HDL Compiler™ for SystemVerilog User Guide

Version O-2018.06, June 2018

SYNOPSYS®

Copyright Notice and Proprietary Information

©2018 Synopsys, Inc. All rights reserved. This Synopsys software and all associated documentation are proprietary to Synopsys, Inc. and may only be used pursuant to the terms and conditions of a written license agreement with Synopsys, Inc. All other use, reproduction, modification, or distribution of the Synopsys software or the associated documentation is strictly prohibited.

Destination Control Statement

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

Disclaimer

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

Trademarks

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

Free and Open-Source Software Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

Third-Party Links

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

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

Copyright Notice for the Command-Line Editing Feature

© 1992, 1993 The Regents of the University of California. All rights reserved. This code is derived from software contributed to Berkeley by Christos Zoulas of Cornell University.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgement:

   This product includes software developed by the University of California, Berkeley and its contributors.

4. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright Notice for the Line-Editing Library

© 1992 Simmule Turner and Rich Salz. All rights reserved.

This software is not subject to any license of the American Telephone and Telegraph Company or of the Regents of the University of California.

Permission is granted to anyone to use this software for any purpose on any computer system, and to alter it and redistribute it freely, subject to the following restrictions:

1. The authors are not responsible for the consequences of use of this software, no matter how awful, even if they arise from flaws in it.
2. The origin of this software must not be misrepresented, either by explicit claim or by omission. Since few users ever read sources, credits must appear in the documentation.
3. Altered versions must be plainly marked as such, and must not be misrepresented as being the original software. Since few users ever read sources, credits must appear in the documentation.
4. This notice may not be removed or altered.

Contents

About This Manual 2

Customer Support. 4

1. SystemVerilog for Synthesis

Supported Constructs 1-3

Coding for QoR. 1-7

Reading SystemVerilog Designs 1-7

Automated Process of Reading Designs With Dependencies 1-8

The -autoread Option 1-9

File Dependencies 1-10

Setting Library Search Order 1-11

Ignoring Modules During the Read Process 1-12

Ignoring Modules During the Read Process (Legacy) 1-14

File Format Inference Based on File Extensions 1-15

Specifying the SystemVerilog Version 1-15

Reading Verilog Designs 1-16

Netlist Reader 1-16

Automatic Detection of Input Type. 1-16

Reading Designs 1-17

Reading Designs Using the VCS Command-Line Options. 1-18

Creating Relative Placement Using HDL Compiler Directives 1-18

HDL Compiler Directives for Relative Placement 1-19

Relative Placement Restrictions 1-19

Specifying Relative Placement Groups 1-20
Specifying Subgroups, Keepouts, and Instances 1-21
Enabling Automatic Cell Placement 1-22
Specifying Placement for Array Elements 1-23
Specifying Cell Alignment 1-24
Specifying Cell Orientation 1-24
Ignoring Relative Placement 1-25
Relative Placement Examples 1-26
Relative Placement Example 1 1-26
Relative Placement Example 2 1-27
Relative Placement Example 3 1-29
Relative Placement Example 4 1-31

Bottom-Up Hierarchical Elaboration 1-32
Running the Bottom-Up Hierarchical Elaboration Flow Using a Wrapper Design 1-33
Preventing Port Name Mismatches During Linking in Bottom-Up Hierarchical Flow 1-35

Shortening Long Module Names in the Netlist 1-36

Reading Designs With Assertion Checker Libraries 1-37

Netlist Wrapper for Testbenches 1-38
Creating a Testbench With a Wrapper 1-40

Customizing Elaboration Reports 1-42

Reporting Elaboration Errors in the Hierarchy 1-43
Example of Reporting Elaboration Errors 1-44

Querying Information about RTL Preprocessing 1-48

Reporting HDL Compiler Variables 1-50

Parameterized Designs 1-50

Defining Macros 1-53
Predefined Macros 1-54
Global Macro Reset: `undefineall 1-55

Using $display During RTL Elaboration 1-55

Elaboration System Tasks 1-56

Inputs and Outputs 1-57
Input Descriptions 1-58
Design Hierarchy 1-59

Component Inference and Instantiation 1-60
Naming Considerations 1-60
Generic Netlists 1-61
Error Messages 1-64

2. Global Name Space ($unit)

About the Global Name Space 2-2
Reading Designs With $unit 2-3
Defining Objects Before Use 2-4
Specifying Global Files First 2-5
Specifying Global Files for Each analyze Command 2-6
Synthesis Restrictions for $unit 2-7
Declarations 2-8
Instantiations 2-8
Static Variables 2-8
Static Tasks and Functions 2-8

3. Packages

About Packages 3-2
Using Packages 3-2
Referencing Declarations in Packages 3-3
Wildcard Imports From Packages Into Modules 3-4
Specific Imports From Packages Into Modules 3-5
Wildcard Imports From Packages Into $unit 3-7
Package Searching 3-8

4. Combinational Logic

Synthetic Operators 4-2
Logic and Arithmetic Expressions 4-3
Basic Operators 4-4
Addition Overflow 4-5
Sign Conversions 4-6
Language Constructs for Combinational Logic Inference 4-11

The always_comb and always Constructs. 4-11
Latches in Combinational Logic. 4-12
The priority if and priority case Constructs 4-13
priority if 4-14
priority case 4-14
The unique if and unique case Constructs 4-15
unique if 4-15
unique case 4-15

Multiplexing Logic 4-16
SELECT_OP Inference 4-17
One-Hot Multiplexer Inference. 4-18
MUX_OP Inference 4-19
Variables Controlling MUX_OP Inference 4-21
MUX_OP Inference Examples 4-21
Using the case Statement for MUX_OP Inference 4-25
MUX_OP Inference Limitations 4-27

MUX_OP Cells With Variable Indexing 4-27

MUX_OP Inference for Bit and Memory Accesses 4-28

Bit-Truncation Coding for DC Ultra Datapath Extraction. 4-28

5. Sequential Logic

Generic Sequential Cell SEQGEN 5-2

Inference Reports for Registers 5-4

Register Inference Guidelines 5-6
Multiple Events in an always Block 5-6
Minimizing Registers 5-7
Keeping Unloaded Registers 5-8
Preventing Unwanted Latches. 5-11
Reset Logic Inference 5-12
Register Inference Limitations 5-14

Register Inference Examples 5-15
Inferring Latches 5-15
Basic D Latch 5-16
D Latch With Asynchronous Set: Use async_set_reset 5-16
D Latch With Asynchronous Reset: Use async_set_reset 5-17

D Latch With Asynchronous Set and Reset:
Use hdlin_latch_always_async_set_reset 5-17
Unintended Logic Inferred Using always_latch. 5-18
Inferring Flip-Flops. 5-19
Basic D Flip-Flop 5-20
D Flip-Flop With Asynchronous Reset Using ?: Construct 5-21
D Flip-Flop With Asynchronous Reset 5-21
D Flip-Flop With Asynchronous Set and Reset. 5-22
D Flip-Flop With Synchronous Set:
Use sync_set_reset 5-22
D Flip-Flop With Synchronous Reset: Use sync_set_reset 5-23
D Flip-Flop With Synchronous and Asynchronous Load 5-23
D Flip-Flops With Complex Set and Reset Signals 5-24
Multiple Flip-Flops With Asynchronous and
Synchronous Controls 5-25
Unintended Logic Inferred Using always_ff 5-26

6. State Machines

Using Enumerated Types 6-2
FSM State Diagram and State Table 6-2
Default Data Type 6-3
Explicit Data Types 6-5

Automatic Increments for the State Variables 6-6

Enumeration Range 6-8

Methods for Enumerated Types 6-9

Separate Sequential and Combinational Assignments. 6-9

FSM Coding Requirements for Automatic Inference 6-10

FSM Inference Variables 6-11

FSM Coding Example. 6-11

FSM Inference Report 6-14

Enumerated Types 6-15

7. Interfaces

Elements of Interfaces 7-2
Example: Interface With Wires 7-3

Example: Interface With Modports. 7-6
Example: Interface With Modport Expressions 7-10
Example: Interface With Functions 7-11
Example: Interface With Functions and Tasks. 7-13
Example: Interface With always Blocks. 7-19

Inputs to Interfaces 7-19
Ports in Interfaces Example 7-20
Parameterized Interfaces Example 7-23

Arrays of Interfaces. 7-24
Coding Styles for Interface Arrays 7-25
Coding Style Restrictions on Array Interfaces 7-29

Renaming Conventions 7-29
Renamed Modules Example 1. 7-30
Renamed Modules Example 2. 7-32
Renamed Modules Example 3. 7-34

Using Interfaces in HDL Compiler. 7-36

Synthesis Restrictions 7-37

8. Modeling Three-State Buffers

Using z Values 8-2

Three-State Driver Inference Report 8-2

Assigning a Single Three-State Driver to a Single Variable 8-3

Assigning Multiple Three-State Drivers to a Single Variable. 8-4

Registering Three-State Driver Data. 8-5

Instantiating Three-State Drivers 8-6

Errors and Warnings. 8-6

9. Other SystemVerilog Features

Variables 9-3

The foreach Loop 9-4

Functions and Tasks 9-5

Binding Function and Task Arguments by Name 9-11

Parameterized Functions and Tasks Using Virtual Classes 9-11

Parameterized Data Types 9-12

Bit-Level Support for Compiler Directives 9-15

Structures 9-16

Unions 9-18

Multidimensional Arrays 9-19

Configurations 9-23
    Configuration Examples 9-24
    Default Statement 9-25
    Instance Bindings 9-26
    Multiple Top-Level Designs 9-28

Implicit Port Connections 9-29

Casting 9-31

Assignment Patterns 9-32

Macro Expansion and Parameter Substitution 9-34

`begin_keywords and `end_keywords 9-34

Predefined SYSTEMVERILOG Macro 9-35

Matching Block Names 9-35

Port Renaming 9-37

Generic Wire Type 9-39

General Verilog Coding Guidelines 9-40

Guidelines for Interacting With Other Flows 9-41
    Synthesis Flows 9-41
    Low-Power Flows 9-41
    Verification Flows 9-44

10. HDL Compiler Synthesis Directives

RTL Pragmas 10-2
    async_set_reset 10-3
    async_set_reset_local 10-3
    async_set_reset_local_all 10-3

dc_tcl_script_begin and dc_tcl_script_end 10-4
enum 10-5
full_case. 10-7
infer_multibit and dont_infer_multibit 10-9
Using the infer_multibit Directive 10-10
Using the dont_infer_multibit Directive 10-11
Reporting Multibit Components 10-12
infer_mux 10-14
infer_mux_override 10-14
infer_onehot_mux 10-15
keep_signal_name 10-15
one_cold 10-15
one_hot 10-15
parallel_case 10-16
preserve_sequential 10-17
sync_set_reset. 10-17
sync_set_reset_local 10-18
sync_set_reset_local_all 10-19
template 10-19
Directive Support by Pragma Prefix. 10-20

SystemVerilog Attributes 10-21
Using SystemVerilog Attributes in Synthesis 10-22
Supported Attributes 10-22
Supported RTL Constructs 10-23
Designs (Modules) 10-23
Ports 10-24
Cells (Instantiations) 10-24
Pins. 10-25
Inferred Register Cells (Sequential Processes) 10-25

11. Troubleshooting Guidelines

Code Expansion for Macros and Conditional Directives. 11-2
Code Expansion Guidelines 11-2
Code Expansion Example 1 11-3
Code Expansion Example 2 11-5

Minimizing Mismatches Between Simulation and Synthesis 11-6
Preventing case Mismatches 11-7

Using unique Instead of full_case and parallel_case 11-7
Using priority Instead of full_case 11-9
Using Void Functions Instead of Tasks Inside always_comb 11-11
Conversion Between Two-State and Four-State Variables 11-13
Data Type Declarations. 11-15
Synthesizable do...while Loops 11-15
Troubleshooting generate Loops 11-16
Assertions in Synthesis. 11-16
Other Troubleshooting Guidelines 11-17

Appendix A. SystemVerilog Design Examples

FIFO Example. A-2
Bus Fabric Design A-4
Coding for Late-Arriving Signals A-18
Duplicating Datapaths A-18
Moving Late-Arriving Signals Close to Output A-21
Overview. A-21
Late-Arriving Data Signal Example 1 . . . . A-23
Late-Arriving Data Signal Example 2 . . . . A-24
Late-Arriving Data Signal Example 3 . . . . A-26
Late-Arriving Control Signal Example 1 . . . . A-29
Late-Arriving Control Signal Example 2 . . . . A-30
Master-Slave Latch Inferences A-31
Overview for Inferring Master-Slave Latches. A-32
Master-Slave Latch With One Master-Slave Clock Pair A-32
Master-Slave Latch With Multiple Master-Slave Clock Pairs. A-33
Master-Slave Latch With Discrete Components A-34
JK Flip-Flop With Synchronous Set and Reset Using sync_set_reset. A-34

Appendix B. Verilog Language Support

Syntax B-2
Comments B-2
Numbers B-2

Verilog Keywords B-3

Unsupported Verilog Language Constructs B-4

Construct Restrictions and Comments B-5

always Blocks B-6

generate Statements B-6

Generate Overview. B-7

Types of generate Blocks B-7

Anonymous generate Blocks B-7

Loop Generate Blocks and Conditional Generate Blocks. B-10

Restrictions. B-11

Conditional Expressions (?:) Resource Sharing B-11

Case B-12

casez and casex B-12

Full Case and Parallel Case B-12

defparam B-14

disable B-14

Blocking and Nonblocking Assignments B-15

Macromodule. B-16

inout Port Declaration B-16

tri Data Type B-17

HDL Compiler Directives B-17

`define. B-17

`include B-18

`ifdef, `else, `endif, `ifndef, and `elsif B-19

`rp_group and `rp_endgroup. B-20

`rp_place. B-20

`rp_fill B-21

`rp_array_dir B-21

rp_align. B-22

rp_orient B-22

rp_ignore and rp_endignore B-22

`undef B-23

reg Types. B-23

Types in Busing B-23

Combinational while Loops B-24

Verilog 2001 and 2005 Supported Constructs B-28

Ignored Constructs B-29

Simulation Directives B-29
Verilog System Functions B-30
Verilog 2001 Feature Examples B-30
Multidimensional Arrays and Arrays of Nets B-30
Signed Quantities B-32
Comparisons With Signed Types. B-34
Controlling Signs With Casting Operators B-35
Part-Select Addressing Operators ([+:] and [-:]). B-35
Variable Part-Select Overview B-35
Example—Ascending Array and -: B-36
Example—Ascending Array and +:. B-37
Example—Descending Array and the -: Operator B-38
Example—Descending Array and the +: Operator B-39
Power Operator (**) B-40
Arithmetic Shift Operators (<<< and >>>) B-40
Verilog 2005 Feature Example B-41
Zero Replication. B-41
Configurations. B-42
Configuration Examples. B-43
Default Statement B-43
Instance Bindings B-45
Multiple Top-Level Designs B-46

Appendix C. Unsupported Constructs

Unsupported SystemVerilog Constructs C-2

Glossary

Index

Preface

This preface includes the following sections:

- About This Manual
- Customer Support

About This Manual

The *HDL Compiler for SystemVerilog User Guide* describes the SystemVerilog constructs supported by the Synopsys synthesis tools.

Audience

This document is for logic designers who are familiar with the Synopsys Design Compiler® tool and the HDL Compiler tool. Knowledge of the Verilog language is required. This document is not a standalone document but must be used in conjunction with the *IEEE Std 1800-2012*.

Related Publications

For additional information about the HDL Compiler tool, see the documentation on the Synopsys SolvNet® online support site at the following address:

https://solvnet.synopsys.com/DocsOnWeb

You might also want to see the documentation for the following related Synopsys products:

- DC Explorer
- Design Compiler®
- DesignWare® components
- Library Compiler™
- VHDL System Simulator

Release Notes

Information about new features, enhancements, changes, known limitations, and resolved Synopsys Technical Action Requests (STARs) is available in the *HDL Compiler Release Notes* on the SolvNet site.

To see the *HDL Compiler Release Notes*,

1. Go to the SolvNet Download Center located at the following address:

   https://solvnet.synopsys.com/DownloadCenter

2. Select HDL Compiler, and then select a release in the list that appears.

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
`Courier`	Indicates syntax, such as `write_file`.
`Courier italic`	Indicates a user-defined value in syntax, such as `write_file` *`design_list`*.
`Courier bold`	Indicates user input—text you type verbatim—in examples, such as `prompt>` **`write_file top`**
[ ]	Denotes optional arguments in syntax, such as `write_file [-format` *`fmt`*`]`
...	Indicates that arguments can be repeated as many times as needed, such as *`pin1 pin2 ... pinN`*
|	Indicates a choice among alternatives, such as `low | medium | high`
Ctrl+C	Indicates a keyboard combination, such as holding down the Ctrl key and pressing C.
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Edit > Copy	Indicates a path to a menu command, such as opening the Edit menu and choosing Copy.

Customer Support

Customer support is available through SolvNet online customer support and through contacting the Synopsys Technical Support Center.

Accessing SolvNet

The SolvNet site includes a knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. The SolvNet site also gives you access to a wide range of Synopsys online services including software downloads, documentation, and technical support.

To access the SolvNet site, go to the following address:

https://solvnet.synopsys.com

If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to sign up for an account.

If you need help using the SolvNet site, click HELP in the top-right menu bar.

Contacting the Synopsys Technical Support Center

If you have problems, questions, or suggestions, you can contact the Synopsys Technical Support Center in the following ways:

- Open a support case to your local support center online by signing in to the SolvNet site at https://solvnet.synopsys.com, clicking Support, and then clicking “Open A Support Case.”
- Send an e-mail message to your local support center.
  - E-mail support_center@synopsys.com from within North America.
  - Find other local support center e-mail addresses at

    https://www.synopsys.com/support/global-support-centers.html
- Telephone your local support center.
  - Call (800) 245-8005 from within North America.
  - Find other local support center telephone numbers at

    https://www.synopsys.com/support/global-support-centers.html

1

SystemVerilog for Synthesis

These topics describe the SystemVerilog constructs supported by the Synopsys synthesis tools:

- Supported Constructs
- Coding for QoR
- Reading SystemVerilog Designs
- Reading Verilog Designs
- Reading Designs Using the VCS Command-Line Options
- Creating Relative Placement Using HDL Compiler Directives
- Bottom-Up Hierarchical Elaboration
- Shortening Long Module Names in the Netlist
- Reading Designs With Assertion Checker Libraries
- Netlist Wrapper for Testbenches
- Customizing Elaboration Reports
- Reporting Elaboration Errors in the Hierarchy
- Querying Information about RTL Preprocessing
- Reporting HDL Compiler Variables

- Parameterized Designs
- Defining Macros
- Using $display During RTL Elaboration
- Inputs and Outputs

For information about troubleshooting guidelines and the tool limitations, see

- Troubleshooting Guidelines
- Unsupported Constructs

Supported Constructs

This table lists the supported SystemVerilog features and provides the usage information for each feature. For information about the syntax, see the *IEEE Std 1800-2005*. To download a copy of the standard at no charge, go to the following address:

http://www.synopsys.com/community/interoperability/pages/systemverilog.aspx

Table 1-1 Supported Constructs

Category	Feature	Usage reference
Literals	Structure literals	Structures
	Unsized literal ('0, '1, 'x, 'z)	Structures
Data types	Logic (4-value) data type	Used in most examples
	Integer data types (`int`, `bit`)	Other SystemVerilog Features
	User-defined types (`typedef`)	Default Data Type Functions and Tasks Data Type Declarations
	Structures (packed and unpacked)	Structures
	Enumerations	Default Data Type
	Enumeration methods	*IEEE Std 1800-2012* For restrictions, see Methods for Enumerated Types.
	Range of `enum` labels	Enumeration Range
	Unions (packed)	Unions
	Casting	Casting
	Void data types	*IEEE Std 1800-2012*
	Generic wire type	Generic Wire Type
Arrays	Packed arrays	Multidimensional Arrays Casting
	Packed array of enumerations	*IEEE Std 1800-2012*

Table 1-1 Supported Constructs (Continued)

Category	Feature	Usage reference
	Unpacked arrays	Multidimensional Arrays
	Array querying (`$size`, `$left`, `$right`, `$low`, `$high`, `$increment`, `$dimensions`, `$unpacked_dimensions`)	Multidimensional Arrays
Data declaration	Scoping	*IEEE Std 1800-2012*
	Constraints	*IEEE Std 1800-2012*
	Variables	Used in many examples
Operators	"." operator	Structures
	+=, -=, ++, --, &=, |=, ^=	Synthetic Operators
	Wildcard equality and inequality operators (==? and !=?)	*IEEE Std 1800-2012*
	Left-to-right stream operator {>>{}}[1]	*IEEE Std 1800-2012*
	<<=, >>=, <<<=, >>>=	Synthetic Operators
	`inside` and `case-inside`	*IEEE Std7 1800-2012* Note: Both operators are supported in procedural context. The `inside` operator is not supported in a continuous assignment, and the `case-inside` operator cannot be used in a continuous assignment.
Procedural statements	`unique if` and `priority if`	unique if priority if
	`unique case`, `priority case`, `casex`, and `casez`	unique case priority case
	Matching end block names	Matching Block Names

Table 1-1 Supported Constructs (Continued)

Category	Feature	Usage reference
Processes	`always_comb`	The always_comb and always Constructs
	`always_latch`	Inferring Latches
	`always_ff`	Inferring Flip-Flops
Functions and tasks	Void functions	Synthesis Restrictions for $unit Functions and Tasks
	• All types as legal task or function argument types • All types as legal function return types • Return statement in functions • Logic default task or function argument type • Input default task or function argument direction	Functions and Tasks
	Binding by name	Binding Function and Task Arguments by Name
	Automatic variable initialization	Variables
	Argument binding using the `.name` syntax	*IEEE Std 1800-2012*
Assertions	Assertions	Unsupported Constructs Reading Designs With Assertion Checker Libraries
Hierarchy	All types as legal module ports	Multidimensional Arrays Functions and Tasks
	`$unit`	About the Global Name Space
	Implicit `.name` and `.*` port connections	Implicit Port Connections
Interfaces	Interface as a signal container and module port replacement	Example: Interface With Wires
	Interface modports	Example: Interface With Modports

Table 1-1 Supported Constructs (Continued)

Category	Feature	Usage reference
	Interface ports	Ports in Interfaces Example
	Parameterized interfaces	Parameterized Interfaces Example
	Interface functions and tasks	Example: Interface With Functions and Tasks
	Generic interface	*IEEE Std 1800-2012*
	Array of interfaces	Arrays of Interfaces
Parameters	Default `logic` type	*IEEE Std 1800-2012*
	Data type parameter	Parameterized Data Types
System tasks and functions	Size system function (`$bits`)	Casting
System tasks	`$fatal`, `$error`, `$warning`, `$info`	Elaboration System Tasks
Compiler directives	`begin_keywords` and `end_keywords`	\`begin_keywords and \`end_keywords
Flow control	`for (int i=0; ...)`	Functions and Tasks
	`break` and `continue`	*IEEE Std 1800-2012*
	`do...while`	Synthesizable do...while Loops
Packages	• Scope extraction using :: • Wildcard imports inside modules • Wildcard imports inside `$unit` • Specific imports	Packages

1. Excess source bits not supported for unpack operation (see the IEEE Std 1800-2012, section 11.4.14.3).

Coding for QoR

The HDL Compiler tool optimizes a design to provide the best QoR independent of the coding style; however, the optimization of the design is limited by the design context information available. You can use the following techniques to provide the information for the tool to produce optimal results:

- The tool cannot determine whether an input of a module is a constant even if the upper-level module connects the input to a constant. Therefore, use a parameter instead of an input port to express an input as a constant.
- During compilation, constant propagation is the evaluation of expressions that contain constants. The tool uses constant propagation to reduce the hardware required to implement complex operators.

  If you know that a variable is a constant, specify it as a constant. For example, a “+” operator with a constant high as an argument causes an increment operator rather than an adder. If both arguments of an operator are constants, no hardware is inferred because the tool can calculate the expression and insert the result into the circuit.

  The same technique applies to designing comparators and shifters. When you shift a vector by a constant, the implementation requires only reordering (rewiring) the bits without hardware implementation.

Reading SystemVerilog Designs

You can use these commands to read SystemVerilog designs into the synthesis tool.

- `read_sverilog` or `read_file -format sverilog`

  For designs containing interfaces or parameterized designs, set the `hdlin_auto_save_templates` variable to `true.`

  For example,

  ```
  set_app_var hdlin_auto_save_templates true
  read_sverilog  parametrized_interface.sv
  current_design top
  link
  compile
  write -format verilog -hierarchy \
     -output gates.parametrized_interface_rd.v
  ```

- `analyze -format sverilog {`*`files`*`}`
`elaborate `*`topdesign`*

  For example,

```
analyze -format sverilog parametrized_interface.sv
elaborate top
compile
write -format verilog -hierarchy \
   -output gates.parametrized_interface_an_elab.v
```

This method is recommended because of the following reasons:

- Recursive elaboration is performed on the entire design, so you do not need an explicit `link` command. The `elaborate` command includes the functions of the `link` command.
- For designs containing interfaces or parameterized designs, you do not need to set the `hdlin_auto_save_templates` variable to `true`.

For designs containing black boxes, use the `hdlin_sv_blackbox_modules` variable to specify the black boxes. For designs containing global declarations, you must read the global files and then the specific design files.

For more information about designs containing black boxes or global declarations, see

- Ignoring Modules During the Read Process
- Reading Designs With $unit

Automated Process of Reading Designs With Dependencies

You can enable the tool to automatically read designs with dependencies in correct order by using the `-autoread` option with the `read_file or analyze` command.

- `read_file -autoread`

  This command reads files with dependencies automatically, analyzes the files, and elaborates the files starting at a specified top-level design. For example,

  ```
  dc_shell> read_file -autoread file_list -top design_name
  ```

  You must specify the file_list argument to list the files, directories, or both to be analyzed. The `-autoread` option locates the source files by expanding each file or directory in the file_list argument. You must specify the top design by using the `-top` option.

- `analyze -autoread`

  This command reads files with dependencies automatically and analyzes the files without elaboration. For example,

  ```
  dc_shell> analyze -autoread file_list -top design_name
  ```

  You must specify the file_list argument to list the files, directories, or both to be analyzed. The `-autoread` option locates the source files by expanding each file or directory in the file_list argument. If you use the `-top` option, the tool analyzes only the source files needed to elaborate the top-level design. If you do not specify the `-top` option, the tool analyzes all the files in the *file_list* argument, grouping them in the order according to the dependencies that the `-autoread` option infers.

Example

The following example specifies the current directory as the source directory. The command reads the source files, analyzes them, and then elaborates the design starting at the top-level design.

```
dc_shell> read_file {.} -autoread -recursive -top E1
```

The following example specifies the file extensions for SystemVerilog files other than the default (.sv and .sverilog) and sets the file source lists. The `read_file -autoread` command specifies the top-level design and includes only files with the specified SystemVerilog file extensions.

```
dc_shell> set_app_var hdlin_autoread_sverilog_extensions {.sve .SVE}
dc_shell> set_app_var my_sources {mod1/src mod2/src}
dc_shell> set_app_var my_excludes {mod1/src/incl_dir/ mod2/src/incl_dir/}
dc_shell> read_file $my_sources -recursive -exclude $my_excludes \
   -autoread -format sverilog -top TOP
```

Excluding directories is useful when you do not want the tool to use those files that have the same file extensions as the source files in the directories.

See Also

- The -autoread Option
- File Dependencies

The -autoread Option

When you use the `-autoread` *`file_list`* option with the `read_file` or `analyze` command, the resulting GTECH representation is retained in memory. Dependencies are determined by the files or directories specified in the file_list argument. If the file_list argument changes between consecutive calls of the `-autoread` option, the tool uses the latest set of files to determine the dependencies. You can use the `-autoread` option on designs written in any VHDL, Verilog, or SystemVerilog language version. If you do not specify this option, only the files specified in the *file_list* argument are processed and the file list cannot include directories.

When you specify a directory as an argument, the command reads files from the directory. If you specify both the -autoread and `-recursive` options, the command also reads files in the subdirectories.

When the -autoread option is set, the command infers RTL source files based on the file extensions set by the variables listed in the following table. If you specify the `-format` option, only files with the specified file extensions are read.

Variable	Description	Default
`hdlin_autoread_exclude_extensions`	Specifies the file extension to exclude files from the analyze process.	" "
`hdlin_autoread_verilog_extensions`	Specifies the file extension to analyze files as Verilog files.	.v
`hdlin_auto_autoread_vhdl_extensions`	Specifies the file extension to analyze files as VHDL files.	.vhd .vhdl
`hdlin_autoread_sverilog_extensions`	Specifies the file extension to analyze files as SystemVerilog files.	.sv .sverilog

File Dependencies

A file dependency occurs when a file requires language constructs that are defined in another file. When you specify the `-autoread` command, the tool analyzes the files (and elaborates the files if you use the `read_file` command) with the following dependencies in the correct order:

- *Analyze dependency*

  If file B defines entity E in SystemVerilog and file A defines the architecture of entity E, file A depends on file B and must be analyzed after file B. Language constructs that can cause analyze dependencies include VHDL package declarations, entity declarations, direct instantiations, and SystemVerilog package definitions and import.

- *Link dependency*

  If module X instantiates module Y in Verilog, you must analyze both of them before elaboration and linking to prevent the tool from inferring a black box for the missing module. Language constructs that can cause link dependencies include VHDL component instantiations and SystemVerilog interface instantiations.

- *Include dependency*

  When file X includes file Y using the `‘include` directive, this is known as an *include dependency*. The `-autoread` option analyzes the file that contains the `include directive statement when any of the included files are changed between consecutive calls of the `-autoread` option.

- *Verilog and SystemVerilog compilation-unit dependency*

The dependency occurs when the tool detects files that must be analyzed together in one compilation unit. For example, Verilog or SystemVerilog macro usage and definition are located in different files but not linked by the `` `include `` directive, such as a macro defined several times in different files. The `-autoread` option cannot determine which file to use. Language constructs that can cause compilation-unit dependencies include SystemVerilog function types, local parameters, and enumerated values defined by the `$unit` scope.

Setting Library Search Order

When multiple design libraries are available during elaboration, the tool searches for a particular design in the libraries that are defined by the `define_design_lib` command. The library defined last is searched first. This library search order is the default and applies to the entire design, including the subdesigns. By default, the tool searches the library of the parent design first for a subdesign. If the subdesign is not found, it searches other libraries in this search order.

For example, the library search order is defined as lib3, lib2, and lib1in the following `define_design_lib` command sequence:

```
dc_shell> define_design_lib lib1 ...
dc_shell> define_design_lib lib2 ...
dc_shell> define_design_lib lib3 ...
```

To change the library search order, list the libraries by using the `–uses` option with the `analyze` command. When a design is analyzed with the `analyze -uses` *`design_libs`* command, the tool searches for the subdesigns of this design in the library order specified by the `–uses` option.

When you use the `-uses` option,

- The parent design library is searched first, followed by libraries in the order specified by the `-uses` option.
- The specified library search order applies only to the specified design and its subdesigns. Other designs use the default.
- The search is restricted to the libraries specified by the `–uses` option. Other libraries are not searched even if no library is found.
- An empty list for the `–uses` option limits the search to the library of the parent design.

For example, in the following design, three different versions of the submod design are analyzed in the lib1, lib2, and lib3 libraries respectively:

top.v

```
module top (...);
...
U0 submod (...);
...
endmodule
```

submod1.v

```
submod (...);
<implementation 1>
endmodule
```

submod2.v

```
submod (...);
<implementation 2>
endmodule
```

submod3.v

```
submod (...);
<implementation 3>
endmodule
```

When you use the following command to analyze the top-level top.v design, the module analyzed using the lib2 library is chosen during elaboration and the modules using the lib1 and lib3 libraries are ignored.

```
dc_shell> analyze ... -uses "lib2 lib1 lib3" top.v
```

Ignoring Modules During the Read Process

During early design stages, you can include incomplete or non-synthesizable designs by using the SystemVerilog *interface-only* feature. This feature allows modules that communicate with or instantiate the unfinished module to connect port signals correctly even for an unfinished design. The unfinished module design can be empty or incomplete, or it can contain unsupported constructs. The module body will eventually be replaced by synthesizable RTL.

To enable the tool to read SystemVerilog designs as interface-only, use the `hdlin_sv_interface_only_modules` variable to list the design modules. The HDL Compiler tool parses only the module interface of the listed designs, skipping the module content, and creates a black box for each module. During elaboration, the tool issues a warning message that the module content is discarded and ignored, as shown in the following example:

```
dc_shell> set hdlin_sv_interface_only_modules {my_module1 my_module2}
dc_shell> analyze -f sverilog top.sv
```

```
Warning: ./rtl/top.sv:21: The body of module 'my_module1' is being
discarded, because the module name is in hdlin_sv_interface_only_modules.
(VER-747)
```

After elaboration of the top-level design, you can use the `is_interface_only` attribute to list all the designs that were read as interface only. For example,

```
dc_shell> get_designs -filter "is_interface_only"
{my_module1_P2}
```

Limitations

The *IEEE Std 1800-2012* (section 23.2.1) defines two module definition styles:

- ANSI header style—all port information within the module header

  ```
  module_name #( parameter_port_list )
   ( port_direction_and_type_list );

  ...design content...
  ```

- Non-ANSI header style—non-name port information follows the module header

  ```
  module_name #( port_name_list ) ;

   parameter_declaration_list
   port_direction_and_size_declarations
   port_direction_and_type_list

  ...design content...
  ```

All modules with ANSI style module headers can be read in as interface-only.

For modules with non-ANSI style module headers, the tool skips the module content after the first occurrence of design content that is not one of the following:

- Port declarations
- Data type definitions
- Parameter declarations
- Net or variable declarations
- Package imports

When using non-ANSI style module headers, keep all port-related declarations together at the beginning of the module to prevent the tool from skipping interface information. Avoid breaking up the port declarations with other statements that are not port declarations.

Ignoring Modules During the Read Process (Legacy)

Important:

> This section documents the legacy module black-boxing functionality. It is recommended that you use the improved functionality described in “Ignoring Modules During the Read Process” on page 1-12.
>
> For information on the difference between these methods, see SolvNet article 2730864, “What is the Difference Between the hdlin_sv_interface_only_modules and hdlin_sv_blackbox_modules Variables?”

You can direct the HDL Compiler tool to ignore modules, such as analog blocks and register files, during the read process.

To enable this capability, specify a list of modules to be ignored as black boxes by setting the `hdlin_sv_blackbox_modules` variable. The tool ignores the specified modules when reading the design with the `read_sverilog` or `analyze -format sverilog` command and treats them as black boxes during the link process. When reading the design, the tool issues VER-746 warning messages indicating which modules are ignored; however, it does not issue any messages if you specify invalid modules names. Valid module names are those coded in the RTL.

For example, the following command specifies the two modules named mod1 and mod2 in the RTL to be ignored:

```
dc_shell> set_app_var hdlin_sv_blackbox_modules "mod1 mod2"
```

When reading the design, the tool issues warning messages similar to the following:

```
Warning: mod1.v:2: The declaration of module 'mod1' is being ignored,
because the module name is in hdlin_sv_blackbox_modules. (VER-746)
```

During the link process performed by the `elaborate` or `link` command, the tool issues warning messages similar to the following:

```
Warning: All references to module 'mod1' are ignored and treated as
black boxes. (LINK-35)
```

The following restrictions apply:

- Support for black-box modules is available only in SystemVerilog, but not in Verilog or VHDL.
- You should not modify the setting of the `hdlin_sv_blackbox_modules` variable between the `read_sverilog` and `link` commands or between the `analyze -format sverilog` and `elaborate` commands.
- You should not use design names reported by the `list_designs` command as module names because they might change during the read process and no longer match the original RTL names. Use the original RTL names to set the variable.

File Format Inference Based on File Extensions

You can specify a file format by using the `-format` option with the `read_file` command. If you do not specify a format, the `read_file` command infers the format based on the file extensions. If the file extension is unknown, the tool assumes the .ddc format.

The file extensions in this table are supported for automatic inference:

Format	File extensions
ddc	.ddc
db	.db, .sldb, .sdb, .db.gz, .sldb.gz, .sdb.gz
SystemVerilog	.sv, .sverilog, .sv.gz, .sverilog.gz

The supported extensions are not case-sensitive. All formats except the .ddc format can be compressed in gzip (.gz) format.

If you use a file extension that is not supported and you omit the `-format` option, the synthesis tool generates an error message. For example, if you specify `read_file test.vlog`, the tool issues the following DDC-2 error message:

```
Error: Unable to open file 'test.vlog' for reading. (DDC-2)
```

Specifying the SystemVerilog Version

To specify which SystemVerilog language version to use during the read process, set the `hdlin_sverilog_std` variable. The valid values for this variable are `2005`, 2009, and `2012`, corresponding to the 2005, 2009, and 2012 SystemVerilog LRM releases respectively. When you set the `hdlin_sverilog_std` variable to a valid version, the SystemVerilog LRM features of this version are enabled when you run the `analyze -format sverilog` command or the `read_sverilog` command. The default for the `hdlin_sverilog_std` variable is `2012`.

Note:

> The `hdlin_vrlg_std` variable sets the language version for the `analyze -format verilog` and `read_verilog` commands. The default is `2005`.

Reading Verilog Designs

When the HDL Compiler tool reads a design, it checks for correct syntax and builds a generic technology (GTECH) netlist that the Design Compiler tool uses to optimize the design. You can use the `read_verilog` command to do both functions or use the `analyze` and `elaborate` commands to do each function separately. If you have parameterized designs, use the `elaborate` command to specify parameter values.

The tool supports automatic linking of mixed-language libraries. In Verilog, the default library is in the work directory, and you cannot have multiple libraries. In VHDL, you can have multiple design libraries.

Specifying the Verilog Version

To specify which Verilog language version to use during the read process, set the `hdlin_vrlg_std` variable. The valid values for the `hdlin_vrlg_std` variable are `1995`, `2001`, and `2005`, corresponding to the 1995, 2001, and 2005 Verilog LRM releases respectively. The default is `2005`.

In addition to RTL designs, the tool can read Verilog gate-level netlists.

See Also

- Parameterized Designs
- Automatic Detection of Input Type

Netlist Reader

Design Compiler contains a specialized reader for gate-level Verilog netlists that has higher capacity on designs that do not use RTL-level constructs, but it does not support the entire Verilog language. The specialized netlist reader reads netlists faster and uses less memory than HDL Compiler.

If you have problems reading a netlist with the netlist reader, try reading it with HDL Compiler by using `read_verilog -rtl` or by specifying `read_file -format verilog -rtl`.

Automatic Detection of Input Type

By default, when you read in a Verilog gate-level netlist, HDL Compiler determines that your design is a netlist and runs the specialized netlist reader.

Important:

> For best memory usage and runtime, do not mix RTL and netlist designs into a single read. The automatic detector chooses one reader—netlist or RTL—to read all files

included in the command. Mixed files default to the RTL reader, because it can read both types; the netlist reader can read only netlists.

