.
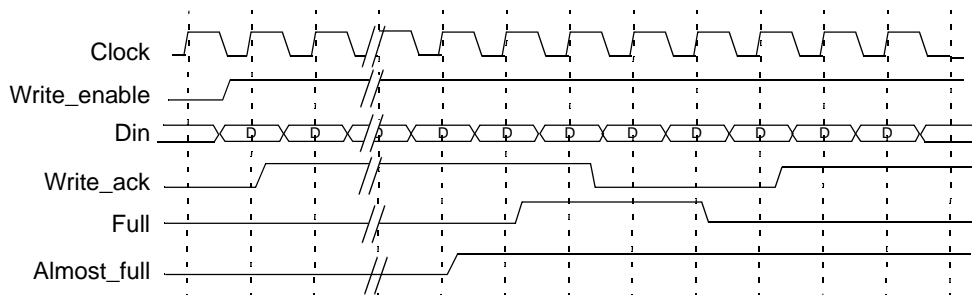
Empty/Almost Empty Flags

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.
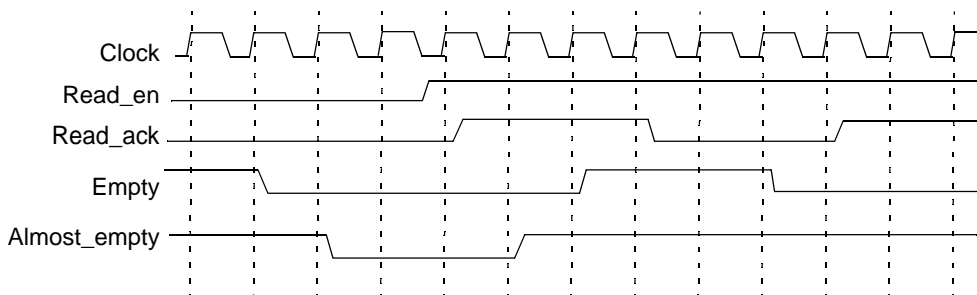
Handshaking Flags

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 613 for more information.

Prog_empty   Indicates that the FIFO has reached a user-defined empty threshold. See Programmable Empty, on page 616 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 de-asserted. 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
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.
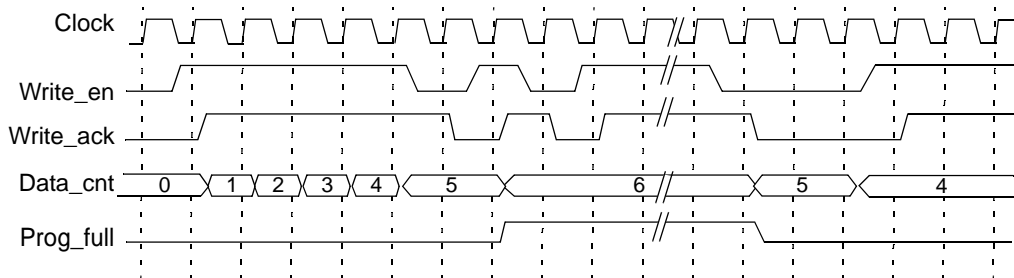
Programmable Full with Multiple Threshold Constants

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.
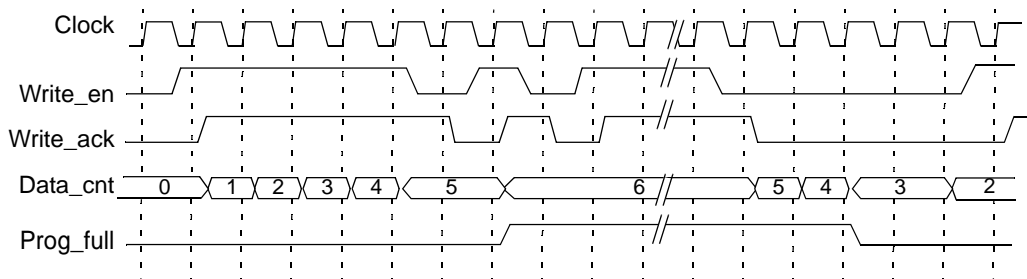
Programmable Full with Single Threshold Input

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.
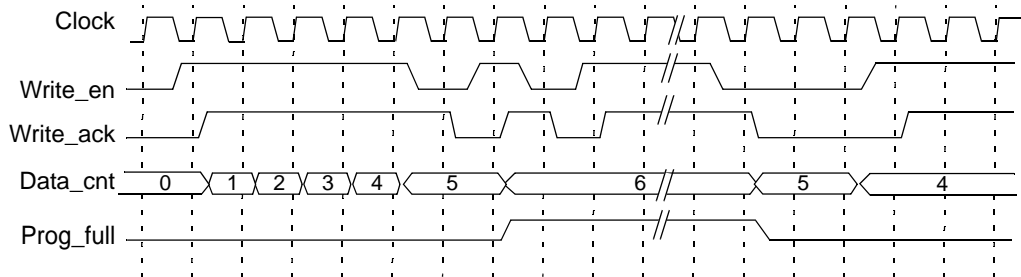
Programmable Full with Multiple Threshold Inputs

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.
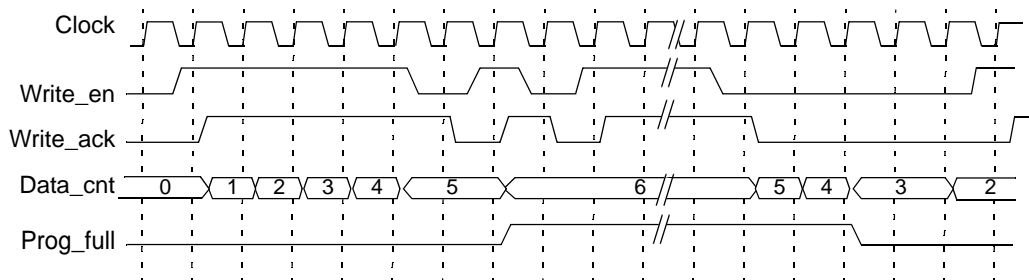
Programmable Empty

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
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.
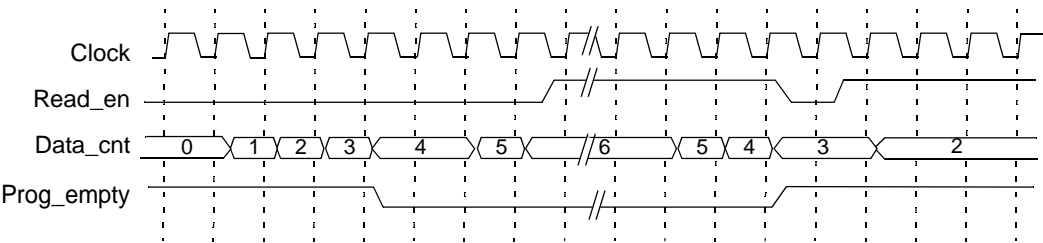
Programmable Empty with Multiple Threshold Constants

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.
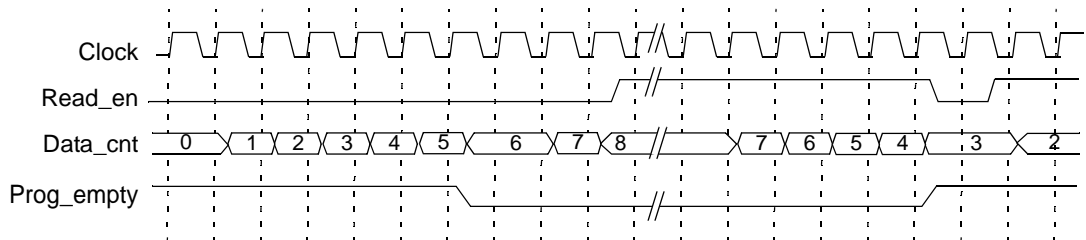
Programmable Empty with Single Threshold Input

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.
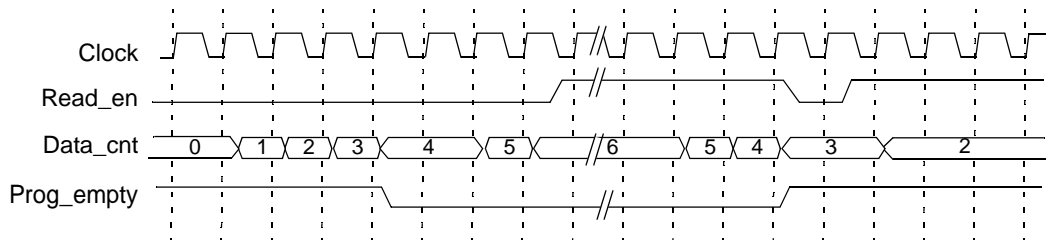
Programmable Empty with Multiple Threshold Inputs
PGM_EMPTY_TYPE = 4

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.
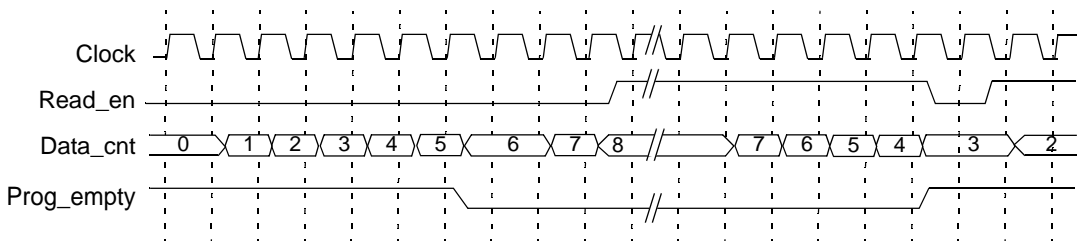
SYNCore RAM Compiler

The SYNCore RAM Compiler generates Verilog code for your RAM implementation. This section describes the following:

- Single-Port Memories, on page 619
- Dual-Port Memories, on page 621
- Read/Write Timing Sequences, on page 626

For further information, refer to the following:

- Specifying RAMs with SYNCore, on page 400 of the *User Guide*, for information about using the wizard to generate FIFOs

- Launch SYNCore Command, on page 221 and SYNCore FIFO Wizard, on page 223 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 400 in the *User Guide* for a procedure.

Single-Port Read

ADDR	00	01	02	03	

CLK

| QOUT | XX | F0 | F1 | F2 | F3 |

MEM1 — F1

MEM0 — F0

MEM3 — F3

MEM2 — F2

MEM4 — F4
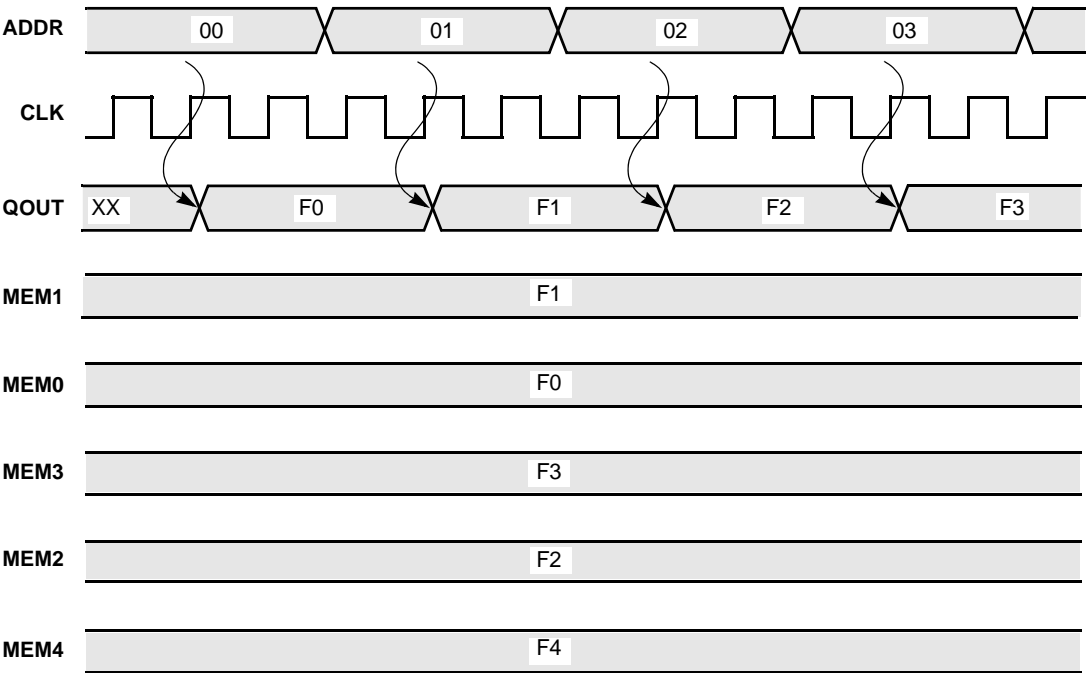
Single-Port Write
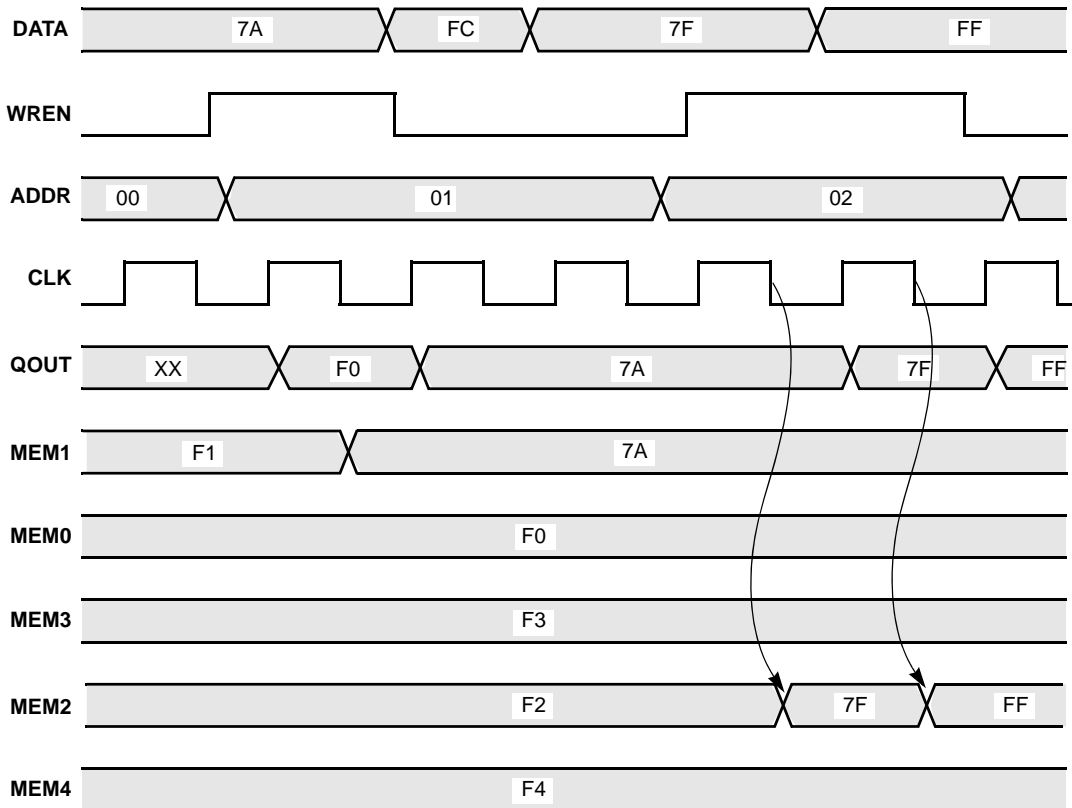
Dual-Port Memories