The following variables apply only to HDL Compiler and are not implemented by the netlist reader:

- `power_preserve_rtl_hier_names` (default is `false`)
- `hdlin_auto_save_templates` (default is `false`)

If you set either of these variables to `true`, automatic netlist detection is disabled and you must use the `-netlist` option to enable the netlist reader.

Reading Designs

Table summarizes the recommended and alternative commands to read your designs.

Type of input	Reading method

RTL	For parameterized designs,
	`analyze -format verilog { files }`
	`elaborate` *`topdesign`*
	is preferred because it does a recursive elaboration of the entire design and lets you pass parameter values to the elaboration. The read method conditionally elaborates all designs with the default parameters.
	
	To enable macro definition from the read method, use
	`read_file -format verilog { files }`
	
	Alternative reading methods:
	`read_verilog -rtl { files }`
	`read_file -format verilog -rtl { files }`

Gate-level netlists	Recommended reading method:
	`read_verilog  { files }`
	
	Alternative reading methods:
	`read_verilog -netlist { files }`
	`read_file -format verilog -netlist { files }`

Reading Designs Using the VCS Command-Line Options

The `analyze` command with the VCS command-line options provides better compatibility and makes reading large designs easier. When you use the VCS command-line options, the tool automatically resolves references for instantiated designs by searching the referenced designs in user-specified libraries and then loading these referenced designs.

Reading Large Designs

To read designs containing many HDL source files and libraries, specify the `-vcs` option with the `analyze` command. You must enclose the VCS command-line options in double quotation marks. For example,

```
dc_shell> analyze -vcs "-verilog -y mylibdir1 +libext+.v -v myfile1 \
   +incdir+myincludedir1 -f mycmdfile2" top.v
```

Reading Designs With Mixed Formats

To read SystemVerilog files with a specified file extension and Verilog files in one `analyze` command, use the `-vcs "+systemverilogext+`*`ext`*`"` option. When you do so, the files must not contain any Verilog 2001 styles.

For example, the following command analyzes SystemVerilog files with the .sv file extension and Verilog files:

```
dc_shell> analyze -format verilog -vcs "-f F +systemverilogext+.sv"
```

Creating Relative Placement Using HDL Compiler Directives

Relative placement technology allows you to create structures in which you specify the relative column and row positions of instances. During placement and optimization, these structures are preserved and the cells in each structure are placed as a single entity.

Relative placement is usually applied to datapaths and registers, but you can apply it to any cells in your design, controlling the exact relative placement topology of gate-level logic groups and defining the circuit layout. You can use the relative placement capability to explore QoR benefits, such as shorter wire lengths, reduced congestion, better timing, skew control, fewer vias, better yield, and lower dynamic and leakage power.

These topics describe how to create relative placement by specifying the HDL compiler directives:

- HDL Compiler Directives for Relative Placement
- Relative Placement Restrictions
- Specifying Relative Placement Groups

- Specifying Subgroups, Keepouts, and Instances
- Enabling Automatic Cell Placement
- Specifying Placement for Array Elements
- Specifying Cell Alignment
- Specifying Cell Orientation
- Ignoring Relative Placement
- Relative Placement Examples

HDL Compiler Directives for Relative Placement

This table lists the HDL compiler directives for relative placement in RTL designs and HDL netlists. A check mark (X) indicates that the directive is applicable and supported for the design format.

Table 1-2 HDL Compiler Directives for Relative Placement

HDL compiler directive	RTL design	HDL netlist	Usage reference
`rp_group` and `rp_endgroup`	X	X	Specifying Relative Placement Groups
rp_place	X	X	Specifying Subgroups, Keepouts, and Instances
rp_fill	X	X	Enabling Automatic Cell Placement
`rp_array_dir`	X		Specifying Placement for Array Elements
`rp_align`		X	Specifying Cell Alignment
`rp_orient`		X	Specifying Cell Orientation
`rp_ignore` and `rp_endignore`		X	Ignoring Relative Placement

Relative Placement Restrictions

You must not specify relative placement HDL compiler directives for RTL design and for HDL netlists in the same file. Other restrictions apply depending on the input format.

The following restrictions apply to RTL designs:

- Specify relative placement directives only on register banks.
- To perform relative placement on leaf-level registers, you must specify the relative placement directives inside an always block that infers registers, but not combinational logic.

  If an always block does not infer registers, the tool generates an ELAB-2 error message.

The following restrictions apply to HDL netlists, including GTECH netlists and mapped netlists:

- You must use the HDL Compiler netlist reader to read HDL netlists.
- For GTECH netlists, apply relative placement directives only to cells that have a one-to-one mapping of the library cell.

  Relative placement directives can be applied to cells such as AND gates, OR gates, and D flip-flops. Relative placement directives cannot be applied to cells such as SEQGENs, SELECT_OPs, MUX_OPs, and DesignWare components.
- For mapped netlists, you can apply relative placement directives to any cell.

See Also

- Generic Sequential Cell SEQGEN
- SELECT_OP Inference
- MUX_OP Inference
- Synthetic Operators

Specifying Relative Placement Groups

To specify a relative placement group in an RTL design or HDL netlist (a GTECH netlist or mapped netlist), use the `rp_group and `rp_endgroup directive pair.

The syntax for the directive pair is as follows:

- Verilog and SystemVerilog syntax for RTL designs

  ```
  ‘rp_group ( group_name {num_cols num_rows} )
  ‘rp_endgroup ( {group_name} )
  ```

  For leaf-level relative placement groups, specify the directives inside an always block. High-level hierarchical groups do not have to be included in an always block.
- Syntax for HDL netlists

  ```
  //synopsys rp_group ( group_name {num_cols num_rows} )
  //synopsys rp_endgroup ( {group_name} )
  ```

Place all cell instances between the directives to declare them as members of the specified relative placement group.

To specify the size of the relative placement group, use the *num_cols* and *num_rows* optional arguments for the number of columns and number of rows. The tool ensures that all instances in the group is placed inside the specified size limits. The tool issues an error message for a size violation.

The following example shows that relative placement group rp_grp1 contains the inferred register:

```
...
always_ff @(posedge)
   ‘rp_group (rp_grp1)
   ...
   ‘rp_endgroup (rp_grp1)
      Q1 <= DATA1;
...
```

Specifying Subgroups, Keepouts, and Instances

To place a subgroup, a keepout region, or an instance in the current relative placement group of an RTL design or HDL netlist, use the `` `rp_place `` directive. When you specify a subgroup at a specific hierarchy, you must instantiate the subgroup instance outside any group declaration in the module.

The syntax for the `` `rp_place `` directive is as follows:

- Verilog and SystemVerilog syntax for RTL designs

```
‘rp_place ( hier group_name col row )
‘rp_place ( keep keepout_name col row width height )
‘rp_place ({leaf} [inst_name] col row )
```

- Syntax for HDL netlists

```
//synopsys rp_place ( hier group_name col row )
//synopsys rp_place ( hier group_name [inst_name] col row )
//synopsys rp_place ({leaf} [inst_name] col row )
//synopsys rp_place ( keep keepout_name col row width height )
```

Use the *col* and *row* optional arguments to specify absolute column and row coordinates in the grid of the relative placement group or a location relative to the current coordinates (the location of the current instance). To specify locations relative to the current coordinates, enclose the column and row coordinates in angle brackets (<>). No brackets for absolute locations. If you do not specify the *col* and *row* arguments, a new instance is automatically placed in the available grid at the specified location. After the instance is placed, the tool increments the column and row coordinates of the location for the next cell.

The following example shows that group my_group_1 is placed at location (0,0) in the grid and group my_group_2 is placed at the next row position (0,1):

```
`rp_place (my_group_1 0 0)
`rp_place (my_group_2 0 <1>)
```

The following example shows that relative placement group my_reg_bank contains four subgroups at the following locations: (0,0), (0,1), (1,*), and (1,*). The wildcard character (*) indicates that the tool can choose any value for a coordinate in the group.

```
`rp_group (my_reg_bank)
`rp_place (hier rp_grp1  0 0)
`rp_place (hier rp_grp4  0 1)
`rp_place (hier rp_grp2  1 *)
`rp_place (hier rp_grp3  1 *)
`rp_endgroup (my_reg_bank)
```

See Also

- Enabling Automatic Cell Placement

Enabling Automatic Cell Placement

To enable the tool to place cells automatically at a specified location, use the `` `rp_fill `` directive for both RTL designs and HDL netlists.

The syntax for the `` `rp_fill `` directive is as follows:

- Verilog and SystemVerilog syntax for RTL designs

  ‘rp_fill ({*col row*} {pattern *pat*})

- Syntax for HDL netlists

  //synopsys rp_fill ({*col row*} {pattern *pat*})

When you specify the `col` and `row` arguments, the tool places each new instance in the grid at the specified location. The default is column zero and row zero (0,0). After the instance is placed, the tool increments the column and row coordinates of the location for the next cell.

- To specify the placement location, use the *col* and *row* arguments.

  The *col* and *row* optional arguments represent absolute column and row coordinates in the grid at the specified location or a location relative to the current coordinates (the location of the current instance). To specify locations relative to the current coordinates, enclose the column and row coordinates in angle brackets (<>). No brackets for absolute locations. You must use positive integers for absolute coordinates and any integer for relative coordinates.

For example, if the specified location is (3,4), you can increment the column coordinate by 1 and set the row coordinate to 0 by specifying `‘rp_fill <1> 0`. The relative coordinates point to location (4,0) for the next cell.

- To specify the placement direction, use the `pattern` argument with the `UX`, `DX`, `RX`, or `LX` keyword for up, down, right, or left direction respectively.

  The default is `UX`, where the tool places cells in the up direction within a column. The `RX` pattern fills a row with cells. If no pattern is specified for a cell, the tool uses the incremental location of the last pattern. If you do not specify the *`col`* and *`row`* arguments, the tool uses the previous pattern. When the tool encounters a group declaration, it initializes the placement location to (0,0) with the `UX` pattern.

When the tool encounters an array of instantiation in Verilog, the array cells are enumerated to match the pattern iterating from the left index to the right index, as shown in the following example:

```
and a[0:2] ( ); // generates a[0], a[1], a[2]
```

The following example uses the `UX` and `RX` patterns for the group_x relative placement group in the HDL netlist:

```
//synopsys rp_group (group_x)
//synopsys rp_fill (0 0 UX)
Cell C1 (...);
Cell C2 (...);
Cell C3 (...);
//synopsys rp_fill (0 <1> RX) // move up a row to the left, fill to R
Cell c4 (...);
Cell C5 (...);
Cell C6 (...);
//synopsys rp_endgroup (group_x)
```

Specifying Placement for Array Elements

To place array elements in ascending or descending order in a relative placement group, use the `` `rp_array_dir `` directive with the `up` or `down` keyword for RTL designs. You cannot use this directive for HDL netlists.

The Verilog syntax for RTL designs is as follows:

```
‘rp_array_dir ( up|down )
```

The `up` keyword indicates to place array elements from the least significant bit to the most significant bit, and the `down` keyword indicates to place array elements from the most significant bit to the least significant bit. The following example shows that the array elements are placed in the up direction:

```
...
always_ff @(posedge CLK)
   ‘rp_group (rp_grp1)
   ‘rp_fill (0 0 UX)
   ‘rp_array_dir (up)
   ‘rp_endgroup (rp_grp1)
      Q1 <= DATA1 ;
...
```

Specifying Cell Alignment

When an instance is smaller than the grid size, use the `‘rp_align` directive to specify the alignment of the instance in HDL netlists. You cannot use this directive for RTL designs.

The syntax for HDL netlists is as follows:

```
//synopsys rp_align ( n|s|e|w|nw|sw|ne|se|pin=name { inst })
```

By default, the tool applies the specified alignment to all subsequent instantiations within the group until the tool encounters another `‘rp_align` directive. If you do not specify an alignment, the default is `sw`. If you specify the *`inst`* instance name argument, the alignment applies only to that instance. If the instance straddles cells, the alignment takes place within the straddled region. The instance is snapped to legal row and routing grid coordinates. You must specify either the alignment or the pin name.

The following example specifies the cell placement of C1 at the northeast corner:

```
//synopsys rp_group (group_x)
//synopsys rp_fill ( 0  0  RX )
//synopsys rp_align (NE C1 )
Cell C1 ...
Cell C2 ...
Cell C3 ...
//synopsys rp_fill ( 0  <1>  RX )
Cell C4 ...
Cell C5 ...
Cell C6 ...
//synopsys rp_endgroup (group_x)
```

Specifying Cell Orientation

To control the placement orientation of library cells in the current group in HDL netlists, use the `‘rp_orient` directive. You cannot use this directive for RTL designs. When you specify a list of possible orientations, the tool chooses the first legal orientation.

The HDL netlist syntax is as follows:

- `//synopsys rp_orient ( {N|W|S|E|FN|FW|FS|FE}* { inst } )`

When you use the *inst* argument, the orientation is applied to the specified instance only. If you do not specify an instance, the orientation applies to all subsequent instances until another orientation is specified.

- //synopsys rp_orient ({N|W|S|E|FN|FW|FS|FE}* { *group_name inst* }))

  When you use the *group_name inst* argument, the orientation is applied to the specified instance and the rest of the group remains unchanged. The default orientation is N, specifying no flipping rotation at the horizontal axis and no mirror orientation at the vertical axis. The FN, FW, FS, and FE orientations specify flipping north, flipping west, flipping south, and flipping east respectively.

The following example specifies the cell placement of C1 in the west direction:

```
//synopsys rp_group (group_x)
...
//synopsys rp_orient (W C1)
Cell C0 ...
Cell C1 ...
...
```

Ignoring Relative Placement

To ignore lines in HDL netlists, use the ‘rp_ignore and ‘rp_endignore directive pair. During relative placement, the tool omits any lines encapsulated by the directive pair except the ‘include and ‘define directives, variable substitution, and cell mapping. You cannot use this directive pair for RTL designs.

You can use this directive pair to place the instantiations of submodules in a relative placement group close to the ‘rp_place hier group(inst)location for relative placement arrays.

The HDL netlist syntax is as follows:

```
//synopsys rp_ignore
...
//synopsys rp_endignore
```

The following example ignores the directive //synopsys rp_fill (0 <1> RX):

```
//synopsys rp_group (group_x)
//synopsys rp_fill ( 0  0  RX )
Cell C1 ...
Cell C2 ...
Cell C3 ...
//synopsys rp_ignore
//synopsys rp_fill ( 0  <1>  RX )
//synopsys rp_endignore
Cell C4 ...
Cell C5 ...
Cell C6 ...
//synopsys rp_endgroup (group_x)
```

Relative Placement Examples

This section provides examples that use HDL compiler directives for relative placement.

- Relative Placement Example 1
- Relative Placement Example 2
- Relative Placement Example 3
- Relative Placement Example 4

Relative Placement Example 1

This example shows how to apply the `` `rp_group ``, `` `rp_place ``, `` `rp_fill ``, and `` `rp_array_dir `` directives to several register banks in an RTL design for relative placement.

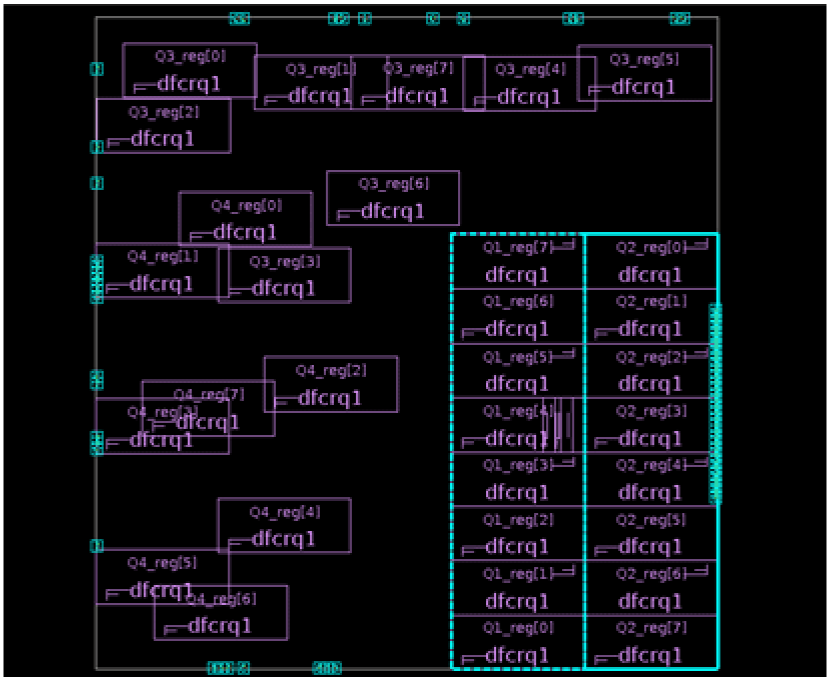
Example 1-1 Relative Placement Using HDL Compiler Directives

```
module dff_async_reset (
   input [7:0] DATA1, DATA2, DATA3, DATA4,
   input CLK, RESET,
   output logic [7:0] Q1, Q2, Q3, Q4
);
`rp_group (my_reg_bank)
`rp_place (hier rp_grp1  * 0)
`rp_place (hier rp_grp2  * 0)
`rp_endgroup (my_reg_bank)

always_ff @(posedge CLK or posedge RESET)
begin
   `rp_group (rp_grp1)
   `rp_fill (0 0 UX)
   `rp_array_dir(up)
   `rp_endgroup (rp_grp1)
   if (RESET) Q1 <= 8'b0;
   else Q1 <= DATA1;
end

always_ff @(posedge CLK or posedge RESET)
   `rp_group (rp_grp2)
   `rp_fill (0 0 UX)
   `rp_array_dir(down)
   `rp_endgroup (rp_grp2)
   if (RESET) Q2 <= 8'b0;
   else Q2 <= DATA2;

always_ff @(posedge CLK or posedge RESET)
```

```
   if (RESET) Q3 <= 8'b0;
   else Q3 <= DATA3;

always_ff @(posedge CLK or posedge RESET)
   if (RESET) Q4 <= 8'b0;
   else Q4 <= DATA4;
endmodule
```

Figure 1-1 shows the layout of this example after relative placement. The register banks that are constrained by the HDL compiler directives have a well-structured layout, while the register banks that are not constrained by the directives are not placed together.

Figure 1-1 Layout With Relative Placement Specified on Several Register Banks



Relative Placement Example 2

This example shows how to use the `‘rp_array_dir` directive to place relative placement groups vertically.

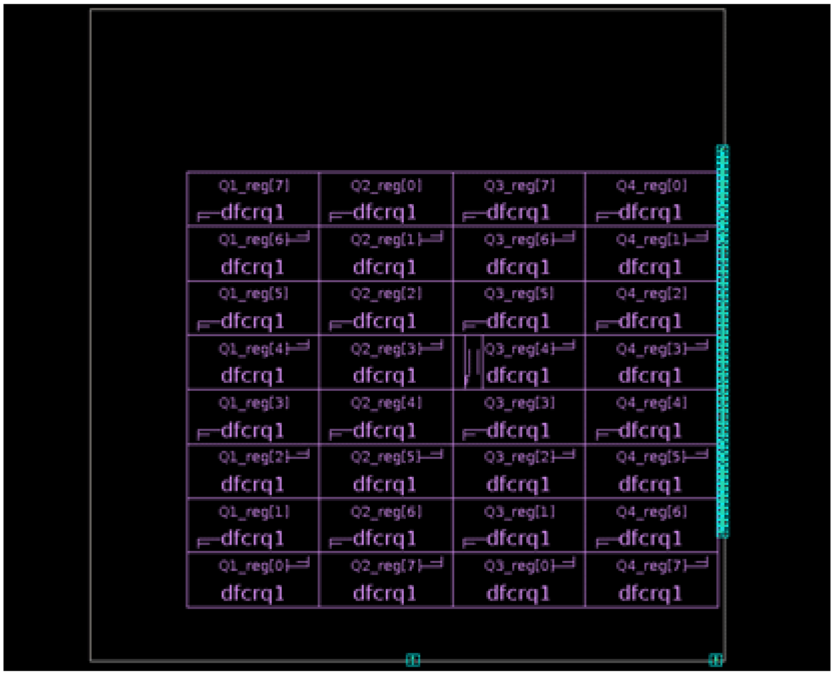
In this example, the array elements in relative placement groups rp_grp2 and rp_grp4 are placed in the down direction from element (7) to element (0) starting at row 0. The array elements in relative placement groups rp_grp1 and rp_grp3 are placed in the up direction from element (0) to element (7) starting at row 0.

Example 1-2 Relative Placement Groups Placed Vertically

```
module dff_async_reset (
   input [7:0] DATA1, DATA2, DATA3, DATA4,
   input CLK, RESET,
   output logic [7:0] Q1, Q2, Q3, Q4
);
`rp_group (my_reg_bank)
`rp_place (hier rp_grp1  * 0)
`rp_place (hier rp_grp2  * 0)
`rp_place (hier rp_grp3  * 0)
`rp_place (hier rp_grp4  * 0)
`rp_endgroup (my_reg_bank)

always_ff @(posedge CLK or posedge RESET)
begin
   `rp_group (rp_grp1)
   `rp_fill (0 0 UX)
   `rp_array_dir(up)
   `rp_endgroup (rp_grp1)
   if (RESET) Q1 <= 8'b0;
   else Q1 <= DATA1;
end

always_ff @(posedge CLK or posedge RESET)
   `rp_group (rp_grp2)
   `rp_fill (0 0 UX)
   `rp_array_dir(down)
   `rp_endgroup (rp_grp2)
   if (RESET) Q2 <= 8'b0;
   else Q2 <= DATA2;

always_ff @(posedge CLK or posedge RESET)
   `rp_group (rp_grp3)
   `rp_fill (0 0 UX)
   `rp_array_dir(up)
   `rp_endgroup (rp_grp3)
   if (RESET) Q3 <= 8'b0;
   else Q3 <= DATA3;

always_ff @(posedge CLK or posedge RESET)
   `rp_group (rp_grp4)
   `rp_fill (0 0 UX)
   `rp_array_dir(down)
   `rp_endgroup (rp_grp4)
   if (RESET) Q4 <= 8'b0;
   else Q4 <= DATA4;
endmodule
```

Figure 1-2 shows the generated layout where each relative placement group consists of one array and each group is placed vertically.

Figure 1-2 Relative Placement Groups Placed Vertically



Relative Placement Example 3

This example shows how to use the `‘rp_fill` directive to place relative placement groups horizontally.

In this example, each relative placement group is placed horizontally because the `‘rp_fill` directive is set to `RX`, which specifies the incremental row coordinate to the right of the initial position. The array elements in relative placement groups rp_grp2 and rp_grp4 are placed in the down direction from element (7) to element (0) starting at column 0. The array elements in relative placement groups rp_grp1 and rp_grp3 are placed in the up direction from the element (0) to element (7) starting at column 0.

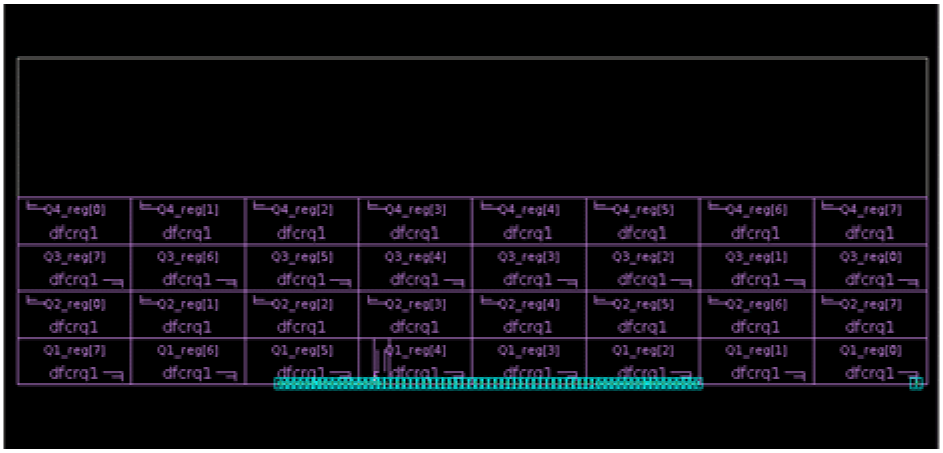
Example 1-3 Relative Placement Groups Placed Horizontally

```
module dff_async_reset (
   input [7:0] DATA1, DATA2, DATA3, DATA4,
   input CLK, RESET,
   output logic [7:0] Q1, Q2, Q3, Q4
);
`rp_group (my_reg_bank)
`rp_place (hier rp_grp1  0 *)
`rp_place (hier rp_grp2  0 *)
`rp_place (hier rp_grp3  0 *)
`rp_place (hier rp_grp4  0 *)
`rp_endgroup (my_reg_bank)
```

```
always_ff @(posedge CLK or posedge RESET)
begin
   `rp_group (rp_grp1)
   `rp_fill (0 0 RX)
   `rp_array_dir(up)
   `rp_endgroup (rp_grp1)
   if (RESET)
      Q1 <= 8'b0;
   else
      Q1 <= DATA1;
end

always_ff @(posedge CLK or posedge RESET)
   `rp_group (rp_grp2)
   `rp_fill (0 0 RX)
   `rp_array_dir(down)
   `rp_endgroup (rp_grp2)
   if (RESET)
      Q2 <= 8'b0;
   else
      Q2 <= DATA2;

always_ff @(posedge CLK or posedge RESET)
   `rp_group (rp_grp3)
   `rp_fill (0 0 RX)
   `rp_array_dir(up)
   `rp_endgroup (rp_grp3)
   if (RESET)
      Q3 <= 8'b0;
   else
      Q3 <= DATA3;

always_ff @(posedge CLK or posedge RESET)
   `rp_group (rp_grp4)
   `rp_fill (0 0 RX)
   `rp_array_dir(down)
   `rp_endgroup (rp_grp4)
   if (RESET)
      Q4 <= 8'b0;
   else
      Q4 <= DATA4;
endmodule
```

Figure 1-3 shows the generated layout where each relative placement group consists of one array and each group is placed horizontally.

Figure 1-3 Relative Placement Groups Placed Horizontally



Q4_reg[0] dfcrq1	Q4_reg[1] dfcrq1	Q4_reg[2] dfcrq1	Q4_reg[3] dfcrq1	Q4_reg[4] dfcrq1	Q4_reg[5] dfcrq1	Q4_reg[6] dfcrq1	Q4_reg[7] dfcrq1
Q3_reg[7] dfcrq1	Q3_reg[6] dfcrq1	Q3_reg[5] dfcrq1	Q3_reg[4] dfcrq1	Q3_reg[3] dfcrq1	Q3_reg[2] dfcrq1	Q3_reg[1] dfcrq1	Q3_reg[0] dfcrq1
Q2_reg[0] dfcrq1	Q2_reg[1] dfcrq1	Q2_reg[2] dfcrq1	Q2_reg[3] dfcrq1	Q2_reg[4] dfcrq1	Q2_reg[5] dfcrq1	Q2_reg[6] dfcrq1	Q2_reg[7] dfcrq1
Q1_reg[7] dfcrq1	Q1_reg[6] dfcrq1	Q1_reg[5] dfcrq1	Q1_reg[4] dfcrq1	Q1_reg[3] dfcrq1	Q1_reg[2] dfcrq1	Q1_reg[1] dfcrq1	Q1_reg[0] dfcrq1

Relative Placement Example 4

This example uses wildcard characters to infer keepouts in relative placement groups.

In this example, the design has a pipelined structure with combinational logic between register banks. At the top level, the top relative placement group contains wildcard characters for columns. Because of the wildcard characters, the tool infers keepouts in the relative placement groups to reserve space for combinational logic. To prevent the tool from inferring keepouts, use numbers instead of wildcard characters.

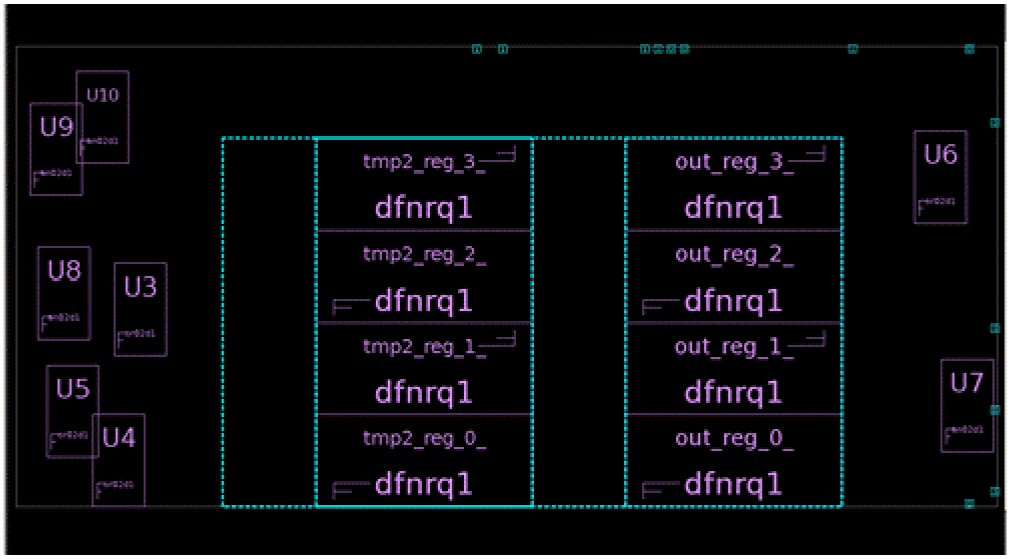
Example 1-4 Relative Placement Groups With Keepouts

```
module pipe (
   input clk,
   input [3:0] in1, in2,
   output logic [3:0] out
);
logic [3:0] tmp1, tmp2, tmp3;
assign tmp1 = in1 & in2;
assign tmp3 = tmp2 | in2;

always_ff @(posedge CLK)
   `rp_group (gp0)
   `rp_fill (0 0 UX)
   `rp_array_dir (up)
   `rp_endgroup (gp0)
   tmp2 <= tmp1;

always_ff @(posedge CLK)
   `rp_group (gp1)
   `rp_fill (0 0 UX)
   `rp_array_dir (up)
   `rp_endgroup (gp1)
```

```
     out <= tmp3;

     `rp_group (top)
     `rp_place (hier gp0 * 0)
     `rp_place (hier gp1 * 0)
     `rp_endgroup (top)
   endmodule
```

Figure 1-4 shows the generated layout where keepouts are inferred in the relative placement groups.

Figure 1-4 Relative Placement Groups With Keepouts



Bottom-Up Hierarchical Elaboration

In SystemVerilog designs, information about how to build a module is often supplied by an external source. These types of designs are called design templates. When you build a design using a top-down approach, all the information (design context) is available to accurately customize the template based on the way this information is used in the larger design. Additional information are required for

- Module parameters
- Interface parameters
- Interface modports
- Generic interface ports

To build a standalone design template without the design context, you need to specify the required information to accurately customize the design template. For example, to specify the information for module parameters, use the `-parameters` option with the `elaborate` command. However, in more complex SystemVerilog designs, this approach is ineffective because the design template has elements, such as interfaces, that are not elaborated and can be overridden on a port-by-port basis.

To overcome that issue, you need to create a simplified SystemVerilog wrapper for the design context of the block to be built. Only the design context of the block is required in the wrapper, so this process can be very simple in most cases.

Running the Bottom-Up Hierarchical Elaboration Flow Using a Wrapper Design

To build designs with type parameters or interfaces, use a streamlined wrapper to specify all the design context information to build the low-level designs separately.

Follow the steps described in these two procedures in sequence to build a bottom-up SystemVerilog design:

To build a low-level design using a wrapper with the design context,

1. Create a SystemVerilog wrapper module for the design.
2. Analyze the SystemVerilog design files being built and the wrapper module by using the `analyze` command.
3. Elaborate the wrapper module by using the `elaborate` command.
4. Remove the wrapper module by using the `remove_design` command.
5. Set the current design to the target design by using the `current_design` command.
6. Proceed with the synthesis flow.
7. Save the design by using the `write` command.

Note:

In the port mapping, you specify only the ports with parameterized interfaces.

To synthesize the top-level design with the synthesized low-level design,

1. Read the complete RTL design by using the `analyze` command.
2. Elaborate the top-level design by using the `elaborate` command.
3. Replace the low-level design with the synthesized version.
4. Proceed with the synthesis flow.

The following example consists of the myInft and mid SystemVerilog design files, wrapper file, Tcl scripts, and guidelines for hierarchical elaboration:

- File myIntf.sv

  ```
  interface myIntf #(parameter SIZE1=10, SIZE2=10) (input logic clk);
     logic [SIZE1-1:0] a, b;
     logic [SIZE2-1:0] c, d;
  modport mod1 (input clk, a, c, output b, d);
  endinterface
  ```

- File mid.sv

  ```
  module mid #(parameter type pType=logic) (
     myIntf.mod1 iport1,
     input pType other1, other2,
     output logic [7:0] other3, other4);
  /* RTL implementation */
  endmodule
  ```

- File wrapper.sv

  The wrapper specifies the actual parameter values only for ports with parameterized interfaces.

  ```
  module wrapper;
  myIntf #(8, 16) i1 ();
  mid #(logic[3:0]) m1 (.iport1(i1.mod1));
  endmodule
  ```

- Tcl script for the low-level design wrapper

  ```
  define_design_lib myLIB -path myLIB_dir
  analyze -library myLIB -format sverilog "myIntf.sv mid.sv wrapper.sv"
  elaborate -library myLIB wrapper
  remove_design [get_designs wrapper*]
  current_design [get_designs mid*]
  compile_ultra ...
  write -hierarchy -format ddc -output mid_mapped.ddc
  ```

- Tcl script for top-level synthesis

  ```
  define_design_lib myLIB -path myLIB_dir
  analyze -library myLIB -format sverilog "myIntf.sv mid.sv top.sv"
  elaborate -library myLIB top
  remove_design [get_designs mid*]
  read_ddc mid_mapped.ddc
  current_design top
  link
  ```

- Guidelines

  - Duplicate the instantiation of the low-level design and interfaces of the wrapper design at the top level, including

  - Same low-level design parameters
  - Same interface parameter assignment style and order
  - Same modports specification in the port list

- When using modport expression, specify modports in the port list.
- Optionally, you can specify only the interface ports at the low-level design instantiation from the wrapper design.
- Use the same design libraries names for the low-level designs in the wrapper and the top-level design.

If the tool cannot link the low-level designs when integrating the top design, the tool elaborates the low-level designs again without issuing any error message. For example,

```
dc_shell> link
Linking design 'top'
Using the following designs and libraries:

----------------------------------------------------------------------
top                          …/top.db
lsi_10k (library)            …/syn/lsi_10k.db
dw_foundation.sldb (library) …/syn/dw_foundation.sldb
bot_B12_B22_I_ifc_IFC_mp_    …/out/t01.bot.mapped.ddc

Information: Building the design 'bot' instantiated from design 'top'
with the parameters "B1=2,B2=2|((N%ifc%I%WORK_HIER/
IFC%mp%P1=2,P2=2)(N%r%)(N%p%))". (HDL-193)
Presto compilation completed successfully.
1
```

Preventing Port Name Mismatches During Linking in Bottom-Up Hierarchical Flow

In a bottom-up synthesis flow, the `change_names` command can change the port names of lower-level, synthesized designs by replacing a period (.) or square bracket ([or]) with an underscore (_). However, the top-level design is unsynthesized and still contains the original port names and this can cause mismatches during linking. Use the following variables to avoid port name mismatch linking errors:

`set link_portname_allow_period_to_match_underscore true`. The default is `false.`

`set link_portname_allow_square_bracket_to_match_underscore true`. The default is `false.` This variable should only be used for matching modport arrays.

In the following example, the linker cannot resolve the B.X port name of the mid1 instance because the port name was changed to B_X in the mid module. When you set the `link_portname_allow_period_to_match_underscore` variable to `true`, the design links.

```
module top (input A, B, output Y);
wire Y;
mid mid1 (.\B.X (B), .A(A), .Y(Y));
endmodule

module mid (input B_X, A, output Y);
assign Y = B_X & A;
endmodule
```

Shortening Long Module Names in the Netlist

If your design contains many interfaces and parameters, the tool creates long module names in the netlist because of inlining. These long names cannot be read by some back-end tools. To shorten the names, set the `hdlin_shorten_long_module_name` variable to `true` and set the `hdlin_module_name_limit` variable to a maximum number of characters allowed in the names. During a compile, when a module name is longer than the specified number, the tool renames the module to the original name plus a hash of the full name and issues a warning about the renamed module.

The following example shows a module name in the RTL, in the original gate-level netlist, and after renaming:

- Script to shorten module names

  ```
  # Enables shortening of names
  set_app_var hdlin_shorten_long_module_name true
  # Specify minimum number of characters. Default: 256
  set hdlin_module_name_limit 100
  ```

- Module name in the RTL

  ```
  module sender
  ```

- Original module name in the gate-level netlist

  ```
  module
  sender_I_i_s1_i_sendmode_I_i_s2_i_sendmode_I_i_s3_i_sendmode_I_i_s4_i_
  sendmode_I_i_s5_i_sendmode_I_i_s6_i_sendmode_I_i_s7_i_sendmode_I_i_s8_
  i_sendmode_I_i_s9_i_sendmode_I_i_s10_i_sendmode_
  ```

- Shortened module name in the gate-level netlist

  ```
  module sender_h_948_242_781
  ```

- Warning message

```
dc_shell> Warning:
Design'sender_I_i_s1_i_sendmode_I_i_s2_i_sendmode_I_i_s3_i_sendmode_I_
i_s4_i_sendmode_I_i_s5_i_sendmode_I_i_s6_i_sendmode_I_i_s7_i_sendmode_
I_i_s8_i_sendmode_I_i_s9_i_sendmode_I_i_s10_i_sendmode_'
was renamed to 'sender_h_948_242_781' to resolve a long name which is
nor supported by some down stream tools. (LINK-26)
```

Reading Designs With Assertion Checker Libraries

When reading designs containing assertion checker libraries, the tool infers extra logic in the gate-level netlist. To prevent the tool from inferring the extra logic, follow these steps to ignore the assertion checker libraries:

1. Include the checker library files in your search path.

   The following command includes the $VCS_ROOT/packages/sva checker library directory in the search path:

   ```
   dc_shell> set search_path \
      [concat $search_path $[VCS_ROOT]/packages/sva]
   ```

2. Create the sva.inc file to include all checker libraries in your design directory.

   For example, the following sva.inc file includes the assert_one_hot checker library.

   ```
   // sva.inc file includes all the assertions that you are using
   `include "assert_one_hot.sv"
   `include "assert_proposition.sv"
   ...
   ```

   For a complete list of assertions, see the VCS MAX documentation.

3. Analyze the checker libraries using the predefined SVA_CHECKER_INTERFACE macro.

   ```
   dc_shell> analyze -define SVA_CHECKER_INTERFACE \
      -format sverilog {sva.inc child.sv}
   ```

   Note:
   You must analyze the check libraries before the files that use the check libraries.