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 400 in the *User Guide* for a procedure to specify a dual-port RAM with SYNCore.

Dual-Port Single Read
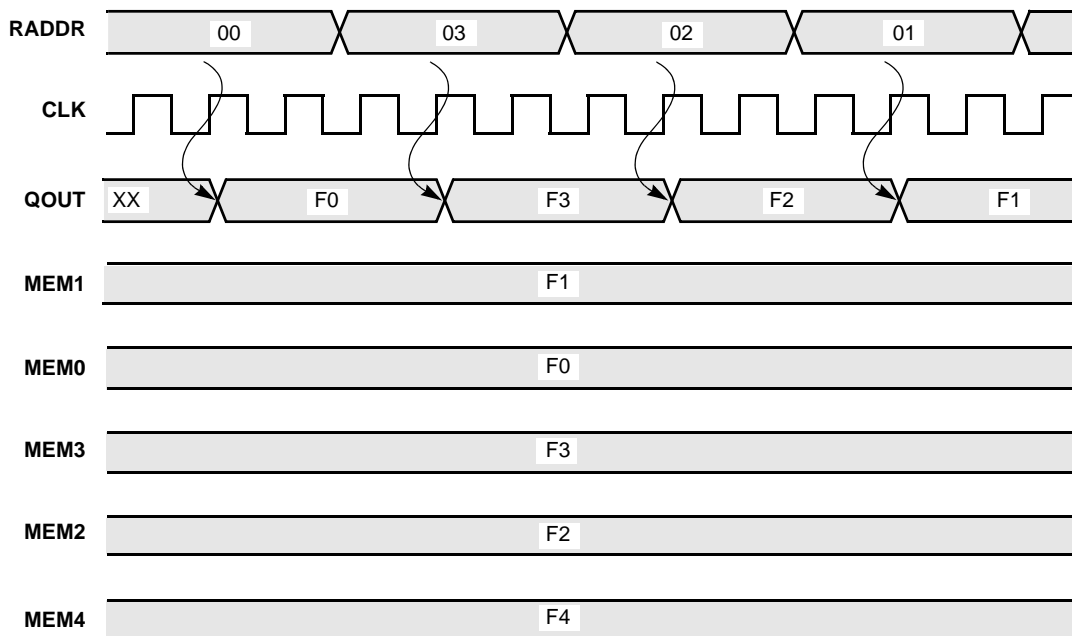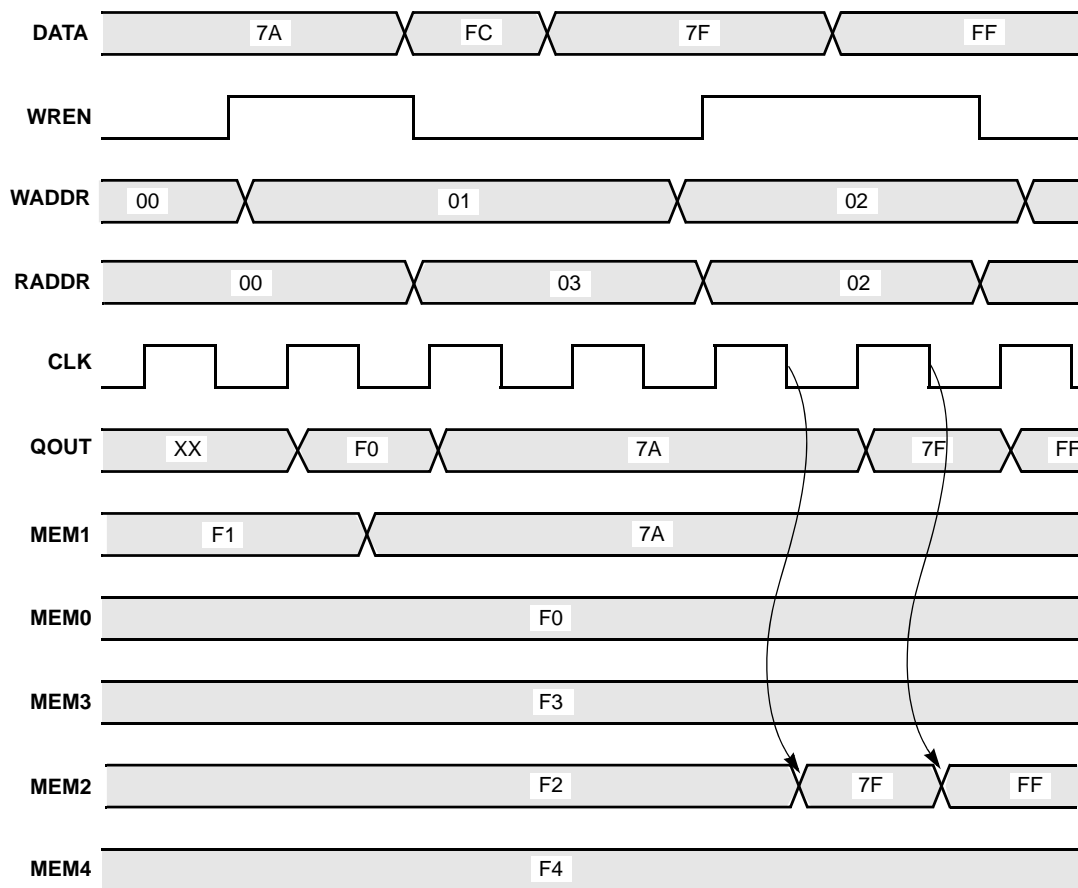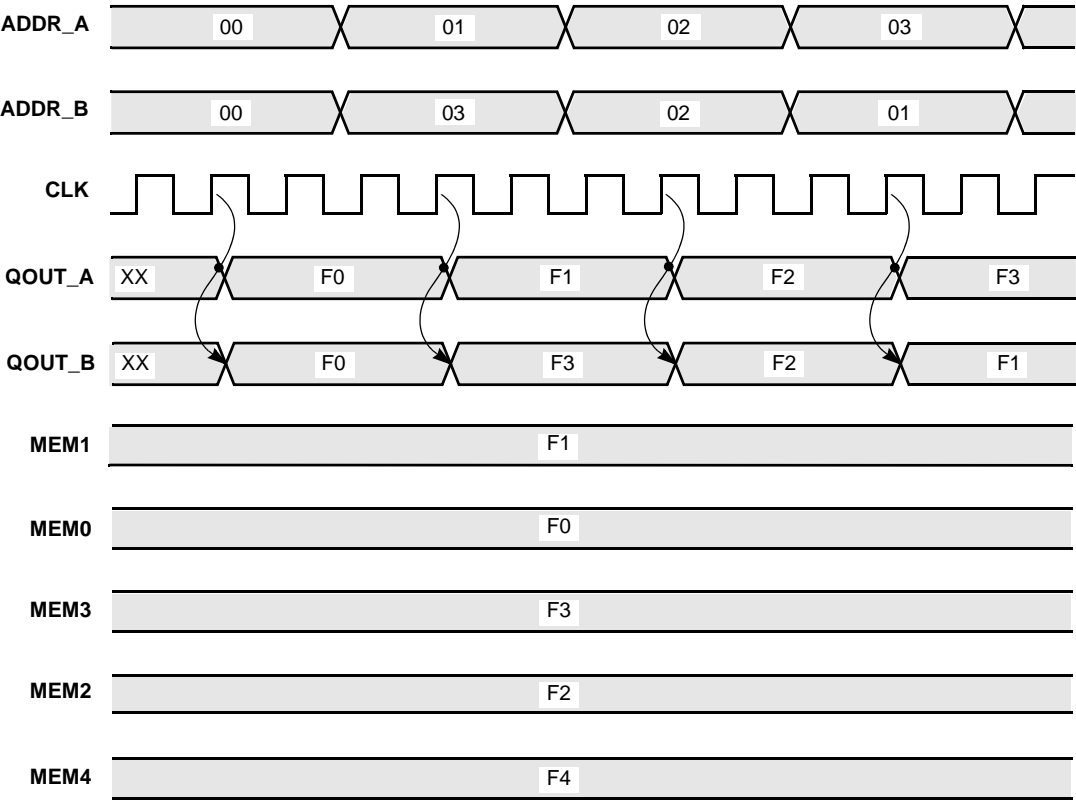
RADDR	00	03	02	01