4. Elaborate the top design.

   The following command elaborates the child.sv module, which uses the assert_one_hot check library:

   ```
   dc_shell> elaborate child
   ```

The child.sv module

```
// child.sv
module child(reset_n, clk);
   input reset_n, clk;
   reg [7:0] count;

initial $monitor("count = %b \n", count);
begin
   if (reset_n == 0) count <= 8'b00000001;
   else count <= ((count << 1) | {7'b0000000, count[7]});
end

// the width is 8 bits
// Coverage level 1 is enabled by default.
assert_one_hot #(0, 8, 0, "ERROR: count is not one-hot") \
invalid_one_hot (clk, reset_n, count);
endmodule
```

See Also

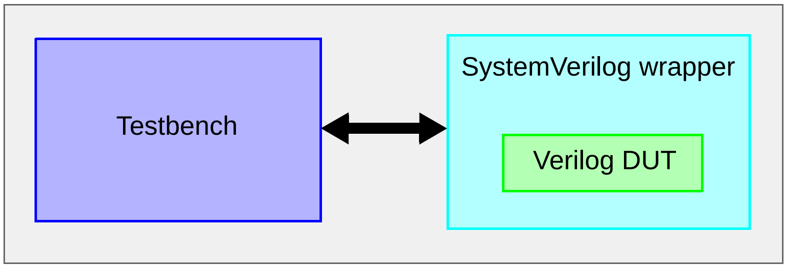
- The *VCS MAX User Guide*

Netlist Wrapper for Testbenches

You cannot use a testbench that is developed for a SystemVerilog design for the gate-level netlist because the port number, port types, and port names are not preserved in the Verilog implementation. For example, Verilog designs have no interface modports, parameter types, unpacked arrays or structs, or enumerations. In addition, each element in a SystemVerilog interface modport is implemented as a separate port by the synthesis tool. Back-end tools can read only netlists in Verilog format.

To use a SystemVerilog testbench for a gate-level Verilog simulation, you must modify the testbench to convert the interface ports to the implementation ports. The `write -format svsim` command can automate this process and write out a SystemVerilog netlist wrapper, which is a SystemVerilog module declaration. The testbench must instantiate the wrapper exactly as the original SystemVerilog module. This creates a SystemVerilog design under test (DUT), provides correct port mapping, and generates a SystemVerilog design instance that can be driven by the testbench, as shown in the following figure:

Figure 1-5 Single DUT Test Environment



The SystemVerilog simulation wrapper only supports module headers that are completely self-contained. If the module header requires definitions that are outside the header, follow these guidelines:

- Module header requiring definitions from the `$unit` global name space

  When building your testbench, ensure that the SystemVerilog netlist wrapper has the same design context as the RTL. You might need to add simulation tool settings or edit the wrapper to include the `$unit` definitions.

- Module header requiring definitions from a package

  In the RTL, import the package as part of the module header. For example,

  ```
  module xyz
  import myPack::*;
  ...
  endmodule;
  ```

- Module header containing forward references to elements that are defined inside the module

  This is not supported.

The following limitations apply:

- You cannot use the `-hierarchy` option with the `-format svsim` option of the `write` command.
- You cannot use one `write` command to create netlist wrappers for multiple designs.
- Only a subset of synthesizable SystemVerilog designs is supported, that is, the design whose root modules use ANSI-style port declarations.
- Only one DUT is allowed in each wrapper.
- The synthesis tool cannot read the netlist wrapper.

Creating a Testbench With a Wrapper

In the following test case, the IFC interface (interface.sv) uses the byte type as the default, and the TOP module (top.sv) uses a generic interface with the i modport. You must use the ANSI-style port declarations.

Follow these steps to create a wrapper and the gate-level netlist for a DUT that uses an interface module with overridden parameter types.

1. Create the dummy_top module (dummy_top.sv) to instantiate the TOP module where the IFC interface is overridden by the MY_T type.
2. Use the get_design_from_inst procedure to retrieve the cell instance.

   ```
   proc get_design_from_inst { inst } {
      return [get_attribute [get_cells $inst] ref_name]
   }
   ```

   In SystemVerilog, module names change based on the interface types, modports, parameter types, parameters, and so on. Because you know the instance name, top_inst, in the dummy_top module, you can retrieve the instance from the reference name (design declaration) by using the get_design_from_inst procedure.
3. Analyze the test case and elaborate the dummy_top module.

   ```
   dc_shell> analyze -format sverilog "interface.sv top.sv dummy_top.sv"
   dc_shell> elaborate dummy_top
   ```
4. Compile the test case.

   ```
   dc_shell> compile
   ```
5. Set the dut variable to the top-level instance by using the get_design_from_inst procedure.

   ```
   dc_shell> set_app_var dut [get_design_from_inst top_inst]
   ```
6. Write out the gate-level DUT.

   ```
   dc_shell> write -format verilog -hierarchy \
      -output compiled_gates.v $dut
   ```
7. Write out the wrapper file.

   ```
   dc_shell> write -format svsim -output netlist_wrapper.sv $dut
   ```

Test case

- interface.sv

```
typedef logic MY_T[0:2];
interface IFC #(parameter type T = byte);
T x, y;
modport mp (input x, output y);
endinterface
```

- top.sv

```
module TOP #(parameter type T = shortint) (interface.mp i);
T temp;
assign temp = i.x;
assign i.y = temp;
endmodule
```

- dummy_top.sv

```
module dummy_top #(parameter type T = MY_T) (
   input T in,
   output T out
);

IFC#(.T(T)) ifc();
assign ifc.x = in;
assign out = ifc.y;

TOP #(.T(T)) top_inst(ifc.mp);
endmodule
```

Tcl Script

The following script creates the gate-level netlist and a wrapper:

```
proc get_design_from_inst { inst } {
   return [get_attribute [get_cells $inst] ref_name]
}
analyze -format sverilog "interface.sv top.sv dummy_top.sv"
elaborate dummy_top
compile
set_app_var dut [get_design_from_inst top_inst]
write -format verilog -hierarchy -output compiled_gates.v $dut
write -format svsim -output netlist_wrapper.sv $dut
```

Wrapper file

The wrapper (netlist_wrapper.sv) contains the module declaration of TOP_svsim. It provides mapping between the original SystemVerilog ports and the Verilog implementation ports using the SystemVerilog streaming operator (>>). It instantiates the gate-level netlist created for the DUT, TOP_I_i_IFC_mp_T_array_1_0_2_logic_DQLcWqa_.

```
// For simulation only. Do not modify.
module TOP_svsim #(parameter type T = shortint) (interface.mp i);
TOP_I_i_IFC_mp_T_array_1_0_2_logic_DQLcWqa_TOP_I_i_IFC_mp_T_array_1_0_2_
logic_DQLcWqa_( {>>{ i.x }}, {>>{ i.y }} );
endmodule
```

Gate-level netlist

The port mapping is expressed as a port connection to the top_inst instance in positional notation. The top.sv module uses the `shortint` type that is overridden but the MY_T type in the top_inst instance.

```
module TOP_I_i_IFC_mp_T_array_1_0_2_logic_DQLcWqa_ ( \i.x , \i.y  );
   input [0:2] \i.x ;
   output [0:2] \i.y ;

assign \i.y  [0] = \i.x  [0];
assign \i.y  [1] = \i.x  [1];
assign \i.y  [2] = \i.x  [2];
endmodule
```

Customizing Elaboration Reports

By default, the tool displays inferred sequential elements, MUX_OPs, and inferred three-state elements in elaboration reports using the `basic` setting, as shown in Table 1-3. You can customize the report by setting the `hdlin_reporting_level` variable to `none`, `comprehensive`, or `verbose`. A true, false, or verbose setting indicates that the corresponding information is included, excluded, or detailed respectively in the report.

Table 1-3 Basic hdlin_reporting_level Variable Settings

Information displayed **(information keyword)**	**basic (default)**	**none**	**comprehensive**	**verbose**
Floating net to ground connections (`floating_net_to_ground`)	false	false	true	true
Inferred state variables (`fsm`)	false	false	true	true
Inferred sequential elements (`inferred_modules`)	true	false	true	true
MUX_OPs (`mux_op`)	true	false	true	true

Table 1-3 Basic hdlin_reporting_level Variable Settings (Continued)

Information displayed **(information keyword)**	**basic (default)**	**none**	**comprehensive**	**verbose**
Synthetic cells (`syn_cell`)	false	false	true	true
Inferred three-state elements (`tri_state`)	true	false	true	true

In addition to the four settings, you can customize the report by specifying the add (+) or subtract (-) option. For example, to report floating-net-to-ground connections, synthetic cells, inferred state variables, and verbose information for inferred sequential elements, but not MUX_OPs or inferred three-state elements, enter

```
dc_shell> set_app_var hdlin_reporting_level verbose-mux_op-tri_state
```

Setting the `hdlin_reporting_level` variable as follows is equivalent to the set_app_var hdlin_reporting_level comprehensive command, which reports comprehensive information.

```
dc_shell> set_app_var hdlin_reporting_level \
   basic+floating_net_to_ground+syn_cell+fsm
```

Reporting Elaboration Errors in the Hierarchy

The tool elaborates designs in a top-down order, and elaboration errors of a top-level module prohibit the elaboration of all associated submodules. To continue the elaboration regardless of the top-level errors, use the `hdlin_elab_errors_deep` variable.

By default, the tool reports only the top-level errors during elaboration. To report all errors in the hierarchy, you must fix the top-level errors and then repeat the elaboration step. However, if you set the `hdlin_elab_errors_deep` variable to `true`, the tool reports all elaboration errors in the hierarchy in one elaboration step.

To report all elaboration errors in the hierarchy, follow these steps:

1. Identify and fix all syntax errors in the design.
2. Set the `hdlin_elab_errors_deep` variable to `true`.

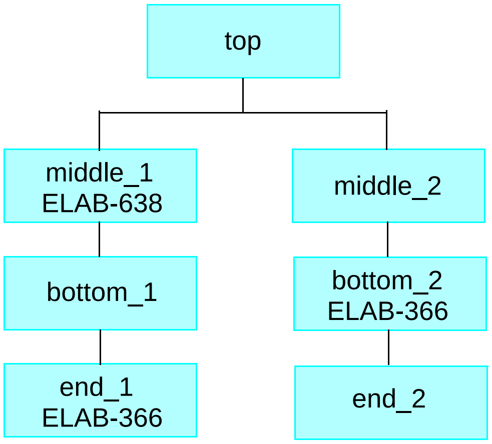
   The tool runs in the RTL debug mode.
3. Elaborate your design using the `elaborate` command.
4. Fix all errors, and fix warnings as needed.

5. Set the `hdlin_elab_errors_deep` variable to `false`.
6. Elaborate the design that contains no errors.
7. Proceed with the synthesis flow.

Example of Reporting Elaboration Errors

This SystemVerilog example uses the top design, as shown in Figure 1-6, to report all elaboration errors in the hierarchy.

Figure 1-6 Hierarchical Design



Example 1-5 SystemVerilog RTL of the top Design

```
module top (input  a, b, output o1, o2 );
middle_1 M1 (a, b, o1);
middle_2 M2 (a, b, o2);
endmodule

module middle_1 (input  a, b, output o);
logic w;
bottom_1 B1 (a, b, w);
logic   bad;
assign bad = a&b&w;
assign bad = 1'b1;
assign  o = bad; // ELAB-368 error
endmodule

module bottom_1 (input  a, b, output c);
end_1 B1 (a, b, c);
endmodule

module end_1 (input  a, b, output c);
logic bad;
assign bad = a;
assign bad = a|b;
assign c =  bad; // ELAB-366 error
endmodule

module middle_2 (input  a, b, output o );
bottom_2 B2 (a, b, o);
endmodule
module bottom_2 (input a, input b, output c);
logic w;
end_2 B2 (a, b, w);
logic   bad;
assign bad = w|a;
assign bad = w&b; // ELAB-366 error
assign c = bad;
endmodule // sub3

module end_2 (input  a, b, output c);
assign c = a^b;
endmodule
```

Example 1-6 Elaboration Results of hdlin_elab_errors_deep Set to false

```
dc_shell> set_app_var hdlin_elab_errors_deep false
false
dc_shell> analyze -f sverilog rtl/test.sv
Running PRESTO HDLC
Searching for ./rtl/test.sv
Compiling source file ./rtl/test.sv
Presto compilation completed successfully.
1
dc_shell> elaborate top
Running PRESTO HDLC
Presto compilation completed successfully.
Elaborated 1 design.
Current design is now 'top'.
Information: Building the design 'middle_1'. (HDL-193)
Error:  ./rtl/test.sv:12: Net 'bad', or a directly connected net, is
driven by more than one source, and at least one source is a constant
net. (ELAB-368)
*** Presto compilation terminated with 1 errors. ***
Information: Building the design 'middle_2'. (HDL-193)
Presto compilation completed successfully.
Information: Building the design 'bottom_2'. (HDL-193)
Error:  ./rtl/test.sv:36: Net 'bad' or a directly connected net is driven
by more than one source, and not all drivers are three-state. (ELAB-366)
*** Presto compilation terminated with 1 errors. ***
Warning: Design 'top' has '2' unresolved references. For more detailed
information, use the "link" command. (UID-341)
1
dc_shell> list_designs
middle_2 top (*)
```

Example 1-7 Elaboration Results of hdlin_elab_errors_deep Set to true

```
dc_shell> set hdlin_elab_errors_deep true
true
dc_shell> analyze -f sverilog rtl/test.sv
Running PRESTO HDLC
Searching for ./rtl/test.sv
Compiling source file ./rtl/test.sv
Presto compilation completed successfully.
1
dc_shell> elaborate top
Running PRESTO HDLC
*** Presto compilation run in rtl debug mode. ***
Presto compilation completed successfully.
Elaborated 1 design.
Current design is now 'top'.
Information: Building the design 'middle_1'. (HDL-193)
*** Presto compilation run in rtl debug mode. ***
Error:  ./rtl/test.sv:12: Net 'bad', or a directly connected net, is
driven by more than one source, and at least one source is a constant
net. (ELAB-368)
Presto compilation completed successfully.
Information: Building the design 'middle_2'. (HDL-193)
*** Presto compilation run in rtl debug mode. ***
Presto compilation completed successfully.
Information: Building the design 'bottom_1'. (HDL-193)
*** Presto compilation run in rtl debug mode. ***
Presto compilation completed successfully.
Information: Building the design 'bottom_2'. (HDL-193)
*** Presto compilation run in rtl debug mode. ***
Error:  ./rtl/test.sv:36: Net 'bad' or a directly connected net is driven
by more than one source, and not all drivers are three-state. (ELAB-366)
Presto compilation completed successfully.
Information: Building the design 'end_1'. (HDL-193)
*** Presto compilation run in rtl debug mode. ***
Error:  ./rtl/test.sv:23: Net 'bad' or a directly connected net is driven
by more than one source, and not all drivers are three-state. (ELAB-366)
Presto compilation completed successfully.
Information: Building the design 'end_2'. (HDL-193)
*** Presto compilation run in rtl debug mode. ***
Presto compilation completed successfully.
1
dc_shell> list_designs
Warning: No designs to list. (UID-275)
0
```

As shown in Example 1-6, the tool reports the following two errors at the top level by default:

- ELAB-368 error from the middle_1 module
- ELAB-366 error from the bottom_2 module

To find the ELAB-366 error in the end1 submodule as shown in Example 1-7, you must fix the error in the middle_1 module. When you set the `hdlin_elab_errors_deep` variable to `true`, the tool reports all errors in the hierarchy in one elaboration step:

- ELAB-368 error from the middle_1 module
- ELAB-366 error from the bottom_2 module
- ELAB-366 error from the end_1 module

The following restrictions apply when the `hdlin_elab_errors_deep` variable is set to `true`:

- No designs are saved because the designs could be erroneous.

  The tool does not create designs when this variable is set to `true`. If you run the `list_designs` command, the tool reports the following warning:

  ```
  Warning: No designs to list (UID-275)
  ```

- You should use the `analyze` command rather than the `read_file` command to read the design because the `read_file` command has no linking functionality and does not accept command-line parameter specifications.
- All syntax errors are reported when you run the `analyze` command. The HDL Compiler tool is not a linting tool, so you should use the `check_design` command in the Design Compiler tool for linting.
- The elaboration runtime might increase slightly.

Querying Information about RTL Preprocessing

You can query information about preprocessing of the RTL, including macro definitions, macro expansions, and evaluations of the conditional statements. You use this information to debug design issues, especially for designs with a large number of macros. To query the preprocessing information, set the `hdlin_analyze_verbose_mode` variable to one of the values listed in Table for the type of information to be reported. The default is 0.

Variable setting	Information reported
0	No preprocessing information.
1	Macro definitions (described by the `define directive in the RTL and specified by the `-define` option on the command line) and evaluations of the conditional statements.
2	Macro expansions and the information reported when the variable is set to 1.

The following example shows how to report preprocessing information by using the `hdlin_analyze_verbose_mode` variable:

- example.v file

```
`define MYMACRO 1'b0

module m (
   input in1,
   output out1
);

`ifdef MYRTL
   assign out1 = `MYMACRO;
`else
   assign out1 = in1;
`endif
endmodule
```

- Excerpt from the log file

```
dc_shell> set hdlin_analyze_verbose_mode 1
1

# Generates messages that `ifdef being skipped and `else analyzed
dc_shell> analyze -f sverilog example.v
...
Information: ./example.v:6: Skipping `ifdef then clause because MYRTL
is not defined.(VER-7)
Information: ./example.v:8: Analyzing `else clause.(VER-7)
...

# Generates messages that `ifdef is analyzed and `else skipped
dc_shell> analyze -f sverilog -define MYRTL example.v
...
Information: ./example.v:6: Analyzing `ifdef then clause because MYRTL
is defined.(VER-7)
Information: ./example.v:8: Skipping `else clause.(VER-7)
...
dc_shell> set hdlin_analyze_verbose_mode 2
2

# Generates messages about evaluation of macro `MUMACRO to 1’b0
dc_shell> analyze -f sverilog -define MYRTL example.v
...
Information: ./example.v:6: Analyzing `ifdef then clause because MYRTL
is defined.(VER-7)
Information: ./example.v:7: Macro |`MYMACRO| expanded to |1'b0|.
(VER-7)
Information: ./example.v:8: Skipping `else clause.(VER-7)
...
```

Reporting HDL Compiler Variables

To get a list of variables that affect RTL reading, use the `printvar` command with the `hdlin*` argument.

All HDL Compiler variables are prefixed with `hdlin`. Running the following command gives you a list of HDL Compiler variables and their defaults:

```
dc_shell> printvar hdlin*
```

Other variables that affect RTL reading include the ones prefixed with `template` and `bus*style`. Use the following commands to report these variables:

```
dc_shell> printvar template*
dc_shell> printvar bus*style
```

For more information about a specific variable, see the man page. For example,

```
dc_shell> man hdlin_analyze_verbose_mode
```

Parameterized Designs

Declaring Parameters Without a Default

Port list parameters can be declared with or without a default. If you declare a parameter without a default, you must specify an override value in every instantiation to prevent a compile error.

As per the *IEEE Std 1800-2012*, parameters without a default are only supported in a parameter port list.

The following design declares the SIZE parameter with no default, and the INSIZE parameter with a default of eight:

Example 1-8 Port List Parameter without a Default

```
module sub #(parameter SIZE)(
   output [SIZE-1:0] out,
   input [SIZE-1:0] in
);

   assign out = ~in;
endmodule

module top (
   output [7:0] b,
   input [7:0] a
);

   sub #(.SIZE(8)) U1 (b,a); //Override value (required)
endmodule
```

The following design declares the SIZE parameter with no default, and the INSIZE parameter with a default of eight:

Example 1-9 Declaring a Parameterized Design

```
module sub #(parameter SIZE, INSIZE=8) (
   output [SIZE-1:0] out,
   input [INSIZE-1:0] in
);
assign out = ~in;
endmodule
```

Instantiating a Parameterized Design

You must specify an override value for the SIZE parameter in every instantiation of the design. The INSIZE parameter can be overridden, or the default can be used. The following examples illustrate the different ways to instantiate a parameterized design.

Example 1-10 overrides both parameters and instantiates U1, a 4-bit wide inverter block.

Example 1-10 Instantiating a Parameterized Design With Override Values.

```
module top (
   output[3:0] b,
   input [3:0] a
);
sub #(.SIZE(4), .INSIZE(4)) U1(.out(b),.in(a));
endmodule
```

In Example 1-11 U2 instantiation, the SIZE parameter is overridden to 8, and the default is used for INSIZE (also 8), creating an 8-bit wide inverter block.

Example 1-11 Instantiating a Parameterized Design With Defaults.

```
module top (
   output[7:0] b,
   input [7:0] a
);
sub #(.SIZE(8)) U2(.out(b),.in(a));
endmodule
```

Example 1-12 does not override either parameter. Parameter SIZE is undefined (no default or override value) causing a compile error.

Example 1-12 Incorrect instantiation: No Override Value or Default for Parameter SIZE.

```
module top (
   output[7:0] b,
   input [7:0] a
);
sub U3(.out(b),.in(a));
endmodule
```

Specifying Parameter Values With the Elaborate Command

Another method to build a parameterized design is with the `elaborate` command. The syntax of the command is:

```
elaborate template_name -parameters parameter_list
```

The syntax of the parameter specifications includes strings, integers, and constants using the following formats `b,`h, b, and h.

You can store parameterized designs in user-specified design libraries. For example,

```
analyze -format verilog n-register.v -library mylib
```

This command stores the analyzed results of the design contained in file n-register.v in a user-specified design library, mylib.

To verify that a design is stored in memory, use the `report_design_lib work` command. The `report_design_lib` command lists designs that reside in the indicated design library.

When a design is built from a template, only the parameters you indicate when you instantiate the parameterized design are used in the template name. For example, suppose the template ADD has parameters N, M, and Z. You can build a design where N = 8, M = 6, and Z is left at its default. The name assigned to this design is `ADD_N8_M6`. If no parameters are listed, the template is built with the default, and the name of the created design is the same as the name of the template.

Designs which declare parameters without a default must have an override value at instantiation or a compile error occurs. In the preceding ADD example, parameter Z must have a default, but N and M do not.

The model in Example 1-13 uses a parameter to determine the register bit-width; the default width is declared as 8.

Example 1-13 Register Model

```
module DFF ( in1, clk, out1 );
  parameter SIZE = 8;
  input [SIZE-1:0] in1;
  input clk;
  output [SIZE-1:0] out1;
  reg [SIZE-1:0] out1;
  reg [SIZE-1:0] tmp;
always @(clk)
   if (clk == 0)
      tmp = in1;
   else //(clk == 1)
      out1 <= tmp;
endmodule
```

If you want an instance of the register model to have a bit-width of 16, use the `elaborate` command to specify this as follows:

```
elaborate DFF -param SIZE=16
```

The `list_designs` command shows the design, as follows:

```
DFF_SIZE16 (*)
```

Using the `read_verilog` command to build a design with parameters is not recommended because you can build a design only with the default of the parameters.

You also need to either set the `hdlin_auto_save_templates` variable to `true` or insert the `template` directive in the module, as follows:

```
module DFF ( in1, clk, out1 );
  parameter SIZE = 8;
  input [SIZE-1:0] in1;
  input clk;
  output [SIZE-1:0] out1;
  // synopsys template
...
```

The `hdlin_template_naming_style`, `hdlin_template_parameter_style`, and `hdlin_template_separator_style` variables control the naming convention for templates.

Defining Macros

You can use `analyze -define` to define macros on the command line.

Note:

When using the `-define` option with multiple analyze commands, you must remove any designs in memory before analyzing the design again. To remove the designs, use the `remove_design -all` command. Because elaborated designs in memory have no timestamps, the tool cannot determine whether the analyzed file has been updated. The tool might assume that the previously elaborated design is up-to-date and reuse it.

See Also

- `define

Predefined Macros

You can also use the following predefined macros:

- `SYNTHESIS`—Used to specify simulation-only code, as shown in Example 1-14.

Example 1-14 Using SYNTHESIS and `ifndef ... `endif Constructs

```
module dff_async (RESET, SET, DATA, Q, CLK);
  input CLK;
  input RESET, SET, DATA;
  output Q;
  reg Q;
  // synopsys one_hot "RESET, SET"

  always @(posedge CLK or posedge RESET or posedge SET)
     if (RESET)
         Q <= 1'b0;
     else if (SET)
         Q <= 1'b1;
     else Q <= DATA;
    `ifndef SYNTHESIS
       always @ (RESET or SET)
         if (RESET + SET > 1)
         $write ("ONE-HOT violation for RESET and SET.");
    `endif
  endmodule
```

In this example, the `SYNTHESIS` macro and the `` `ifndef ... `endif `` constructs determine whether or not to execute the simulation-only code that checks if the `RESET` and `SET` signals are asserted at the same time. The main always block is both simulated and synthesized; the block wrapped in the `` `ifndef `` ... `` `endif `` construct is executed only during simulation.

- `VERILOG_1995`, `VERILOG_2001`, `VERILOG_2005`—Used for conditional inclusion of Verilog 1995, Verilog 2001, or Verilog 2005 features respectively. When you set the `hdlin_vrlg_std` variable to `1995`, `2001`, or `2005`, the corresponding macro

VERILOG_1995, VERILOG_2001, or VERILOG_2005 is predefined. By default, the hdlin_vrlg_std variable is set to 2005.

Global Macro Reset: `undefineall

The `` `undefineall `` directive is a global reset for all macros that causes all the macros defined earlier in the source file to be reset to undefined.

Using $display During RTL Elaboration

The $display system task is usually used to report simulation progress. In synthesis, HDL Compiler executes $display calls as it sees them and executes all the display statements on all the paths through the program as it elaborates the design. It usually cannot tell the value of variables, except compile-time constants like loop iteration counters.

Note that because HDL Compiler executes all $display calls, error messages from the Verilog source can be executed and can look like unexpected messages.

Using $display is useful for printing out any compile-time computations on parameters or the number of times a loop executes, as shown in Example 1-15.

Example 1-15 $display Example

```
module F (in, out, clk);
  parameter SIZE = 1;
  input [SIZE-1: 0] in;
  output  [SIZE-1: 0] out;
  reg [SIZE-1: 0] out;
  input clk;
  // ...
  `ifdef SYNTHESIS
    always $display("Instantiating F, SIZE=%d", SIZE);
  `endif
endmodule

module TOP (in, out, clk);
  input  [33:0] in;
  output [33:0] out;
  input clk;

  F #( 2)  F2 (in[ 1:0] ,out[ 1:0], clk);
  F #(32) F32 (in[33:2], out[33:2], clk);
endmodule
```

HDL Compiler produces output such as the following during elaboration:

```
dc_shell> elaborate  TOP
Running HDLC
```

```
HDLC compilation completed successfully.
Elaborated 1 design.
Current design is now 'TOP'.
Information: Building the design 'F' instantiated from design 'TOP' with
        the parameters "2". (HDL-193)
$display output: Instantiating F, SIZE=2
HDLC compilation completed successfully.
Information: Building the design 'F' instantiated from design 'TOP' with
        the parameters "32". (HDL-193)
$display output: Instantiating F, SIZE=32
HDLC compilation completed successfully.
```

Elaboration System Tasks

SystemVerilog elaboration system tasks provide the ability to check parameter values used in module instantiations and report status, warnings, or errors during elaboration. Four elaboration system tasks are supported.

SystemVerilog syntax	Tool output format	Message number
`$fatal(finish_num, user_message);`	`Error: file name: line number: $fatal(finish_num) output: user_message`	`ELAB-2050`
`$error(user_message);`	`Error: file name: line number: $error output: user_message`	`ELAB-2051`
`$warning(user_message);`	`Warning: file name: line number: $warning output: user_message`	`ELAB-2052`
`$info(user_message);`	`Information: file name: line number: $info output: user_message`	`ELAB-2053`

SystemVerilog elaboration system tasks have the following details:

- These elaboration system tasks are called outside procedural code in a `generate` or conditional `generate` construct.
- The `user_message` argument contains a formatting string and constant expressions, including constant function calls.
- The `$fatal` and `$error` tasks terminate HDL-Compiler compilation with an error.
- The `finish_num` argument only applies to the `$fatal` task. It has a value of 0, 1, or 2. It is printed in the output message, but the value has no meaning to the tool.
- The `$warning` and `$info` tasks output a message, but continue compilation without an error.

Note:

The elaboration system tasks use the same names as the SystemVerilog Severity Tasks. The tasks are differentiated by the context of the system task call. The Severity Tasks are called within procedural code (for example inside an `always` block) while the elaboration system tasks must be called from outside procedural code. HDL-Compiler continues to parse and ignore SystemVerilog Severity Tasks.

For more information, see the *IEEE Std 1800-2012* (Sections 20.10 and 20.11).

Example 1-16 Checking a Parameter Value With a SystemVerilog Elaboration System Task

```
module sub #(parameter SIZE) (
   output [SIZE-1:0] out,
   input [SIZE-1:0] in);
if ((SIZE < 1) || (SIZE > 8)) // conditional generate construct
   $fatal(1, “Parameter SIZE has an invalid value of %d”, SIZE);
assign out = ~in;
endmodule

module top (
   output [15:0] out,
   input [15:0] in);
sub #(.SIZE(16)) U1 (.out(out), .in(in));
endmodule
```

Example 1-17 Error Messages When Elaborating a Design With an Invalid Parameter Value

```
dc_shell> elaborate top
Error:  ./parameter.sv:9: $fatal(1) output: Parameter SIZE has an invalid
value of 16 (ELAB-2050)
*** Presto compilation terminated with 1 error. ***
```

Inputs and Outputs

This section contains the following topics:

- Input Descriptions
- Design Hierarchy
- Component Inference and Instantiation
- Naming Considerations
- Generic Netlists
- Error Messages

Input Descriptions

Verilog code input to HDL Compiler can contain both structural and functional (RTL) descriptions. A Verilog structural description can define a range of hierarchical and gate-level constructs, including module definitions, module instantiations, and netlist connections.

The functional elements of a Verilog description for synthesis include

- always statements
- Tasks and functions
- Assignments
  - Continuous—are outside always blocks
  - Procedural—are inside always blocks and can be either blocking or nonblocking
- Sequential blocks (statements between a begin and an end)
- Control statements
- Loops—for, while, forever

  The forever loop is only supported if it has an associated disable condition, making the exit condition deterministic.
- case and if statements

Functional and structural descriptions can be used in the same module, as shown in Example 1-18.

In this example, the `detect_logic` function determines whether the input bit is a 0 or a 1. After making this determination, `detect_logic` sets `ns` to the next state of the machine. An always block infers flip-flops to hold the state information between clock cycles. These statements use a functional description style. A structural description style is used to instantiate the three-state buffer t1.

Example 1-18 Mixed Structural and Functional Descriptions

```
// This finite state machine (Mealy type) reads one
// bit per clock cycle and detects three or more
// consecutive 1s.
module three_ones( signal, clock, detect, output_enable );
input signal, clock, output_enable;
output detect;
// Declare current state and next state variables.
reg [1:0] cs;
reg [1:0] ns;
wire ungated_detect;

// Declare the symbolic names for states.
parameter NO_ONES = 0, ONE_ONE = 1,
          TWO_ONES = 2, AT_LEAST_THREE_ONES = 3;
// ************* STRUCTURAL DESCRIPTION ****************
// Instance of a three-state gate that enables output
three_state t1 (ungated_detect, output_enable, detect);

// ************* FUNCTIONAL DESCRIPTION ****************
// always block infers flip-flops to hold the state of
// the FSM.
always @ ( posedge clock ) begin
     cs = ns;
end
// Combinational function
function detect_logic;
input [1:0] cs;
input signal;

begin
   detect_logic = 0;    //default
   if ( signal == 0 )   //bit is zero
      ns = NO_ONES;
   else                 //bit is one, increment state
   case (cs)
   NO_ONES: ns = ONE_ONE;
   ONE_ONE: ns = TWO_ONES;
   TWO_ONES, AT_LEAST_THREE_ONES:
   begin
      ns = AT_LEAST_THREE_ONES;
      detect_logic = 1;
   end
   endcase
end
endfunction
assign ungated_detect = detect_logic( cs, signal );
endmodule
```

Design Hierarchy

The HDL Compiler tool maintains the hierarchical boundaries you define when you use structural Verilog. These boundaries have two major effects:

- Each module in HDL descriptions is synthesized separately and maintained as a distinct design. The constraints for the design are maintained, and each module can be optimized separately in the Design Compiler tool.
- Module instantiations within HDL descriptions are maintained during input. The instance names that you assign to user-defined components are propagated through the gate-level implementation.

Note:

The HDL Compiler tool does not automatically create the hierarchy for nonstructural Verilog constructs, such as blocks, loops, functions, and tasks. These elements of HDL descriptions are translated in the context of their designs. To group the gates in a block, function, or task, you can use the `group -hdl_block` command after reading in a Verilog design. The HDL Compiler tool supports only the top-level `always` blocks. Due to optimization, small blocks might not be available for grouping. To report blocks available for grouping, use the `list_hdl_blocks` command. For information about how to use the `group` command with Verilog designs, see the man page.

Component Inference and Instantiation

There are two ways to define components in your Verilog description:

- You can directly instantiate registers into a Verilog description, selecting from any element in your ASIC library, but the code is technology dependent and the description is difficult to write.
- You can use Verilog constructs to direct HDL Compiler to infer registers from the description. The advantages are these:
  - The Verilog description is easier to write and the code is technology independent.
  - This method allows Design Compiler to select the type of component inferred, based on constraints.

If a specific component is necessary, use instantiation.

Naming Considerations

The bus output instance names are controlled by the following variables: `bus_naming_style` (controls names of elements of Verilog arrays) and `bus_inference_style` (controls bus inference style). To reduce naming conflicts, use caution when applying nondefault naming styles. For details, see the man pages.

Generic Netlists

After HDL Compiler reads a design, it creates a generic netlist consisting of generic components, such as SEQGENs.

For example, after HDL Compiler reads the my_fsm design in Example 1-19, it creates the generic netlist shown in Example 1-20.

Example 1-19 my_fsm Design

```
module my_fsm (clk, rst, y);
      input clk, rst;
      output y;
      reg  y;
      reg [2:0] current_state;
       parameter
           red    = 3'b001,
           green  = 3'b010,
           yellow = 3'b100;
      always @ (posedge clk or posedge rst)
          if (rst)
               current_state = red;
          else
              case (current_state)
                  red:
                      current_state = green;
                  green:
                      current_state = yellow;
                  yellow:
                      current_state = red;
              default:
                      current_state = red;
              endcase
      always @ (current_state)
          if (current_state == yellow)
              y = 1'b1;
          else
              y = 1'b0;
      endmodule
```

After the tool reads in the my_fsm design, it outputs the generic netlist shown in Example 1-20.

Example 1-20 Generic Netlist

```
module my_fsm ( clk, rst, y );
  input clk, rst;
  output y;
  wire   N0, N1, N2, N3, N4, N5, N6, N7, N8, N9, N10, N11, N12, N13, N14,
N15,
         N16, N17, N18;
  wire   [2:0] current_state;

  GTECH_OR2 C10 ( .A(current_state[2]), .B(current_state[1]), .Z(N1) );
  GTECH_OR2 C11 ( .A(N1), .B(N0), .Z(N2) );
  GTECH_OR2 C14 ( .A(current_state[2]), .B(N4), .Z(N5) );
  GTECH_OR2 C15 ( .A(N5), .B(current_state[0]), .Z(N6) );
  GTECH_OR2 C18 ( .A(N15), .B(current_state[1]), .Z(N8) );
  GTECH_OR2 C19 ( .A(N8), .B(current_state[0]), .Z(N9) );
  \**SEQGEN**  \current_state_reg[2]  ( .clear(rst), .preset(1'b0),
        .next_state(N7), .clocked_on(clk), .data_in(1'b0), .enable(1'b0),
.Q(
        current_state[2]), .synch_clear(1'b0), .synch_preset(1'b0),
        .synch_toggle(1'b0), .synch_enable(1'b1) );
  \**SEQGEN**  \current_state_reg[1]  ( .clear(rst), .preset(1'b0),
        .next_state(N3), .clocked_on(clk), .data_in(1'b0), .enable(1'b0),
.Q(
        current_state[1]), .synch_clear(1'b0), .synch_preset(1'b0),
        .synch_toggle(1'b0), .synch_enable(1'b1) );
  \**SEQGEN**  \current_state_reg[0]  ( .clear(1'b0), .preset(rst),
        .next_state(N14), .clocked_on(clk), .data_in(1'b0),
.enable(1'b0), .Q(
        current_state[0]), .synch_clear(1'b0), .synch_preset(1'b0),
        .synch_toggle(1'b0), .synch_enable(1'b1) );
  GTECH_NOT I_0 ( .A(current_state[2]), .Z(N15) );
  GTECH_OR2 C47 ( .A(current_state[1]), .B(N15), .Z(N16) );
  GTECH_OR2 C48 ( .A(current_state[0]), .B(N16), .Z(N17) );
  GTECH_NOT I_1 ( .A(N17), .Z(N18) );
  GTECH_OR2 C51 ( .A(N10), .B(N13), .Z(N14) );
  GTECH_NOT I_2 ( .A(current_state[0]), .Z(N0) );
  GTECH_NOT I_3 ( .A(N2), .Z(N3) );
  GTECH_NOT I_4 ( .A(current_state[1]), .Z(N4) );
  GTECH_NOT I_5 ( .A(N6), .Z(N7) );
  GTECH_NOT I_6 ( .A(N9), .Z(N10) );
  GTECH_OR2 C68 ( .A(N7), .B(N3), .Z(N11) );
  GTECH_OR2 C69 ( .A(N10), .B(N11), .Z(N12) );
  GTECH_NOT I_7 ( .A(N12), .Z(N13) );
  GTECH_BUF B_0 ( .A(N18), .Z(y) );
endmodule
```

The `report_cell` command lists the cells in a design. Example 1-21 shows the `report_cell` output for my_fsm design.

Example 1-21   report_cell Output

```
dc_shell> report_cell
Information: Updating design information... (UID-85)

****************************************
Report : cell
Design : my_fsm
Version: B-2008.09
Date   : Tue Jul 15 07:11:02 2008
****************************************

Attributes:
    b - black box (unknown)
    c - control logic
    h - hierarchical
    n - noncombinational
    r - removable
    u - contains unmapped logic

Cell                      Reference       Library             Area
Attributes
-------------------------------------------------------------------------
B_0                       GTECH_BUF       gtech           0.000000  u
C10                       GTECH_OR2       gtech           0.000000  u
C11                       GTECH_OR2       gtech           0.000000  c, u
C14                       GTECH_OR2       gtech           0.000000  u
C15                       GTECH_OR2       gtech           0.000000  c, u
C18                       GTECH_OR2       gtech           0.000000  u
C19                       GTECH_OR2       gtech           0.000000  c, u
C47                       GTECH_OR2       gtech           0.000000  u
C48                       GTECH_OR2       gtech           0.000000  u
C51                       GTECH_OR2       gtech           0.000000  u
C68                       GTECH_OR2       gtech           0.000000  c, u
C69                       GTECH_OR2       gtech           0.000000  c, u
I_0                       GTECH_NOT       gtech           0.000000  u
I_1                       GTECH_NOT       gtech           0.000000  u
I_2                       GTECH_NOT       gtech           0.000000  u
I_3                       GTECH_NOT       gtech           0.000000  u
I_4                       GTECH_NOT       gtech           0.000000  u
I_5                       GTECH_NOT       gtech           0.000000  u
I_6                       GTECH_NOT       gtech           0.000000  u
I_7                       GTECH_NOT       gtech           0.000000  c, u
current_state_reg[0]      **SEQGEN**                      0.000000  n, u
current_state_reg[1]      **SEQGEN**                      0.000000  n, u
current_state_reg[2]      **SEQGEN**                      0.000000  n, u
-------------------------------------------------------------------------
Total 23 cells                                            0.000000
1
```

Error Messages

If the design contains syntax errors, these are typically reported as ver-type errors; mapping errors, which occur when the design is translated to the target technology, are reported as elab-type errors. An error causes the script you are currently running to terminate; an error terminates your Design Compiler session. Warnings are errors that do not stop the read from completing, but the results might not be as expected.