CLK

QOUT	XX	F0	F3	F2	F1

MEM1	F1

MEM0	F0

MEM3	F3

MEM2	F2

MEM4	F4

Dual-Port Single Write

DATA	7A	FC	7F	FF

WREN

| WADDR | 00 | 01 | 02 | |

| RADDR | 00 | 03 | 02 | |

CLK

| QOUT | XX | F0 | 7A | 7F | FF |

| MEM1 | F1 | 7A |

| MEM0 | F0 |

| MEM3 | F3 |

| MEM2 | F2 | 7F | FF |

| MEM4 | F4 |

Dual-Port Read

ADDR_A	00	01	02	03
ADDR_B	00	03	02	01

CLK

QOUT_A	XX	F0	F1	F2	F3
QOUT_B	XX	F0	F3	F2	F1

MEM1   F1

MEM0   F0

MEM3   F3

MEM2   F2

MEM4   F4

Dual-Port Write

Signal	Values
DATA_A	7A, FC, 7F, FF
WREN_A	
ADDR_A	00, 01, 02
DATA_B	04, 4A, 4F, F4
WREN_B	
ADDR_B	00, 03, 02
CLK	
QOUT_A	XX, F0, 7A, 7F, FF
QOUT_B	XX, F0, 04, F3, XX
MEM1	F1, 7A
MEM0	F0, 7A
MEM3	F3
MEM2	F2, 7F, FF
MEM4	F4
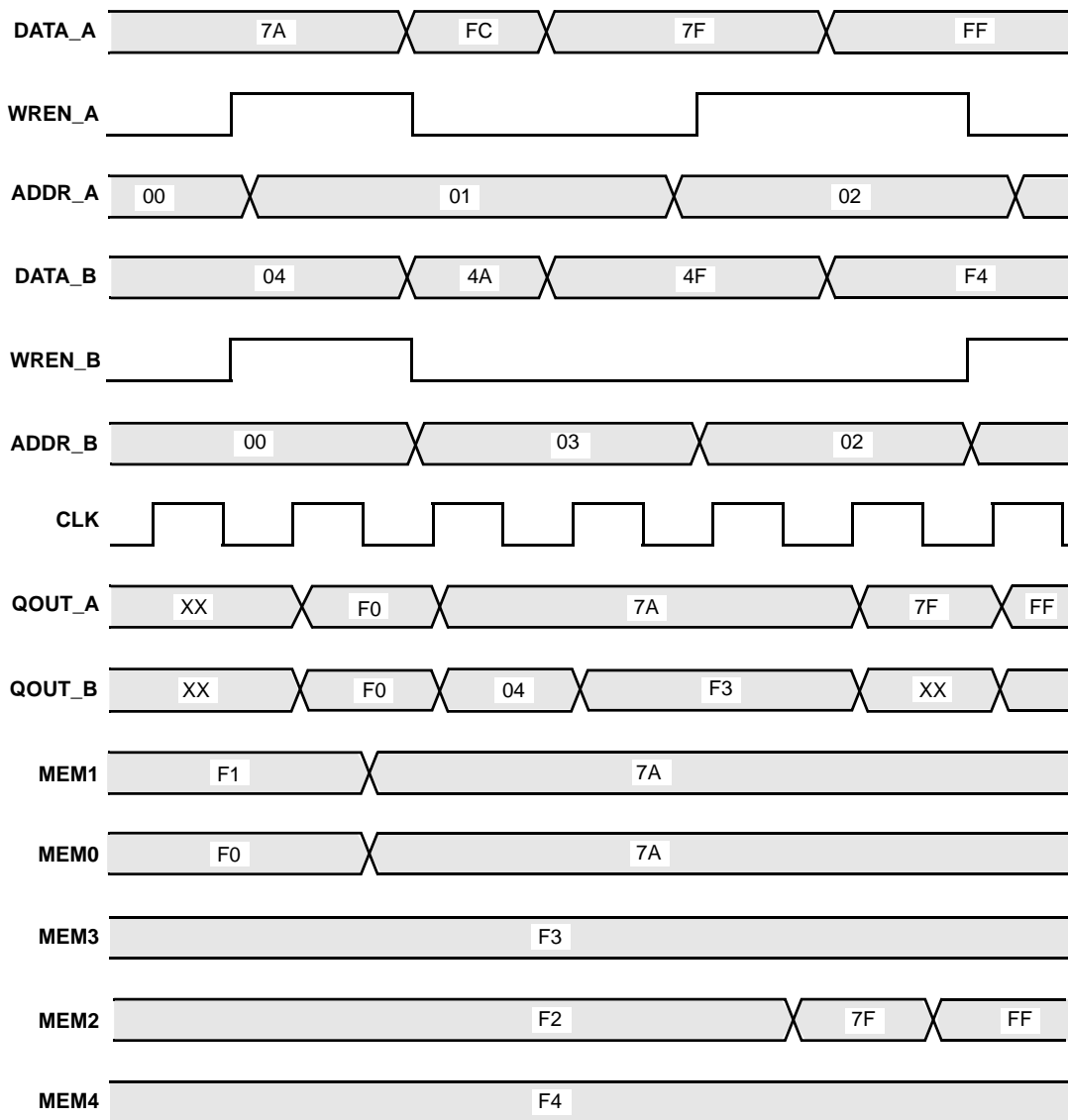
Read/Write Timing Sequences

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
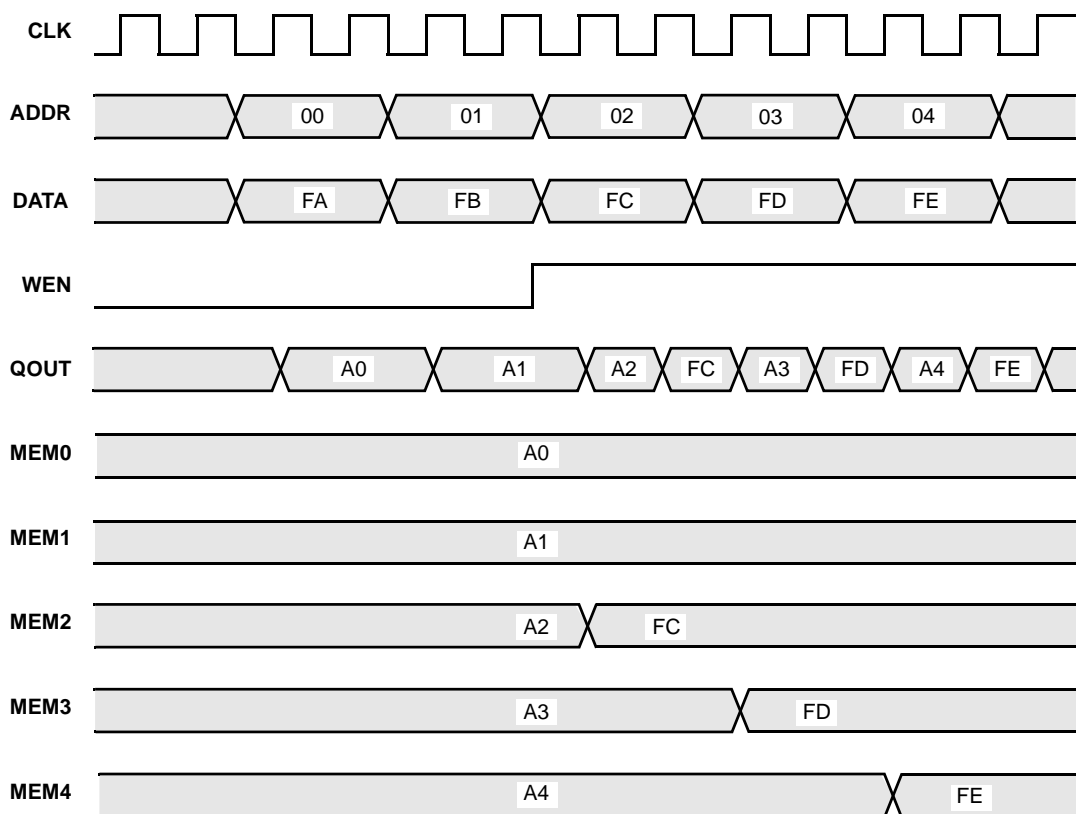
Write Before Read
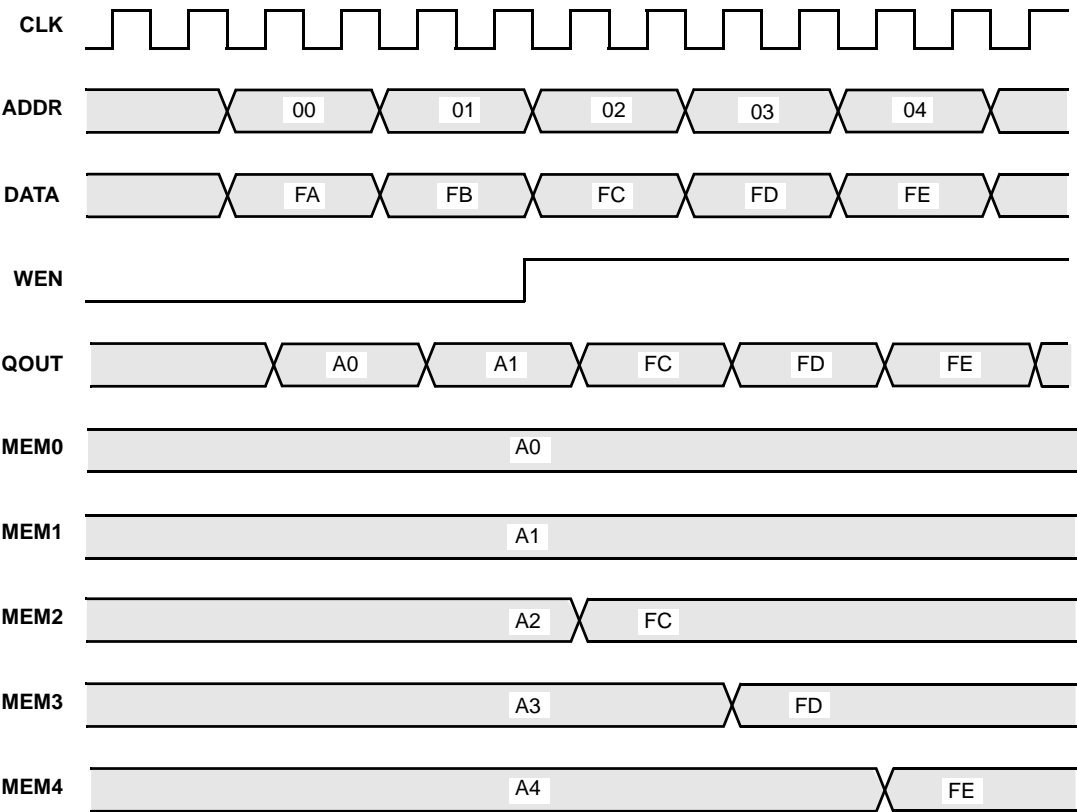
No Read on Write
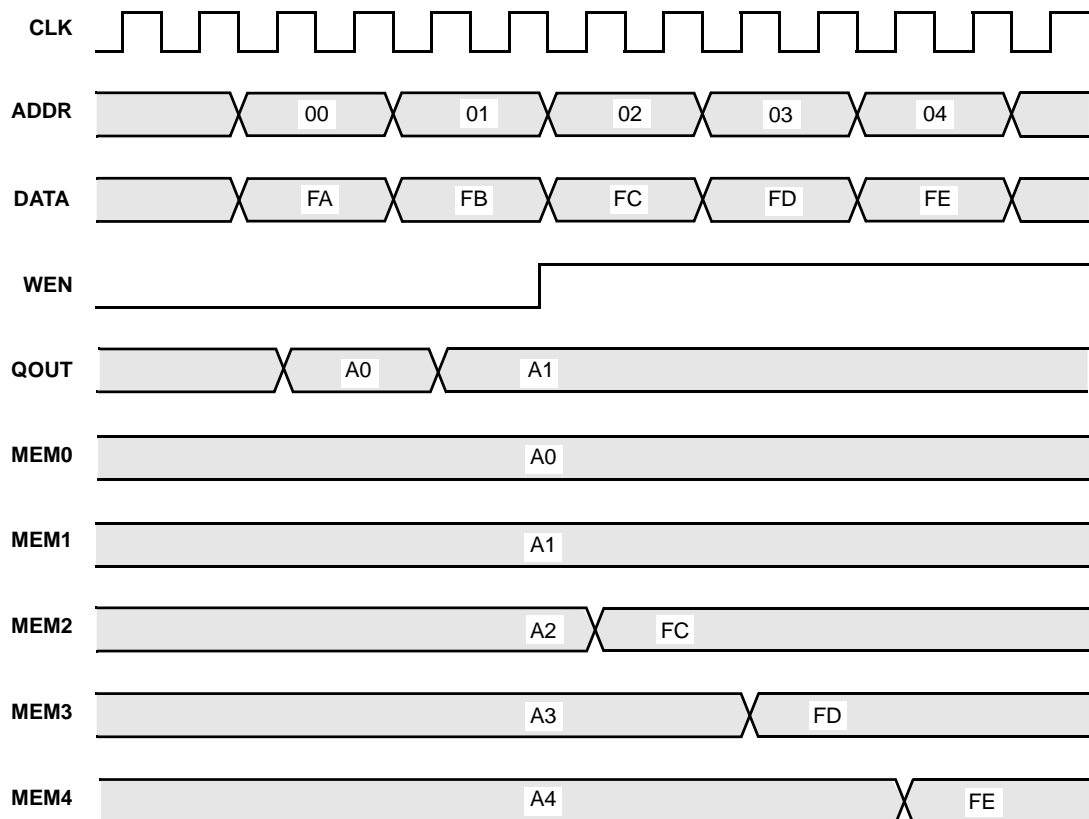
SYNCore Byte-Enable RAM Compiler

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 408 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 236 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.
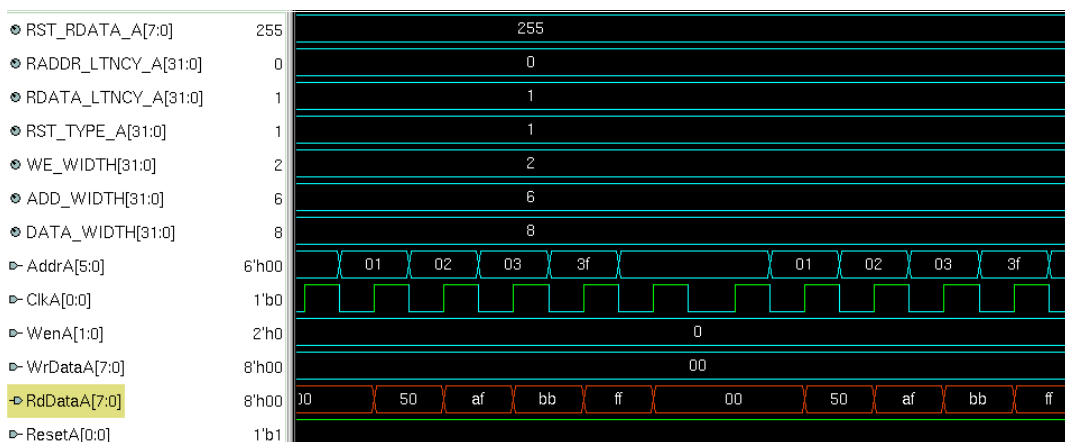
As shown in the above waveform, output read data changes on the same clock following the input address changed. When the address changes from 'h00 to 'h01, read data changes to 50 on the same clock, and data will be valid on the next clock edge.