You set the `suppress_errors` variable to suppress warnings when reading Verilog source files. By default, the tool does not suppress any warnings. You can specify a list of warning codes for which warning messages are to be suppressed during the current shell session. This variable has no effect on error messages that stop the reading process.

You can also use this variable to disable specific warnings: set `suppress_errors` to a space-separated string of the error ID codes you want suppressed. Error ID codes are printed immediately after warning and error messages. For example, to suppress the following warning, set the variable to HDL-193:

```
Warning: Assertion statements are not supported. They are
ignored near symbol "assert" on line 24 (HDL-193).
```

2

Global Name Space ($unit)

The following topics describe how the Synopsys synthesis tools support SystemVerilog global declarations:

- About the Global Name Space
- Reading Designs With $unit
- Synthesis Restrictions for $unit

About the Global Name Space

The Synopsys synthesis tools support SystemVerilog global declarations through the global name space ($unit). This is a top-level name space, which is outside any modules and is visible to all modules at all hierarchical levels. While the global name space allows you to share common function and variable declarations among several modules, various tools treat the declarations in $unit differently. As a result, you should use packages instead of $unit for this purpose.

You can specify the following objects in $unit:

- Type definitions
- Enumerated types
- Local parameter (`localparam`) declarations

  The `parameter` keyword can also be used to declare local parameters.
- Automatic tasks and automatic functions
- Constant declarations
- Simulation-related constructs, such as `timeunit`, `timeprecision`, and `timescale`
- Directives
- Verilog 2001 compiler directives, such as `` `include `` and `` `define ``

Example 2-1 shows how to use $unit. This example combines objects that include enums, typedefs, parameter declarations, tasks, functions, and structure variables in $unit; these objects are used by the test module.

Example 2-1 $unit Usage

```
typedef enum logic {FALSE, TRUE} my_reg;
localparam a = ’1;
typedef struct {
   my_reg [a:0] orig;
   my_reg [a:0] orig_inverted;
} my_struct;

function automatic my_reg [a:0] invert (my_reg [a:0] value);
return (~value);
endfunction

task automatic check_invert(my_struct struct_in);
begin
   if(struct_in.orig == struct_in.orig_inverted)
      $display("\n ERROR: Value not inverted\n");
   else
      $display("\n CORRECT: Value is inverted\n");
```

```
end
endtask

module test(
   input my_reg [a:0] din,
   output my_struct dout
);

assign dout.orig = din;
assign dout.orig_inverted = invert(din);
always_comb check_invert(dout);
endmodule
```

See Also

- Packages
- Specifying Global Files for Each analyze Command

Reading Designs With $unit

The following topics describe the guidelines on how to read designs that use the $unit name space:

- Defining Objects Before Use
- Specifying Global Files First
- Specifying Global Files for Each analyze Command

The examples provided in each topic use the `analyze` command, but the guidelines apply to both the `analyze` and read commands, such as `read_file` and `read_sverilog`.

The following four files are used in the examples:

- global.sv—contains a global declaration of the structure data type that is used by other modules.

  ```
  typedef struct {
     logic a, b;
  } data;
  ```

- and_struct.sv—assigns the structure data type in $unit to the din input.

  ```
  module and_struct(
     input data din,
     output a_and_b
  );
  assign a_and_b = din.a & din.b;
  endmodule
  ```

- or_struct.sv—assigns the structure data type in $unit to the din input.

  ```
  module or_struct(
     input data din,
     output a_or_b
  );
  assign a_or_b = din.a | din.b;
  endmodule
  ```

- top.sv—assigns the structure data type in $unit to the din input and instantiates the and_struct module with the name u1 and the or_struct module with the name u2.

  ```
  module top(
     input data din,
     output or_result, and_result
  );
  and_struct u1(.din, .a_and_b(and_result));
  or_struct u2(.din, .a_or_b(or_result));
  endmodule
  ```

Defining Objects Before Use

You must define an object before you use it. In Example 2-2, one `analyze` command reads all the specified files. The first file is global.sv, which is followed by three files, and_struct.sv, or_struct.sv, and top.sv. These three files all use the global declaration in the global.sv file. Because the global.sv file is read first, the tool generates the netlist without errors.

Example 2-2

```
dc_shell> analyze -format sverilog \
   {global.sv and_struct.sv or_struct.sv top.sv}
dc_shell> elaborate top
dc_shell> write -format verilog -hierarchy -output gtech.sample1.v
```

However, if the global.sv file is read after the other files, as shown in Example 2-3, the tool issues a VER-518 error message.

Example 2-3

```
dc_shell> analyze -format sverilog \
   {and_struct.sv or_struct.sv top.sv global.sv}
Running HDLC
Searching for ./and_struct.sv
Searching for ./or_struct.sv
Searching for ./top.sv
Searching for ./global.sv
Compiling source file ./and_struct.sv
Error:  ./and_struct.sv:1: Syntax error at or near token 'din'. (VER-294)
Compiling source file ./or_struct.sv
Error:  Cannot recover from previous errors. (VER-518)
*** HDLC compilation terminated with 2 errors. ***
0
```

Specifying Global Files First

When using read commands to read files individually, you must include all applicable global files for each read command by using either one of the following two methods. The tool issues an error message if you mix both methods.

- The $unit method

  In Example 2-4, a separate `analyze` command reads each of the files, and_struct.sv, or_struct.sv, and top.sv, individually. Because a separate $unit name space is created for each `analyze` command, you must specify the global file first for each `analyze` command.

Example 2-4

```
dc_shell> analyze -format sverilog {global.sv and_struct.sv}
dc_shell> analyze -format sverilog {global.sv or_struct.sv}
dc_shell> analyze -format sverilog {global.sv top.sv}
dc_shell> elaborate top
dc_shell> write -format verilog -hierarchy -output gtech.sample3.v
```

  Because all required global declarations are available in $unit when the modules are read, the tool generates the netlist without errors.

  However, if the global file is not read before each design file, as shown in Example 2-5, the tool issues an error message. The error occurs because the structure data type is applied to the din input before the data type is defined.

Example 2-5

```
dc_shell> analyze -format sverilog {global.sv and_struct.sv}
Running HDLC
...
dc_shell> analyze -format sverilog {or_struct.sv}
Running HDLC
Searching for ./or_struct.sv
Compiling source file ./or_struct.sv
Error: ./or_struct.sv:1: Syntax error at or near token 'din'.
(VER-294)
*** HDLC compilation terminated with 1 error. ***
0
```

- The `` `include `` construct

  You can use this method to fix the problem described in Example 2-5. Example 2-6 shows how to include the global file.

Example 2-6

```
‘include "global.sv"

module or_struct(
   input data din,
   output a_or_b
);
assign a_or_b = din.a | din.b;
endmodule
```

Example 2-7 shows the script for the `` `include `` method. The tool reads all the files and generates the netlist without errors.

Example 2-7

```
dc_shell> analyze -format sverilog {global.sv and_struct.sv}
dc_shell> analyze -format sverilog {or_struct_modified.sv}
dc_shell> analyze -format sverilog {global.sv top.sv}
dc_shell> elaborate top
dc_shell> write -format verilog -hierarchy -output gtech.sample4.v
```

Specifying Global Files for Each analyze Command

For each `analyze` command, a separate $unit name space is created for the files read. Multiple `analyze` commands do not share the name spaces. Therefore, you must specify all the global files that are used by the files analyzed for each `analyze` command. Example 2-8, which shows how to apply this guideline, uses the following five files:

- global.sv—contains the global declaration of the structure data type that is used by other modules.

  ```
  typedef struct {
     logic a, b;
  } data;
  ```

- global2.sv—contains the global function parity that uses the structure data type from the global.sv file.

  ```
  function automatic parity (input data din);
  return(^{din.a, din.b});
  endfunction
  ```

- and_struct_exor.sv—assigns the structure data type to the din input. The design computes the parity of the din input using the parity function from the global2.sv file.

```
module and_struct_exor (
   input data din,
   output a_and_b, a_exor_b
);
assign a_and_b = din.a & din.b;
assign a_exor_b = parity(din);
endmodule
```

- or_struct.sv—assigns the structure data type to the din input.

```
module or_struct (
   input data din,
   output a_or_b
);
assign a_or_b = din.a | din.b;
endmodule
```

- top_modified.sv—assigns the structure data type to the din input and instantiates the and_struct_exor module with the name u1 and the or_struct module with the name u2.

```
module top(
   input data din,
   output or_result, and_result, parity_result
);
and_struct_exor u1(.din, .a_and_b(and_result),
.a_exor_b(parity_result));
or_struct u2(.din, .a_or_b(or_result));
endmodule
```

In Example 2-8, the first `analyze` command reads the global.sv and global2.sv global files before the and_struct_exor file so that the data structure and the parity function are available to the and_struct_exor module. Because the tool creates a separate $unit for each `analyze` command, the global.sv file must be read individually for the or_struct and top_modified.sv files.

Example 2-8

```
dc_shell> analyze -format sverilog \
   {global.sv global2.sv and_struct_exor.sv}
dc_shell> analyze -format sverilog {global.sv or_struct.sv}
dc_shell> analyze -format sverilog {global.sv top_modified.sv}
dc_shell> elaborate top
dc_shell> write -format verilog -hierarchy -output gtech.sample6.v
```

The tool generates the netlist without any errors because all required global declarations are available in $unit when the modules are read.

Synthesis Restrictions for $unit

The following objects are not allowed in $unit because of synthesis restrictions:

- Declarations
- Instantiations
- Static Variables
- Static Tasks and Functions

Declarations

Declarations of nets and variables are not allowed in $unit. For example, the tool cannot synthesize the following declarations:

```
logic a, c;
wire b;
```

Instantiations

Module, interface, and gate instantiations are not allowed in $unit. For example, the tool cannot synthesize the following code:

```
and and_gate(out, in1, in2);
half_adder U1(sum, ain, bin); // where half_adder is module
nameiface if1();              // where iface is the interface name
```

Static Variables

Static variables inside automatic functions or automatic tasks are not allowed in $unit. For example, the tool cannot synthesize the following code:

```
function automatic [31:0] incr_by_value (logic [31:0] val);
static logic [31:0] sum = 0; //static variable here
sum += val;
return(sum);
endfunction
```

Static Tasks and Functions

Static tasks or static functions are not allowed in $unit. For example, the tool cannot synthesize the following code:

```
function static logic non_zero_int_is_true (logic [31:0] val);
if (val == 0) return ('0);
else return('1);
endfunction
```

If you use the previous code in $unit, the tool issues an error message similar to the following:

```
Error:  ...: Static function 'adder' is not synthesizable
in $unit, expecting "automatic" keyword (VER-523)
```

Verilog functions are static by default. For example, if you do not use the `automatic` keyword in the following code, the tool assumes it is a static function and issues a VER-523 error message.

```
function void adder (
   input cin, [7:0] in1, [7:0] in2,
   output [8:0] result
);
begin
   assign result[0] = cin;
end
endfunction
```

3

Packages

You can use packages to share parameters, types, tasks, and functions among multiple modules and interfaces in SystemVerilog. Packages are explicitly named scopes appearing at the same level as the top-level modules. The following topics describe various ways to reference such declarations in modules, interfaces, and other packages:

- About Packages
- Using Packages
- Referencing Declarations in Packages
- Wildcard Imports From Packages Into Modules
- Specific Imports From Packages Into Modules
- Wildcard Imports From Packages Into $unit
- Package Searching

About Packages

In any SystemVerilog design projects, it is common for a design team to reuse types, functions, and tasks. When you put these common constructs in packages, they can be shared among the team. This allows developers to use existing code based on their requirements without any ambiguity.

After specifying all types, functions, and tasks in a package, you analyze the package. Modules that use the package declarations can be analyzed separately without the need to reanalyze the package. This can save runtime when large packages are used.

The following restrictions apply when you use packages:

- Wire and variable declarations in packages are not allowed.

  The tool issues an error message.

- Functions and tasks that are declared inside packages need to be automatic.
- Sequence, property, and program blocks are ignored.

  The tool issues a warning. For more information about ignored assertions, see Assertions in Synthesis.

Using Packages

To use a package in SystemVerilog,

1. Analyze the package by using the `analyze` command.

   The command creates a temporary *package_name.pvk* file. If you modify this analyzed package by adding or removing functions, tasks, types, and so on, the tool overwrites this temporary file and issues a VER-26 warning message similar to the following:

   ```
   Warning:  ./test.sv:1: The package p has already been analyzed. It is
   being replaced. (VER-26)
   ```

2. Analyze and elaborate the modules that use the package created in step 1 by using the `analyze` and `elaborate` commands respectively.

   If your modules were analyzed using a previous version of the package, repeat step 2 so that the tool uses the latest declarations from the package.

Referencing Declarations in Packages

Example 3-1 uses the following three files. To access the declarations in the pkg1 package, the test1.sv and test2.sv files use the scope resolution operator (::) to reference the type and function declarations with the package_name::type_name and package_name::function_name syntax.

- package.sv—contains two types, my_struct and my_T, and two functions, subtract and complex_add.

```
package pkg1;
typedef struct {int a;logic b;} my_struct;
typedef logic [127:0] my_T;

function automatic my_T subtract(my_T one, two);
return(one - two);
endfunction

function automatic my_struct complex_add(my_struct one, two);
complex_add.a = one.a + two.a;
complex_add.b = one.b + two.b;
endfunction
endpackage : pkg1
```

- test1.sv—uses the my_T type and the subtract function to compute the result and equal values.

```
module test1 (
   input pkg1::my_T in1, in2,
   input [127:0]test_vector,
   output pkg1::my_T result,
   output equal
);
assign result = pkg1::subtract(in1, in2);
assign equal = (in1 == test_vector);
endmodule
```

- test2.sv—uses the my_struct type and complex_add function to compute the result2 values.

```
module test2 (
   input pkg1::my_struct in1, in2,
   output pkg1::my_struct result2
);
assign result2 = pkg1::complex_add(in1, in2);
endmodule
```

In Example 3-1, you analyze the package.sv file first to create the temporary pkg1.pvk file. Then, you can analyze and elaborate the test1 and test2 files individually because the pkg1.pvk file already exists.

Example 3-1 Script

```
# Analyze the package file the first time, it creates pkg1.pvk file
analyze -format sverilog package.sv

# Analyze/elaborate the first module, test1, that uses the package
analyze -format sverilog test1.sv
elaborate test1

# Analyze/elaborate the second module, test2, that uses the package
analyze -format sverilog test2.sv
elaborate test2
```

Alternatively, you can analyze all the package and module files at the same time.

Wildcard Imports From Packages Into Modules

Example 3-2 uses wildcard imports to import all declarations, the enum identifier color and its literal values, into the module scope. Both imported items are used in the finite state machine.

Example 3-2 Wildcard Imports

```
package p;
typedef enum logic [1:0] {red, blue, yellow, green} color;
endpackage

module fsm_controller (
   output logic [2:0] result,
   input [1:0] read_value,
   input clock, reset
);

/*
Using wildcard imports, both the enum identifier and literals become
available and are kept because they are used inside the module.
*/
import p::*;
color State;

always_ff @(posedge clock, negedge reset)
begin
   if (!reset)
   begin
      State <= red;
   end
   else
   begin
      State <= color'(read_value);
   end
end

always_comb
begin
   case(State)
      red   : result = 3'b101;
      yellow: result = 3'b001;
      blue  : result = 3'b000;
      green : result = 3'b010;
   endcase
end
endmodule
```

Specific Imports From Packages Into Modules

Packages can hold many declarations, but not all the decorations are needed by all the modules. For example, if the entire design uses one global package for all declarations, you can import specific type or function declarations for your modules from the global package.

In Example 3-3, the p package contains the color, SWITCH_VALUES, packet_t, and sw_lgt_pair types. Because the fsm_controller module uses only the color type, you import the color type and its literal values from the package.

Example 3-3 Package p

```
package p;

typedef enum logic [1:0] {red, blue, yellow, green} color;
typedef enum logic {OFF, ON} SWITCH_VALUES;

typedef struct {
   logic [7:0] src;
   logic [7:0] dst;
   logic [31:0] data;
} packet_t;

typedef struct packed {
   SWITCH_VALUES switch; // 1 bit
   color light;          // 2 bits
   logic test_bit;       // 1 bit
   sw_lgt_pair;          // 4 bits
}
endpackage

module fsm_controller (
   output logic [2:0] result,
   input [1:0] read_value,
   input clock, reset
);

import p::color; //use specific imports to import the identifier color
import p::red;   //use specific imports to import the enum literal values
import p::blue;
import p::yellow;
import p::green;

color State;

always_ff @(posedge clock, negedge reset) begin
begin
   if (!reset)
   begin
      State <= red;
   end
   else
   begin
      State <= color'(read_value);
   end
end

always_comb
begin
```

```
   case(State)
      red   : result = 3'b101;
      yellow: result = 3'b001;
      blue  : result = 3'b000;
      green : result = 3'b010;
   endcase
end
endmodule
```

Note:

Even if you use wildcard imports (`import p::*;`) in the fsm_controller module, the module uses only the required types. When creating large designs with packages, use specific imports so that you know what types are imported into the designs. If you use wildcard imports, debugging might be difficult because you have to step through your entire module to find what types are actually used.

Wildcard Imports From Packages Into $unit

The tool supports wildcard imports into the $unit name space from packages without requiring access by using the scope resolution operator (::). Example 3-4 shows the code compaction benefit of using wildcard imports.

Example 3-4

```
package pkg;
typedef struct {byte a, b;} packet;
typedef enum logic[1:0] {ONE,TWO,THREE} state_t;
endpackage

import pkg::*; //wildcard import into $unit
module test (
   output packet packet1,
   input clk, rst,
   input state_t data1, data2
);

/* Using scope resolution without imports in $unit syntax:
module test (
   output pkg::packet packet1,
   input clk, rst,
   input pkg::state_t data1, data2
);
*/
...
endmodule
```

Package Searching

The HDL Compiler tool can search for a previously analyzed package (.pvk file) in different directories when analyzing modules that contain imports from this package.

When searching for a package, the tool first looks at the current working directory regardless of whether the working directory is specified in the `search_path` variable. If the tool cannot find the package, it looks at other directories, starting from the left most directory path specified in the `search_path` variable, and uses the first matching package it finds. You should specify directory paths in correct order for the `search_path` variable.

For example, the RTL design contains the test1.sv and test2.sv files that need the user-defined types in the pkg1.sv package. You use the script shown in Example 3-5 to analyze the pkg1.sv package and the test1.sv file in the test1 library, but you need to analyze the test2.sv file in the test2 library. To use this analyzed package, include the directory path of the analyzed pkg1.pvk package in the `search_path` variable, and then analyze the test2.sv file without reanalyzing the pkg1.sv package, as shown in Example 3-6.

Example 3-5

```
dc_shell> define_design_lib test1 -path ../test1
dc_shell> analyze -format sverilog \
   -library test1 {list ../rtl/pkg1.sv ../rtl/test1.sv}
dc_shell> elaborate -library test1 my_test1
```

Example 3-6

```
dc_shell> lappend search_path {list ../test1}
dc_shell> define_design_lib test2 –path ../test2
dc_shell> analyze -format sverilog -library test2 {list ../rtl/test2.sv}
dc_shell> elaborate -library test2 my_test2
```

The tool issues the following information message, showing which previously analyzed package is used and where the package resides:

```
Found package 'pkg1' via search_path at ./test1/pkg1.pvk.
```

When the tool cannot find the package, it issues the following error message:

```
Error:  test1.sv:1: Package 'pkg1' has not been analyzed for import or
content extraction. (VER-224)
```

4

Combinational Logic

These topics describe how to model combinational logic using HDL operators, MUX_OP cells, and SystemVerilog constructs, such as `always_comb`, `unique if`, `priority if`, `priority case`, and `unique case`.

- Synthetic Operators
- Logic and Arithmetic Expressions
- Language Constructs for Combinational Logic Inference
- Multiplexing Logic
- MUX_OP Cells With Variable Indexing
- MUX_OP Inference for Bit and Memory Accesses
- Bit-Truncation Coding for DC Ultra Datapath Extraction

Synthetic Operators

Synopsys provides the DesignWare Library, which is a collection of intellectual property (IP), to support the synthesis products. Basic IP provides implementations of common arithmetic functions that can be referenced by HDL operators in the RTL.

The DesignWare IP solutions are built on a hierarchy of abstractions. HDL operators (either the built-in operators or HDL functions and procedures) are associated with synthetic operators, which are bound to synthetic modules. Each synthetic module can have multiple architectural realizations called implementations. When you use the HDL addition operator in a design, the HDL Compiler tool infers an abstract representation of the adder in the netlist. The same inference applies when you use a DesignWare component. For example, a DW01_add instantiation is mapped to the synthetic operator associated with it, as shown in Figure 4-1.

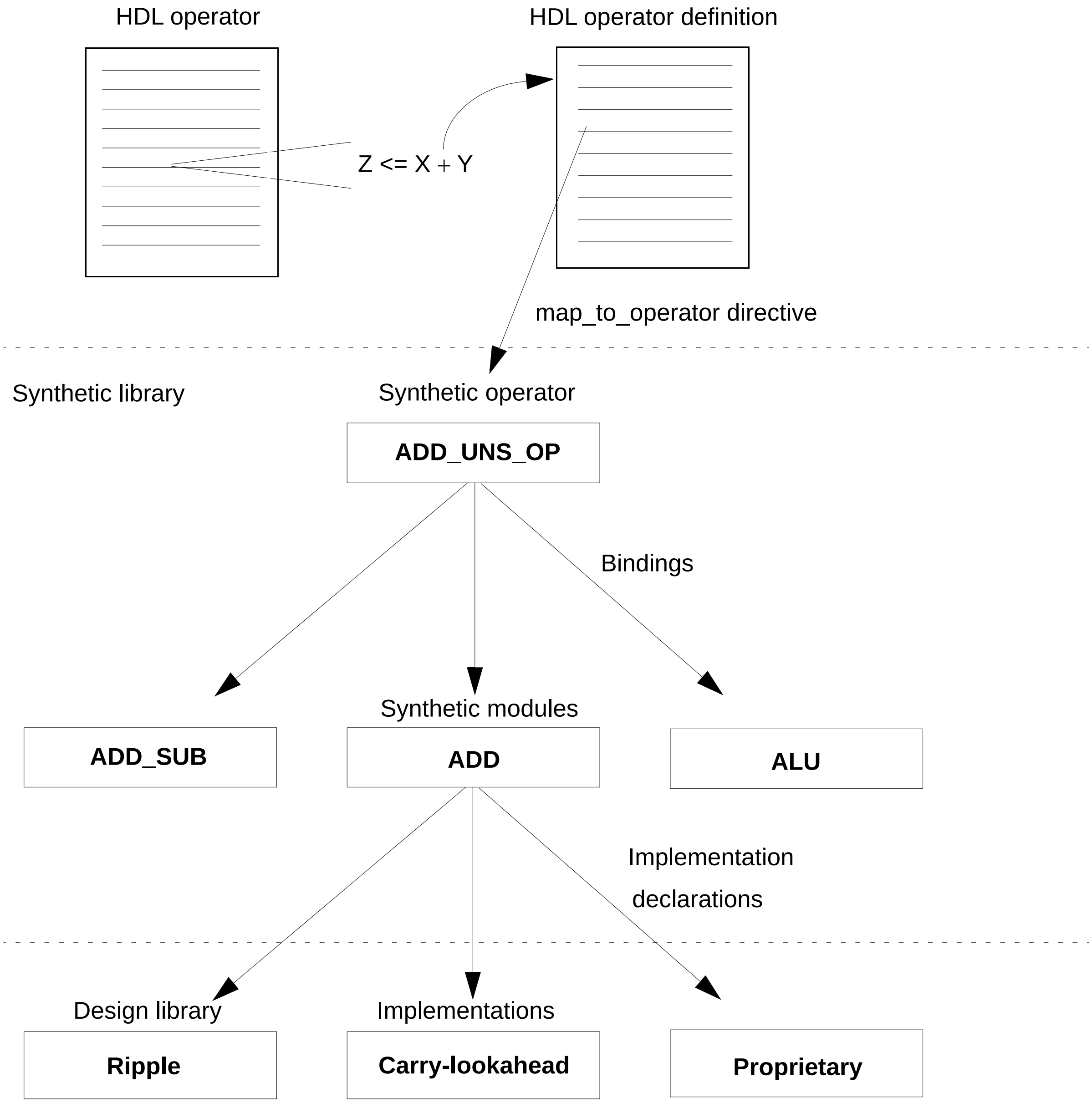
A synthetic library contains definitions for synthetic operators, synthetic modules, and bindings. It also contains declarations that associate synthetic modules with their implementations. To display information about the standard synthetic library that is included with the Design Compiler license, use the `report_synlib` command.

For example,

```
report_synlib standard.sldb
```

For more information about the DesignWare synthetic operators, modules, and libraries, see the DesignWare documentation.

Figure 4-1 DesignWare Hierarchy



Logic and Arithmetic Expressions

These topics discuss synthesis for logic and arithmetic expressions.

- Basic Operators

- Addition Overflow
- Sign Conversions

Basic Operators

When the HDL Compiler tool elaborates a design, it maps HDL operators to synthetic (DesignWare) operators in the netlist. When the Design Compiler tool optimizes the design, it maps these operators to the DesignWare synthetic modules and chooses the best implementation based on the constraints, option settings, and wire load models.

The HDL Compiler tool maps HDL operators, such as comparison (> or <), addition (+), decrement (-), and multiplication (*), to synthetic operators from the Synopsys standard synthetic library, standard.sldb. Table 4-1 shows the complete list of the standard synthetic operators. For more information, see the DesignWare Library documentation.

Table 4-1 HDL Operators Mapped to Standard Synthetic Operators

HDL operator(s)	Synthetic operator(s)
+	ADD_UNS_OP, ADD_UNS_CI_OP, ADD_TC_OP, ADD_TC_CI_OP
-	SUB_UNS_OP, SUB_UNS_CI_OP, SUB_TC_OP, SUB_TC_CI_OP
*	MULT_UNS_OP, MULT_TC_OP
<	LT_UNS_OP, LT_TC_OP
>	GT_UNS_OP, GT_TC_OP
<=	LEQ_UNS_OP, LEQ_TC_OP
>=	GEQ_UNS_OP, GEQ_TC_OP
if, case	SELECT_OP
division (/)	DIV_UNS_OP, MOD_UNS_OP, REM_UNS_OP, DIVREM_UNS_OP, DIVMOD_UNS_OP,DIV_TC_OP, MOD_TC_OP, REM_TC_OP, DIVREM_TC_OP, DIVMOD_TC_OP
=, !=	EQ_UNS_OP, NE_UNS_OP, EQ_TC_OP, NE_TC_OP
<<, >> (logic)	ASH_UNS_UNS_OP, ASH_UNS_TC_OP, ASH_TC_UNS_OP, ASH_TC_TC_OP
<<<, >>> (arith)	ASHR_UNS_UNS_OP, ASHR_UNS_TC_OP, ASHR_TC_UNS_OP, ASHR_TC_TC_OP

Table 4-1 HDL Operators Mapped to Standard Synthetic Operators (Continued)

HDL operator(s)	Synthetic operator(s)
Barrel Shift ror, rol	BSH_UNS_OP, BSH_TC_OP, BSHL_TC_OP BSHR_UNS_OP, BSHR_TC_OP
Shift and Add srl, sll, sra, sla	SLA_UNS_OP, SLA_TC_OP SRA_UNS_OP, SRA_TC_OP

Note:

Depending on the selected implementation, a DesignWare license might be needed during optimization. To find out the implementation options and license requirements, see the DesignWare Datapath and Building Block IP Quick Reference.

Addition Overflow

When the Design Compiler tool performs arithmetic optimization, it considers how to handle addition overflow caused by carry bits. The optimized structure is affected by the bit-widths that you declare for storing the intermediate results.

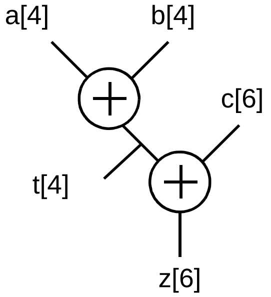
4-Bit Temporary Variable

For example, an expression that adds two 4-bit numbers and stores the result in a 4-bit register can overflow the 4-bit output and truncate the most significant bit. In Example 4-1, three variables are added (a + b + c). The temporary variable, t, holds the intermediate result of a + b. If t is declared as a 4-bit variable, the overflow bits from the addition of a + b are truncated. Figure 4-2 shows how the HDL Compiler tool determines the default structure.

Example 4-1 Adding Numbers of Different Bit-Widths

```
t <= a + b;  // a and b are 4-bit numbers
z <= t + c;  // c is a 6-bit number
```

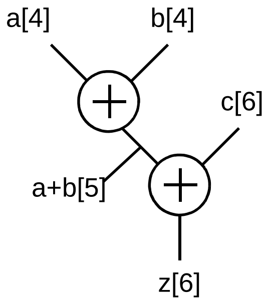
Figure 4-2 Default Structure for a 4-Bit Temporary Variable



5-Bit Intermediate Result

To perform the previous addition (z = a + b + c) without a temporary variable, the HDL Compiler tool determines that 5 bits are needed to store the intermediate result to avoid overflow, as shown in Figure 4-3. This result might be different from the previous case, where a 4-bit temporary variable truncates the intermediate result. Therefore, these two structures do not always yield the same result.

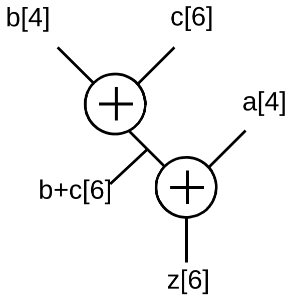
Figure 4-3 Structure for a 5-Bit Intermediate Result



Optimization for Delay

If the same expression is optimized for the late-arriving signal, a, the tool restructures the expression so that signals b and c are added first. Because signal c is declared as 6 bits, the tool determines that the intermediate result must be stored in a 6-bit variable. Figure 4-4 shows the structure for this example.

Figure 4-4 Structure for a Late-Arriving Signal



Sign Conversions

When reading a design that contains signed expressions and assignments, the tool issues VER-318 warnings for sign assignment mismatches.

No warnings are issued for the following conditions:

- The conversion is necessary only for constants in the expression.
- The width of the constant does not change as a result of the conversion.

- The most significant bit of the constant is zero (not negative).

In the following example, though the tool implicitly converts the signed constant 1 to unsigned, no warning is issued because the conversion meets the previously mentioned three conditions. By default, integer constants are treated as signed types with signed values.

```
module t (
   input [3:0] a, b,
   output [5:0] z
);
assign z = a + b + 1;
endmodule
```

A VER-318 warning indicates that the tool implicitly performs one of the following operations:

- Conversion
  - ❍ An unsigned expression to a signed expression
  - ❍ A signed expression to an unsigned expression
- Assignment
  - ❍ An unsigned right side to a signed left side
  - ❍ A signed right side to an unsigned left side

In the following example, signed logic a is converted to an unsigned value and not sign-extended, and the tool issues a VER-318 warning. This behavior complies with the SystemVerilog and Verilog 2001 styles.

```
module t (/*...*/);
logic signed [3:0] a;
logic [7:0] c;
assign a = 4'sb1010;
assign c = a+7'b0101011;
endmodule
```

When explicit type casting is used, no VER-318 warning is issued. For example, to force logic a to be unsigned, assign logic c as follows:

```
c = unsigned'(a)+7'b0101011;
```

For Verilog designs, you can use the `$signed` and `$unsigned` system tasks to do the sign conversion. For more information, see the *IEEE Std 1364-2005*.

In the following example, the left side is unsigned, but the right side is sign-extended; that is, logic a contains the value of 8'b11111010 after the assignment. A VER-318 warning is issued.

```
module t (/*...*/)
logic signed [3:0] a;
assign a = 4’sb1010;
endmodule
```

If a line contains more than one implicit conversion, such as the expression that is assigned to logic c in the following example, the tool issues only one warning. In this example, logic a and b are converted to unsigned values and the right side is unsigned. Assigning the right-side value to logic c results in a VER-318 warning.

```
module t (/*...*/)
logic signed [3:0] a;
logic signed [3:0] b;
logic signed [7:0] c;
assign c = a+4'b0101+(b*3'b101);
endmodule
```

The following examples show sign conversions and the cause of each VER-318 warning:

- In the m1 module, the signs are consistently applied and no warning is issued.

  ```
  module m1 (
     input signed [0:3] a,
     output signed [0:4] z
  );
  assign z = a;
  endmodule
  ```

- In the m2 module, input a is signed and added to 3'sb111, which is a signed value of -1. Output z is not signed, so the signed value of the expression on the right side is converted to unsigned and assigned to output z.

  ```
  module m2 (
     input signed [0:2] a,
     output [0:4] z
  );
  assign z = a + 3'sb111;
  endmodule

  Warning:  ./test.sv:5: signed to unsigned assignment occurs. (VER-318)
  ```

- In the m3 module, input a is unsigned but becomes signed when it is assigned to signed logic x, and the tool issues a VER-318 warning. In the z = x < 4'sd5 expression, the comparison result of signed x to a signed 4'sd5 value is put into unsigned logic z. This appears to be a sign mismatch; however, no VER-318 warning is issued because comparison results are always considered unsigned for all relational operators.

```
module m3 (
   input [0:3] a,
   output logic z
);
logic  signed [0:3] x;
always_comb
begin
   x = a;
   z = x < 4'sd5;
end
endmodule

Warning:  ./test.sv:8: unsigned to signed assignment occurs. (VER-318)
```

- In the m4 module, the signs are consistently applied and no warning is issued.

```
module m4 (
   input signed [7:0] in1, in2,
   output signed [7:0] out
);
assign out = in1 * in2;
endmodule
```

- In the m5 module, inputs, a and b, are unsigned but they are assigned to signed signals x and y respectively. Two VER-318 warnings are issued. In addition, logic y is subtracted from logic x and assigned to unsigned output z; the expression results in a VER-318 warning.

```
module m5 (
   input [1:0] a, b,
   output [2:0] z
);
logic signed [1:0] x, y;
assign x = a;
assign y = b;
assign z = x - y;
endmodule

Warning:  ./test.sv:6: unsigned to signed assignment occurs. (VER-318)
Warning:  ./test.sv:7: unsigned to signed assignment occurs. (VER-318)
Warning:  ./test.sv:8: signed to unsigned assignment occurs. (VER-318)
```

- In the m6 module, input a is unsigned but put into signed register x.

```
module m6 (
   input [3:0] a,
   output z
);
logic signed [3:0] x;
always @(a) x = a;
assign      z = x < -4'sd5;
endmodule

Warning:  ./test.sv:6: unsigned to signed assignment occurs. (VER-318)
```

- In the m7 module, the tool issues no warning because all signs are properly applied. Comparing a signed constant results in a signed comparison.

```
module m7 (
   input signed [7:0] in1, in2,
   output lt, in1_lt_64
);
assign lt = in1 < in2;
assign in1_lt_64 = in1 < 8'sd64;
endmodule
```

- In the m8 module, signed input in1 is compared with unsigned input in2. Because comparison is unsigned, a VER-318 warning is issued. In addition, the unsigned 8‘d64 constant causes an unsigned comparison; a VER-318 warning is issued.

```
module m8 (
   input signed [7:0] in1,
   input [7:0] in2,
   output lt
);
wire uns_lt, uns_in1_lt_64;
assign uns_lt = in1 < in2;
assign uns_in1_lt_64 = in1 < 8'd64;
assign lt = uns_lt + uns_in1_lt_64;
endmodule

Warning:  ./test.sv:7: signed to unsigned conversion occurs. (VER-318)
Warning:  ./test.sv:8: signed to unsigned conversion occurs. (VER-318)
```

- In the m9 module, even though inputs, in1 and in2, are mismatched in signs, the casting operator converts input in2 to a signed signal. When a casting operator is used and a sign conversion occurs, no warning is issued.

```
module m9 (
   input signed [7:0] in1;
   input [7:0] in2;
   output lt;
);
assign lt = in1 < signed?({1'b0, in2});
endmodule
```

Language Constructs for Combinational Logic Inference

This section describes combinational logic inference for the `always`, `always_comb`, `priority if`, `priority case`, `unique if`, and `unique case` constructs.

- The always_comb and always Constructs
- Latches in Combinational Logic
- The priority if and priority case Constructs
- The unique if and unique case Constructs

The always_comb and always Constructs

In SystemVerilog, you can use the `always_comb` construct to model combinational logic. The following example describes an AND operation using the `always_comb` construct:

```
module test (
   input a, b,
   output logic y
);
always_comb
   y = a & b;
endmodule
```

In Verilog, you use the `always @*` construct to infer the same logic. For example,

```
module test (
   input a, b,
   output reg y
);
//always_comb
always @*
   y = a & b;
endmodule
```

When the `always_comb` construct is used, the tool checks whether the logic described in the `always_comb` block represents combinational logic. When the tool synthesizes the `always_comb` block and infers a latch, it issues an ELAB-974 warning.

In the following example, the tool issues a warning because of the missing `else` condition.

```
module unintended_latch (
   input a, b,
   output logic c
);
always_comb
   if (a)
   c = b;
endmodule

Warning:  …/test.sv:5: Netlist for always_comb block contains a latch.
(ELAB-974)
```

If the tool does not infer combinational logic that is described in the `always_comb` block, it issues an ELAB-982 warning. As shown in the following example, logic tmp is not an output and might be removed during synthesis, so the tool issues an ELAB-982 warning:

```
module test (
   input a, b,
   output c
);
logic tmp;
assign c = a | b;
always_comb tmp = a & b;
endmodule
```

See Also

- Latches in Combinational Logic

Latches in Combinational Logic

When a variable in a combinational logic block (an `always` block without a `posedge` or `negedge` keyword) is not specified in all the branches, the Verilog code can imply combinational feedback paths or latches in the synthesized logic. A variable is fully specified when it is assigned a value under all conditions.

Example 4-2 shows that variable Q is not assigned when GATE equals 1’b0 and a latch is inferred to store its previous value. To avoid the latch inference, assign a value to the variable in all the branches of the `always` block.