The following waveform shows the read sequence with both the read data and address registered in single-port mode.
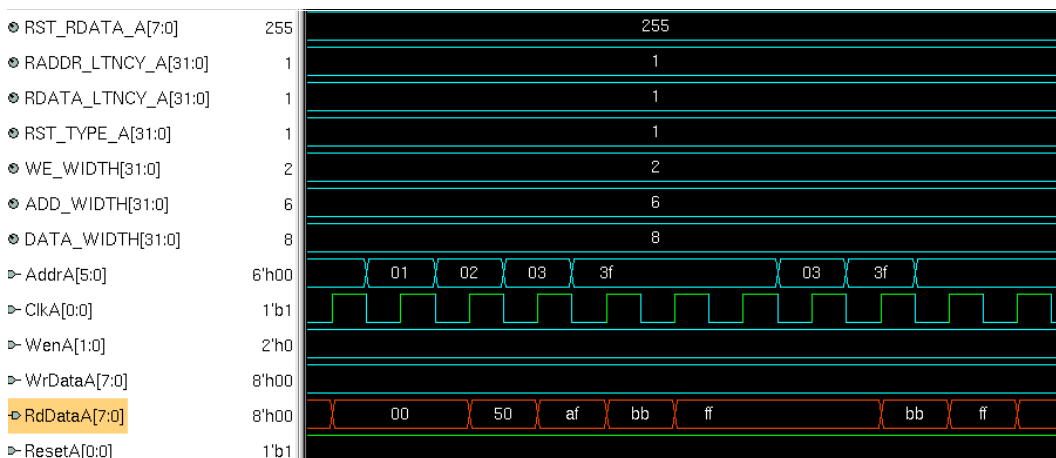
As shown in the above waveform, output read data changes on the next clock edge after the input address changes. When the address changes from 'h00 to 'h01, read data changes to 50 on the next clock, and data is valid on the next clock edge.

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.

◉ RST_RDATA_A[7:0]	255	255
◉ RADDR_LTNCY_A[31:0]	0	0
◉ RDATA_LTNCY_A[31:0]	1	1
◉ RST_TYPE_A[31:0]	1	1
◉ WE_WIDTH[31:0]	2	2
◉ ADD_WIDTH[31:0]	6	6
◉ DATA_WIDTH[31:0]	8	8
▷ AddrA[5:0]	6'h00	00 01 02 03 3f 00 01 02 03 3f
▷ ClkA[0:0]	1'b0	
▷ WenA[1:0]	2'h0	0 1 2 0 3 0
▷ WrDataA[7:0]	8'h00	00 aa ff 00
▷ RdDataA[7:0]	8'h00	00 55 ff bb 99 00 50 af bb ff
▷ ResetA[0:0]	1'b1	
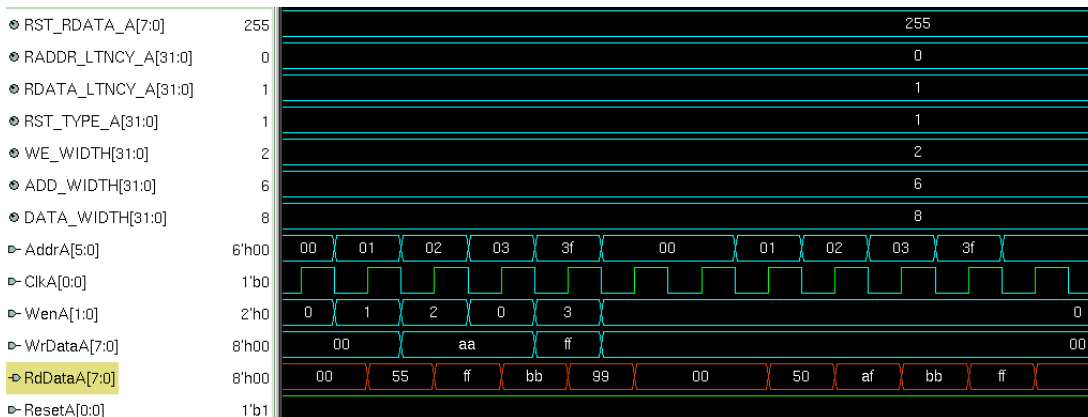
On each active edge of the clock when there is a change in address with an active enable, data is written into memory on the same clock. When enable is not active, any change in address or data is ignored. Active reset ignores any change in input address and data.

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 634
- Single-Port Read Operation, on page 636
- Dual-Port Read Operation, on page 637
- Parameter List, on page 637
- Clock Latency, on page 639

For further information, refer to the following:

- Specifying ROMs with SYNCore, on page 414 of the *User Guide*, for information about using the wizard to generate ROMs
- Launch SYNCore Command, on page 221 and SYNCore ROM Wizard, on page 239 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.
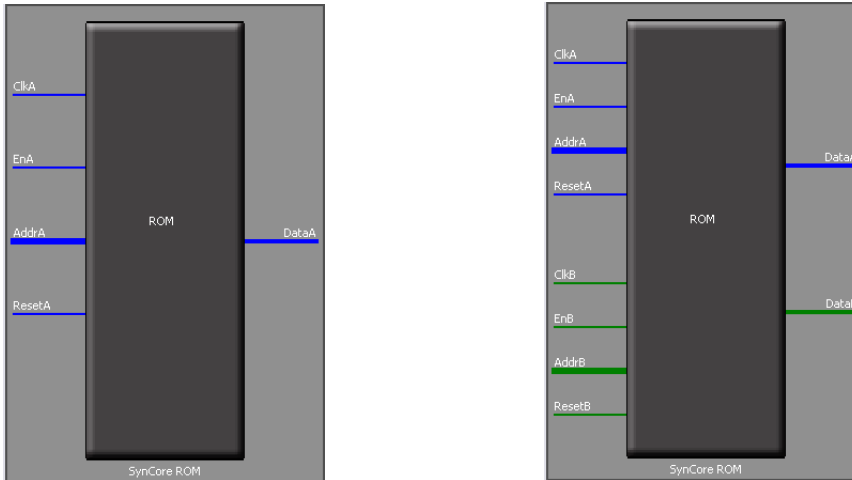
In the single-port (Port A) configuration, signals are synchronized to ClkA; ResetA can be synchronous or asynchronous depending on parameter selection. The read address (AddrA) and/or data output (DataA) can be registered to increase memory performance and improve timing. Both the read address and data output are subject to clock latency based on the ROM configuration (see Clock Latency, on page 639). 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 414 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.
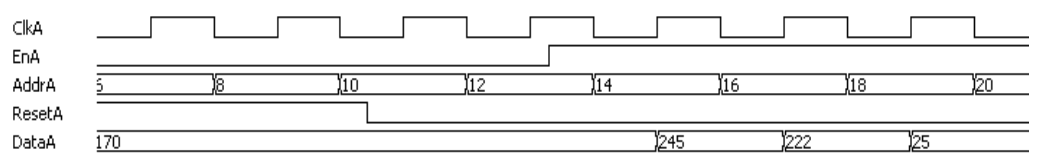
When reset is active, the output data holds the initialization value (i.e., 255). When reset goes inactive (and enable is active), data is read form the addressed location of ROM. Reset has priority over enable and always sets the output to the predefined initialization value. When both enable and reset 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 639.

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 414 in the *User Guide*). The following diagram shows the read timing for a dual-port ROM.
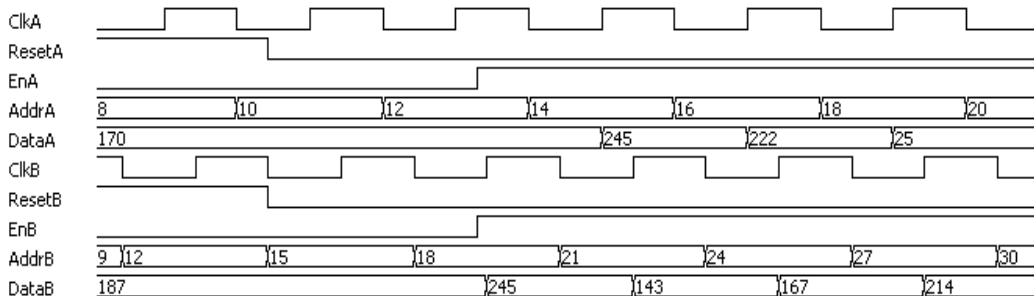
When either reset is active, the corresponding output data holds the initialization value (i.e., 255). When a reset goes inactive (and its enable is active), data is read form the addressed location of ROM. Reset has priority over enable and always sets the output to the predefined initialization value. 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 639.

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.
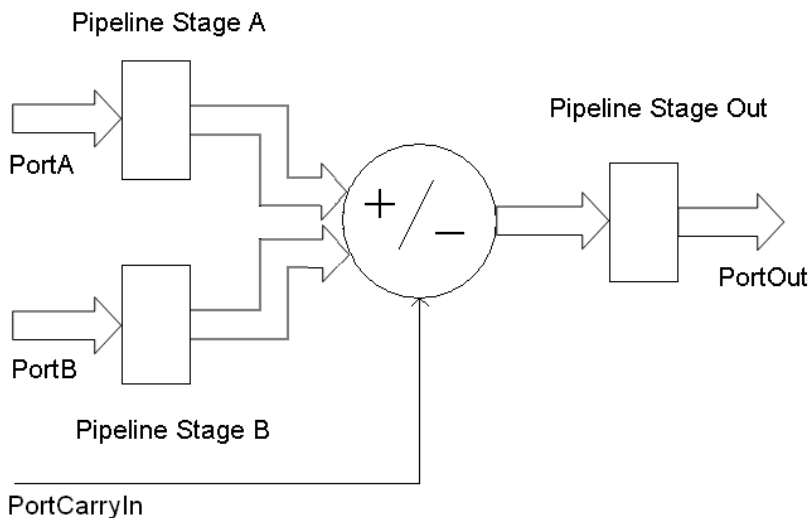
SYNCore adder/subtractor has ADD_N_SUB parameter, which can take three values ADD, SUB, or DYNAMIC. Based on this parameter value, the adder/subtractor can be configured as follows.

- 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'

PortA	0	4	9	13	5	1
PortB	0	1	3	5	2	5
PortOut	0	5	12	18	7	6

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'
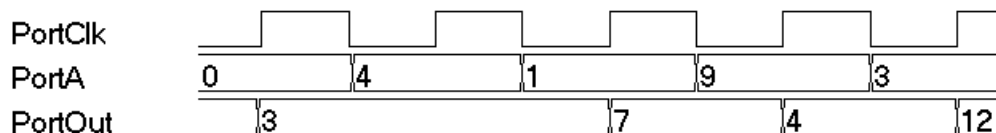
Subtractor

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'

PortClk						
PortA	0	4	9	13	5	
PortB	0	1	3	5	2	
PortOut	0		3	6	8	3
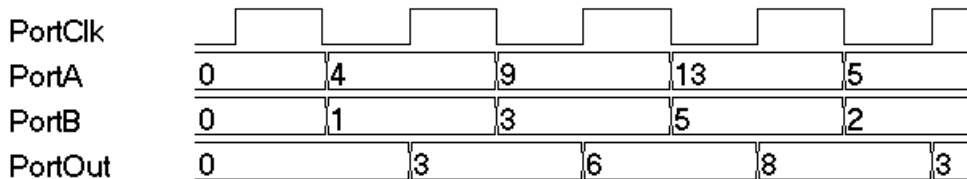
Subtractor with Pipeline Stages at Input and Output – In this mode, input PortA and PortB are pipelined and then subtracted, and the result is pipelined. The result is valid only at the second rising edge of the clock.