Example 4-2 Latch Inference

```
always @ (DATA or GATE) begin
   if (GATE) begin
      Q = DATA;
   end
end
```

In Example 4-3 and Example 4-4, variable Q is assigned the value of 0 when GATE equals 1’b0; it is assigned in all the branches of the `always` block. Example 4-3 and Example 4-4 are not equivalent to Example 4-2, in which Q holds its previous value when GATE equals 1’b0.

Example 4-3 Avoiding Latch Inference—Method 1

```
always @ (DATA, GATE) begin
   Q = 0;
   if (GATE)
   Q = DATA;
end
```

Example 4-4 Avoiding Latch Inference—Method 2

```
always @ (DATA, GATE) begin
   if (GATE)
      Q = DATA;
   else
      Q = 0;
end
```

Example 4-5 results in a latch because the variable is not assigned in all the branches of the `always` block. To avoid the latch inference, add the following statement before the `endcase` statement:

```
default: decimal= 10’b0000000000;
```

Example 4-5 Latch Inference Using a case Statement

```
always @(I) begin
   case(I)
      4'h0: decimal= 10'b0000000001;
      4'h1: decimal= 10'b0000000010;
      4'h2: decimal= 10'b0000000100;
      4'h3: decimal= 10'b0000001000;
      4'h4: decimal= 10'b0000010000;
      4'h5: decimal= 10'b0000100000;
      4'h6: decimal= 10'b0001000000;
      4'h7: decimal= 10'b0010000000;
      4'h8: decimal= 10'b0100000000;
      4'h9: decimal= 10'b1000000000;
   endcase
end
```

When a variable is not assigned in all the branches of a `for` loop or no initial value before the loop, latches are also inferred.

The priority if and priority case Constructs

This section describes how to direct the tool to infer multiplexers using the `priority if` and `priority case` constructs.

priority if

Example 4-6 shows how to direct the tool to infer a multiplexer by using the `priority if` construct.

Example 4-6 Multiplexer Inference Using priority if

```
module priority_if (
   input a, b, c, d, [3:0] sel,
   output logic z
);

always_comb
begin
   priority if (sel[3]) z = d;
   else if     (sel[2]) z = c;
   else if     (sel[1]) z = b;
   else if     (sel[0]) z = a;
end
endmodule
```

priority case

Example 4-7 shows how to direct the tool to infer a multiplexer by using the `priority case` construct. Using the `case` keyword qualified by the priority keyword without coding a default case is the same as using the Synopsys `full_case` directive. However, you should use the `priority case` construct to prevent simulation and synthesis mismatches, which can occur when the directive is used.

Example 4-7 Multiplexer Inference Using priority case

```
// priority_case.sv
module my_priority_case (
   input [1:0] in, a, b, c,
   output logic [1:0] out
);

always_comb
   priority case (in)
      0: out = a;
      1: out = b;
      2: out = c;
   endcase
endmodule
```

See Also

- Preventing case Mismatches

The unique if and unique case Constructs

This section describes how to direct the tool to infer combinational logic using the `unique if` and `unique case` constructs.

unique if

Example 4-8 shows how to direct the tool to infer a multiplexer by using the `unique if` construct.

Example 4-8 Multiplexer Inference Using unique if

```
module unique_if (
   input a, b, c, d, [3:0] sel,
   output logic z
);

always_comb
begin
   unique if (sel[3]) z = d;
   else if   (sel[2]) z = c;
   else if   (sel[1]) z = b;
   else if   (sel[0]) z = a;
end
endmodule
```

unique case

Example 4-9 describes a state machine and uses the `unique case` construct for the state control. Using the `case` keyword qualified by the unique keyword is the same as using the Synopsys `full_case` and `parallel_case` directives. However, you should use the unique case construct to prevent simulation and synthesis mismatches, which can occur when the directives are used.

Example 4-9 State Machine Using unique case

```
// State machine using unique case: unique_case.sv
module fsm_cc1_3oh (
   input in1, a, b, c, d, clk,
   output logic o1
);

logic [3:0] state, next;

always_ff @(posedge clk)
state <= next;

always_comb
begin
   next = state;
   unique case (1'b1)
      state[0]: begin
                next[0] = (in1 == 1'b1);
                o1 = a;
                end
      state[1]: begin
                next[1] = 1’b1;
                o1 = b;
                end
      state[2]: begin
                next[2] = 1'b1;
                o1 = c;
                end
      state[3]: begin
                next[3] = 1'b1;
                o1 = d;
                end
   endcase
end
endmodule
```

See Also

- Preventing case Mismatches

Multiplexing Logic

Multiplexers are commonly modeled with the `case` statements. The `if` statements are occasionally used, but they are difficult to code. To implement multiplexing logic, the tool uses SELECT_OP cells, which are mapped to combinational logic or multiplexers in the logic library during synthesis. If you want the Design Compiler tool to map multiplexing logic to multiplexers or multiplexer trees in the logic library, you must use MUX_OP cells.

The following topics describe multiplexer inference:

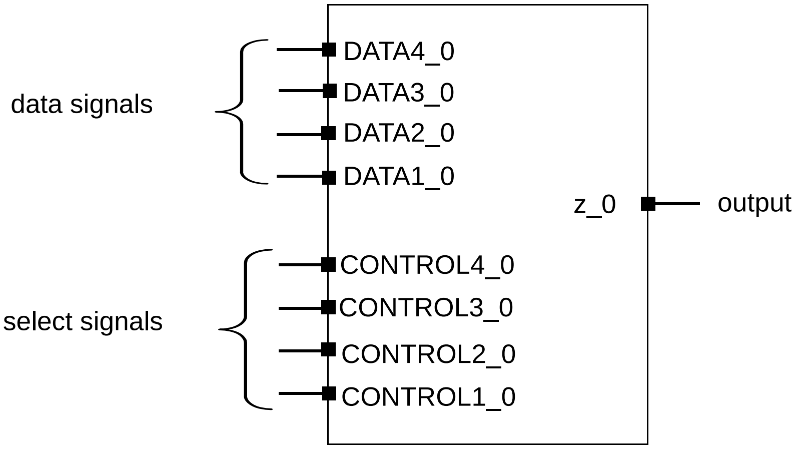
- SELECT_OP Inference
- One-Hot Multiplexer Inference
- MUX_OP Inference
- Variables Controlling MUX_OP Inference
- Using the case Statement for MUX_OP Inference
- MUX_OP Inference Limitations

SELECT_OP Inference

By default, the HDL Compiler tool uses SELECT_OP cells to implement conditional operations implied by the `if` and `case` statements.

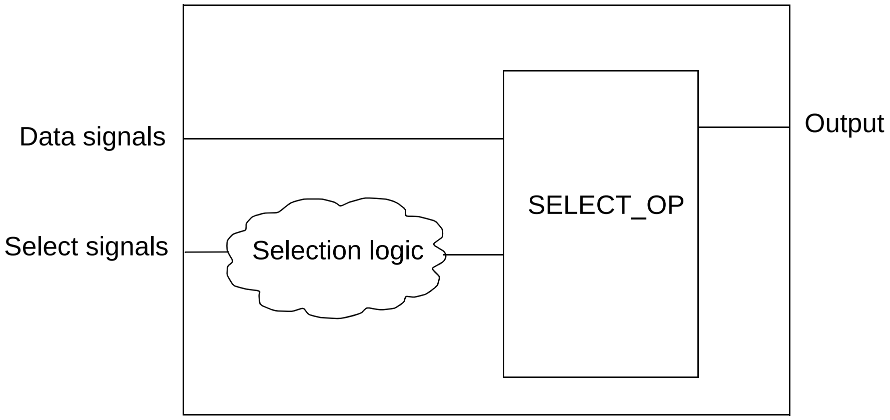
A 4-bit design that requires four selection bits is called a one-hot implementation. As shown in Figure 4-5, the 4-bit SELECT_OP design behaves like a one-hot multiplexer. The control inputs are mutually exclusive, and each control input allows the corresponding input data to pass to the output.

Figure 4-5 SELECT_OP Implementation for a 4-bit Data Signal



To determine which data signal is chosen, the HDL Compiler tool generates selection logic, as shown in Figure 4-6. Depending on the design constraints, the Design Compiler tool implements the SELECT_OP cell with either combinational logic or multiplexer cells from the logic library.

Figure 4-6 HDL Compiler Output—SELECT_OP and Selection Logic



See Also

- One-Hot Multiplexer Inference

One-Hot Multiplexer Inference

To direct the Design Compiler tool to map a SELECT_OP cell to a one-hot multiplexer in the logic library, use the `infer_onehot_mux` directive as shown in Example 4-10 and Example 4-11.

Example 4-10 One-Hot Multiplexer Coding Style One

```
module onehot_1 (
   input in1, in2, in3,
   input sel1, sel2, sel3,
   output logic out
);
always_comb
begin
unique case (1'b1)  // synopsys infer_onehot_mux
   sel1: out  = in1;
   sel2: out  = in2;
   sel3: out  = in3;
endcase
end
endmodule
```

Example 4-11 One-Hot Multiplexer Coding Style Two

```
module onehot_2 (
   input in1, in2, in3,
   input sel1, sel2, sel3,
   output logic out
);
always_comb
begin
unique case ({sel3, sel2, sel1}) // synopsys infer_onehot_mux
   3'b001: out  = in1;
   3'b010: out  = in2;
   3'b100: out  = in3;
endcase
end
endmodule
```

For Verilog designs, use the `parallel_case` and `full_case` directives instead of the `unique case` construct, which generates the same logic as these directives. For SystemVerilog designs, you should use the `unique case` construct to prevent simulation and synthesis mismatches that can occur when you use these directives.

See Also

- Minimizing Mismatches Between Simulation and Synthesis
- The *Design Compiler User Guide*

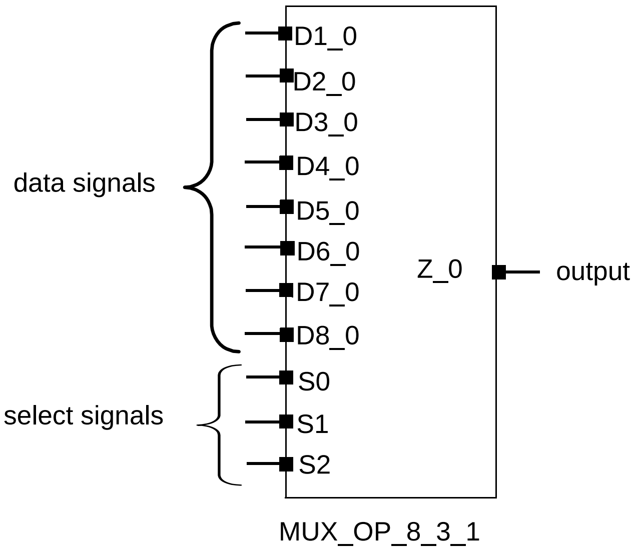
MUX_OP Inference

To enable the Design Compiler tool to map multiplexing logic in the RTL to multiplexers (or multiplexer trees), you need to direct the HDL Compiler tool to infer MUX_OP cells.

MUX_OP cells are hierarchical generic cells that are optimized to use minimum select signals. They are typically faster than SELECT_OP cells, which use the one-hot implementation. Although MUX_OP cells improve design speed, they might increase area. During optimization, the Design Compiler tool maps MUX_OP cells to multiplexers in the logic library. When the tool prohibits the tradeoff between speed and area, combinational logic is used instead.

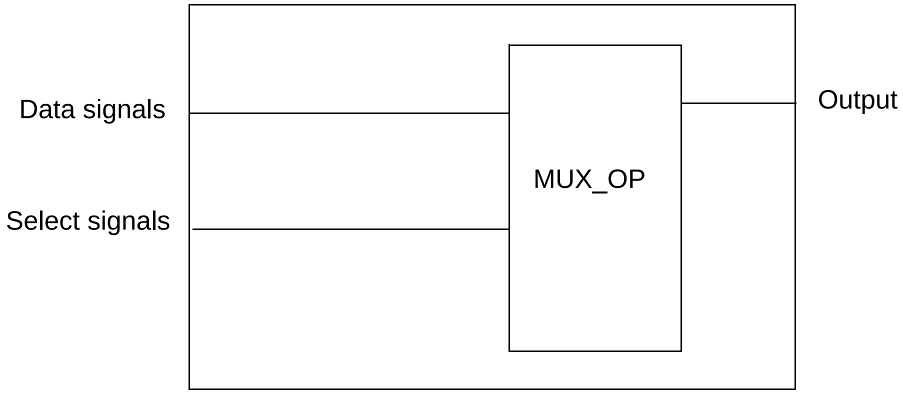
Figure 4-7 shows an 8-bit MUX_OP cell. To select an output, the MUX_OP cell needs only three control signals (log2(8)) as compared to eight control signals for a SELECT_OP cell.

Figure 4-7 8-Bit MUX_OP Generic Cell l



The MUX_OP cell contains internal selection logic to determine which signal is chosen, the HDL Compiler tool does not need to generate any selection logic, as shown in Figure 4-8.

Figure 4-8 HDL Compiler Output—MUX_OP Generic Cell for 8-Bit Data



Use the following methods for MUX_OP inference:

- MUX_OP inference for a `case` or `if` statement

  Use the `infer_mux` directive and a simple variable as the control signal. For example, use input A, but not the negation of input A.

  ```
  module mux (
     input [1:0] SEL,
     input [3:0] DIN,
     output logic DOUT
  );
  always_comb
  case (SEL)  // synopsys infer_mux
     2'b00: DOUT <= DIN[0];
     2'b01: DOUT <= DIN[1];
     2'b10: DOUT <= DIN[2];
     2'b11: DOUT <= DIN[3];
  endcase
  ```

  Use the following directive to direct the HDL Compiler tool to infer MUX_OP cells for all `case` statements in the design block:

  ```
  // synopsys infer_mux block_label_list
  ```

- MUX_OP inference for all `case` and `if` statements

  Use the `hdlin_infer_mux` variable and a simple variable as the control signal.

If you set the `infer_mux` directive on the `case` statement that has two or more synthetic operators as data inputs, the tool does not infer a MUX_OP cell but issues an ELAB-370 warning because there is no resource sharing.

See Also

- Using the case Statement for MUX_OP Inference
- The *Design Compiler User Guide*

Variables Controlling MUX_OP Inference

The variables that control MUX_OP inference are listed in Table 4-2.

Table 4-2 MUX_OP Inference Variables

Variable	Description
`hdlin_infer_mux`	Controls MUX_OP inference for all designs in the same Design Compiler session.
`hdlin_mux_size_limit`	Sets the maximum size of a MUX_OP cell that the HDL Compiler tool can infer. The default is 32. Setting this variable to a value greater than 32 might cause long elaboration runtime.
`hdlin_mux_size_min`	Sets the minimum number of data inputs for a MUX_OP cell. The default is 2.
`hdlin_mux_oversize_ratio`	Specifies the ratio of the number of MUX_OP inputs to the number of data inputs. When this ratio is exceeded, SELECT_OP cells are inferred instead. The default is 100.
`hdlin_mux_size_only`	Sets the `size_only` attribute on all MUX_OP cells automatically to ensure these cells are mapped to the library cells and to prevent logic decomposition during optimization.
`hdlin_mux_for_array_read_sparseness_limit`	Prevents inference of a sparse multiplexer for array read operations when the percentage of MUX_OP connected inputs is below the setting of this variable.

MUX_OP Inference Examples

In Example 4-12, two MUX_OP cells and one SELECT_OP cell are inferred as follows:

- For the first `always_comb` block, the tool infers a MUX_OP cell because the `infer_mux` directive is set on the `case` statement.
- For the second `always_comb` block, a SELECT_OP cell is inferred for the first `case`, while a MUX_OP cell is inferred for the second `case` statement with the `infer_mux` directive.

Example 4-12 Two MUX_OP Cells and One Select_OP Cell Inferred

```
module test_muxop_selectop (
   input [7:0] DIN1, DIN2,
   input [3:0] DIN3,
   input [2:0] SEL1, SEL2,
   input [1:0] SEL3,
   output logic DOUT1, DOUT2, DOUT3
);

always_comb
begin
case (SEL1) //synopsys infer_mux
   3'b000: DOUT1 = DIN1[0];
   3'b001: DOUT1 = DIN1[1];
   3'b010: DOUT1 = DIN1[2];
   3'b011: DOUT1 = DIN1[3];
   3'b100: DOUT1 = DIN1[4];
   3'b101: DOUT1 = DIN1[5];
   3'b110: DOUT1 = DIN1[6];
   3'b111: DOUT1 = DIN1[7];
endcase
end

always_comb
begin
case (SEL2)
   3'b000: DOUT2 = DIN2[0];
   3'b001: DOUT2 = DIN2[1];
   3'b010: DOUT2 = DIN2[2];
   3'b011: DOUT2 = DIN2[3];
   3'b100: DOUT2 = DIN2[4];
   3'b101: DOUT2 = DIN2[5];
   3'b110: DOUT2 = DIN2[6];
   3'b111: DOUT2 = DIN2[7];
endcase

case (SEL3) //synopsys infer_mux
   2'b00: DOUT3 = DIN3[0];
   2'b01: DOUT3 = DIN3[1];
   2'b10: DOUT3 = DIN3[2];
   2'b11: DOUT3 = DIN3[3];
endcase
end
endmodule
```

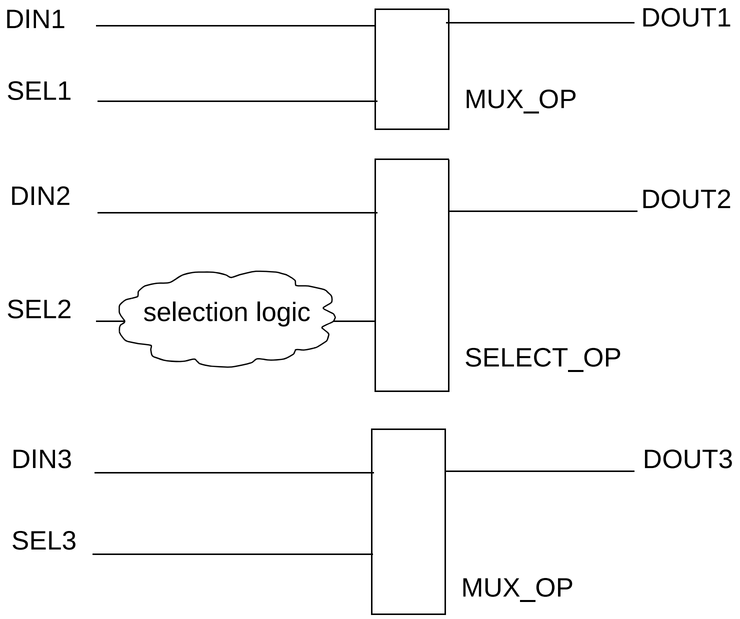
Example 4-13 shows the MUX_OP inference report for Example 4-12. By default, the tool displays the inference report. To exclude the inference report, use the `hdlin_reporting_level` variable.

Example 4-13 MUX_OP Inference Report

```
Statistics for case statements in always block at line 23 in file ...
===============================================
|           Line           |  full/ parallel  |
===============================================
|            23            |    auto/auto     |
===============================================
Statistics for MUX_OPs
============================================================
|    block name/line     | Inputs | Outputs | # sel inputs |
============================================================
| test_muxop_selectop/10 |   8    |    1    |      3       |
| test_muxop_selectop/33 |   4    |    1    |      2       |
============================================================
```

Example 4-9 shows an implementation of the design in Example 4-12 by the HDL Compiler tool.

Figure 4-9 HDL Compiler Implementation



Example 4-14 sets the `infer_mux` directive on the `always_comb` block.

Example 4-14 MUX_OP Inference for a Block

```
module muxtwo(
   input [7:0] DIN1,
   input [3:0] DIN2,
   input [2:0] SEL1,
   input [1:0] SEL2,
   output logic DOUT1, DOUT2
);
```

```
//synopsys infer_mux "blk1"
always_comb
begin: blk1
// This case statement infers an 8-to-1 MUX_OP
case (SEL1)
   3'b000: DOUT1 = DIN1[0];
   3'b001: DOUT1 = DIN1[1];
   3'b010: DOUT1 = DIN1[2];
   3'b011: DOUT1 = DIN1[3];
   3'b100: DOUT1 = DIN1[4];
   3'b101: DOUT1 = DIN1[5];
   3'b110: DOUT1 = DIN1[6];
   3'b111: DOUT1 = DIN1[7];
endcase

// This case statement infers a 4-to-1 MUX_OP
case (SEL2)
   2'b00: DOUT2 = DIN2[0];
   2'b01: DOUT2 = DIN2[1];
   2'b10: DOUT2 = DIN2[2];
   2'b11: DOUT2 = DIN2[3];
endcase
end: blk1
endmodule
```

Example 4-15 sets the `infer_mux` directive on the `case` statement. This `case` statement contains eight unique values, and the HDL Compiler tool infers an 8-to-1 MUX_OP cell.

Example 4-15 MUX_OP Inference for the case Statement

```
module mux8to1 (
   input [7:0] DIN,
   input [2:0] SEL,
   output logic DOUT
);

always_comb
begin: blk1
case (SEL) // synopsys infer_mux
   3'b000: DOUT = DIN[0];
   3'b001: DOUT = DIN[1];
   3'b010: DOUT = DIN[2];
   3'b011: DOUT = DIN[3];
   3'b100: DOUT = DIN[4];
   3'b101: DOUT = DIN[5];
   3'b110: DOUT = DIN[6];
   3'b111: DOUT = DIN[7];
endcase
end: blk1
endmodule
```

In Example 4-16, the tool infers a MUX_OP cell for the `if-else` statement. You must use a simple variable as the control signal for this coding style. For more information about special coding considerations, see Using the case Statement for MUX_OP Inference.

Example 4-16 MUX_OP Inference Using if-else Statements

```
module test (
   input sel, a, b,
   output logic dout
);
always_comb
if(sel) //synopsys infer_mux
   dout = a;
else
   dout = b;
endmodule
```

In Example 4-17, the tool infers a MUX_OP cell for the conditional operator (?:). You must place the `infer_mux` directive right after the “?” construct.

Example 4-17 MUX_OP Inference Using a case Statement

```
module test (sig, A, B, C);
   input A, B, C;
   output  sig;
assign sig = A ? /* synopsys infer_mux */ B :C ;
endmodule
```

See Also

- infer_mux
- Customizing Elaboration Reports

Using the case Statement for MUX_OP Inference

When you design multiplexing logic, you should use the `case` statement. Using the `if` statement might result in slower and larger designs and inflexible code. To direct the tool to infer MUX_OP cells, use the `case` statement.

Although the `infer_mux` directive is set on the `if` statement in Example 4-18, the coding style is too complex for the tool to infer a MUX_OP cell. Instead, the tool infers a SELECT_OP cell to build the multiplexing logic. To infer a 2-to-1 MUX_OP cell, you must modify the `if` statement and assign it to a variable so that it behaves like a `case` statement.

Example 4-18 No MUX_OP Inference for Complex Expressions

```
module mux21 (
   input a, b, c, d,
   input [1:0] sel,
   output logic dout
);

always_comb
if (sel[0] == 1'b0) /* synopsys infer_mux */
   dout <= a and b;
else
   dout <= d and c;
endmodule
```

In Example 4-19, the tool infers a SELECT_OP even though the `infer_mux` directive is used. To enable the tool to infer a MUX_OP cell, you must modify the code as shown in Example 4-20.

Example 4-19 SELECT_OP Inference for Multiplexing Logic of Complex Expressions

```
module mux41 (
   input a, b, c, d,
   output logic dout
);
always_comb
if (a == 1'b0 && b == 1'b0) /* synopsys infer_mux */
   dout <= c & d;
else if (a == 1'b0 && b == 1'b1)
   dout <= c | d;
else if (a == 1'b1 && b == 1'b0)
   dout <= !c & d;
else if (a == 1'b1 && b == 1'b1)
   dout <= c | !d;
endmodule
```

Example 4-20 MUX_OP Inference for Multiplexing Logic

```
module mux41 (
   input a, b, c, d,
   output logic dout
);
logic [1:0] tmp;
always_comb
begin
   tmp = {a, b};
   if (tmp == 2'b00) /* synopsys infer_mux */
      dout <= c & d;
   else if (tmp == 2'b01)
      dout <= c | d;
   else if (tmp == 2'b10)
      dout <= !c & d;
   else if (tmp == 2'b11)
      dout <= c | !d;
end
endmodule
```

MUX_OP Inference Limitations

The HDL Compiler tool does not infer MUX_OP cells for the `case` or `if` statement that uses a complex control expression. You must use a simple variable as the control expression.

MUX_OP cells are inferred for incomplete `case` statements if the `case` statements

- Contain an `if` statement that covers more than one value
- Have a missing branch or a missing assignment in a branch
- Contain don’t care values

When the tool infers MUX_OP cells for these `case` statements, the logic might not be optimal because other optimizations are disabled, such as optimizing the default branch. If the `infer_mux` directive is set on the `case` statement, no optimization is performed.

When inferring a MUX_OP cell for an incomplete `case` statement, the tool issues the following ELAB-304 warning:

```
Warning: Case statement has an infer_mux attribute and a
default branch or incomplete mapping. This can cause
nonoptimal logic if a mux is inferred. (ELAB-304)
```

MUX_OP Cells With Variable Indexing

The tool generates a MUX_OP cell to index into a data variable by using a variable address. For example,

```
module E (
   input [7:0] data,
   input [2:0] addr,
   output out
);
assign out = data[addr];
endmodule
```

In this example, a MUX_OP cell is used to implement data[addr] because the addr subscript is not a known constant.

MUX_OP inference for array read operation is controlled only by the `hdlin_mux_for_array_read_sparseness_limit` variable. The `hdlin_infer_mux`, `hdlin_mux_size_limit` and `hdlin_mux_oversize_ratio` variables control MUX_OP inference for the `if`, `case`, and ?: operations.

See Also

- Variables Controlling MUX_OP Inference

MUX_OP Inference for Bit and Memory Accesses

By default, the tool infers MUX_OP cells for bit and memory accesses, as shown in Example 4-21and Example 4-22.

Example 4-21 MUX Inference for Bit Access

```
module mux_infer_bit (
   input [15:0] x,
   input [3:0] a,
   output logic y
);
always_comb y = x[a];
endmodule
```

Example 4-22 MUX Inference for Memory Access

```
module mux_infer_memory (
   input rw,
   input [3:0] addr,
   inout [7:0] word
);
logic [7:0] mem [15:0];
assign word = (rw) ? mem[addr] : 8'hz ;
always @*
if (!rw) x[addr] = word;
endmodule
```

Bit-Truncation Coding for DC Ultra Datapath Extraction

Datapaths are commonly used in applications that contain extensive data manipulation, such as 3-D, multimedia, and digital signal processing (DSP) designs. Datapath extraction transforms arithmetic operators into datapath blocks to be implemented by a datapath generator.

The DC Ultra tool enables datapath extraction after timing-driven resource sharing and explores various datapath and resource-sharing options during compile.

Note:
> This feature is not available in DC Expert. For more information about datapath optimization, see the Design Compiler documentation.

The DC Ultra datapath optimization supports datapath extraction of expressions containing truncated operands. To prevent extraction, both of the following conditions must exist:

- The operands have upper bits truncated. For example, if d is 16-bit, d[7:0] truncates the upper eight bits.

- The width of the resulting expression is greater than the width of the truncated operand. In the following example, if e is 9-bit, the width of e is greater than the width of the truncated operand d[7:0]:

  ```
  assign e = c + d[7:0];
  ```

For lower-bit truncations, the datapath is extracted in all cases. As described in Table , bit truncation can be either explicit or implicit.

Truncation type	Description

Explicit bit truncation	An explicit upper-bit truncation occurs when you specify the bit range for truncation.
	The following code indicates explicit upper-bit truncation of operand A because p is smaller than q:
	`wire [q:0] A;`
	`out = A [p:0];`

Implicit bit truncation	An implicit upper-bit truncation occurs through assignment. Unlike explicit upper-bit truncation, you do not explicitly define the range for truncation.
	The following code indicates implicit upper-bit truncation of operand Y:
	`input [7:0] A, B;`
	`output [14:0] Y;`
	`assign Y = A*B;`
	Because A and B are 8-bit, their product is 16-bit. However, the 15-bit Y is assigned to the 16-bit product and the most significant bit (MSB) of the product is implicitly truncated. In this example, the MSB is the carryout bit.

Example 4-23 shows how bit truncation affects datapath extraction. When the a*b operation is assigned to wire d, the upper bits are implicitly truncated and the width of output e is less than the width of wire d. This code meets the first condition but not the second, so the code is extracted.

Example 4-23 Design test1: Truncated Operand Is Extracted

```
module test1 (
   input [7:0] a, b, c,
   output [7:0] e
);

wire [14:0] d;
assign d = a * b; // Implicit upper-bit truncation
assign e = c + d; // Width of e is less than d
endmodule
```

Example 4-24 shows how bit truncation prevents extraction. When the a*b operation is assigned to wire d, the upper bits are implicitly truncated and the width of output e is greater than the width of wire d. This code meets both the first and second conditions, so the code is not extracted.

Example 4-24 Design test2: Truncated Operand Is Not Extracted

```
module test2 (
   input [7:0] a, b, c,
   output [8:0] e
);

wire [7:0] d;
assign d = a * b; // Implicit upper-bit truncation
assign e = c + d; // Width of e is greater than d
endmodule
```

Example 4-25 shows how bit truncation prevents extraction. The upper bits of wire d are explicitly truncated, and the width of output e is greater than the width of wire d. This code meets both the first and second conditions, so the code is not extracted.

Example 4-25 Design test3: Truncated Operand Is Not Extracted

```
module test3 (
   input [7:0] a, b, c,
   output [8:0] e
);

wire [15:0] d;
assign d = a * b;      // d is not truncated
assign e = c + d[7:0]; // Explicit upper-bit truncation of d
                       // Width of e is greater than d[7:0]
endmodule
```

Example 4-26 shows how bit truncation does not prevent extraction. The lower bits of wire d are explicitly truncated. For expressions involving lower-bit truncations, the truncated operands are extracted regardless of the bit-width of the truncated operands and the expression result. This code is extracted.

Example 4-26 Design test4: Truncated Operand Is Extracted

```
module test4 (
   input [7:0] a, b, c,
   output [9:0] e
);

wire [15:0] d;
assign d = a * b;       // No implicit upper-bit truncation
assign e = c + d[15:8]; // "explicit lower" bit truncation of d
endmodule
```

5

Sequential Logic

The term register refers to a 1-bit memory device, either a flip-flop or latch. A flip-flop is an edge-triggered memory device, while a latch is a level-sensitive memory device. The following topics describe flip-flop and latch inference:

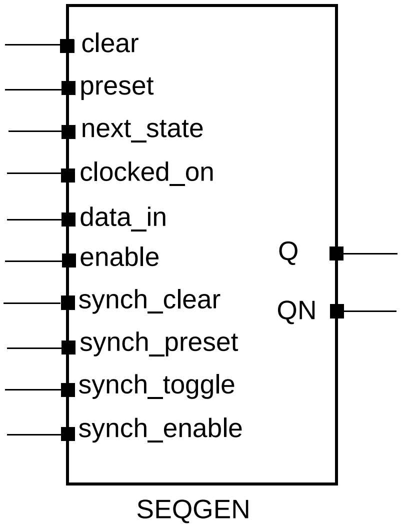
- Generic Sequential Cell SEQGEN
- Inference Reports for Registers
- Register Inference Guidelines
- Register Inference Examples

For a complete FIFO design example that uses the `always_ff` construct, see SystemVerilog Design Examples.

Generic Sequential Cell SEQGEN

When the HDL Compiler tool reads a design, it uses a generic sequential cell SEQGEN shown in Figure 5-1 to represent an inferred flip-flop or latch.

Figure 5-1 SEQGEN Cell and Pin Assignments



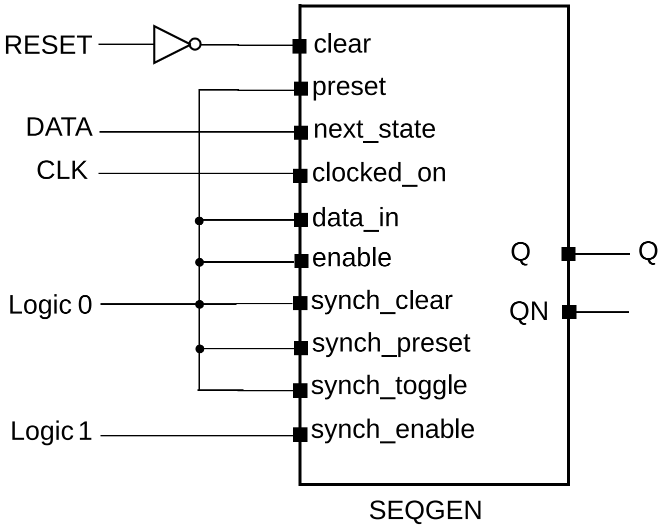
Example 5-1 shows how to direct the HDL Compiler tool to use a SEQGEN cell to implement a D flip-flop with an asynchronous reset.

Example 5-1 D Flip-Flop With Asynchronous Reset

```
module dff_async_set (
   input DATA, CLK, RESET,
   output logic Q
);
always_ff @(posedge CLK or negedge RESET)
if (~RESET) Q <= 1'b0;
else        Q <= DATA;
endmodule
```

Figure 5-2 shows the SEQGEN implementation.

Figure 5-2 SEQGEN Implementation



Example 5-2 shows the `report_cell` output, where the inferred Q_reg flip-flop is mapped to a SEQGEN cell.

Example 5-2 report_cell Output

```
****************************************
Report : cell
Design : dff_async_set
Version: J-2014.09
Date   : Tue Oct 7 14:42:54 2014
****************************************

Attributes:
    b - black box (unknown)
    h - hierarchical
    n - noncombinational
    r - removable
    u - contains unmapped logic

Cell                   Reference       Library             Area  Attributes
--------------------------------------------------------------------------
I_0                    GTECH_NOT       gtech           0.000000   u
Q_reg                  **SEQGEN**                      0.000000   n, u
--------------------------------------------------------------------------
Total 2 cells                                          0.000000
1
```

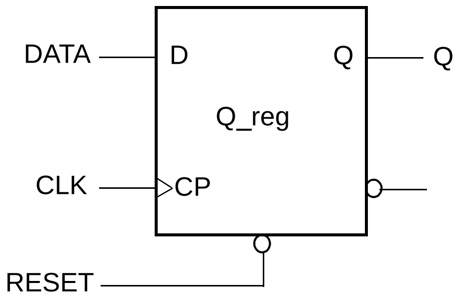
Example 5-3 shows the GTECH netlist.

Example 5-3 GTECH Netlist

```
module dff_async_set ( DATA, CLK, RESET, Q );
  input DATA, CLK, RESET;
  output Q;
  wire   N0;

  \**SEQGEN**  Q_reg ( .clear(N0), .preset(1'b0), .next_state(DATA),
        .clocked_on(CLK), .data_in(1'b0), .enable(1'b0), .Q(Q),
        .synch_clear(1'b0), .synch_preset(1'b0), .synch_toggle(1'b0),
        .synch_enable(1'b1)
        );
  GTECH_NOT I_0 ( .A(RESET), .Z(N0) );
endmodule
```

After Design Compiler compiles the design, the SEQGEN is mapped to the appropriate flip-flop in the logic library. Figure 5-3 shows an example of an implementation after compile.

Figure 5-3 Design Compiler Implementation



Note:

If the logic library does not contain the inferred flip-flop or latch, the Design Compiler tool creates combinational logic for the missing function. For example, if you describe a D flip-flip with a synchronous set but your target library does not contain this type of flip-flop, the tool creates combinational logic for the synchronous set function. The tool cannot create logic to duplicate an asynchronous preset or reset. Your library must contain the sequential cell with the asynchronous control pins. For more information, see Register Inference Limitations.

Inference Reports for Registers

HDL Compiler provides inference reports that describe each inferred flip-flop or latch. You can enable or disable the generation of inference reports by using the `hdlin_reporting_level` variable. By default, `hdlin_reporting_level` is set to `basic`.

When `hdlin_reporting_level` is set to `basic` or `comprehensive`, HDL Compiler generates a report similar to Example 5-4. This basic inference report shows only which type of register was inferred.

Example 5-4 Inference Report for a D Flip-Flop With Asynchronous Reset

```
=========================================================================
| Register Name |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
=========================================================================
|   Q_reg       | Flip-flop |   1   |  N  | N  | Y  | N  | N  | N  | N  |
=========================================================================
```

In the report, the columns are abbreviated as follows:

- MB represents multibit cell
- AR represents asynchronous reset
- AS represents asynchronous set
- SR represents synchronous reset
- SS represents synchronous set
- ST represents synchronous toggle

A “Y” in a column indicates that the respective control pin was inferred for the register; an “N” indicates that the respective control pin was not inferred for the register. For a D flip-flop with an asynchronous reset, there should be a “Y” in the AR column. The report also indicates the type of register inferred, latch or flip-flop, and the name of the inferred cell.

When the `hdlin_reporting_level` variable is set to `verbose`, the report indicates how each pin of the SEQGEN cell is assigned, along with which type of register was inferred. Example 5-5 shows a verbose inference report.

Example 5-5 Verbose Inference Report for a D Flip-Flop With Asynchronous Reset

```
=========================================================================
| Register Name |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
=========================================================================
|    Q_reg      | Flip-flop |   1   |  N  | N  | Y  | N  | N  | N  | N  |
=========================================================================
Sequential Cell (Q_reg)
      Cell Type: Flip-Flop
      Multibit Attribute: N
      Clock: CLK
      Async Clear: RESET
      Async Set: 0
      Async Load: 0
      Sync Clear: 0
      Sync Set: 0
      Sync Toggle: 0
      Sync Load: 1
```

If you do not want the inference report, set the `hdlin_reporting_level` variable to `none`.

See Also

- Reporting Elaboration Errors in the Hierarchy

Register Inference Guidelines

When inferring registers, restrict each always block so that it infers a single type of memory element and check the inference report to verify that HDL Compiler inferred the correct device.

Register inference guidelines are described in the following sections:

- Multiple Events in an always Block
- Minimizing Registers
- Keeping Unloaded Registers
- Preventing Unwanted Latches
- Reset Logic Inference
- Register Inference Limitations

Multiple Events in an always Block

HDL Compiler supports multiple events in a single `always` block, as shown in Example 5-6.

Example 5-6 Multiple Events in a Single always Block

```
module test (
   input [7:0] din,
   input clk,
   output logic [7:0] result
);
always_ff
begin
   @ (posedge clk) result <= din;
   @ (posedge clk) result <= result + din;
   @ (posedge clk) result <= result + din;
end
endmodule
```

Minimizing Registers

An `always` or `always_ff` block that contains a clock edge in the sensitivity list causes a flip-flop inference for each variable assigned a value in that block. It might not be necessary to infer as flip-flops all variables in the always block. Make sure your HDL description builds only as many flip-flops as the design requires.

Example 5-7 infers six flip-flops: three to hold the values of count and one each to hold and_bits, or_bits, and xor_bits. However, the output values of the and_bits, or_bits, and xor_bits depend solely on the value of count. Because count is registered, there is no reason to register the three outputs.

Example 5-7 Inefficient Circuit Description With Six Inferred Registers

```
module count (
input clock, reset,
output logic and_bits, or_bits, xor_bits
);
logic [2:0] count;
// synopsys sync_set_reset "reset"
always_ff @(posedge clock)
begin
if (reset) count <= 0;
else       count <= count + 1;
and_bits <= & count;
or_bits  <= | count;
xor_bits <= ^ count;
end
endmodule
```

Example 5-8 shows the inference report which contains the six inferred flip-flops.

Example 5-8 Inference Report

```
=========================================================================
|Register Name |    Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
=========================================================================
|  count_reg   | Flip-flop |   3   |  Y  | N  | N  | N  | Y  | N  | N  |
| and_bits_reg | Flip-flop |   1   |  N  | N  | N  | N  | N  | N  | N  |
|  or_bits_reg | Flip-flop |   1   |  N  | N  | N  | N  | N  | N  | N  |
| xor_bits_reg | Flip-flop |   1   |  N  | N  | N  | N  | N  | N  | N  |
=========================================================================
```

To avoid inferring extra registers, you can assign the outputs from within an asynchronous always block. Example 5-9 shows the same function described with two `always` blocks, one synchronous and one combinational, that separate registered or sequential logic from combinational logic. This technique is useful for describing finite state machines. Signal assignments in the synchronous always block are registered, but signal assignments in the asynchronous always block are not. The code in Example 5-9 creates a more area-efficient design.

Example 5-9 Circuit With Three Inferred Registers

```
module count (
   input clock, reset,
   output logic and_bits, or_bits, xor_bits
);
logic [2:0] count;
// synopsys sync_set_reset "reset"
always_ff @(posedge clock)
if (reset) count <= 0;
else       count <= count + 1;

always_comb
begin
   and_bits = & count;
   or_bits  = | count;
   xor_bits = ^ count;
end
endmodule
```

Example 5-10 shows the inference report, which contains three inferred flip-flops.

Example 5-10 Inference Report

```
=========================================================================
| Register Name |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
=========================================================================
|  count_reg    | Flip-flop |   3   |  Y  | N  | N  | N  | Y | N  | N  |
=========================================================================
```

See Also

- D Flip-Flop With Synchronous Reset: Use sync_set_reset

Keeping Unloaded Registers

The tool does not keep unloaded or undriven flip-flops and latches in a design during optimization. You can use the `hdlin_preserve_sequential` variable to control which cells to preserve:

- To preserve unloaded/undriven flip-flops and latches in your GTECH netlist, set `hdlin_preserve_sequential` to `all`.
- To preserve all unloaded flip-flops only, set `hdlin_preserve_sequential` to `ff`.
- To preserve all unloaded latches only, set `hdlin_preserve_sequential` to `latch`.
- To preserve all unloaded sequential cells, including unloaded sequential cells that are used solely as loop variables, set `hdlin_preserve_sequential` to `all+loop_variables`.

- To preserve flip-flop cells only, including unloaded sequential cells that are used solely as loop variables, set `hdlin_preserve_sequential` to `ff+loop_variables`.
- To preserve unloaded latch cells only, including unloaded sequential cells that are used solely as loop variables, set `hdlin_preserve_sequential` to `latch+loop_variables`.

If you want to preserve specific registers, use the `preserve_sequential` directive as shown in Example 5-11 and Example 5-12.

Important:

> To preserve unloaded cells through compile, you must set `compile_delete_unloaded_sequential_cells` to `false`. Otherwise, Design Compiler removes them during optimization.

Example 5-11 uses the `preserve_sequential` directive to save the unloaded cell, sum2, and the combinational logic preceding it; note that the combinational logic after it is not saved. If you also want to save the combinational logic after sum2, you need to recode design mydesign as shown in Example 5-12.

Example 5-11 Retains an Unloaded Cell (sum2) and Two Adders

```
module mydesign (
   input clk,
   input [0:1] in1, in2, in3,
   output [0:3] out
);
logic sum1, sum2 /* synopsys preserve_sequential */;
logic [0:4] save;
always_ff @ (posedge clk)
begin
   sum1 <= in1 + in2;
   // sum2 register is preserve
   sum2 <= in1 + in2 + in3;
end
assign out = ~sum1;
assign save = sum1 + sum2;
endmodule
```

Example 5-12 preserves all combinational logic before reg save.

Example 5-12 Retains an Unloaded Cell and Three Adders

```
module adders (
   input clk,
   input [0:1] in1, in2, in3,
   output [0:3] out
);
logic sum1, sum2 ;
logic [0:4] save /* synopsys preserve_sequential */;

// sum2 register is preserved
always_ff @ (posedge clk)
begin
   sum1 <= in1 + in2;
   sum2 <= in1 + in2 + in3;
end

// save register is preserved
always_ff @ (posedge clk)
save <= sum1 + sum2;

assign out = ~sum1;
endmodule
```

The `preserve_sequential` directive and the `hdlin_preserve_sequential` variable enable you to preserve cells that are inferred but optimized away by HDL Compiler. If a cell is never inferred, the `preserve_sequential` directive and the `hdlin_preserve_sequential` variable have no effect because there is no inferred cell to act on. In Example 5-13, sum2 is not inferred, so `preserve_sequential` does not save sum2.

Example 5-13 preserve_sequential Has No Effect on Cells Not Inferred

```
module adders (
   input clk,
   input [0:1] in1, in2,
   output [0:3] out
);
logic sum1, sum2 /* synopsys preserve_sequential */;
wire [0:4] save;
always_ff @ (posedge clk)
begin
   sum1 <= in1 + in2;
end

/*
Although the preserve_sequential directive is on
sum2, it is not saved due to sum2 is not inferred
*/
assign out = ~sum1;
assign save = sum2;
endmodule
```

Note:

By default, the `hdlin_preserve_sequential` variable does not preserve variables used in for loops as unloaded registers. To preserve such variables, you must set `hdlin_preserve_sequential` to `ff+loop_variables`.

In addition to preserving sequential cells with the `hdlin_preserve_sequential` variable and the `preserve_sequential` directive, you can also use the `hdlin_keep_signal_name` variable and the `keep_signal_name` directive. For more information, see Keeping Signal Names.

Note:

The tool does not distinguish between unloaded cells (those not connected to any output ports) and feedthroughs. See Example 5-14 for a feedthrough.

Example 5-14

```
module test (
   input clk,in,
   output logic out
);
logic tmp1;
always_ff @ (posedge clk)
begin
   tmp1 <= in;
   out  <= tmp1;
end
endmodule
```

With `hdlin_preserve_sequential` set to `ff`, the tool builds two registers; one for the feedthrough cell (temp1) and the other for the loaded cell (temp2) as shown in the following memory inference report:

Example 5-15 Feedthrough Register temp1

```
=================================================================
| Register Name |    Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
=================================================================
|   tmp1_reg    | Flip-flop |   1   |  N  | N  |  N | N  | N  | N  | N  |
|    out_reg    | Flip-flop |   1   |  N  | N  |  N | N  | N  | N  | N  |
=================================================================
```

Preventing Unwanted Latches

When you do not specify a signal or variable in all branches of a combinational logic block, the tool infers latches (see Latches in Combinational Logic). Use the SystemVerilog always_comb construct to model combinational logic, so the tool checks whether the logic inferred within the block represents combinational logic. If a latch is inferred, the tool issues an ELAB-974 warning. In addition, you can set the `hdlin_check_no_latch` variable to check for ELAB-395 message, which indicates latch inference.

As shown in Example 5-16, one branch of the case statement is commented out, so output DOUT is not fully specified and the tool infers a latch.

Example 5-16

```
module selector (
   input [1:0] SEL,
   input [3:0] DIN,
   output logic DOUT
);
always_comb
case (SEL)
  2'b00: DOUT = DIN[0];
  2'b01: DOUT = DIN[1];
  2'b10: DOUT = DIN[2];
//  2'b11: DOUT = DIN[3];
endcase
endmodule
```

Reset Logic Inference

To enable the tool to recognize reset signals and infer proper reset logic, you can use the sync_set_reset directive, the hdlin_ff_always_sync_set_reset variable, or the hdlin_ff_always_async_set_reset variable.

- The sync_set_reset directive

  When the directive is set on single-bit signals, the tool infers flip-flops with synchronous set and reset logic using those signals. For more information about this directive, see sync_set_reset.

  For example, the following code enables the tool to recognize the reset and int_reset signals as the reset logic:

  //synopsys sync_set_reset "reset, int_reset"

- The hdlin_ff_always_sync_set_reset variable

  When this variable is set to false (the default), the tool infers synchronous set and reset logic only for flip-flops that have the sync_set_reset directive. When you set this variable to true without specifying the sync_set_reset directive, the tool tries to infer synchronous set and reset logic for flip-flops on which a constant 0 or constant 1 is loaded under the clock event.

- The hdlin_ff_always_async_set_reset variable

  When this variable is set to true (the default) and the async_set_reset directive is not set, the tool infers asynchronous set and reset logic for flip-flops by checking for asynchronous set and reset conditions on the flip-flops. When you set this variable to false, the tool does not attempt to identify any asynchronous set and reset condition and it uses the async_set_reset directive to identify the asynchronous set and reset signals for each flip-flop.

In an always block, you should always give the synchronous reset the highest priority so that the tool recognizes the reset signal. For example,

```
//synopsys sync_set_reset "reset"
always_ff @(posedge CLK)
begin
   if reset
      out_reg <= 1'b0;
   else if (signal1)
      out_reg <= input1;
   else
      out_reg <= input2;
end
```

When more than one reset signal is in a block, you should specify the reset in the first and second if statements. For example,

```
//synopsys sync_set_reset "reset, reset_int"
always_ff @(posedge CLK)
begin
   if reset
      out_reg <= 1'b0;
   else if reset_int
      out_reg <= 1'b0;
   else if (signal1)
      out_reg <= input1;
   else
      out_reg <= input2;
end
```

This coding style enables the tool to recognize the reset signals and display them in the summary report of registers, as shown in the following inference report:

```
=========================================================================
| Register Name |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
=========================================================================
|    out_reg    | Flip-flop |   1   |  N  | N  |  N | N  | Y  | N  | N  |
=========================================================================
```

When you assign an initial value to a register upon reset, you must set it to a known value. If reset registers are pipelined, such as a register feeding into another register, the tool does not recognize the reset signal at the register that is not initialized. For example,

```
//synopsys sync_set_reset "reset"
always_ff @(posedge CLK)
begin
   if reset
      out_reg <= Out_reg_first;
   else if (signal1)
      out_reg <= input1;
   else
      out_reg <= input2;
end
```

See Also

- D Flip-Flop With Synchronous Reset: Use sync_set_reset

Register Inference Limitations

Note the following limitations when inferring registers:

- The tool does not support more than one independent if-block when asynchronous behavior is modeled within an always block. If the always block is purely synchronous, multiple independent if-blocks are supported by the tool.
- The HDL Compiler tool cannot infer flip-flops and latches with three-state outputs. You must instantiate these components in your Verilog description.
- The HDL Compiler tool cannot infer flip-flops with bidirectional pins. You must instantiate these components in the RTL.
- The HDL Compiler tool cannot infer flip-flops with multiple clock inputs. You must instantiate these components in the RTL.
- The HDL Compiler tool cannot infer multiport latches. You must instantiate these components in the RTL.
- The HDL Compiler tool cannot infer register banks (register files). You must instantiate these components in the RTL.
- Although you can instantiate flip-flops with bidirectional pins, the Design Compiler tool interprets these cells as black boxes.
- If you use an `if` statement to infer D flip-flops, the `if` statement must occur at the top level of the `always` block.

  The following example is invalid because the `if` statement does not occur at the top level:

```
module invalid (
   input clk, reset,
   input d,
   output logic q
);

logic temp;
always_ff @(posedge clk or posedge reset)
begin
   temp <= reset;
   if (reset) q <= 1'b0;
   else       q <= d;
end
endmodule
```

The tool issues the following message when the `if` statement does not occur at the top level:

```
Error:  .../test.sv:8: The statements in this 'always' block are
outside the scope of the synthesis policy. Only an 'if' statement is
allowed at the top level in this always block. (ELAB-302)
```

Register Inference Examples

The following sections describe register inference examples:

- Inferring Latches
- Inferring Flip-Flops

Inferring Latches

The tool infers latches when variables are conditionally assigned. A variable is conditionally assigned if there is a path that does not explicitly assign a value to that variable.

- Basic D Latch
- D Latch With Asynchronous Set: Use async_set_reset
- D Latch With Asynchronous Reset: Use async_set_reset
- D Latch With Asynchronous Set and Reset: Use hdlin_latch_always_async_set_reset
- Unintended Logic Inferred Using always_latch

Basic D Latch

To direct the tool to infer a D latch, you need to control the gate and data signals from the top-level ports or through combinational logic, so simulation can initialize the design. Example 5-17 shows that a D latch is inferred for the `always_latch` and `always@` constructs.

Example 5-17 D Latch Code

```
module d_latch_A (
   input GATE, DATA,
   output logic Q
);
always_latch
if (GATE) Q <= DATA;
endmodule

module d_latch_B (
   input GATE, DATA,
   output reg Q
);
always @(GATE or DATA)
if (GATE) Q <= DATA;
endmodule
```

The HDL Compiler tool generates the inference report shown in Example 5-18.

Example 5-18 Inference Report

```
=========================================================================
|   Register Name   | Type  | Width | Bus | MB | AR | AS | SR | SS | ST |
=========================================================================
|       Q_reg       | Latch |   1   |  N  | N  | N  | N  | -  | -  | -  |
=========================================================================
```

D Latch With Asynchronous Set: Use async_set_reset

Example 5-19 shows the recommended coding style for an asynchronously set latch using the `async_set_reset` directive.

Example 5-19 D Latch With Asynchronous Set: Uses async_set_reset

```
module d_latch_async_set (
   input GATE, DATA, SET,
   output logic Q
);
// synopsys async_set_reset "SET"
always_latch
if (~SET)      Q <= 1'b1;
else if (GATE) Q <= DATA;
endmodule
```

The tool generates the inference report shown in Example 5-20.

Example 5-20 Inference Report for D Latch With Asynchronous Set

```
===========================================================================
|    Register Name    | Type  | Width | Bus | MB | AR | AS | SR | SS | ST |
===========================================================================
|        Q_reg        | Latch |   1   |  N  | N  | N  | Y  | -  | -  | -  |
===========================================================================
```

D Latch With Asynchronous Reset: Use async_set_reset

Example 5-21 shows the recommended coding style for an asynchronously reset latch using the `async_set_reset` directive.

Example 5-21 D Latch With Asynchronous Reset: Uses async_set_reset

```
module d_latch_async_reset (
   input RESET, GATE, DATA,
   output logic Q
);
//synopsys async_set_reset "RESET"
always_latch
if (~RESET)    Q <= 1'b0;
else if (GATE) Q <= DATA;
endmodule
```

The tool generates the inference report shown in Example 5-22.

Example 5-22 Inference Report for D Latch With Asynchronous Reset

```
=============================================================================
|    Register Name    | Type  | Width | Bus | MB | AR | AS | SR | SS | ST |
=============================================================================
|        Q_reg        | Latch |   1   |  N  | N  | Y  | N  | -  | -  | -  |
=============================================================================
```

D Latch With Asynchronous Set and Reset: Use hdlin_latch_always_async_set_reset

To infer a D latch with an active-low asynchronous set and reset, set the `hdlin_latch_always_async_set_reset` variable to true and use the coding style shown in Example 5-23.

Note:

> This example uses the `one_cold` directive to prevent priority encoding of the set and reset signals. Although this saves area, it might cause a simulation/synthesis mismatch if both signals are low at the same time.

Example 5-23 D Latch With Asynchronous Set and Reset: Uses hdlin_latch_always_async_set_reset

```
module d_latch_async (
   input GATE, DATA, RESET, SET,
   output logic Q
);
// synopsys one_cold "RESET, SET"
always_latch
if (!SET)        Q <= 1'b1;
else if (!RESET) Q <= 1'b0;
else if (GATE)   Q <= DATA;
endmodule
```

Example 5-24 shows the inference report.

Example 5-24 Inference Report D Latch With Asynchronous Set and Reset

```
===================================================================
|    Register Name    | Type  | Width | Bus | MB | AR | AS | SR | SS | ST |
===================================================================
|       Q_reg         | Latch |   1   |  N  | N  | Y  | Y  | -  | -  | -  |
===================================================================
```

See Also

- Unintended Logic Inferred Using always_latch

Unintended Logic Inferred Using always_latch

Although you use the `always_latch` construct to describe sequential logic, the tool might not infer the intended logic when synthesizing your code. For example, when one of the signals driven from an `always_latch` block is needed to compute an output of a module. As shown in Example 5-25, the tmp logic is not defined as an output port and might be removed during synthesis. An unintended empty block might be inferred, and the tool issues an ELAB-983 warning message.

Example 5-25 Unintended Empty Block

```
module empty_always_latch(
   input logic clk, in,
   output logic out
);

logic tmp;

always_latch
begin
   if(clk)
   tmp <= in;
end
endmodule
```

Inferring Flip-Flops

Synthesis of sequential elements, such as various types of flip-flops, often involves signals that set or reset the sequential device. Synthesis tools can create a sequential cell that has built-in set and reset functionality. This is referred to as set/reset inference. For an example using a flip-flop with reset functionality, consider the following RTL code:

```
module m (
   input clk, set, reset, d,
   output reg q
);
always_ff @ (posedge clk)
if (reset) q <= 1'b0;
else       q <= d;
endmodule
```

There are two ways to synthesize an electrical circuit with a reset signal based on the previous code. You can either synthesize the circuit with a simple flip-flop with external combinational logic to represent the reset functionality, as shown in Figure 5-4, or you can synthesize a flip-flop with built-in reset functionality, as shown in Figure 5-5.

Figure 5-4 Flip-Flop With External Combinational Logic to Represent Reset

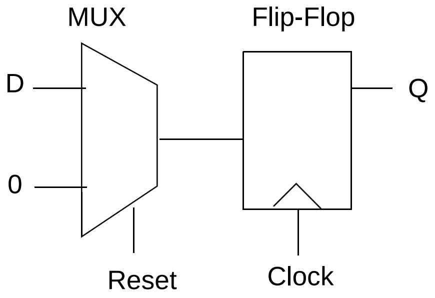
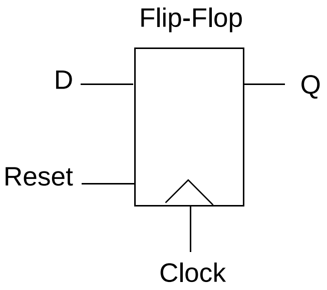


Figure 5-5 Flip-Flop With Built-In Reset Functionality



The intended implementation is not apparent from the RTL code. You should specify HDL Compiler synthesis directives or Design Compiler variables to guide the tool to create the proper synchronous set and reset signals.

SystemVerilog provides the `always_ff` construct for modeling sequential logic. The tool checks whether the logic inferred represents sequential logic.

The following sections provide examples of these flip-flops:

- Basic D Flip-Flop
- D Flip-Flop With Asynchronous Reset Using ?: Construct
- D Flip-Flop With Asynchronous Reset
- D Flip-Flop With Asynchronous Set and Reset
- D Flip-Flop With Synchronous Set: Use sync_set_reset
- D Flip-Flop With Synchronous Reset: Use sync_set_reset
- D Flip-Flop With Synchronous and Asynchronous Load
- D Flip-Flops With Complex Set and Reset Signals
- Multiple Flip-Flops With Asynchronous and Synchronous Controls
- Unintended Logic Inferred Using always_ff

Basic D Flip-Flop

When you infer a D flip-flop, make sure you can control the clock and data signals from the top-level design ports or through combinational logic. Controllable clock and data signals ensure that simulation can initialize the design. If you cannot control the clock and data signals, infer a D flip-flop with an asynchronous reset or set or with a synchronous reset or set.

Example 5-26 infers a basic D flip-flop.

Example 5-26 Basic D Flip-Flop

```
module dff_pos (
   input DATA, CLK,
   output logic Q
);
always_ff @(posedge CLK)
   Q <= DATA;
endmodule
```

HDL Compiler generates the inference report shown in Example 5-27.

Example 5-27 Inference Report

```
===============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|        Q_reg        | Flip-flop |   1   |  N  | N  | N  | N  | N  | N  | N  |
===============================================================================
```

D Flip-Flop With Asynchronous Reset Using ?: Construct

Example 5-28 uses the ?: construct to infer a D flip-flop with an asynchronous reset. Note that the tool does not support more than one ?: operator inside an always block.

Example 5-28 D Flip-Flop With Asynchronous Reset Using ?: Construct

```
module dff_async_reset (
   input CLK, RESET, DATA,
   output logic Q
);
always_ff @ (posedge CLK or negedge RESET)
   Q <= (!RESET) ? 1'b0 : DATA;
endmodule
```

HDL Compiler generates the inference report shown in Example 5-29.

Example 5-29 D Flip-Flop With Asynchronous Reset Inference Report

```
==============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
==============================================================================
|       Q_reg         | Flip-flop |   1   |  N  | N  | Y  | N  | N  | N  | N  |
==============================================================================
```

D Flip-Flop With Asynchronous Reset

Example 5-30 infers a D flip-flop with an asynchronous reset.

Example 5-30 D Flip-Flop With Asynchronous Reset

```
module dff_async_reset (
   input DATA, CLK, RESET,
   output logic Q
);
always_ff @(posedge CLK or posedge RESET)
if (RESET) Q <= 1'b0;
else       Q <= DATA;
endmodule
```

HDL Compiler generates the inference report shown in Example 5-31.

Example 5-31 D Flip-Flop With Asynchronous Reset Inference Report

```
==============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
==============================================================================
|       Q_reg         | Flip-flop |   1   |  N  | N  | Y  | N  | N  | N  | N  |
==============================================================================
```

D Flip-Flop With Asynchronous Set and Reset

Example 5-32 infers a D flip-flop with asynchronous set and reset pins. The example uses the `one_hot` directive to prevent priority encoding of the set and reset signals. If signals SET and RESET are asserted at the same time, the synthesized hardware is unpredictable. To check for this condition, use the SYNTHESIS macro and the `` `ifndef ... `endif `` constructs (see Predefined SYSTEMVERILOG Macro).

Example 5-32 D Flip-Flop With Asynchronous Set and Reset

```
module dff_async (
   input CLK, RESET, SET, DATA,
   output logic Q
);
// synopsys one_hot "RESET, SET"
always_ff @(posedge CLK or posedge RESET or posedge SET)
if (RESET)    Q <= 1'b0;
else if (SET) Q <= 1'b1;
else          Q <= DATA;

`ifndef SYNTHESIS
always @ (RESET or SET)
if (!$onehot0 ({RESET, SET}))
$write ("\nONE-HOT violation for RESET and SET.\n");
`endif
endmodule
```

Example 5-33 shows the inference report.

Example 5-33 D Flip-Flop With Asynchronous Set and Reset Inference Report

```
============================================================================
|   Register Name   |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
============================================================================
|      Q_reg        | Flip-flop |   1   |  N  | N  | Y  | Y  | N  | N  | N  |
============================================================================
```

D Flip-Flop With Synchronous Set: Use sync_set_reset

This example shows a D flip-flop design with a synchronous set.

The `sync_set_reset` directive is applied to the SET signal. If the target library does not have a D flip-flop with synchronous set, the Design Compiler tool infers synchronous set logic as the input to the D pin of the flip-flop. If the set logic is not directly in front of the D pin of the flip-flop, initialization problems can occur during gate-level simulation of the design. The `sync_set_reset` directive ensures that this logic is as close to the D pin as possible.

Design of a D Flip-Flop With Synchronous Set

```
module dff_sync_set (
   input DATA, CLK, SET,
   output logic Q
);
//synopsys sync_set_reset "SET"
always_ff @(posedge CLK)
if (SET) Q <= 1'b1;
else     <= DATA;
endmodule
```

Inference Report

```
=========================================================================
| Register Name |    Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
=========================================================================
|     Q_reg     | Flip-flop |   1   |  N  | N  | N  | N  | N  | Y  | N  |
=========================================================================
```

D Flip-Flop With Synchronous Reset: Use sync_set_reset

Example 5-34 infers a D flip-flop with synchronous reset. The `sync_set_reset` directive is applied to the RESET signal.

Example 5-34   D Flip-Flop With Synchronous Reset: Use sync_set_reset

```
module dff_sync_reset (
   input DATA, CLK, RESET,
   output logic Q
);
//synopsys sync_set_reset "RESET"
always_ff @(posedge CLK)
if (~RESET)  Q <= 1'b0;
else         Q <= DATA;
endmodule
```

HDL Compiler generates the inference report shown in Example 5-35.

Example 5-35   D Flip-Flop With Synchronous Reset Inference Report

```
=========================================================================
| Register Name |    Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
=========================================================================
|     Q_reg     | Flip-flop |   1   |  N  | N  | N  | N  | Y  | N  | N  |
=========================================================================
```

D Flip-Flop With Synchronous and Asynchronous Load

Use the coding style in Example 5-36 to infer a D flip-flop with both synchronous and asynchronous load signals.

Example 5-36 Synchronous and Asynchronous Loads

```
module dff_a_s_load (
   input ALOAD, ADATA, SLOAD, SDATA, CLK,
   output logic Q
);
wire asyn_rst, asyn_set;
assign asyn_rst = ALOAD && !ADATA;
assign asyn_set = ALOAD && ADATA;

//synopsys one_cold "ALOAD, ADATA"
always_ff @ (posedge CLK or posedge asyn_rst or posedge asyn_set)
begin
   if (asyn_set)       Q <= 1'b1;
   else if (asyn_rst)  Q <= 1'b0;
   else if (SLOAD)     Q <= SDATA;
end
endmodule
```

HDL Compiler generates the inference report shown in Example 5-37.

Example 5-37 D Flip-Flop With Synchronous and Asynchronous Load Inference Report

```
==============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
==============================================================================
|       Q_reg        | Flip-flop |   1   |  N  | N  | Y  | Y  | N  | N  | N  |
==============================================================================
Sequential Cell (Q_reg)
        Cell Type: Flip-Flop
        Multibit Attribute: N
        Clock: CLK
        Async Clear: ADATA' ALOAD
        Async Set: ADATA ALOAD
        Async Load: 0
        Sync Clear: 0
        Sync Set: 0
        Sync Toggle: 0
        Sync Load: SLOAD
```

D Flip-Flops With Complex Set and Reset Signals

While many set and reset signals are simple signals, some include complex logic. To enable HDL Compiler to generate a clean set/reset (that is, a set/reset signal attached only to the appropriate set/reset pins), use the following coding guidelines:

- Apply the appropriate set/reset compiler directive (`//synopsys sync_set_reset` or `//synopsys async_set_reset`) to the set/reset signal.
- Use no more than two operands in the set/reset logic expression conditional.
- Use the set/reset signal as the first operand in the set/reset logic expression conditional.

This coding style supports usage of the negation operator on the set/reset signal and the logic expression. The logic expression can be a simple expression or any expression contained inside parentheses. However, any deviation from these coding guidelines is not supported. For example, using a more complex expression other than the OR of two expressions, or using a rst (or ~rst) that does not appear as the first argument in the expression is not supported.

Examples

```
//synopsys sync_set_reset "rst"
always_ff @(posedge clk)
if (rst | logic_expression)
   q <= 0;
else ...
else ...
...

//synopsys sync_set_reset "rst"
assign a = rst |  ~( a | b & c);
always_ff @(posedge clk)
if (a)
   q <= 0;
else ...;
else ...;
...

//synopsys sync_set_reset "rst"
always_ff @(posedge clk)
if ( ~ rst |  ~ (a | b | c))
   q <= 0;
else ...
else ...
...

//synopsys sync_set_reset "rst"
assign a =  ~ rst |  ~ logic_expression;
always_ff @(posedge clk)
if (a)
   q <= 0;
else ...;
else ...;
...
```

Multiple Flip-Flops With Asynchronous and Synchronous Controls

In Example 5-38, the infer_sync block uses the reset signal as a synchronous reset and the infer_async block uses the reset signal as an asynchronous reset.

Example 5-38 Multiple Flip-Flops With Asynchronous and Synchronous Controls

```
module multi_attr (
   input DATA1, DATA2, CLK, RESET, SLOAD,
   output logic Q1, Q2
);

//synopsys sync_set_reset "RESET"
always_ff @(posedge CLK)
begin: infer_sync
   if (~RESET)      Q1 <= 1'b0;
   else if (SLOAD)  Q1 <= DATA1;
// note: else hold Q1
end: infer_sync

always_ff @(posedge CLK or negedge RESET)
begin: infer_async
   if (~RESET)      Q2 <= 1'b0;
   else if (SLOAD)  Q2 <= DATA2;
// note: else hold Q1
end: infer_async
endmodule
```

Example 5-39 shows the inference report.

Example 5-39 Inference Report

```
==============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
==============================================================================
|       Q1_reg        | Flip-flop |   1   |  N  | N  | N  | N  | Y  | N  | N  |
==============================================================================

=============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
=============================================================================
|       Q2_reg        | Flip-flop |   1   |  N  | N  | Y  | N  | N  | N  | N  |
=============================================================================
```

Unintended Logic Inferred Using always_ff

Although you use the `always_ff` construct to describe flip-flops, the tool might not infer the intended logic when synthesizing your code. For example, when one of the signals driven from an `always_ff` block is needed to compute an output of a module. As shown in Example 5-40, the tmp logic is not defined as an output port and might be removed during synthesis. An unintended empty block might be inferred, and the tool issues an ELAB-984 warning message.

Example 5-40

```
module empty_alway_ff (
   input logic clk, in,
   output logic out
);

logic tmp;
always_ff @(posedge clk)
begin
   tmp <= in;
end
endmodule
```

6

State Machines

You can assign integral named constants to state variables in state machines by using enumerations. This feature allows you to identify the state assignments for the synthesis tool.

The following topics provide state machine examples that use enumerations:

- Using Enumerated Types
- Automatic Increments for the State Variables
- Enumeration Range
- Methods for Enumerated Types
- Separate Sequential and Combinational Assignments

The following topics describe state machine inference:

- FSM Coding Requirements for Automatic Inference
- FSM Inference Variables
- FSM Coding Example
- FSM Inference Report
- Enumerated Types

Using Enumerated Types

You can use enumerated types to describe the state variables in state machines. The default data type for enumerations is `int`, which is 32-bit. You can also declare other explicit data type.

The following two state machine examples describe how to use enumerations with

- Default Data Type
- Explicit Data Types

See Also

- FSM State Diagram and State Table

FSM State Diagram and State Table

The two state machine examples, Example 6-1 and Example 6-4, use the state diagram and state table shown in Figure 6-1and Table 6-1.

Figure 6-1 Finite State Machine State Diagram

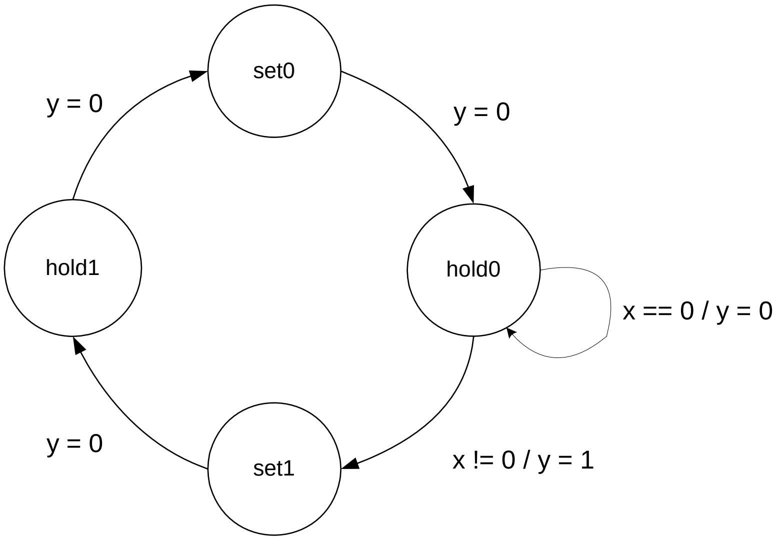


Table 6-1 shows the state table for the state diagram shown in Figure 6-1.

Table 6-1 Finite State Machine State Table

Current state	Input (x)	Next state	Output (y)
0001 (set0)	0	0010 (hold0)	0
0001 (set0)	1	0010 (hold0)	0
0010 (hold0)	0	0010 (hold0)	0
0010 (hold0)	1	0100 (set1)	1
0100 (set1)	0	1000 (hold1)	0
0100 (set1)	1	1000 (hold1)	0
1000 (hold1)	0	0001 (set0)	0
1000 (hold1)	1	0001 (set0)	0

Default Data Type

This example uses the state diagram and state table shown in FSM State Diagram and State Table. The next_state and current_state states have no explicit data type, so they use the default data type, `int`, which is 32 bits.

Example 6-1 State Machine

```
module fsm1cs3 (input x, clk, rst, output y);
enum {set0 = 1, hold0 = 2, set1 = 4, hold1 = 8} current_state,
next_state;

always_ff @ (posedge clk or posedge rst)
if (rst)
   current_state <= set0;
else
   current_state <= next_state;

always_comb
case (current_state)
   set0   : next_state = hold0;
   hold0  : if (x == 0)
            next_state = hold0;
            else
            next_state = set1;
   set1   : next_state = hold1;
   hold1  : next_state = set0;
   default: next_state = set0;
endcase
assign y = current_state == hold0 & x;
endmodule
```

To report state machine inference, set the `hdlin_reporting_level` variable to `basic+fsm`, as shown in Example 6-2. The default is `basic`, which means that an FSM inference report is not generated.

Example 6-2 Script

```
dc_shell> set_app_var hdlin_reporting_level basic+fsm
dc_shell> read_sverilog example_6-1.sv
dc_shell> write -format verilog -hierarchy -output gtech.example_6-1.v
dc_shell> write -format verilog -hierarchy -output gates.example_6-1.v
```

Example 6-3 shows the FSM inference, which is 32-bit, for the state machine in Example 6-1.

Example 6-3 Inference Report

```
statistics for FSM inference:
  state register: current_state
  states
  ======
  set0:00000000000000000000000000000001
  hold0:00000000000000000000000000000010
  set1:00000000000000000000000000000100
  hold1:00000000000000000000000000001000

  total number of states: 4
```

See Also

- Customizing Elaboration Reports

Explicit Data Types

Example 6-4 uses the state diagram and state table shown in FSM State Diagram and State Table. To avoid the inference of 32-bit state encoding, you can specify the state variables with a `logic` type of four bits, as shown in the following code:

```
enum logic [3:0] {set0 = 4'b0001, hold0 = 4'b0010,
set1 = 4'b0100, hold1 = 4'b1000} current_state, next_state;
```

Example 6-4

```
module fsm1cs1 (
   input logic x, clk, rst,
   output logic y
);

enum logic [3:0] {set0 = 4'b0001, hold0 = 4'b0010, set1 = 4'b0100,
hold1 = 4'b1000} current_state, next_state;

always_ff @ (posedge clk or posedge rst)
if (rst)
   current_state <= set0;
else
      current_state <= next_state;

always_comb
priority case (current_state)
   set0:  begin
          next_state = hold0;
          y = 0;
          end
   hold0: begin
          if (x == 0)
          begin
             next_state = hold0;
             y = 0;
          end
          else
          begin
             next_state = set1;
             y = 1;
          end
          end
   set1:  begin
          next_state = hold1;
          y = 0;
          end
   hold1: begin
          next_state = set0;
          y = 0;
          end
endcase
endmodule
```

Example 6-5 shows the FSM inference, which is four-bit, for the state machine in Example 6-4.

Example 6-5 FSM Inference Report

```
statistics for FSM inference:
  state register: current_state
  states
  ======
  set0:0001
  hold0:0010
  set1:0100
  hold1:1000

  total number of states: 4
```

Automatic Increments for the State Variables

You do not need to specify a value for each variable in the enumeration list. By default, the tool assigns an increment of the previous variable value to each variable without a value. In Example 6-6, only the IDLE and FIVE states are explicitly specified, as shown in the following enumeration:

```
typedef enum logic [3:0] {IDLE = 1, FIVE = 4, TEN, TWENTY_FIVE, FIFTEEN,
THIRTY, TWENTY, OWE_DIME} state_assignments;
```

The tool assigns the values of 5, 6, 7, 8, 9, and 10 to the TEN, TWENTY_FIVE, FIFTEEN, THIRTY, TWENTY, and OWE_DIME states respectively. These values are increments of the FIVE state value, which is 4.

Example 6-6

```
`define vend_a_drink  {D, dispense, collect} = {IDLE,2'b11}

module drink_machine(
   input logic reset, clk, nickel_in, dime_in, quarter_in,
   output logic collect, nickel_out, dime_out, dispense
);
typedef enum logic [3:0]{IDLE = 1, FIVE = 4, TEN, TWENTY_FIVE, FIFTEEN,
THIRTY, TWENTY,OWE_DIME} state_assignments;
state_assignments D, Q;

always_comb
begin
   nickel_out = 0;
   dime_out   = 0;
   dispense   = 0;
   collect    = 0;
   if ( reset ) D = IDLE;
   else begin
   case ( Q )
      IDLE:
         if (nickel_in)       D = FIVE;
         else if (dime_in)    D = TEN;
         else if (quarter_in) D = TWENTY_FIVE;
      FIVE:
         if(nickel_in)        D = TEN;
         else if (dime_in)    D = FIFTEEN;
         else if (quarter_in) D = THIRTY;
      TEN:
         if (nickel_in)       D = FIFTEEN;
         else if (dime_in)    D = TWENTY;
         else if (quarter_in) `vend_a_drink;
      TWENTY_FIVE:
         if( nickel_in)       D = THIRTY;
         else if (dime_in)    `vend_a_drink;
         else if (quarter_in)
         begin
         `vend_a_drink;
         nickel_out = 1;
         dime_out = 1;
         end
      FIFTEEN:
         if (nickel_in)       D = TWENTY;
         else if (dime_in)    D = TWENTY_FIVE;
         else if (quarter_in)
         begin
         `vend_a_drink;
         nickel_out = 1;
         end
      THIRTY:
         if (nickel_in)       `vend_a_drink;
         else if (dime_in)
         begin
            `vend_a_drink;
            nickel_out = 1;
         end
         else if (quarter_in)
         begin
            `vend_a_drink;
            dime_out = 1;
            D = OWE_DIME;
         end
      TWENTY:
         if (nickel_in)       D = TWENTY_FIVE;
         else if (dime_in)    D = THIRTY;
         else if (quarter_in)
         begin
            `vend_a_drink;
            dime_out = 1;
         end
      OWE_DIME:
         begin
         dime_out = 1;
         D = IDLE;
         end
   endcase
   end
end
always_ff @(posedge clk ) begin
  Q <= D;
end
endmodule
```

Example 6-7 shows the FSM inference report.