Parameters: PORTA_PIPELINE_STAGE= '1'
                     PORTB_PIPELINE_STAGE= '1'
                     PORTOUT_PIPELINE_STAGE= '1'

PortClk						
PortA	0	4	9	13	5	
PortB	0	1	3	5	2	
PortOut	0			3	6	8
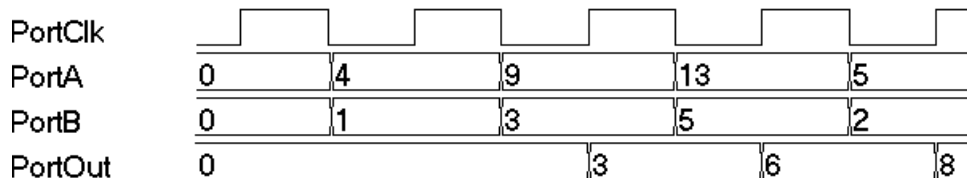
Subtractor with a Port Constant

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'

PortA	0	4	1	9	3
PortOut	0	3	0	8	2

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'

PortClk					
PortA	0	4	1	9	3
PortOut	0		3	0	8	2

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'
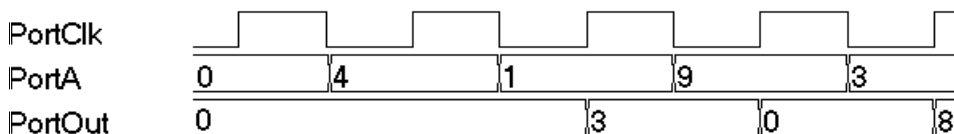
Dynamic Adder/Subtractor

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'
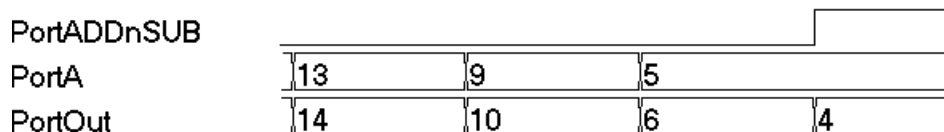
Dynamic Adder/Subtractor with Pipeline Stages at Input Only – In this mode, a constant value is either added or subtracted from the pipelined version of port A based on the value of port PortADDnSUB. Because there is no pipeline stage on the output port, the result changes immediately with respect to 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'
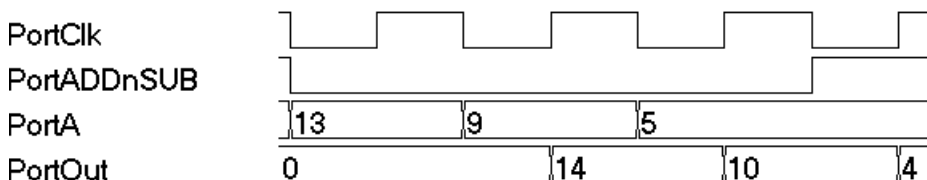
Dynamic Adder/Subtractor with Carry Input

The following waveform shows the behavior of the dynamic adder/subtractor with a carry input (the carry input is assumed to be 0).
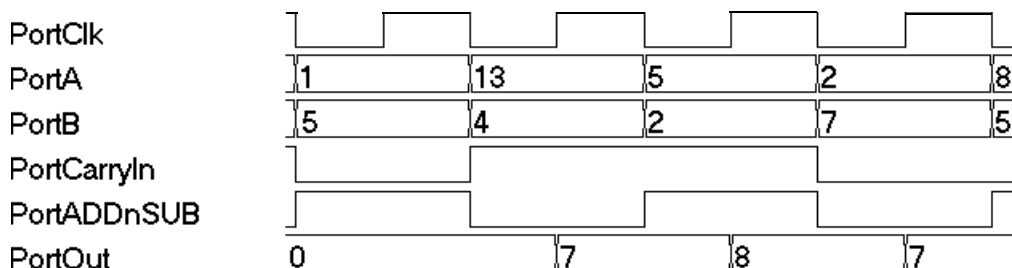
Dynamic Adder/Subtractor with Complete Control Signals

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.
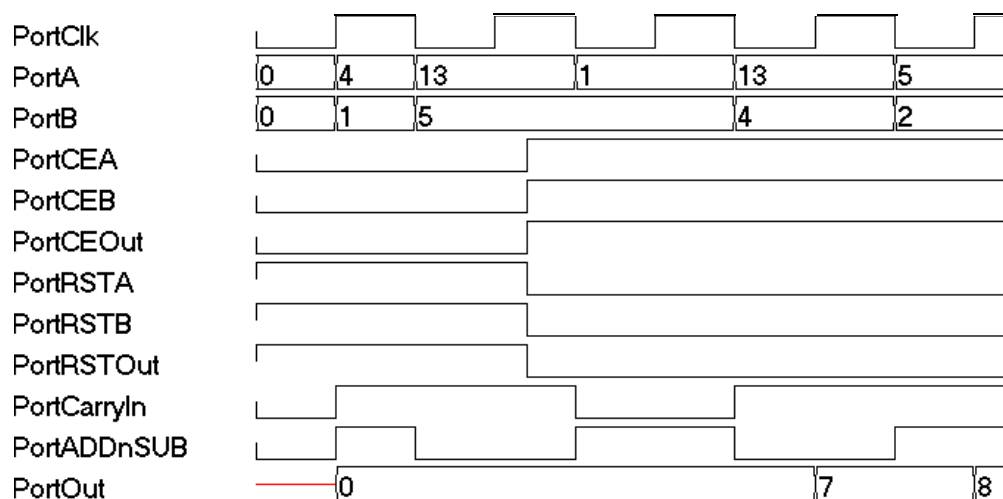
SYNCore Counter Compiler

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 652
- UP Counter Operation, on page 653
- Down Counter Operation, on page 653
- Dynamic Counter Operation, on page 654

For further information, refer to the following:

- Specifying Counters with SYNCore, on page 426 of the *User Guide,* for information about using the wizard to generate a counter core.
- Launch SYNCore Command, on page 221 and SYNCore Counter Wizard, on page 247 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 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'
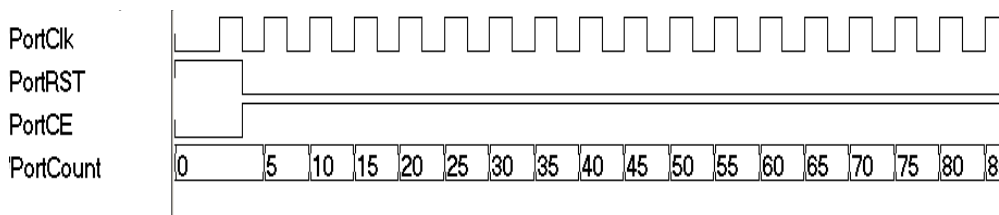
Note: Counter core can be configured to use a constant or dynamic load value in Up Counter mode (for the counter to load the PortLoadValue, PortCE must be active). This functionality is explained in .

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'

PortClk											
PortRST											
PortCE											
PortCount	53	48	43	38	33	28	23	18	13	8	3
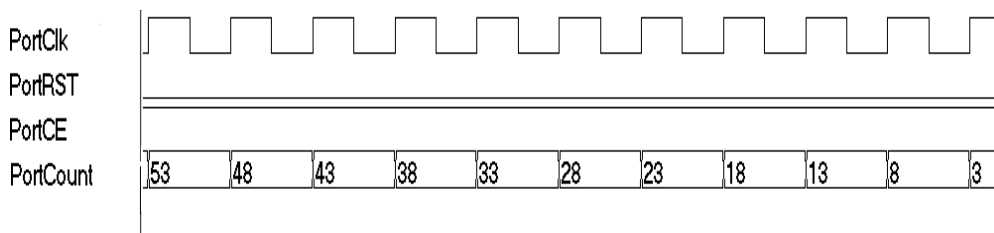
Note: Counter core can be configured to use a constant or dynamic load value in Down Counter mode (for the counter to load the PortLoadValue, PortCE must be active). This functionality is explained in .

Dynamic Counter Operation

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'
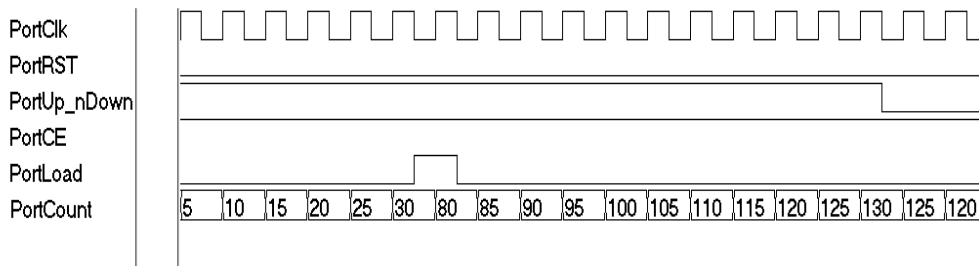
Note: *For counter to load the PortLoadValue, PortCE must be active.

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'
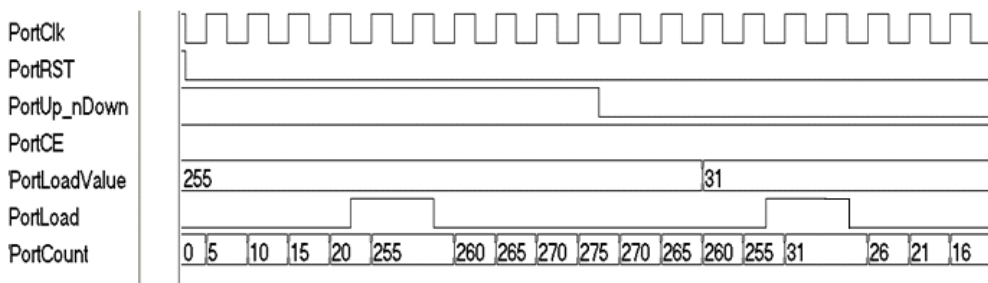
Note: ***** For counter to load the PortLoadValue, PortCE should be
active.

Encryption Scripts

There are two FPGA encryption methods available to Synopsys FPGA synthesis tool users: P1735, and OpenIP. With both encryption methods, the IP vendor can encrypt their IP from their own website. From the synthesis tool, the synthesis user has access o the IP that the vendor makes available for download and evaluation within a synthesis design.

Each encryption method has corresponding scripts that the user can run. The following sections provide an overview of encryption and decryption methodologies and descriptions of the two encryption scripts:

Encryption and Decryption Methodologies

This section describes common encryption schemes. There are two major classes of encryption/decryption algorithms: symmetric, and asymmetric.

Symmetric Encryption
With this kind of encryption, a special number is used as a key to encrypt the files. The same key is used to decrypt the file, so you must have access to the same key.
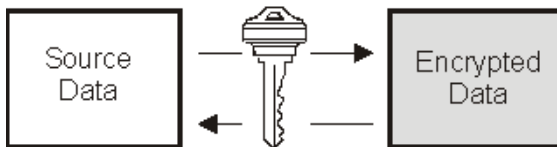
These are some examples of this kind of algorithm: Data Encryption Standard (DES); Triple DES or TDES or TDEA (Triple Data Encryption Algorithm), which uses DES three times; and Advanced Encryption Standard (AES).

- Asymmetric Encryption
  This encryption scheme uses different keys to encode and decode data. The end user generates the keys and makes a public key to everyone

who needs it for encryption. The public key cannot be used for decryp-
tion. The end user uses the private key to decrypt the data. Examples of
asymmetric encryption are DH (Diffie-Hellman) and RSA (Rivest,
Shamir, and Adelman).
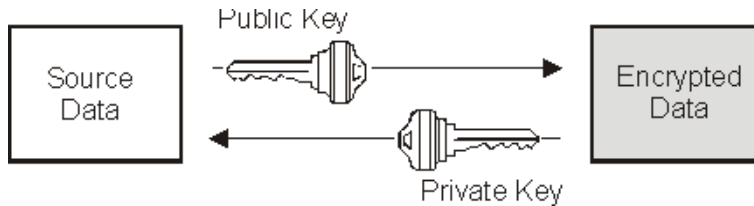
The encryptP1735 Script

The encryptP1735 script is available to IP vendors who wish to provide evalu-
ation IP to their synthesis users. The script is a Perl script that uses the
P1735 standard to let IP vendors encrypt modules or components, which can
then be downloaded for evaluation or use by a Synopsys FPGA user. The
script can be run according to any of three use models to encrypt the
associated RTL files (see Encrypting IP with the encryptP1735.pl Script, on
page 440 in the *User Guide*).

You run the script with the encryptP1735 command, the complete syntax for
which is described in encryptP1735, on page 37 in the *Command Reference*
manual.

The following sections describe details of the encrypP1735 script files:

- Input Files for the P1735 Encryption Script, on page 659
- Public Keys Repository File, on page 659
- Pragmas for the P1735 Encryption Script, on page 659
- Adding Multiple Keys, on page 660

Input Files for the P1735 Encryption Script

The encryptP1735 encryption script reads an HDL file, with or without encryption attributes, according to the selected use model. Additionally, the script reads the keys repository file that contains the public keys for the IP consumer tools.

Public Keys Repository File

The encryptP1735 encryption script requires public keys from the default keys.txt file from the directory *installLocation*/lib to create the decryption envelope. This file includes public keys for each of the tools that require a key block in the encrypted file. The public keys file includes a Synopsys synthesis tool public key; the file can be expanded by the user to include public keys for other tools.

Pragmas for the P1735 Encryption Script

The header blocks in the encryptP1735.pl script support the pragmas described in the following table.