Example 6-7 FSM Inference Report

```
statistics for FSM inference:
  state register: Q
  states
  ======
  IDLE:        0001
  FIVE:        0100
  TEN:         0101
  TWENTY_FIVE: 0110
  FIFTEEN:     0111
  THIRTY:      1000
  TWENTY:      1001
  OWE_DIME:    1010

  total number of states: 8
```

Enumeration Range

To simplify state machine coding, you can specify an enumeration range for the tool to determine the states. The coding style in Example 6-8 requires you to specify each state, whereas the coding style in Example 6-9 allows you to specify the enumeration range. In Example 6-9, the tool evaluates the state[2] range and determines the individual states to be state0 and state1.

Example 6-8 Specifying Each State

```
enum { state0, state1 } current_state;
```

Example 6-9 Specifying an Enumeration Range

```
enum { state[2] } current_state;
```

Methods for Enumerated Types

You can use different methods, such as `.first`, `.last`, `.next`, `.prev`, and `.num`, to access enumerated literals of enumerated types. These methods provide alternative coding techniques for finite state machines. For more information, see the *IEEE Std 1800-2012*.

The following enumeration methods are not supported:

- Methods to access explicit state encoding

  For example, `enum {red = 2'd1, blue = 2'd2, green = 2'd4} state;`

- Optional arguments, such as `light.next(5)`

Separate Sequential and Combinational Assignments

To create separate sequential and combinational assignments, use an edge-triggered `always_ff` block for the sequential assignments and a signal-triggered `always` or `always_comb` block for the combinational assignments. Use this technique to create state machines. Example 6-10 shows the implementation of a Mealy state machine.

Example 6-10 Mealy State Machine Example

```
module mealy (
   input in1, in2, clk, reset,
   output logic out
);

enum logic {S1, S2} current_state, next_state;

// Sequential block
always @(posedge clk or negedge reset)
if (!reset) current_state <= S1;
else current_state <= next_state;

// Combinational block
always_comb
unique case (current_state)
// output and state vector decode (combinational)
case (current_state)
   S1: begin
```

```
          next_state = S2;
          out        = 1'b0;
          end
      S2: begin
          next_state = in1?S1:S2;
          out        = in1?in2:~in2;
          end
    endcase
    endmodule
```

FSM Coding Requirements for Automatic Inference

To enable HDL Compiler to automatically infer an FSM, follow the coding guidelines in Table 6-2.

Table 6-2 Code Requirements for FSM Inference

Item	Description
Registers	To infer a register as an FSM state register, the register - Must never be assigned a value other than the defined state values. - Must always be inferred as a flip-flop (and not a latch). - Must never be a module port, function port, or task port. This would make the encoding visible to the outside. Inside expressions, FSM state registers can be used only as an operand of "==" or "!=" comparisons, or as the expression in a case statement (that is, "case (cur_state) ...") that is, an implicit comparison to the label expressions. FSM state registers are not allowed to occur in other expressions—this would make the encoding explicit.
Function	There can be only one FSM design per module. State variables cannot drive a port. State variables cannot be indexed.
Ports	All ports of the initial design must be either input ports or output ports. Inout ports are not supported.
Combinational feedback loops	Combinational feedback loops are not supported although combinational logic that does not depend on the state vector is accurately represented.
Clocks	FSM designs can include only a single clock and an optional synchronous or asynchronous reset signal.

FSM Inference Variables

Finite state machine inference variables are listed in Table 6-3.

Table 6-3 Variables Specific to FSM Inference

Variable	Description
`hdlin_reporting_level`	Default is `basic`. Enables and disables FSM inference reports. When set to `comprehensive`, FSM inference reports are generated when HDL Compiler reads the code. By default, FSM inference reports are not generated. For more information, including valid values, see Customizing Elaboration Reports.
`fsm_auto_inferring`	Determines whether Design Compiler automatically extracts the FSM during compile. This option controls Design Compiler extraction. To automatically infer and extract an FSM, this variable must be `true`. Default is `false`.

FSM Coding Example

HDL Compiler infers an FSM for the design in Example 6-11. Figure 6-2 shows the state diagram for fsm1.

Example 6-11 Finite State Machine fsm1

```
module fsm1 (x, clk, rst, y);
  input x, clk, rst;
  output y;

  parameter [3:0]
  set0 = 4'b0001, hold0 = 4'b0010, set1 = 4'b0100, hold1 = 4'b1000;

  reg [3:0] current_state, next_state;

  always @ (posedge clk or posedge rst)
    if (rst)
        current_state <= set0;
    else
        current_state <= next_state;

  always @ (current_state or x)
    case (current_state)
        set0:
            next_state = hold0;
        hold0:
            if (x == 0)
                next_state = hold0;
            else
                next_state = set1;
        set1:
            next_state = hold1;
        hold1:
            next_state = set0;
        default :
            next_state = set0;
    endcase
  assign y = current_state == hold0 & x;
endmodule
```

Figure 6-2 State Diagram for fsm1

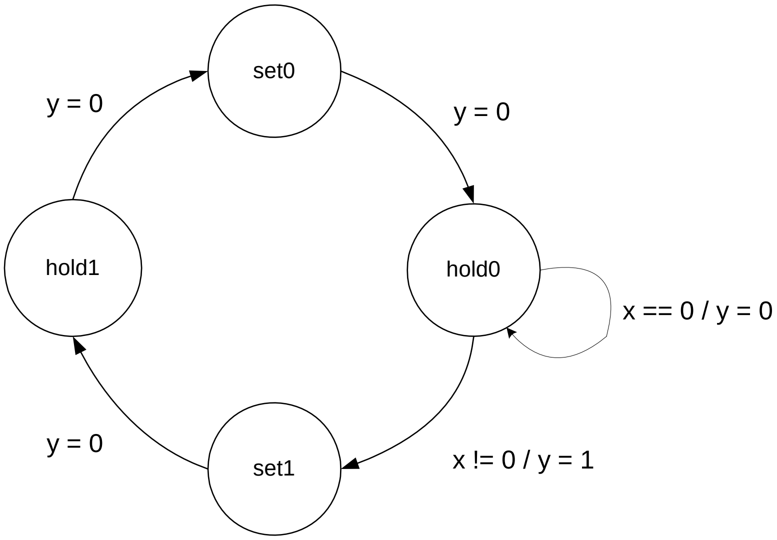


Table 6-4 shows the state table for fsm1.

Table 6-4 State Table for fsm1

Current state	Input (x)	Next state	Output (y)
0001 (set0)	0	0010 (hold0)	0
0001 (set0)	1	0010 (hold0)	0
0010 (hold0)	0	0010 (hold0)	0
0010 (hold0)	1	0100 (set1)	1
0100 (set1)	0	1000 (hold1)	0
0100 (set1)	1	1000 (hold1)	0
1000 (hold1)	0	0001 (set0)	0
1000 (hold1)	1	0001 (set0)	0

FSM Inference Report

HDL Compiler creates a finite state machine inference report when you set hdlin_reporting_level to comprehensive. The default is basic, meaning that an FSM inference report is not generated. For more information about the hdlin_reporting_level variable, see Customizing Elaboration Reports.

Consider the code in Example 6-12.

Example 6-12 FSM Code

```
module fsm (clk, rst, y);
  input clk, rst;
  output y;
  parameter [2:0]
  red = 3'b001, green = 3'b010, yellow = 3'b100;
  reg [2:0] current_state, next_state;

  always @ (posedge clk or posedge rst)
    if (rst)
      current_state <= red;
    else
      current_state <= next_state;

  always @(*)
      case (current_state)
        yellow:
           next_state = red;
        green:
           next_state = yellow;
        red:
           next_state = green;
        default:
          next_state = red;
      endcase
   assign y = current_state == yellow;
endmodule // fsm
```

Example 6-13 shows the FSM inference report.

Example 6-13 FSM Inference Report

```
statistics for FSM inference:
  state register: current_state
  states
  ======
  fsm_state_0:100
  fsm_state_1:010
  fsm_state_2:001
  total number of states: 3
```

Enumerated Types

HDL Compiler simplifies equality comparisons and detection of full cases in designs that contain enumerated types. A variable has an enumerated type when it can take on only a subset of the values it could possibly represent. For example, if a 2-bit variable can be set to 0, 1, or 2 but is never assigned to 3, then it has the enumerated type {0, 1, 2}. Enumerated types commonly occur in finite state machine state encodings. When the number of states needed is not a power of 2, certain state values can never occur. In finite state machines with one-hot encodings, many values can never be assigned to the state vector. For example, for a vector of length *n*, there are *n* one-hot values and (2***n* - *n*) values will never be used.

HDL Compiler infers enumerated types automatically; user directives can be used in other situations. When all variable assignments are within a module, HDL Compiler usually detects if the variable has an enumerated type. If the variable is assigned a value that depends on an input port or if the design assigns individual bits of the variable separately, HDL Compiler requires the `/* synopsys enum */` directive in order to consider the variable as having an enumerated type.

When enumerated types are inferred, HDL Compiler generates a report similar to Example 6-14.

Example 6-14 Enumerated Type Report

```
=======================================================================
|         Symbol Name         |  Source  |    Type    |  # of values  |
=======================================================================
|        current_state        |   auto   |   onehot   |       4       |
=======================================================================
```

This report tells you the source of the enumerated type,

- user directive (lists users under Source)
- HDL Compiler inferred (lists auto under Source)

It also tells you the type of encoding—whether onehot or enumerated (`enum`). HDL Compiler recognizes a special case of enumerated types in which each possible value has a single bit set to 1 and all remaining bits set to zero. This special case allows additional optimization opportunities. An enumerated type that fits this pattern is described as onehot in the enumerated type report. All other enumerated types are described as `enum` in the report.

Example 6-15 is a combination of both FSM and enumerated type optimization. The first case statement infers an FSM; the second case statement uses enumerated type optimization. These are two independent processes.

Example 6-15 Design: my_design

```
module my_design (clk, rst, x, y);
  input clk, rst;
  input [5:0] x;
  output [5:0] y;

  parameter [5:0]
    zero = 6'b000001, one = 6'b000010, two = 6'b000100,
 three = 6'b001000, four = 6'b010000, five = 6'b100000;

  reg [5:0] tmp;
  reg [5:0] y;

always @ (posedge clk or posedge rst)
    if (rst)
         tmp <= zero;
    else
         case (x)
             one     : tmp = zero;
             two     : tmp = one;
             three   : tmp = two;
             four    : tmp = three;
             five    : tmp = four;
             default : tmp = five;
        endcase

  always @ (tmp)
    case (tmp)
        five    : y = 6'b100110;
        four    : y = 6'b010100;
        three   : y = 6'b001001;
        two     : y = 6'b010010;
        one     : y = 6'b111111;
        zero    : y = 6'b100100;
        default : y = 6'b110101;
    endcase

endmodule
```

7

Interfaces

A SystemVerilog interface construct is a named bundle of nets, variables, or both. To simplify bus specification and bus management, use interfaces to encapsulate communications between modules. In addition to connectivity management, you can use interfaces as communication protocol handlers by embedding tasks, functions, and `always` blocks in the interfaces for other modules to access.

For synthesis, an interface is an inline instantiation, so any wires or logic defined in an interface are created inside the module that instantiates the interface.

To learn how to use interfaces, see

- Elements of Interfaces
- Inputs to Interfaces
- Arrays of Interfaces
- Renaming Conventions
- Using Interfaces in HDL Compiler
- Synthesis Restrictions

Elements of Interfaces

Interfaces can contain the following elements:

- Wires
- Modports
- Modport Expression
- Function and Tasks
- always Blocks

Wires

Wires or variables that are defined inside an interface are synthesized as nets. These nets connect to all modules that include the interface in the module port lists by using an inout port, unless restricted by a modport.

Modports

Modports inside an interface are used by modules to restrict the signals from the interface to the modules and the directions of these signals. Follow these guidelines when you use modports:

- For synthesis, you should specify modports in both the module definition and port list during instantiation, like the sendmode modport in the following examples:
  - Module definition: `module sender (try_i.sendmode try, ...)`
  - Instantiation: `sender (t.sendmode, ...)`
- Module port names should match the interface signal names unless modport expressions are used.
- The tool supports the `input`, `output`, `inout`, and `import` keywords inside a modport.

If a signal is used in a design without going through a modport, the tool connects it to an inout port and issues an information message similar to the following:

```
Information: ./test.sv:29: Variables crossing hierarchy: interface
content '%s' might become connected to an inout port (VER-735)
```

Modport Expression

A modport expression is explicitly named with a port identifier that is visible only through the modport connection. It provides a consistent port naming scheme for blocks that use the interface while hiding some of the code complexity in the interface.

A modport expression allows the following elements in a modport list:

- Elements of arrays and structures
- Concatenations of elements
- Assignment pattern expressions

Function and Tasks

You define functions and tasks in interfaces following these guidelines:

- For synthesis, you must define all functions and tasks using the `automatic` keyword.
- To be used inside modules, the function or task must be provided through the modport by using the `import` keyword.
- Functions and tasks inside an interface have access to signals defined in the interface. You must include these signals in the modport.
- Logic created by a function or task call is created at the site of the call, that is, inside the module that uses the function or task.

always Blocks

Synthesis supports `always` blocks inside interfaces. The tool creates the logic for an `always` block in the module that instantiates the block, often at the top level.

The following examples show how to define these elements in interfaces:

- Example: Interface With Wires
- Example: Interface With Modports
- Example: Interface With Modport Expressions
- Example: Interface With Functions
- Example: Interface With Functions and Tasks
- Example: Interface With always Blocks

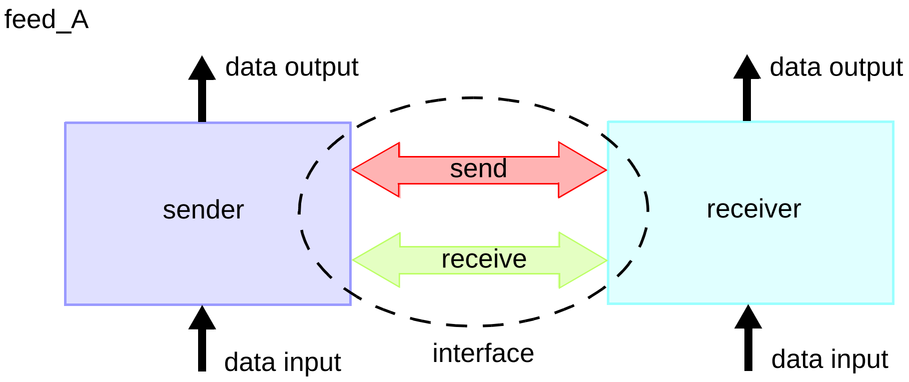
Example: Interface With Wires

The feed_A design uses an interface for the send and receive buses as shown in Figure 7-1. The interface consists of a bundle of bidirectional wires or nets. In this design,

- The sender transmits its input to the receiver through the interface; the receiver accepts the input through the interface and outputs this data.

- The receiver transmits its input to the sender through the interface; the sender accepts the input through the interface and outputs this data.

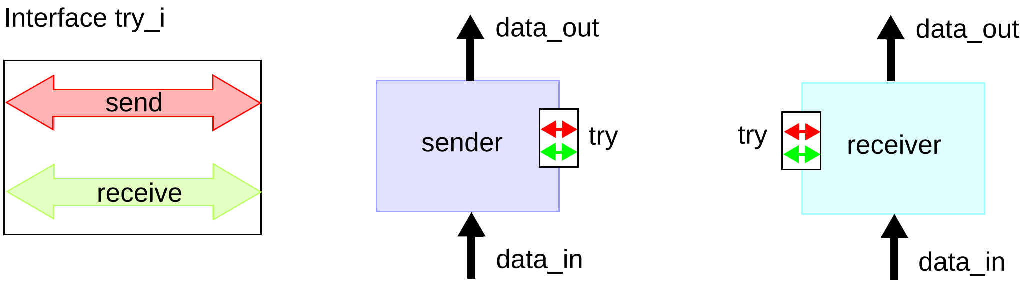
Figure 7-1 Design feed_A



Interface With Wires Only

The following figure shows the feed_A design connects two modules using a basic interface with wires only. This coding style is not recommended for synthesis.

Figure 7-2 Interface With Wires Only



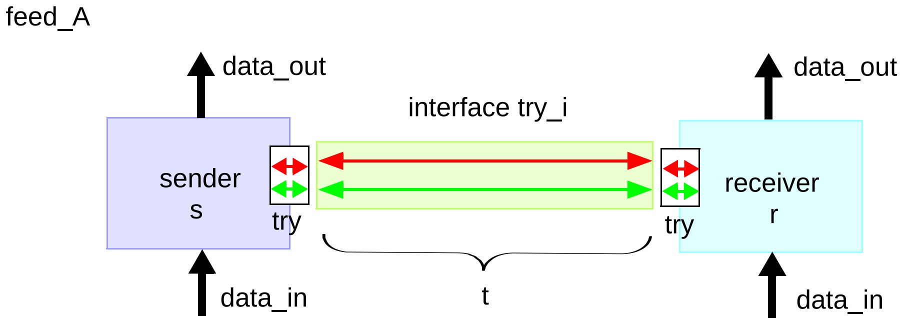
This example shows the RTL for the interface with wires only.

```
// interface definition
interface try_i;
wire  [7:0] send, receive;
endinterface : try_i

// sender module definition
module (
   try_i  try,
   input logic [7:0] data_in,
   output logic [7:0] data_out
);
assign  data_out = try.receive;
assign try.send = data_in;
endmodule

// receiver module definition
module receiver (
   try_i  try,
   input logic [7:0] data_in,
   output logic [7:0] data_out
);
assign data_out = try.send;
assign try.receive = data_in;
endmodule

// top design definition
module feed_A (
   input wire [7:0] di1, di2,
   output wire [7:0] do1, do2
);
try_i    t();
sender   s(t, di1, do1);
receiver r(t, di2, do2);
endmodule
```

Block Diagram of the feed_A Design

The following block diagram shows that the feed_A design contains the t and try instances of the try_i interface. All signals in the t and try instances are bidirectional. The feed_A design is called a basic interface because it does not contain modports. For basic interfaces, the tool assigns the `inout` port to wires and the `ref` port to variables by default.

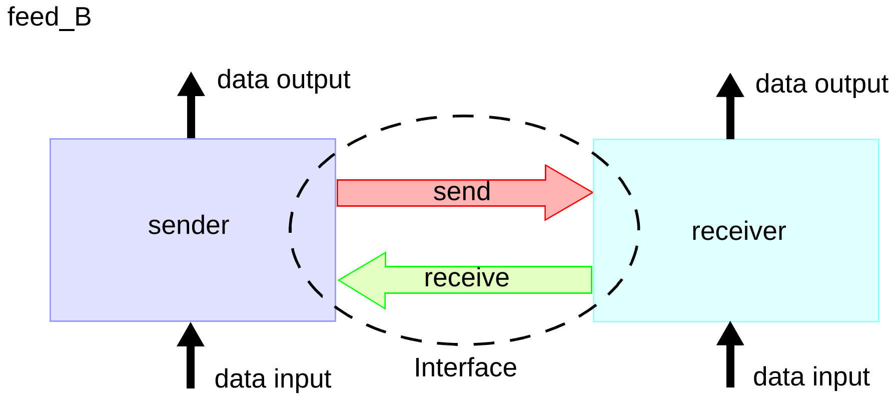
Figure 7-3 Block Diagram of the feed_A Design



Example: Interface With Modports

The following figure shows a design of an interface with modports. The feed_B design includes the functions of the feed_A design described in Example: Interface With Wires and modports with directions defined in the interface.

Figure 7-4 Design feed_B: Interface With Modports



The following sections describe the subdesigns and complete RTL of the feed_B design:

- The try_i Interface
- The sender Module
- The receiver Module
- Block Diagram of the feed_B Design
- Complete RTL of the feed_B Design

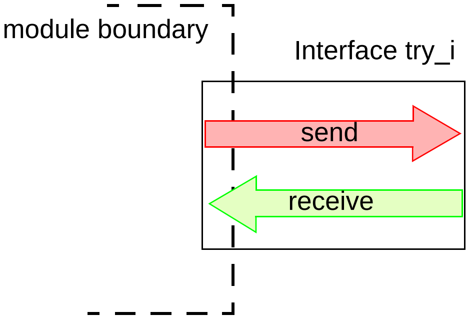
The try_i Interface

The following code shows the definition of the try_i interface, which contains the sendmode and receivemode modports. When the interface is instantiated in modules, you can use these modports to specify the signal directions.

```
interface try_i;
logic [7:0] send;
logic [7:0] receive;
modport sendmode (output  send, input receive);
modport receivemode (input  send, output receive);
endinterface
```

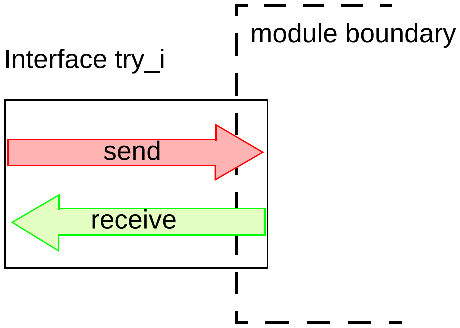
The following figure shows the block diagram of the sendmode modport in the try_i interface used by a module. The send bus is the output from the module, and the receive bus is the input to the module.

Figure 7-5 The try_i Interface With the sendmode Modport



The following figure shows the block diagram of the receivemode modport in the try_i interface used by a module. The send bus is the input to the module, and the receive bus is the output from the module.

Figure 7-6 try_i Interface With the receivemode Modport



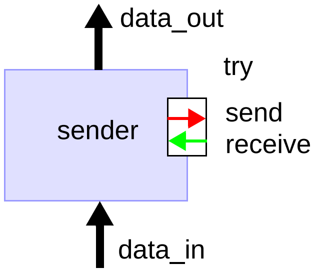
The sender Module

The following code shows that the sender module uses the sendmode modport of the try_i interface as a port named try. In the sendmode modport, the send bus is an output, and the receive bus is an input.

```
module sender (
   try_i.sendmode  try,
   input logic [7:0] data_in,
   output logic [7:0] data_out
);
assign data_out = try.receive;
assign try.send = data_in;
endmodule
```

This figure shows the block diagram of the sender module, which uses the sendmode modport as a port.

Figure 7-7 The sender Module With the sendmode Modport



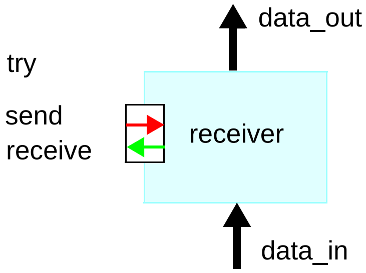
The receiver Module

The following code shows that the receiver module uses the receivemode modport of the try_i interface as a port named try. In the receivemode modport, the receive bus is an output, and the send bus is an input.

```
module receiver (
   try_i.receivemode  try,
   input logic [7:0] data_in,
   output logic [7:0] data_out
);
assign data_out = try.send;
assign try.receive = data_in;
endmodule
```

This figure shows the block diagram of the receiver module, which uses the receivemode modport as a port.

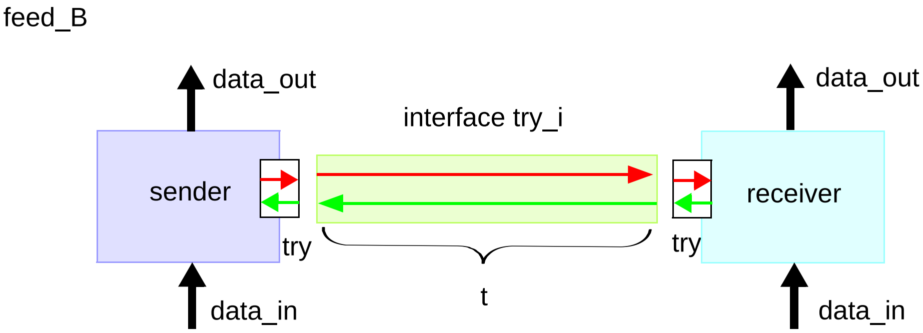
Figure 7-8 The receiver Module With the receivemode Modport



Block Diagram of the feed_B Design

The following block diagram shows that the top-level feed_B design contains the t instance of the try_i interface. The receiver module uses the receivemode modport of the interface as a port named try. The sender module uses the sendmode modport of the interface as a port named try.

Figure 7-9 Block Diagram of the feed_B Design



Complete RTL of the feed_B Design

The following code shows that the top-level feed_B design instantiates the try_i interface, sender module, and receiver module with the t, s, and r names respectively.

Example 7-1 Complete RTL of the feed_B Design

```
interface try_i;
logic [7:0] send;
logic [7:0] receive;
modport sendmode (output  send, input receive);
modport receivemode (input  send, output receive);
endinterface

module sender (
   try_i.sendmode try,
   input logic [7:0] data_in,
   output logic [7:0] data_out
);
assign data_out = try.receive;
assign try.send = data_in;
endmodule

module receiver (
   try_i.receivemode try,
   input logic [7:0] data_in,
   output logic [7:0] data_out
);
assign data_out = try.send;
assign try.receive = data_in;
endmodule

module feed_B (
   input wire [7:0] di1, di2,
   output wire [7:0] do1, do2
);
try_i t();
sender s (t.sendmode, di1, do1);
receiver r (t.receivemode, di2, do2);
endmodule
```

Example: Interface With Modport Expressions

In the following example, the myIf interface

- Uses modport expressions to rename the clk1 and clk2 clocks to a consistent port named clk.
- Distributes the 8-bit a logic to two modports, a[3:0] through modport consumer1 and a[4:7] through modport consumer2.
- Reassembles the 8-bit b output logic, b[3:0] through modport consumer1 and b[4:7] through modport consumer2.

Example 7-2 Interface With Modport Expressions

```
interface myIf (
   input logic clk1, clk2
);
logic [7:0] a, b;

modport consumer1 (input .clk(clk1), .din(a[7:4]), output .dout(b[7:4]));
modport consumer2 (input .clk(clk2), .din(a[3:0]), output .dout(b[3:0]));
endinterface

module top (
   input logic clk1, clk2,
   input logic [7:0] din,
   output logic [7:0] dout
);

myIf i1 (.clk1, .clk2);
regBlock rb1(i1.consumer1);
regBlock rb2(i1.consumer2);

assign i1.a = din;
assign dout = i1.b;
endmodule
```

Including the signal complexity in the myIf interface makes the regBlock specification simple. As shown in the following code, the regBlock module accesses the elements in the interface using the renamed ports:

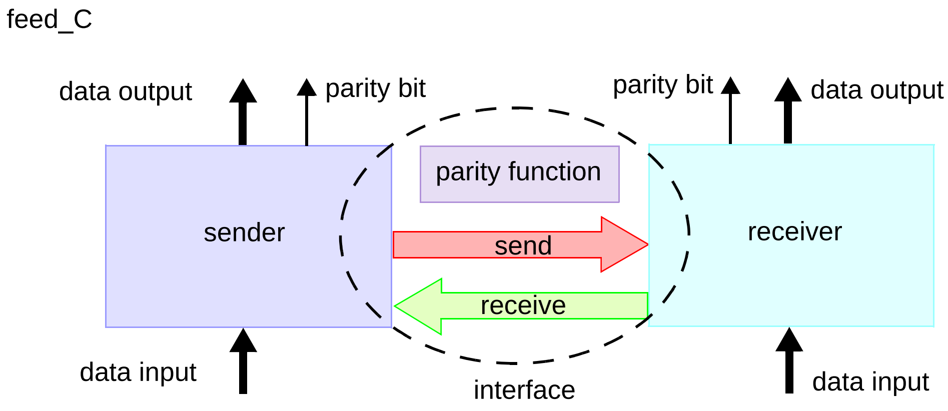
```
module regBlock (myIf iPort);
always @(posedge iPort.clk)
   iPort.dout <= iPort.din;
endmodule
```

Example: Interface With Functions

An interface can be a placeholder for functions that are needed by modules. To enable modules to access the functions in the interface, use modports and the `import` keyword. The hardware implementations are created only in the module that calls the functions.

The following figure shows that the feed_C design contains the functions of the feed_B design described in Example: Interface With Modports and the parity function in the interface.

Figure 7-10 Design feed_C: Interface With Modports and a Function



Complete RTL of the feed_C Design

As shown in Example 7-3, both the sender and receiver modules in the feed_C design need the parity function to calculate the parity values of the send and receive buses. You place the parity function in the interface and import the parity function from the try_i interface by using the sendmode and receivemode modports. This function becomes available to the sender and receiver modules through the sendmode and receivemode modports respectively. The hardware implementations of the parity function are created in the sender and receiver modules.

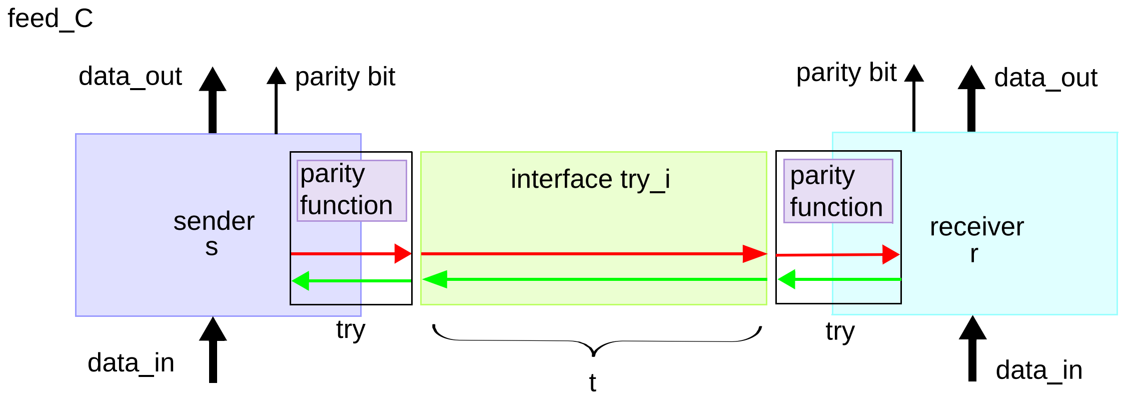
Example 7-3 Design feed_C: Interface With Modports and a Function

```
interface try_i;
logic [7:0] send;
logic [7:0] receive;
logic internal;

function automatic logic parity (logic [7:0] data);
return(^data);
endfunction
modport sendmode (output send, input receive, import parity );
modport receivemode (input send, output receive, import parity);
endinterface

module sender (
   try_i.sendmode  try,
   input logic [7:0] data_in,
   output logic [7:0] data_out, logic data_out_parity
     );
     assign data_out = try.receive;
     assign data_out_parity = try.parity(try.receive);
assign try.send = data_in;
endmodule

module receiver (
   try_i.receivemode  try,
   input logic [7:0] data_in,
   output logic [7:0] data_out, logic data_out_parity
);
assign data_out = try.send;
     assign data_out_parity =  try.parity(try.send);
assign try.receive = data_in;
endmodule

module feed_C (
   input wire [7:0] di1, di2,
   output wire [7:0] do1, do2, logic p1, p2
);
try_i t();
sender s (t.sendmode, di1, do1, p1);
receiver r (t.receivemode, di2, do2, p2);
endmodule
```

Block Diagram of the feed_C Design

The following figure shows the interface block diagram of the feed_C design, which contains the t instance of the try_i interface. The receiver module uses the receivemode modport of the interface as the try port. The sender module uses the sendmode modport of the interface as the try port. Both the sender and receiver modules import the parity function.

Figure 7-11 Block Diagram of the feed_C Design

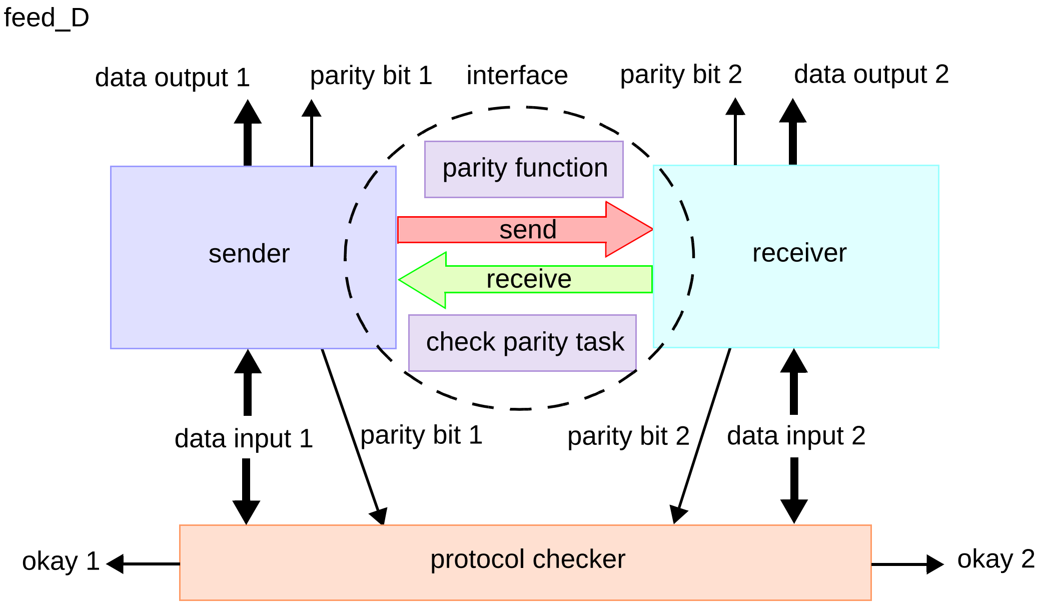


Example: Interface With Functions and Tasks

An interface can be a placeholder for tasks that are needed by modules. To enable modules to access the tasks in the interface, use modports and the `import` keyword. The hardware implementations are created only in the modules that call the tasks.

The following figure shows that the feed_D design contains the functions of the feed_C design described in Example: Interface With Modport Expressions and a task to check the parity.

Figure 7-12 Design feed_D: Interface With Modports, a Function, and a Task

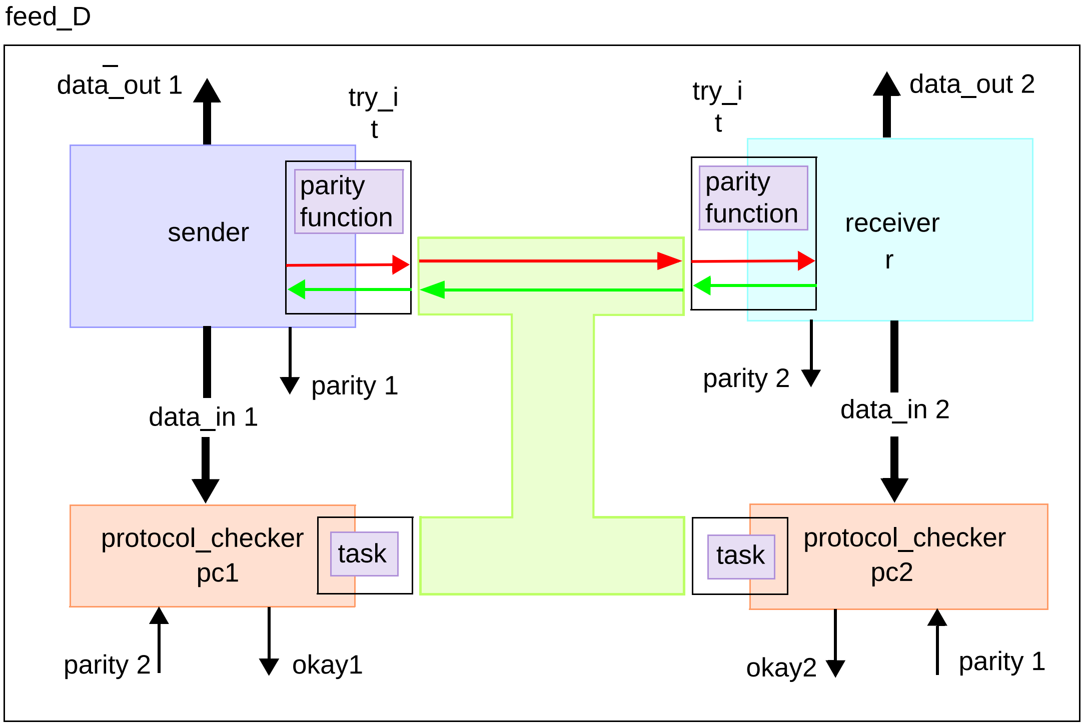


Block Diagram of the feed_D Design

As shown in Figure 7-13, the feed_D design contains the following three modports:

- The sendmode modport imports the parity function and defines the signal directions of the send and receive buses. The sender module uses this modport.
- The receivemode modport imports the parity function and defines the signal directions of the send and receive buses. The receiver module uses this modport.
- The protocol_checkermode modport imports the parity_check task. The pc1 and pc2 instantiations of the protocol_checker module use this modport.

Figure 7-13 Block Diagram of the feed_D Design



Complete RTL of the feed_D Design

This example uses modports, a function, and a task to create an interface for the send and receive data buses. In the feed_D design, the protocol_checker module contains the parity_check task that checks the parity. The hardware implementations of the task are created only in the pc1 and pc2 modules, which call the task using the modports.

Example 7-4 Design feed_D: An Interface With Modports, a Function, and a Task

```
interface try_i;
logic [7:0] send;
logic [7:0] receive;
function automatic logic parity ([7:0] data);
return(^data);
endfunction

task automatic parity_check (
   input logic [7:0] data_sent, logic exp_parity,
   output logic okay
);
if (exp_parity == ^data_sent)
   okay = '1;
else
   okay = '0;
endtask
```

```
modport sendmode (output send, input receive, import parity );
modport receivemode (input send, output receive, import parity );
modport protocol_checkermode (import parity_check  );
endinterface

module sender (
   try_i.sendmode try,
   input logic [7:0] data_in,
   output logic [7:0] data_out, logic data_out_parity
);
assign data_out = try.receive;
assign data_out_parity = try.parity(try.receive);
assign try.send = data_in;
endmodule

module receiver(
   try_i.receivemode try,
   input logic [7:0] data_in,
   output logic [7:0] data_out, logic data_out_parity
);
assign data_out = try.send;
assign data_out_parity = try.parity(try.send);
assign try.receive = data_in;
endmodule

module protocol_checker (
   input logic [7:0] data_sent, logic exp_parity,
   output logic okay,
   try_i.protocol_checkermode try
);
always @ (data_sent)
try.parity_check (data_sent, exp_parity, okay);
endmodule

module feed_D (
   input wire [7:0] di1, di2,
   output wire [7:0] do1, do2, logic p1, p2, okay1, okay2
);
try_i t();
sender s (t.sendmode, di1, do1, p1);
receiver r (t.receivemode, di2, do2, p2);
protocol_checker pc1(di1, p2, okay1, t.protocol_checkermode);
protocol_checker pc2(di2, p1, okay2, t.protocol_checkermode);
endmodule
```

GTECH Netlist

This GTECH netlist shows that the tool creates the hardware implementations of the task or function only in the modules that call the task or function.