Pragma Keyword	Description
begin	Opens a new encryption envelope
end	Closes an encryption envelope
begin_protected	Opens a new decryption envelope
end_protected	Closes a decryption envelope
author	Identifies the author of an envelope
author_info	Specifies additional author information
encoding	Specifies the coding scheme for the encrypted data
data_keyowner	Identifies the owner of the data encryption key
data_method	Identifies the data encryption algorithm
data_keyname	Specifies the name of the data encryption key
data_block	Begins an encoded block of encrypted data
encrypt_agent	Identifies the encryption service

Pragma Keyword	Description
encrypt_agent_info	Specifies additional encryption-agent information
key_keyowner	Identifies the owner of the key encryption key
key_method	Specifies the key encryption algorithm
key_keyname	Specifies the name of the key encryption key
key_public_key	Specifies the public key for key encryption
key_block	Begins an encoded block of key data
version	P1735 encryption version
comment	Uninterrupted documentation string

Adding Multiple Keys

It may become necessary to add multiple keys to the RTL to support how
multiple vendors access to the same RTL. Multiple vendor access is done by
editing the Synopsys synthesis tool public key included in the
install/lib/keys.txt key file shown below:

```
// Use verilog pragma syntax in this file

`pragma protect version=1
`pragma protect author="default"
`pragma protect author_info="default"

`pragma protect key_keyowner="Synplicity", key_keyname="SYNP05_001", key_method="rsa"
`pragma protect key_public_key

MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAybsQaMidiCHZyh14wbXn
UpP8lK+jJY5oLpGqDfSW5PMXBVp0WFd1d32onXEpRkwxEJLlK4RgS43d0FG2ZQ1l
irdimRKNnUtPxsrJzbMr74MQkwmG/X7SEe/lEqwK9Uk77cMEncLycI5yX4f/K9Q9
WS5nLD+Nh6BL7kwR0vSevfePC1fkOa1uC7b7Mwb1mcqCLBBRP9/eF0wUIoxVRzjA
+pJvORwhYtZEhnwvTblBJsnyneT1LfDi/D5WZoikTP/0KBiP87QHMSuVBydMA7J7
g6sxKB92hx2Dpv1ojds1Y5ywjxFxOAA93nFjmLsJq3i/P0lv5TmtnCYX3Wkryw4B
eQIDAQAB

// Add additional public keys below this line
// Add additional public keys above this line

`pragma protect data_keyowner="default-ip-vendor"
`pragma protect data_keyname="default-ip-key"
`pragma protect data_method="aes128-cbc"

// End of file
```

The file is expanded to include public keys for other tools. Please add any
other key between the lines:

```
                // Add additional public keys below this line

                // Add additional public keys above this line
```

The following is an example of an expanded public keys file that contains dummy keys with key_keyname ="DUMMY":

```
// Use verilog pragma syntax in this file

`pragma protect version=1
`pragma protect author="default"
`pragma protect author_info="default"

`pragma protect key_keyowner="Synplicity", key_keyname="SYNP05_001", key_method="rsa"
`pragma protect key_public_key

MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAybsQaMidiCHZyh14wbXn
UpP8lK+jJY5oLpGqDfSW5PMXBVp0WFd1d32onXEpRkwxEJLlK4RgS43d0FG2ZQ1l
irdimRKNnUtPxsrJzbMr74MQkwmG/X7SEe/lEqwK9Uk77cMEncLycI5yX4f/K9Q9
WS5nLD+Nh6BL7kwR0vSevfePC1fkOa1uC7b7Mwb1mcqCLBBRP9/eF0wUIoxVRzjA
+pJvORwhYtZEhnwvTblBJsnyneT1LfDi/D5WZoikTP/0KBiP87QHMSuVBydMA7J7
g6sxKB92hx2Dpv1ojds1Y5ywjxFxOAA93nFjmLsJq3i/P0lv5TmtnCYX3Wkryw4B
eQIDAQAB

// Add additional public keys below this line

`pragma protect key_keyowner="Synopsys", key_keyname="DUMMY", key_method="rsa"
`pragma protect key_public_key

…AAAAAAAA…

// Add additional public keys above this line

`pragma protect data_keyowner="default-ip-vendor"
`pragma protect data_keyname="default-ip-key"
`pragma protect data_method="aes128-cbc"

// End of file
```

If you are using a partial file with all pragmas use model, the keyowner entries also must be edited:

```
`protect key_keyowner="Synplicity", key_keyname="SYNP05_001", key_method="rsa", key_block
`protect key_keyowner=" Synopsys ", key_keyname=" DUMMY ", key_method="rsa", key_block
`protect data_keyowner="ip-vendor-a", data_keyname="fpga-ip", data_method="des-cbc"
```

Be sure to include the below and above entries within your RTL to be able to generate the decryption envelope properly. It is very important that you follow this process when using the partial file with all pragmas use model. If you are using the full-file use model or partial file with minimal pragmas use model, editing the RTL is unnecessary when adding additional keys to the default public key to create expanded keys.

Note: The expanded key shown is only an example and is not intended
be used with the encryptP1735.pl script; the actual key must be
obtained from a valid EDA vendor and added to the keys.txt file.

The encryptIP Script

The encryptIP Perl script is a Synopsys FPGA IP script that is provided to IP
vendors who wish to provide evaluation IP to synthesis users. The script uses
the OpenIP scheme to encrypt modules or components, which can then be
downloaded for evaluation or use by the Synopsys FPGA user. You can
download the encryptIP Perl script from SolvNet
(https://solvnet.synopsys.com/retrieve/032343.html).

You run the script with the encryptIP command, the complete syntax for which
is described in encryptIP, on page 33 in the *Command Reference* manual.

For details, see the following:

- The encryptIP Script Run, on page 662
- Pragmas Used in the encryptIP Script, on page 664

The encryptIP Script Run

The following example describes the various steps that the encryptIP script
executes. For descriptions of the pragmas used in the encryptIP script,
Pragmas Used in the encryptIP Script, on page 664.

1. For each RTL file, the script creates a data block using symmetric
   algorithm and your own session key.

   You can use any of the CBC encryption modes listed in encryptIP, on
   page 33 in the *Command Reference* Manual. The initialization vector is a
   constant, and the block is encoded in base 64. The following excerpt
   uses the data encryption key ABCDEFG:

```
%%% protect data_block
UWhcm3CPmGz27DXAWQZF8rY7hSsvLwedXiP59HYZHJfoMIMkJ0W+6H7vmJEXZ/
    dTxnAgGB2YJCF7lsaZ6x6kRisdtBtIo8+0Glskcykt7FtpAjpz24cJ9hoSYMu
    HCmG70dHDNzTHWjkwBs2fxo5S6559d3pW+SDutrvrsHntyvHYiqxUZPsGce
    ZZQJpqQIpqo24uFCVHNSX/URvL47CUBWoKB2XEpyRv5Zgd1F52YjBLIpET
    +kBEzutAorF5rD9eZSALSo0kvVb7MPXFxmfCF8wHwTnRtkPthNCMq0t3iCgf9EH
```

2. The script precedes the data block with a small data block header that describes the data method:

```
%%% protect data_method=aes128-cbc
%%% protect data_block
UWhcm3CPmGz27DXAWQZF8rY7hSsvLwedXiP59HYZHJfoMIMkJ0W+6H7vmJEXZ/
    dTxnAgGB2YJCF7lsaZ6x6kRisdtBtIo8+0Glskcykt7FtpAjpz24cJ9hoSYMu
    HCmG70dHDNzTHWjkwBs2fxo5S6559d3pW+SDutrvrsHntyvHYiqxUZPsGce
    ZZQJpqQIpqo24uFCVHNSX/URvL47CUBWoKB2XEpyRv5Zgd1F52YjBLIpET
    +kBEzutAorF5rD9eZSALSo0kvVb7MPXFxmfCF8wHwTnRtkPthNCMq0t3iCgf9EH
```

3. The script then prepares a key block for the tool, which contains your session key and Synplicity-specific directives. Note that the data_decrypt_key is required.

```
output_method=blackbox  (Your IP will be a black box in the output netlist)
data_decrypt_key=ABCDEFG  (Session key you used to encrypt your data block)
```

See encryptIP, on page 33 in the *Command Reference* Manual for information about output methods.

4. Use Synplify Public key (Synplicity has an executable that returns this key) and RSA2048 asymmetric encryption to create a key block. Encode it in base64.

```
%%% protect key_block
U9n263KwF7RWb8GSz7C+700tKshqQgTmb8UdRxISekIJDfonqfqzjzEQ+xQ4
    wyh65wo6X56Jm+ClaVuZjgQKK0c4y47nyA1iWcuq1Nh6KeuUscxp+nL6yT9Am
    +nv+c57jSCMG0QsFbRBAIhdlohQAbYbSIuFxdLFEFxW4znF3+YDAsMHeIs
    1tqxKqhQzYQ2fGJdQz0NVRi1hFjx/RpGmoXmzvSTX2xsre+ZNDh3r9qvj37
    QGwLH2erPt/iXcUVnlnPCOaV5z8M1YLrKY8ui7KBs/HhyP7L2mAMPQAFY3i
    DhycUcJ5sirBgKZycpkhP8jQ02yjTZMb7z9KyYTHrzDdA==
```

```
%%% protect key_block
+700tKshqQgTmb8UdRU9n263KwF7RWb8GSz7Cwo6X56Jm+ClxISekIJDfon
    qfqzjzEQ+xQ4wyh657nyA1iWcuq1Nh6KeuUscxp+nL6yT9AaVuZjG0QsFb
    RBAIhdlohQAbYbgQKK0c4y4m+nv+c57jSCMxW4znF3+YDAsMHeIs1tqx
    KqhQzYQ2fGJdQz0NXmzvSTX2xsre+ZNVRi1hFjSIuFxdLFEFx/RpGmo9qvj
    37QGwLH2erPt/iXcUVnlnPCO7KBs/HhyP7L2mAMPQAFY3iDhycUcJaV5z8MD
    h3r1YLrKY8ui8jQ02yjTZMb7z9pkhPYTHrzDdAKy5sirBgKZyc==
```

5. The script adds a small key block header to each key block.

```
%%% protect key_keyowner=Synplicity
%%% protect key_keyname=SYNP05_001
%%% protect key_block
U9n263KwF7RWb8GSz7C+700tKshqQgTmb8UdRxISekIJDfonqfqzjzEQ+xQ4
```

6. Your final encrypted IP key and data blocks looks like this:

```
%%% protect protected_file 1.0
<optional unencrypted HDL>
%%% protect begin_protected
%%% protect key_keyowner=Synplicity
%%% protect key_keyname=SYNP05_001

%%% protect key_block
U9n263KwF7RWb8GSz7C+700tKshqQgTmb8UdRxISekIJDfonqfqzjzEQ+xQ4
    wyh65wo6X56Jm+ClaVuZjgQKK0c4y47nyA1iWcuq1Nh6KeuUscxp+nL6yT9Am
    +nv+c57jSCMG0QsFbRBAIhdlohQAbYbSIuFxdLFEFxW4znF3+YDAsMHeIs
    1tqxKqhQzYQ2fGJdQz0NVRi1hFjx/RpGmoXmzvSTX2xsre+ZNDh3r9qvj37
    QGwLH2erPt/iXcUVnlnPCOaV5z8M1YLrKY8ui7KBs/HhyP7L2mAMPQAFY3i
    DhycUcJ5sirBgKZycpkhP8jQ02yjTZMb7z9KyYTHrzDdA==

<other key blocks>

%%% protect data_method=aes128-cbc
%%% protect data_block
UWhcm3CPmGz27DXAWQZF8rY7hSsvLwedXiP59HYZHJfoMIMkJ0W+6H7vmJEXZ/
    dTxnAgGB2YJCF7lsaZ6x6kRisdtBtIo8+0Gls/kcykt7FtpAjpz24cJ9ho
    SYMuHCmG70dHDNzTHWjkwBs2fxo5S6559d3pW+SDutrvrsHntyv
    HYiqxUZPsGceZZQJpqQIpqo24uFCVHNSX/URvL47CUBWoKB2XEpyRv5Zg
    …

%%% protect end_protected
<optional unencrypted HDL>
```

Pragmas Used in the encryptIP Script

The header blocks in the encryptIP script use the pragmas described in the following tables. Note the following:

- The %%% protect directive must be placed at the exact beginning of a line.

- Exactly one white-space character must separate the %%% from the command that follows

The following table describes the general pragmas used:

%%% protect protected_file 1.0	Line 1 of protected file
%%% protect begin_protected	Begin protected section
%%% protect end_protected	Ends protected section

%%% protect comment *comment*	Single line plain-text comment
%%% protect begin_comment	Begin block of plain-text comments
%%% protect end_comment	End block of plain-text block comments

The following table describes the data block pragmas:

%%% protect author=*string*	Arbitrary string
%%% protect data_method=*string*	For all supported FPGA vendors, one of the DES encryption methods described in The encryptP1735 Script, on page 658.
%%% protect data_block	Immediately precedes encrypted data block

The following table describes the key block pragmas:

%%% protect key_keyowner=*string*	Arbitrary string
%%% protect key_keyname=*string*	Name recognized by the Synplify software to select key block
%%% protect key_method=*string*	Encryption algorithm. Currently we support "rsa"
w %%% protect key_block	Immediately precedes encrypted data block

C H A P T E R   1 4

Scripts

This chapter describes Tcl commands and scripts.

- synhooks File Syntax, on page 668
- Tcl Script Examples, on page 670

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 471.

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 below.
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]
}
```

Tcl Script Examples

This section provides the following examples of Tcl scripts:

Using Target Technologies

```
# Run synthesis multiple times without exiting while trying different
# target technologies. View the 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 {

        ProASIC3 ProASIC3E Fusion # list of technologies
   }

# Loop through synthesis for each target technology.
   foreach technology $try_these {
           impl -add
           set_option -technology $technology
           project -run -fg
           open_file -rtl_view
   }
```

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 676
- Microsemi Features, on page 676
- Synthesis Constraints and Attributes for Microsemi, on page 677

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 192).

The following technologies are supported:

Mixed-Signal FPGAs
- SmartFusion2 and SmartFusion
- Fusion

Low-Power FPGAs
- IGLOO Series (IGLOO2, IGLOO, IGLOO+, and IGLOOE)
- ProASIC3 Series (ProASIC3, ProASIC3E, and ProASIC3L)

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 699 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 677.

Microsemi I/O Standards

The following table lists the supported I/O standards for the ProASIC3L, ProASIC3E, Fusion, and IGLOOe families, ProASIC3, IGLOO, and IGLOO+ families. 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.

ProASIC3L, ProASIC3E, IGLOOe, Fusion	IGLOO, IGLOO+, ProASIC3
GTL25	LVCMOS_12
GTL+25	LVCMOS_15
GTL33	LVCMOS_18
GTL+33	LVCMOS_33
HSTL_Class_I	LVCMOS_5
HSTL_Class_II	LVDS
LVCMOS_12	LVPECL
LVCMOS_15	LVTTL
LVCMOS_18	PCI
LVCMOS_33	PCIX

ProASIC3L, ProASIC3E, IGLOOe, Fusion	IGLOO, IGLOO+, ProASIC3
LVCMOS_5	
LVDS	
LVPECL	
LVTTL	
PCI	
PCIX	
SSTL_2_Class_I	
SSTL_2_Class_II	
SSTL_3_Class_I	
SSTL_3_Class_II	

See Also:

- Industry I/O Standards, on page 190 for a list of industry I/O standards.

- Microsemi Attribute and Directive Summary, on page 713 for a list of Microsemi attributes and directives.

Microsemi Components

The following topics describe how the synthesis tool handles 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 679

- DSP Block Inference, on page 681

- Microsemi RAM Implementations, on page 685

- Instantiating RAMs with SYNCORE, on page 693

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

- Microsemi Fusion Analog Blocks

- SmartFusion Macros

- SmartFusion2 MACC Block

For general information on instantiating black boxes, see Instantiating Black Boxes in VHDL, on page 552, and Instantiating Black Boxes in Verilog, on page 357. 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 294

- Using Predefined Microsemi Black Boxes, on page 480

- Using Smartgen Macros, on page 481

---

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 ProASIC3, ProASIC3E, ProASIC3L, IGLOO, IGLOOe, IGLOO+, SmartFusion, and Fusion devices.

MATH18X18 Block

The synthesis software supports instantiation of the MATH18X18 block. The MATH18X18 block is useful for mapping arithmetic functions.

Microsemi Fusion Analog Blocks

Microsemi Fusion has several analog blocks built into the ProASIC3E device. The synthesis tool treats them as black boxes. The following is a list of the available analog blocks.

- AB
- NVM
- XTLOSC
- RCOSC
- CLKSRC
- NGMUX
- VRPSM
- INBUF_A
- INBUF_DA
- OUTBUF_A
- CLKDIVDLY
- CLKDIVDLY1

SmartFusion Macros

The synthesis software supports the following SmartFusion macros:

- FAB_CCC
- FAB_CCC_DYN

SmartFusion2 MACC Block

SmartFusion2 devices support bit-signed 18x18 multiply-accumulate blocks. This architecture provides dedicated components called SmartFusion2 MACC blocks, for which DSP-related operations can be performed like multiplication followed by addition, multiplication followed by subtraction, and multiplication with accumulate. For more information, see DSP Block Inference, on page 681.

DSP Block Inference

This feature allows the synthesis tools to infer DSP or MATH18x18 blocks for SmartFusion2 devices only. The following structures are supported:

- DOTP Support

  MACC block, when configured in DOTP mode, has two independent signed 9-bit x 9-bit multipliers followed by addition. The sum of the dual independent 9x9 multiplier (DOTP) result is stored in the upper 35 bits of the 44-bit output. In DOTP mode, the MACC block implements the following equation:

  $P = D + (CARRYIN + C) + 512 * ((AL * BH) + (AH * BL))$, when SUB = 0

  $P = D + (CARRYIN + C) - 512 * ((AL * BH) + (AH * BL))$, when SUB = 1

  Below is an example RTL which infers MACC block in DOTP mode after synthesis:

  ```
  module dotp_add_unsign_syn ( ina, inb, inc, ind, ine, dout);

  parameter widtha = 6;
  parameter widthb = 7;
  parameter widthc = 7;
  parameter widthd = 8;
  ```

```
parameter widthe = 30;
parameter width_out = 44;

input [widtha-1:0] ina;
input [widthb-1:0] inb;
input [widthc-1:0] inc;
input [widthd-1:0] ind;
input [widthe-1:0] ine;
output reg [width_out-1:0] dout;

always @(ina or inb or inc or ind or ine)  begin

        dout      <= (ina * inb) + (inc * ind) + ine ;

end

endmodule
```

MACC block does not support DOTP mode when:

– Width of the multiplier inputs is greater than 9-bits for signed.

– Width of the multiplier inputs is greater than 8-bits for unsigned.

– Width of the non-multiplier inputs is greater than 36-bits.

• 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, on page 96 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 (MACC) 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:

- Overflow extraction

- Arithmetic right shift for operand C

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:

- RAM Read Enable Extraction
- ProASIC3/3E/3L and IGLOO+/IGLOO/IGLOOe
- SmartFusion2

RAM Read Enable Extraction

RAM Read Enable extraction currently supports RAMs for output registers, with an enable. This feature is available for ProASIC3E 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
```
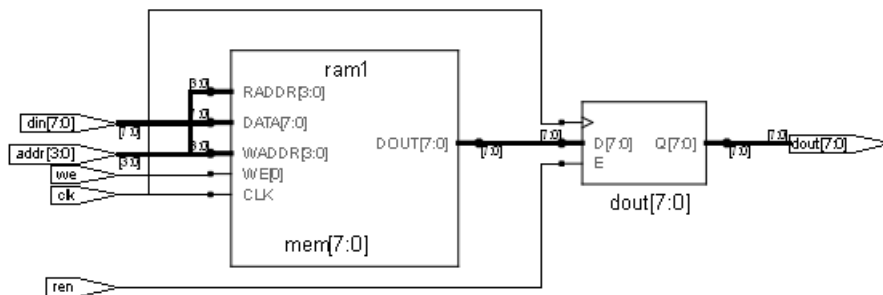
ProASIC3/3E/3L and IGLOO+/IGLOO/IGLOOe

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 ProASIC3/3E/3L or
IGLOO+/IGLOO/IGLOOe, 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;
```
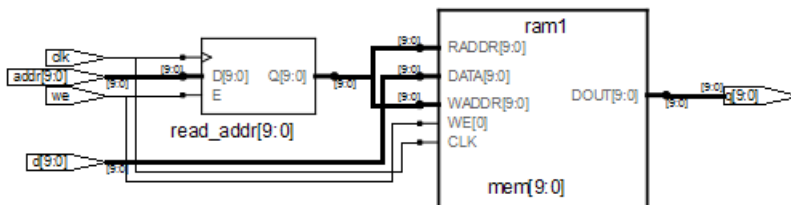
The following is an example of the RAM512X18 configuration:

```
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;
```
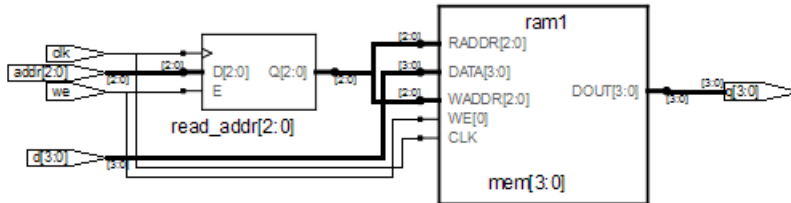

SmartFusion2

SmartFusion2 devices support two types of RAM macros: RAM1K18 and RAM64X18. The synthesis software extracts the RAM structure from the RTL and infers RAM1K18 or RAM64X18 based on the size of the RAM.

The default criteria for specifying the macro is described in the table below for the following RAM types.

True Dual-Port Synchronous Read Memory	The synthesis tool maps to RAM1K18, regardless of its memory size.

Simple Dual-Port or Single-Port Synchronous Memory	If the size of the memory is: • 4608 bits or greater, the synthesis tool maps to RAM1K18. • Greater than 12 bits and less than 4608 bits, the synthesis tool maps to RAM64X18. • Less than or equal to 12 bits, the synthesis tool maps to registers.
Simple Dual-Port or Single-Port Asynchronous Memory	When the size of the memory is 12 bits or greater, the synthesis tool maps to RAM64x18. Otherwise, it maps to registers.
Three Port RAM Inference Support	This feature is supported on SmartFusion2 and IGLOO2 devices only. RAM64x18 is a 3-port memory providing one Write port and two Read ports. Write operation is synchronous while read operations can be asynchronous or synchronous. The tool infers RAM64X18 for such structures.

You can override the default behavior by applying the syn_ramstyle attribute to control how the memory gets mapped. To map to

- RAM1K18 set syn_ramstyle = "lsram"
- RAM64X18 set syn_ramstyle = "uram"
- Registers set syn_ramstyle = "registers"

The value you set for this attribute always overrides the default behavior.

Three Port RAM inference support

Example 1) Three Port RAM Verilog Example—Synchronous Read

```
module
ram_infer15_rtl(clk,dinc,douta,doutb,wrc,rda,rdb,addra,addrb,addrc
);
input clk;
input [17:0] dinc;
input wrc,rda,rdb;
input [5:0] addra,addrb,addrc;
output [17:0] douta,doutb;
reg [17:0] douta,doutb;
reg [17:0] mem [0:63];
```

```
always@(posedge clk)
begin
if(wrc)
mem[addrc] <= dinc;
end

always@(posedge clk)
begin
douta <= mem[addra];
end

always@(posedge clk)
begin
doutb <= mem[addrb] ;
end
endmodule
```

RTL view:
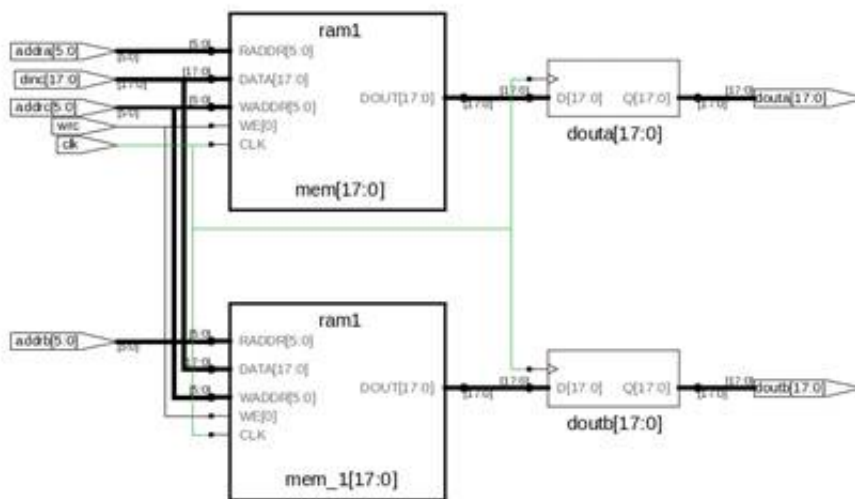
The tool infers one RAM64X18.

Example 2) Three Port RAM VHDL Example—Asynchronous Read

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity ram_singleport_noreg is
port (d : in std_logic_vector(7 downto 0);
addw : in std_logic_vector(6 downto 0);
addr1 : in std_logic_vector(6 downto 0);
addr2 : in std_logic_vector(6 downto 0);
we : in std_logic;
clk : in std_logic;
q1 : out std_logic_vector(7 downto 0);
q2 : out std_logic_vector(7 downto 0) );
end ram_singleport_noreg;
architecture rtl of ram_singleport_noreg 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 (addw)) <= d;
end if;
end if;
end process;
q1<= mem(conv_integer (addr1));
q2<= mem(conv_integer (addr2));
end rtl;
```

RTL View:
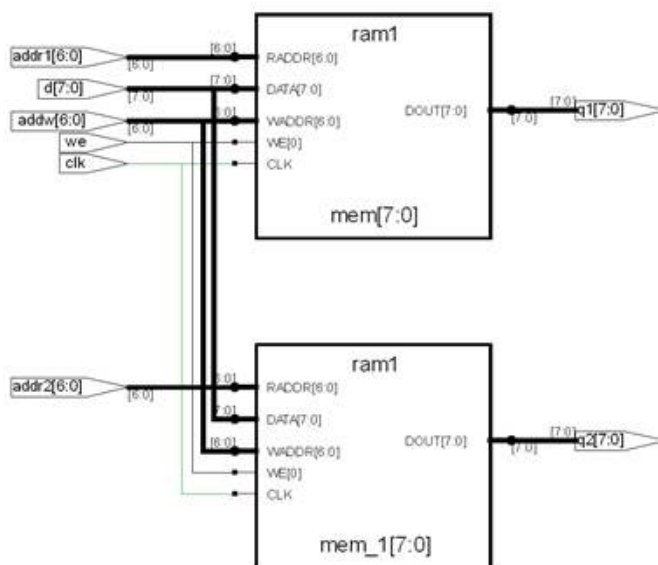
The tool infers one RAM64X18.

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 400 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.

- VM Flow Support, on page 694
- Forward-annotating Constraints for Placement and Routing, on page 695
- Synthesis Reports, on page 696

VM Flow Support

The tool generates a Verilog output netlist (*.vm*) for SmartFusion2 and IGLOO2 for P&R flow. After synthesis, the tool:

- Writes a separate SDC file (**_vm.sdc*).
- Write a separate TCL file (**_partition_vm.tcl*) to forward-annotate the timestamps on instances in incremental compile point flow.
- Forward-annotates properties like RTL attributes in *.vm* netlist and constraints in SDC file.

By default, the tool generates a *.edn* netlist. You can change the netlist from EDIF to Verilog.

To select Verilog output netlist, go to Implementation Options->Implementation Results->Result Format. Select vm from the drop-down menu, click OK, and save the project.
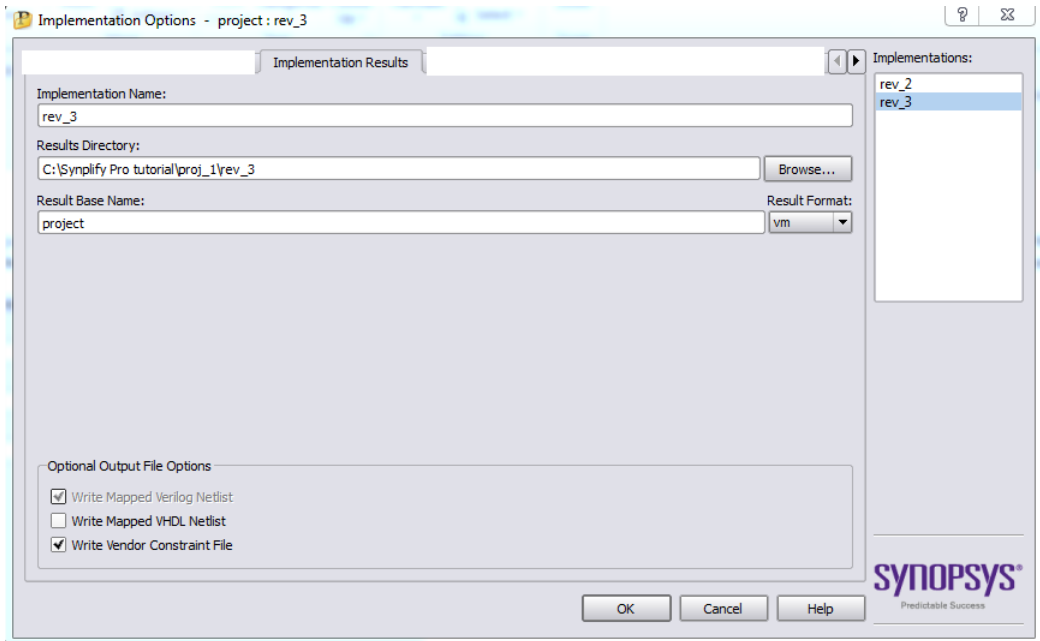
Forward-annotating Constraints for Placement and Routing

For Microsemi, Fusion, IGLOO+/IGLOO/IGLOOe, ProASIC3/3E/3L, and SmartFusion 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
create_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.
set_max_delay	set_max_delay
	The set_max_delay constraint is allowed for all NGT families. Wildcards are accepted.
set_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 ProASIC3/3E technologies. Multicycle constraints with a -through argument are not supported for any NGT family.
set_false_path	set_false_path
	Only false path constraints with a -through argument are supported for NGT families. 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 SmartFusion, Fusion, IGLOO and ProASIC3 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

SmartFusion2 Global Buffer Promotion

The synthesis software inserts the global buffer (CLKINT) on clock, asynchronous set/reset, and data nets based on a threshold value. SmartFusion2 devices have specific threshold values that cannot be changed for the different types of nets in the design. Inserting global buffers on nets with fanout greater than the threshold can help reduce the route delay during place and route.

The threshold values for SmartFusion2 devices are the following:

Net	Global buffer inserted for threshold value > or =
Clock	2
Asynchronous Set/Reset	12
Data	50000

To override these default option settings you can:

- Use the syn_noclockbuf attribute on a net that you do not want a global buffer inserted, even though fanout is greater than the threshold.

- Use syn_insert_buffer="CLKINC" so that the tool inserts a global buffer on the particular net, which is less than the threshold value. You can only specify CLKINC as a valid value for SmartFusion2 devices.

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 SmartFusion, Fusion, IGLOO+/IGLOO/IGLOOe, and ProASIC3/3E/3L 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 SmartFusion, Fusion, IGLOO+/IGLOO/IGLOOe, and ProASIC3/3E/3L 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, on page 33* and the *User Guide* for more information.

Update Compile Point Timing Data Option

In SmartFusion, Fusion, IGLOO+/IGLOO/IGLOOe, and ProASIC3/3E/3L 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 378 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.

Synplify Pro for Microsemi Edition Reference Manual
700   January 2014

> **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 consid-
> ered 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:

- ProASIC3/3E/3L

- IGLOO+/IGLOO/IGLOOe

- SmartFusion, FUSION

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	

Dropdown list:
Default
COMWC-2
COMWC-1
COMWCSTD
COMWC-F
COMTC-2
COMTC-1
COMTCSTD
COMTC-F
COMBC-2

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.

Note that even when a particular operating condition is valid for a family, it may not be applicable to every part/package/speed-grade combination in that family. Consult 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 481
- syn_radhardlevel, on page 141

Integration with Microsemi Tools and Flows

The following describe how the synthesis tools support various tools and flows for Microsemi designs:

Compile Point Synthesis

Compile-point synthesis is available only for the Microsemi SmartFusion, SmartFusion2, Fusion, IGLOO+/IGLOO/IGLOOe/IGLOO2, and ProASIC3/3E/3L 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 378 for a description, and *Working with Compile Points, on page 359* 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 360, and Update Compile Point Timing Data Option, on page 700.)

Incremental Flow Synthesis

Incremental flow synthesis is supported only for SmartFusion2 and IGLOO2.

The FPGA synthesis tool provides timestamps for each manual compile point in the *_partition.tcl file. The timestamps can be used to check if the compile point was re-synthesized in an incremental run of the tool.

The flow is as follows:

1. Define compile point constraint on the modules in the design. For example:

   ```
   define_compile_point              {<viewname>} -type {locked,
   partition} -cpfile {}
   ```

2. Run through the normal synthesis flow. The tool writes timestamps for each compile point in the *<designname>*_partition.tcl file. For example:

   ```
   set_partition_info -name <partition_name> -timestamp <timestamp>
   ```

   With incremental tool run, only affected compile points display new timestamps, unaffected compile points retain the old timestamps.

   The Compile Point Summary report is available in the .srr file.

Microsemi Place-and-Route Tools

You can run place and route automatically after synthesis. For details on how to set options, see Running P&R Automatically after Synthesis, on page 492 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 710 for more information about the preserve_registers option.
Disable I/O Insertion	I/O Insertion, on page 699.
Fanout Guide	Setting Fanout Limits, on page 341 of the *User Guide* and The syn_maxfan Attribute in Microsemi Designs, on page 697.
Hard Limit to Fanout	Setting Fanout Limits, on page 341 of the *User Guide* and The syn_maxfan Attribute in Microsemi Designs, on page 697.
Max number of critical paths in SDF (certain technologies)	Number of Critical Paths, on page 700.
Operating Conditions (certain technologies)	Operating Condition Device Option, on page 702
Promote Global Buffer Threshold	Controlling Buffering and Replication, on page 343 of the *User Guide* and Promote Global Buffer Threshold, on page 698.

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, on page 144.
Retiming	Retiming, on page 328 of the *User Guide and* Retiming, on page 700.
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 700

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 191).

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 65.

 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: FUSION, IGLOO, IGLOOE, IGLOO+, ProASIC3, ProASIC3E, ProASIC3L, SmartFusion, and SmartFusion2.
-part *partName*	Specifies a part for the implementation. Refer to the Implementation Options dialog box for available choices.
-package *packageName*	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.
-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 699.

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 697.
-globalthreshold *value*	Sets the minimum fanout load value. This option applies only to the SmartFusion, Fusion, IGLOO+/IGLOO/IGLOOe, and ProASIC3/3E/3L technologies. For more information, see Promote Global Buffer Threshold, on page 698.
-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 697.
-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 SmartFusion, Fusion, IGLOO+/IGLOO/IGLOOe, and ProASIC3/3E/3L technologies. Values are Default, MIL-WC, IND-WC, COM-WC, and Automotive-WC. See Operating Condition Device Option, on page 702 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 SmartFusion, Fusion, IGLOO+/IGLOO/IGLOOe, and ProASIC3/3E/3L technologies. For information about setting critical paths, see Number of Critical Paths, on page 700.

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 SmartFusion, Fusion, IGLOO+/IGLOO/IGLOOe, and ProASIC3/3E/3L technologies. For additional information about retiming, see Retiming, on page 700
-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, on page 144.
-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 SmartFusion, Fusion, IGLOO+/IGLOO/IGLOOe, ProASIC (500K), ProASIC^{PLUS} (PA), and ProASIC3/3E/3L technologies. See Update Compile Point Timing Data Option, on page 700, 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 How Attributes and Directives are Specified, on page 4.

Attribute/Directive	Description
alsloc	Forward-annotates the relative placements of macros and IP blocks to Microsemi Designer.
alspin	Assigns scalar or bus ports to Microsemi I/O pin numbers.
alspreserve	Specifies that a net be preserved, and prevents it from being removed during place-and-route optimization.
black_box_pad_pin (D)	Specifies that a pin on a black box is an I/O pad. It is applied to a component, architecture, or module, with a value that specifies the set of pins on the module or entity.
black_box_tri_pins (D)	Specifies that a pin on a black box is a tristate pin. It is applied to a component, architecture, or module, with a value that specifies the set of pins on the module or entity.
full_case (D)	Specifies that a Verilog case statement has covered all possible cases.
loop_limit (D)	Specifies a loop iteration limit for for loops.
parallel_case (D)	Specifies a parallel multiplexed structure in a Verilog case statement, rather than a priority-encoded structure.
syn_allow_retiming	Specifies whether registers can be moved during retiming.
syn_black_box (D)	Defines a black box for synthesis.
syn_encoding	Specifies the encoding style for state machines.
syn_enum_encoding (D)	Specifies the encoding style for enumerated types (VHDL only).

(D) indicates directives; all others are attributes.

Attribute/Directive	Description
syn_global_buffers	Sets the number of global buffers to use in a ProASIC3/3E/3L.
syn_hier	Controls the handling of hierarchy boundaries of a module or component during optimization and mapping.
syn_insert_buffer	Inserts a clock buffer according to the specified value.
syn_isclock (D)	Specifies that a black-box input port is a clock, even if the name does not indicate it is one.
syn_keep (D)	Prevents the internal signal from being removed during synthesis and optimization.
syn_maxfan	Overrides the default fanout guide for an individual input port, net, or register output.
syn_multstyle	Determines how multipliers are implemented for Microsemi devices.
syn_netlist_hierarchy	Determines whether the EDIF output netlist is flat or hierarchical.
syn_noarrayports	Prevents the ports in the EDIF output netlist from being grouped into arrays, and leaves them as individual signals.
syn_noclockbuf	Turns off the automatic insertion of clock buffers.
syn_noprune (D)	Controls the automatic removal of instances that have outputs that are not driven.
syn_pad_type	Specifies an I/O buffer standard.
syn_preserve (D)	Prevents sequential optimizations across a flip-flop boundary during optimization, and preserves the signal.
syn_probe	Adds probe points for testing and debugging.
syn_radhardlevel	Specifies the radiation-resistant design technique to apply to a module, architecture, or register.

(D) indicates directives; all others are attributes.

Attribute/Directive	Description
syn_ramstyle	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	Specifies a clock frequency other than that implied by the signal on the clock pin of the register.
syn_replicate	Controls replication.
syn_resources	Specifies resources used in black boxes.
syn_sharing (D)	Specifies resource sharing of operators.
syn_state_machine (D)	Determines if the FSM Compiler extracts a structure as a state machine.
syn_tco<n> (D)	Defines timing clock to output delay through the black box. The *n* indicates a value between 1 and 10.
syn_tpd<n> (D)	Specifies timing propagation for combinational delay through the black box. The *n* indicates a value between 1 and 10.
syn_tristate (D)	Specifies that a black-box pin is a tristate pin.
syn_tsu<n> (D)	Specifies the timing setup delay for input pins, relative to the clock. The *n* indicates a value between 1 and 10.
translate_off/translate_on (D)	Specifies sections of code to exclude from synthesis, such as simulation-specific code.

(D) indicates directives; all others are attributes.

APPENDIX C

Example Code

This appendix contains the code samples that are referenced by the corresponding chapter.

Example - Constant function

```verilog
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 - 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

```verilog
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
```

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
```

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
```

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
```

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
```

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)
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
```

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
```

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
```

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

```vhdl
--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);
```

```
        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

Index

Symbols

A