Example 7-5 GTECH Netlist

```
module sender_I_try_try_i_sendmode_ ( \try.send , \try.receive , data_in,
        data_out, data_out_parity );
  output [7:0] \try.send ;
  input [7:0] \try.receive ;
  input [7:0] data_in;
  output [7:0] data_out;
  output data_out_parity;
  wire   N0, N1, N2, N3, N4, N5;
  assign \try.send  [7] = data_in[7];
  assign \try.send  [6] = data_in[6];
  assign \try.send  [5] = data_in[5];
  assign \try.send  [4] = data_in[4];
  assign \try.send  [3] = data_in[3];
  assign \try.send  [2] = data_in[2];
  assign \try.send  [1] = data_in[1];
  assign \try.send  [0] = data_in[0];
  assign data_out[7] = \try.receive  [7];
  assign data_out[6] = \try.receive  [6];
  assign data_out[5] = \try.receive  [5];
  assign data_out[4] = \try.receive  [4];
  assign data_out[3] = \try.receive  [3];
  assign data_out[2] = \try.receive  [2];
  assign data_out[1] = \try.receive  [1];
  assign data_out[0] = \try.receive  [0];

  GTECH_XOR2 C7 ( .A(N5), .B(data_out[0]), .Z(data_out_parity) );
  GTECH_XOR2 C8 ( .A(N4), .B(data_out[1]), .Z(N5) );
  GTECH_XOR2 C9 ( .A(N3), .B(data_out[2]), .Z(N4) );
  GTECH_XOR2 C10 ( .A(N2), .B(data_out[3]), .Z(N3) );
  GTECH_XOR2 C11 ( .A(N1), .B(data_out[4]), .Z(N2) );
  GTECH_XOR2 C12 ( .A(N0), .B(data_out[5]), .Z(N1) );
  GTECH_XOR2 C13 ( .A(data_out[7]), .B(data_out[6]), .Z(N0) );
endmodule

module receiver_I_try_try_i_receivemode_ ( \try.send , \try.receive ,
        data_in, data_out, data_out_parity );
  input [7:0] \try.send ;
  output [7:0] \try.receive ;
  input [7:0] data_in;
  output [7:0] data_out;
  output data_out_parity;
  wire   N0, N1, N2, N3, N4, N5;
  assign \try.receive  [7] = data_in[7];
  assign \try.receive  [6] = data_in[6];
  assign \try.receive  [5] = data_in[5];
  assign \try.receive  [4] = data_in[4];
  assign \try.receive  [3] = data_in[3];
  assign \try.receive  [2] = data_in[2];
  assign \try.receive  [1] = data_in[1];
  assign \try.receive  [0] = data_in[0];
  assign data_out[7] = \try.send  [7];
  assign data_out[6] = \try.send  [6];
  assign data_out[5] = \try.send  [5];
  assign data_out[4] = \try.send  [4];
  assign data_out[3] = \try.send  [3];
  assign data_out[2] = \try.send  [2];
  assign data_out[1] = \try.send  [1];
  assign data_out[0] = \try.send  [0];

  GTECH_XOR2 C7 ( .A(N5), .B(data_out[0]), .Z(data_out_parity) );
  GTECH_XOR2 C8 ( .A(N4), .B(data_out[1]), .Z(N5) );
  GTECH_XOR2 C9 ( .A(N3), .B(data_out[2]), .Z(N4) );
  GTECH_XOR2 C10 ( .A(N2), .B(data_out[3]), .Z(N3) );
  GTECH_XOR2 C11 ( .A(N1), .B(data_out[4]), .Z(N2) );
  GTECH_XOR2 C12 ( .A(N0), .B(data_out[5]), .Z(N1) );
  GTECH_XOR2 C13 ( .A(data_out[7]), .B(data_out[6]), .Z(N0) );
endmodule

module protocol_checker_I_try_try_i_protocol_checkermode_ ( data_sent,
   exp_parity, okay );
  input [7:0] data_sent;
  input exp_parity;
  output okay;
  wire   N0, N1, N2, N3, N4, N5, N6, N7, N8;

  GTECH_XOR2 C5 ( .A(exp_parity), .B(N1), .Z(N0) );
  GTECH_NOT I_0 ( .A(N0), .Z(N2) );
  GTECH_XOR2 C14 ( .A(N8), .B(data_sent[0]), .Z(N1) );
  GTECH_XOR2 C15 ( .A(N7), .B(data_sent[1]), .Z(N8) );
  GTECH_XOR2 C16 ( .A(N6), .B(data_sent[2]), .Z(N7) );
  GTECH_XOR2 C17 ( .A(N5), .B(data_sent[3]), .Z(N6) );
  GTECH_XOR2 C18 ( .A(N4), .B(data_sent[4]), .Z(N5) );
  GTECH_XOR2 C19 ( .A(N3), .B(data_sent[5]), .Z(N4) );
  GTECH_XOR2 C20 ( .A(data_sent[7]), .B(data_sent[6]), .Z(N3) );
  GTECH_BUF B_0 ( .A(N2), .Z(okay) );
endmodule

module feed_D ( di1, di2, do1, do2, p1, p2, okay1, okay2 );
  input [7:0] di1;
  input [7:0] di2;
  output [7:0] do1;
  output [7:0] do2;
  output p1, p2, okay1, okay2;
  wire   [7:0] \t.send ;
  wire   [7:0] \t.receive ;

  sender_I_try_try_i_sendmode_ s ( .\try.send (\t.send ), .\try.receive (
  \t.receive ), .data_in(di1), .data_out(do1), .data_out_parity(p1) );
  receiver_I_try_try_i_receivemode_ r ( .\try.send (\t.send ),
  .\try.receive ( \t.receive ), .data_in(di2), .data_out(do2),
  .data_out_parity(p2) );
  protocol_checker_I_try_try_i_protocol_checkermode_ pc1 (
  .data_sent(di1), .exp_parity(p2), .okay(okay1) );
  protocol_checker_I_try_try_i_protocol_checkermode_ pc2 (
  .data_sent(di2), .exp_parity(p1), .okay(okay2) );
endmodule
```

Example: Interface With always Blocks

The following example shows that the I interface contains a flip-flop:

```
interface I (input clk, rst, d, output logic q);
always_ff @(posedge clk, negedge rst)
if (!rst)
   q <= 0;
else
   q <= d;
endinterface
module top (
   input clock, reset, data_in,
   output q_out
);
I inst1(clock, reset, data_in, q_out);
endmodule
```

When the I interface is instantiated in the top module, the tool creates a D flip-flop as shown in the following inference report:

```
=========================================================================
| Register Name |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
=========================================================================
|  inst1.q_reg  | Flip-flop |   1   |  N  | N  | Y  | N  | N  | N  | N  |
=========================================================================
```

Inputs to Interfaces

Interfaces can have ports and parameters as inputs.

- Ports

  An interface can have input and output ports. Only the signals declared in the port list of the interface can connect to modules or be used in the functions, tasks, and `always` blocks. To connect external nets or variables to an interface, use the interface ports that in turn connect the external signals to the lower-level modules.

- Parameters

  Elaboration time constants can be passed into interfaces using parameters. The way to define and use parameters in interfaces is identical to that of modules. If the elaborated module refers to interfaces with parameters that are not instantiated in the design, the parameter information needs to come from an external source. For more details, see Bottom-Up Hierarchical Elaboration.

The following examples show how to provide inputs and parameters for interfaces:

- Ports in Interfaces Example
- Parameterized Interfaces Example

Ports in Interfaces Example

When external signals are declared in interface ports, you can connect these signals to the modules that instantiate the interface. As shown in Figure 7-14, the ALU design uses an interface to connect all the top-level inputs to the subdesigns.

The ALU design contains the following interface ports:

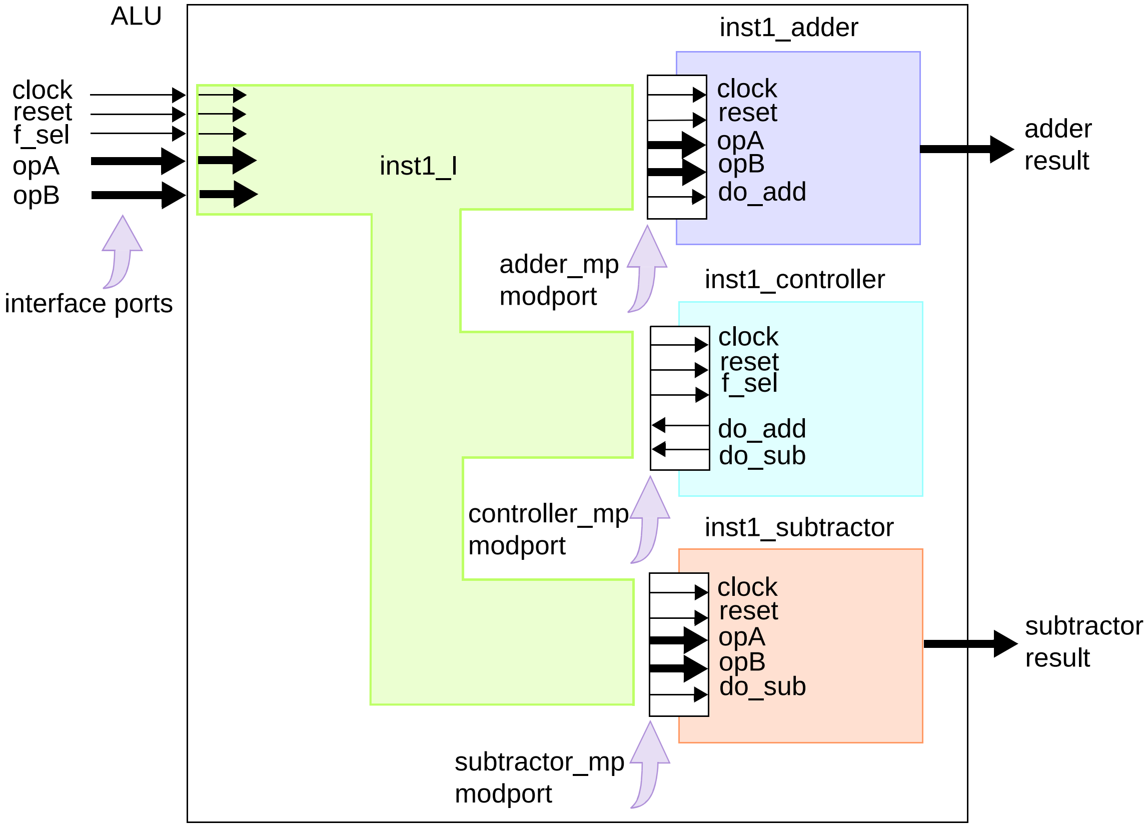
- Input ports: clock, reset, f_sel, opA, and opB
- Output ports: adder_result and subtractor_result

The I interface contains the following elements:

- Modports: adder_mp, subtractor_mp, and controller_mp
- Local signals (not interface ports): do_add and do_sub

The interface ports connect the top-level signals to the interface, whereas the modports only allow intermodule communications. The do_add and do_sub signals are declared inside the I interface. The adder, subtractor, and controller modules use the adder_mp, subtractor_mp, and controller_mp modports of the interface respectively.

Figure 7-14 Design ALU: Creating Ports in an Interface



Complete RTL of the ALU Design

Top-level signals are shared through the interface ports. To create the interface ports, include the clock, reset, f_sel, opA, and opB signals in the port list of the interface as follows:

```
interface I (input logic clock, reset, f_sel, logic [7:0] opA, opB);
```

The adder module uses the clock, reset, opA, and opB global signals and the do_add local signal, which are specified in the adder_mp modport as follows:

```
modport adder_mp (input clock, reset, do_add, opA, opB);
```

The subtractor module uses the clock, reset, opA, and opB global signals and the do_sub local signal, which are specified in the subtractor_mp modport as follows:

```
modport subtractor_mp (input clock, reset, do_sub, opA, opB);
```

The controller module uses the clock, reset, and f_sel global signals and the do_add and do_sub local signals, which are specified in the controller_mp modport as follows:

```
modport controller_mp (input clock, reset, f_sel, output do_add, do_sub);
```

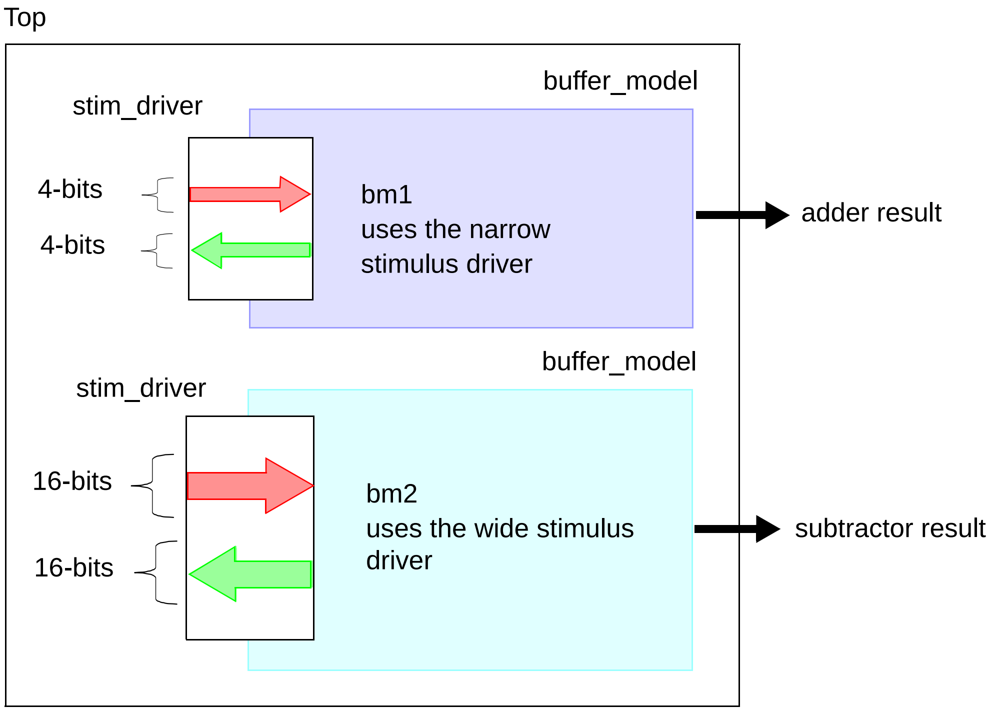
Example 7-6 Complete RTL of the ALU Design

```
interface I (
   input logic clock, reset, f_sel,
   logic [7:0] opA, opB
);
logic do_add;
logic do_sub;
modport adder_mp (input clock, reset, do_add, opA, opB);
modport subtractor_mp (input clock, reset, do_sub, opA, opB);
modport controller_mp (input clock, reset, f_sel, output do_add, do_sub);
endinterface

module adder (
   I.adder_mp adder_signals,
   output logic [7:0] sum
);
always_ff @(posedge adder_signals.clock, negedge adder_signals.reset)
if (!adder_signals.reset)
   sum <= '0;
else if (adder_signals.do_add)
   sum <= adder_signals.opA +adder_signals.opB;
endmodule

module subtractor (
   I.subtractor_mp sub_signals,
   output logic [7:0] difference
);
always_ff @(posedge sub_signals.clock, negedge sub_signals.reset)
if (!sub_signals.reset)
   difference <= '0;
else if (sub_signals.do_sub)
   difference <= sub_signals.opA + sub_signals.opB;
endmodule : subtractor

module controller (I.controller_mp controller_signals);
always_ff @(posedge controller_signals.clock, negedge
controller_signals.reset)
begin
   if (!controller_signals.reset)
   begin
      controller_signals.do_add <= '0;
      controller_signals.do_sub <= '0;
   end
   else if (~controller_signals.f_sel) //decode logic
   begin
      controller_signals.do_add <= '1;
      controller_signals.do_sub <= '0;
   end
   else if (controller_signals.f_sel)
   begin
      controller_signals.do_add <= '0;
      controller_signals.do_sub <= '1;
   end
end
endmodule

module alu (
   input clock, reset, f_sel, [7:0] opA, opB,
   output [7:0] adder_result, subtractor_result);
);
I inst1_I(.clock, .reset, .f_sel, .opA, .opB);
adder inst1_adder(inst1_I.adder_mp, adder_result);
subtractor inst1_subtractor(inst1_I.subtractor_mp, subtractor_result);
controller inst1_controller(inst1_I.controller_mp);
endmodule
```

Parameterized Interfaces Example

An interface can be parameterized in the same way as a module. The parameters can be modified on each instantiation of the interface. This figure shows that the Top design contains a 4-bit stimulus driver and a 16-bit stimulus driver.

Figure 7-15 Parameterized Interface



The following example shows how to create interfaces of different sizes by using parameters. The top design contains two instances of the stim_driver interface. The parameter in interface narrow_stimulus_interface is set to 4, and all the buses are 4 bits wide. The parameter in interface wide_stimulus_interface is set to 16, and all the buses are 16 bits wide in the top module.

Example 7-7 Using a Parameterized Interface

```
interface stim_driver;
```

```
parameter BUS_WIDTH = 8;
logic [BUS_WIDTH-1:0] sig1, sig2;
function automatic logic parity_gen ([BUS_WIDTH-1:0] bus);
return( ^bus );
endfunction
modport buffer_side (input sig1,output sig2,import parity_gen);
endinterface

module buffer_model #(parameter DELAY =1)(
   stim_driver.buffer_side a,
   output logic par_rslt
);
always
begin
   a.sig2 = #DELAY ~a.sig1;
   par_rslt =  a.parity_gen(a.sig2);
end
endmodule

module top # (parameter WIDTH1 = 4, WIDTH2 = 16)(output logic pr1, pr2);
stim_driver # (WIDTH1)narrow_stimulus_interface();
stim_driver # (WIDTH2)wide_stimulus_interface();
buffer_model bm1 (narrow_stimulus_interface.buffer_side, pr1);
buffer_model bm2 (wide_stimulus_interface.buffer_side, pr2);
endmodule
```

Arrays of Interfaces

You can use arrays of interfaces in the bottom or top modules and connect them using the following methods:

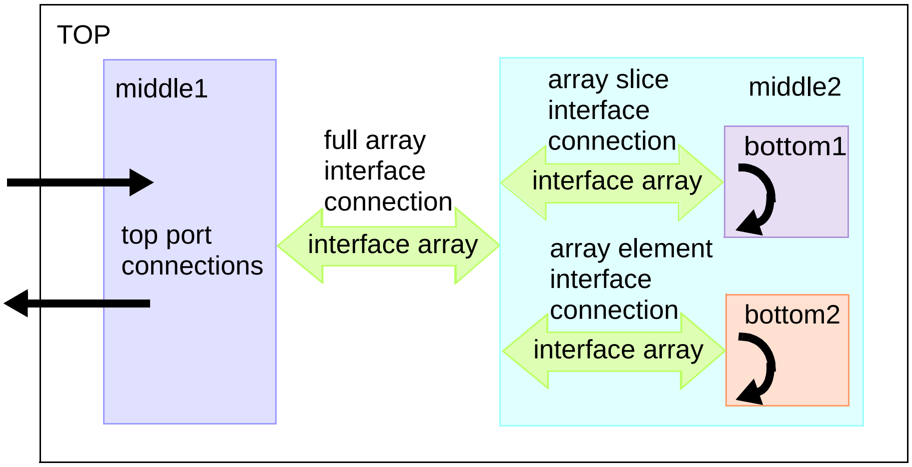
- Full array interface connection
- Array slice connection for interfaces with modports
- Array element connection for interfaces with modports

Follow these guidelines when you design arrays of interfaces:

- To avoid potential renumbering of array of interface ports during linking, define these arrays with a lower bound of 0 and in ascending order, for example, intfArr[0:n] or C-style intfArr[n+1].
- You cannot use the array slice operators (+:, &, and -:) with arrays of interfaces. If the array slice operators are used, the tool issues a VER-721 error message.

The following figure shows that the middle1 and middle2 blocks in the TOP design communicate with each other using a full array interface connection. In the middle2 block, the interface array is split into two portions, one slice of the array to communicate with the bottom1 block and one element of the array to communicate with the bottom2 block. The bottom1 and bottom2 blocks perform some processing of the signals of the interface array.

Figure 7-16 Arrays of Interfaces



Coding Styles for Interface Arrays

The following examples show the supported coding styles for the interface arrays shown in Figure 7-16:

Full Array Connection in the TOP Design

The TOP module uses the Inf interface array to communicate with the middle1 and middle2 modules by using a full array interface connection and modport specifications.

```
module top(
   input [0:2] x,
   output [0:2] y
);
Inf i1 [0:2] ();
// Full array interface connection to an interface with modports array
middle1 M1(i1.modA, x, y);
middle2 M2(i1.modB );
endmodule

// middle1 design with interface with modport array port pm1
module middle1(
   Inf.modA pm1[3],
   input [0:2] x,
   output [0:2] y
);
...
endmodule
...
// middle2 design with interface with modport array port pm2
module middle2(Inf.modB pm2[0:2]);
...
endmodule
```

Array Slice Connection to the middle2 Module, B1 Instance

In the middle2 module declaration, the B1 instantiation of the bottom1 module is connected using the pm2[0:1] syntax, which is a slice of the pm2[0:2] array connection for interfaces with modports.

```
module middle2(Inf.modB pm2[0:2]);
// Interface's array slice connection
bottom1 B1 (pm2[0:1]);
...
endmodule

module bottom1 (Inf.modB pb1[0:1]);
...
endmodule
```

Array Element Connection to the middle2 Module, B2 Instance

In the middle2 module declaration, the B2 instantiation of the bottom2 module is connected using the pm2[2] syntax, which is one element of the pm2[0:2] array connection for interfaces with modports.

```
module middle2(Inf.modB pm2[0:2]);
// Interface's array element connection
bottom2 B2 (pm2[2]);
...
endmodule

// bottom2 with non-array interface with modport port pb2
module bottom2 (Inf.modB pb2);
...
endmodule
```

Accessing the Modport Signals From Interfaces With Modport Arrays

The middle1 and bottom1 modules access the modport signals from array interfaces.

```
module bottom1 (Inf.modB pb1[0:1]);
assign pb1[1].b = ~pb1[1].a;
...
endmodule
```

Complete Implementation of the TOP Design

```
// Interface declarations
interface Inf ();
logic a, b;
// All signals are used on modports
modport modA (output a, input b);
modport modB (input a,  output b);
endinterface

// TOP design declaration
module TOP (
   input [0:2] x,
   output [0:2] y
);

// Array of interface instantiation
Inf i1 [0:2] ();
//Full interface array connection to an interface with modports array
middle1 M1(i1.modA, x, y);
middle2 M2(i1.modB );
endmodule

// middle1 design with interface with modport array port pm1
module middle1 (
   Inf.modA pm1[3],
   input [0:2] x,
   output [0:2] y
);
assign pm1[2].a = ~x[2];
assign y[0] = pm1[0].b;
  ...
endmodule
// middle2 design with interface with modport array port pm2
module middle2 (Inf.modB pm2[0:2]);
// bottom1 instantiation, connecting a slice of pm2
bottom1 B1 (pm2[0:1]);
// bottom1 instantiation, connecting one element of pm2
bottom2 B2 (pm2[2]);
endmodule

// bottom1 with interface with modport array port pb1
module bottom1 (Inf.modB pb1[0:1]);
// modport signal manipulation from and array interface with modports
assign pb1[1].b = ~pb1[1].a;
assign pb1[0].b = ~pb1[0].a;
endmodule

// bottom2 with non-array interface with modport port pb2
module bottom2 (Inf.modB pb2);
assign pb2.b = ~pb2.a;
endmodule
```

Coding Style Restrictions on Array Interfaces

When using array interfaces, you should avoid the following coding styles:

- The following example of array slice connections for interfaces with modports, i1.modA[0:1] and i1[0:1].modB, is not supported.

  ```
  module top();
     Inf i1[0:7];
     middle1 M1 (i1.modA[0:1]); // not supported
     middle2 M2 (i1[0:1].modB); // not supported
  endmodule
  ```

- The following example of array element connection for interfaces with modports, i1.modA[2], is not supported.

  ```
  module top();
     Inf i1[0:7];
     middle1 M1 (i1.modA[2]);   // not supported
  endmodule
  ```

  However, the following example of array element connection for interfaces with modports, i1[2].modB, is supported.

  ```
  module top();
     Inf i1[0:7];
     middle2 M2 (i1[2].modB);   // supported
  endmodule
  ```

- Access to slice of elements from an interface with modport arrays or full array is not supported. For example,

  ```
  // middle1 design with interface with modport array port pm1
  module middle1(Inf.modA pm1[3],  input [0:2] x, output [0:2] y);
     assign pm1[0:1].a = ~x;    // not supported
     assign y = pm1.b;          // not supported
  endmodule
  ```

To implement a bus fabric structure that encapsulates complex interconnection between modules, see Bus Fabric Design.

Renaming Conventions

Modules can contain parameters, interfaces, and interface modports as ports, and interfaces can use parameters. These various connecting methods affect the module names in the GTECH and gate-level netlist. To rename the modules, the tool uses the following format:

```
modulename_p1v1_p2v2...I_portname_interfacename_modportname_vi1_vi2...
```

This table describes the format in details.

Item	Description
modulename	Name of the module
p1	First parameter name inside the module
v1	Value of the first parameter
p2	Second parameter name inside the module
v2	Value of the second parameter
...	
I	Indicates an interface
portname	Name of the port inside the module that uses the interface as a port
interfacename	Name of the interface used in the module port
modportname	Name of the modport used with the interface as a module port
vi1	Value of the first parameter
vi2	Value of the second parameter
...	

To understand the renaming conventions, see

- Renamed Modules Example 1
- Renamed Modules Example 2
- Renamed Modules Example 3

Renamed Modules Example 1

In the following example, the feed_A design contains the sender and receiver modules. The GTECH netlists show the renamed sender and receiver modules.

Example 7-8 feed_A Design

```
interface try_i;
wire  [7:0] send, receive;
endinterface : try_i

module sender (
   try_i  try,
   input logic [7:0] data_in,
   output logic [7:0] data_out
);
assign  data_out = try.receive;
assign try.send = data_in;
endmodule

module receiver(
   try_i  try,
   input logic [7:0] data_in,
   output logic [7:0] data_out
);
assign data_out = try.send;
assign try.receive = data_in;
endmodule
module feed_A (
   input wire [7:0] di1, di2,
   output wire [7:0] do1, do2
);
try_i t();
sender s (t, di1, do1);
receiver r (t, di2, do2);
endmodule
```

The sender Module

The data for renaming the sender module is as follows:

```
modulename : sender
p1: none
v1:none
I : indicates interface
portname: try
interfacename: try_i
modportname : none
pi1: none
vi1: none
```

Based on this data, the tool renames the sender module to sender_I_try_try_i_. The following netlist shows a portion of the GTECH netlist of the renamed module.

```
module sender_I_try_try_i_ ( try_send, try_receive, data_in, data_out );
  inout [7:0] try_send;
  inout [7:0] try_receive;
  input [7:0] data_in;
...
```

The receiver Module

The data for renaming the receiver module is as follows:

```
modulename : receiver
p1: none
v1:none
I : indicates interface
portname: try
interfacename: try_i
modportname : none
pi1: none
vi1: none
```

Based on this data, the tool renames the receiver module to module receiver_I_try_try_i_. The following netlist shows a portion of the GTECH netlist of the renamed module.

```
module receiver_I_try_try_i_ (try_send, try_receive, data_in, data_out);
  inout [7:0] try_send;
  inout [7:0] try_receive;
  input [7:0] data_in;
...
```

See Also

- Example: Interface With Wires

Renamed Modules Example 2

In the following example, the feed_B design contains the sender and receiver modules. The GTECH netlist shows the renamed sender and receiver modules.

Example 7-9 feed_B Design

```
interface try_i;
logic [7:0] send;
logic [7:0] receive;
modport sendmode (output  send, input receive);
modport receivemode (input  send, output receive);
endinterface

module sender (
   try_i.sendmode try,
   input logic [7:0] data_in,
   output logic [7:0] data_out
);
assign data_out = try.receive;
assign try.send = data_in;
endmodule

module receiver (
   try_i.receivemode  try,
   input logic [7:0] data_in,
   output logic [7:0] data_out
);
assign data_out = try.send;
assign try.receive = data_in;
endmodule

module feed_B (
   input [7:0] di1, di2,
   output [7:0] do1, do2
);
try_i t();
sender s (t.sendmode, di1, do1);
receiver r (t.receivemode, di2, do2);
endmodule
```

The sender Module

The data for renaming the sender is as follows:

```
modulename : sender
p1: none
v1:none
I : indicates interface
portname: try
interfacename: try_i
modportname : sendmode
pi1: none
vi1: none
```

Based on this data, the tool renames the sender module to sender_I_try_try_i_sendmode_. The following netlist shows a portion of the GTECH netlist of the renamed module.

```
module sender_I_try_try_i_sendmode_ ( try_send, try_receive, data_in,
data_out );
  output [7:0] try_send;
  input  [7:0] try_receive;
  input  [7:0] data_in;
...
```

The receiver Module

The data for renaming the receiver is as follows:

```
modulename : receiver
p1: none
v1:none
I : indicates interface
portname: try
interfacename: try_i
modportname : receivemode
pi1: none
vi1: none
```

The tool renames the receiver module to receiver_I_try_try_i_receivemode_ based on this data. The following netlist shows a portion of the GTECH netlist of the renamed module.

```
module receiver_I_try_try_i_receivemode_ ( try_send, try_receive,
data_in, data_out );
  input  [7:0] try_send;
  output [7:0] try_receive;
  input  [7:0] data_in;
...
```

See Also

- Example: Interface With Modports

Renamed Modules Example 3

In the following example, the tool renames the bm1 and bm2 modules of the top design.

Example 7-10 RTL Design

```
interface stim_driver;
parameter BUS_WIDTH = 8;
logic [BUS_WIDTH-1:0] sig1, sig2;
function automatic logic parity_gen ([BUS_WIDTH-1:0] bus);
return( ^bus );
endfunction
modport buffer_side (input sig1,output sig2,import parity_gen);
endinterface

module buffer_model #(parameter DELAY =1)(
   stim_driver.buffer_side a,
   output logic par_rslt
);
always
begin
   a.sig2 = #DELAY ~a.sig1;
   par_rslt =  a.parity_gen(a.sig2);
end
endmodule

module top # (parameter WIDTH1 = 4, WIDTH2 = 16)(output logic pr1, pr2);
stim_driver #(WIDTH1)narrow_stimulus_interface();
stim_driver #(WIDTH2)wide_stimulus_interface();
buffer_model bm1(narrow_stimulus_interface.buffer_side, pr1);
buffer_model bm2(wide_stimulus_interface.buffer_side, pr2);
endmodule
```

The data for renaming the modules is as follows:

```
modulename : buffer_model
p1: DELAY
v1:1   (default value)
I : indicates interface
portname: a
interfacename: stim_driver
modportname : buffer_side
pi1: BUS_WIDTH
vi1: 4 (explicit modification)
modulename : buffer_model
p1: DELAY
v1:1   (default value)
I : indicates interface
portname: a
interfacename: stim_driver
modportname : buffer_side
pi1: BUS_WIDTH
vi1: 16 (explicit modification)
```

Based on this data, the tool renames the bm1 and bm2 modules to buffer_model_I_a_stim_driver_buffer_side_4 and buffer_model_I_a_stim_driver_buffer_side_16 respectively.

See Also

- Parameterized Interfaces Example

Using Interfaces in HDL Compiler

Analyzing Interfaces

Interfaces are modular standalone design elements in SystemVerilog and, as such, must be analyzed before elaborating the design. They are treated like modules in the `analyze` command line; there is no order dependency between analyzing the interface definitions and analyzing the modules that use them:

```
analyze -format sverilog {block.sv top.sv myIntf.sv}
```

Using Interfaces at the Top Level of Elaboration

In normal usage, interfaces must be instantiated in the design before they are passed to the port map of a lower module. To support hierarchical flows, HDL Compiler allows an exception to that rule and allows interfaces (and interface modports) to be specified on the top-level ports of a module with no corresponding instantiation. The interface still must be analyzed prior to elaboration.

```
interface myIntf;
    logic a,b,c;
    modport modA (input a, b, output c);
endinterface

module top (myIntf.modA  topPorts);
```

This allows easy encapsulation of top-level ports that are eventually referenced at a higher level. This also allows for design reuse or even for a communication channel to the testbench.

Note:
Generic interfaces and interfaces with nondefault parameters are not allowed at the top level of elaboration because there is not enough information available to resolve the interface. However, such modules can elaborated with the use of a context wrapper, described in “Running the Bottom-Up Hierarchical Elaboration Flow Using a Wrapper Design” on page 1-33.

Passing Parameter Overrides to Interfaces From the Elaborate Command

The `elaborate` command has a `-parameter` option to specify parameter overrides for the top-level module being elaborated. There is no equivalent setting for interfaces that are being used without instantiation. This type of interface usage is supported through the use of context wrappers, described in "Running the Bottom-Up Hierarchical Elaboration Flow Using a Wrapper Design" on page 1-33.

Synthesis Restrictions

The following synthesis restrictions apply when you use interfaces:

- Interfaces must contain only automatic tasks and functions. If you do not use the `automatic` keyword, the tool assumes a static function and issues an error message.

  The exceptions are as follows:

  - The interface variables that are used by the interface method are listed in the modport.
  - The interface instance is in the same module that calls the interface method.
- Exporting tasks and functions from one module into an interface is not supported for synthesis.
- External `fork` and `join` constructs in interfaces are not supported for synthesis.

8

Modeling Three-State Buffers

HDL Compiler infers a three-state driver when you assign the value z (high impedance) to a variable. HDL Compiler infers 1 three-state driver per variable per always block. You can assign high-impedance values to single-bit or bused variables. A three-state driver is represented as a TSGEN cell in the generic netlist. Three-state driver inference and instantiation are described in the following sections:

- Using z Values
- Three-State Driver Inference Report
- Assigning a Single Three-State Driver to a Single Variable
- Assigning Multiple Three-State Drivers to a Single Variable
- Registering Three-State Driver Data
- Instantiating Three-State Drivers
- Errors and Warnings

Using z Values

You can use the z value in the following ways:

- Variable assignment
- Function call argument
- Return value

You can use the z value only in a comparison expression, such as in

```
if (IN_VAL == 1'bz) y=0;
```

This statement is permissible because `IN_VAL == 1'bz` is a comparison. However, it always evaluates to false, so it is also a simulation/synthesis mismatch. See Unknowns and High Impedance in Comparison.

This code,

```
OUT_VAL = (1'bz && IN_VAL);
```

is not a comparison expression. HDL Compiler generates an error for this expression.

Three-State Driver Inference Report

The `hdlin_reporting_level` variable determines whether HDL Compiler generates a three-state inference report. If you do not want inference reports, set `hdlin_reporting_level` to `none`. The default is `basic`, which indicates to generate a report. Example 8-1 shows a three-state inference report:

Example 8-1 Three-State Inference Report

```
============================================
| Register Name |       Type       | Width |
============================================
|   T_tri       | Tri-State Buffer |   1   |
============================================
```

The first column of the report indicates the name of the inferred three-state device. The second column indicates the type of inferred device. The third column indicates the width of the inferred device. HDL Compiler generates the same report for the default and verbose reports for three-state inference. For more information about the `hdlin_reporting_level` variable, see Customizing Elaboration Reports.

Assigning a Single Three-State Driver to a Single Variable

Example 8-2 infers a single three-state driver and shows the associated inference report.

Example 8-2 Single Three-State Driver

```
module three_state (ENABLE, IN1, OUT1);
  input IN1, ENABLE;
  output OUT1;
  reg OUT1;
always @(ENABLE or IN1) begin
  if (ENABLE)
    OUT1 = IN1;
  else
    OUT1 = 1'bz;  //assigns high-impedance state
end
endmodule


Inference Report
============================================
| Register Name |       Type       | Width |
============================================
|   OUT1_tri    | Tri-State Buffer |   1   |
============================================
```

Example 8-3 infers a single three-state driver with MUXed inputs and shows the associated inference report.

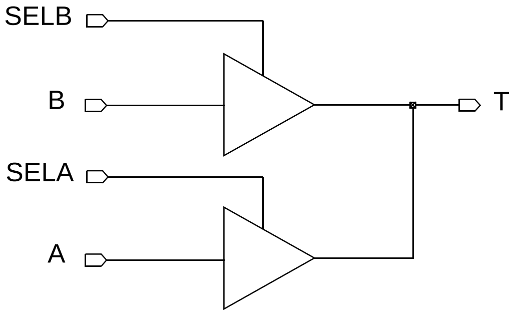
Example 8-3 Single Three-State Driver With MUXed Inputs

```
module three_state (A, B, SELA, SELB, T);
  input  A, B, SELA, SELB;
  output T;
  reg T;
  always @(SELA or SELB or A or B) begin
    T = 1'bz;
    if (SELA)
      T = A;
      if (SELB)
      T = B;
    end
endmodule


Inference Report
============================================
| Register Name |       Type       | Width |
============================================
|   T_tri       | Tri-State Buffer |   1   |
============================================
```

Assigning Multiple Three-State Drivers to a Single Variable

When assigning multiple three-state drivers to a single variable, as shown in Figure 8-1, always use assign statements, as shown in Example 8-4.

Figure 8-1 Two Three-State Drivers Assigned to a Single Variable



Example 8-4 Correct Method

```
module three_state (A, B, SELA, SELB, T);
 input A, B, SELA, SELB;
 output T;
 assign T = (SELA) ? A : 1'bz;
 assign T = (SELB) ? B : 1'bz;
endmodule
```

Do not use multiple always blocks (shown in Example 8-5). Multiple always blocks cause a simulation/synthesis mismatch because the reg data type is not resolved. Note that the tool does not display a warning for this mismatch.

Example 8-5 Incorrect Method

```
module three_state (A, B, SELA, SELB, T);
  input  A, B, SELA, SELB;
  output T;
  reg T;
  always @(SELA or A)
    if (SELA)
      T = A;
    else
      T = 1'bz;
  always @(SELB or B)
    if (SELB)
      T = B;
    else
      T = 1'bz;
endmodule
```

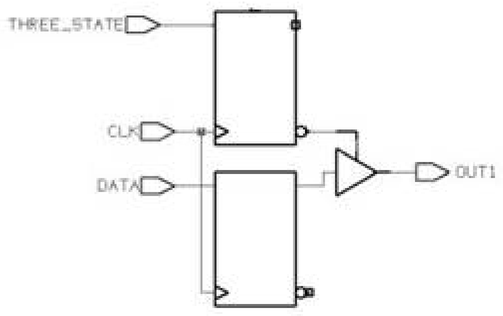
Registering Three-State Driver Data

When a variable is registered in the same block in which it is defined as a three-state driver, HDL Compiler also registers the driver’s enable signal, as shown in Example 8-6. Figure 8-2 shows the compiled gates and the associated inference report.

Example 8-6 Three-State Driver With Enable and Data Registered

```
module ff_3state (DATA, CLK, THREE_STATE, OUT1);
  input DATA, CLK, THREE_STATE;
  output OUT1;
  reg OUT1;
always @ (posedge CLK) begin
  if (THREE_STATE)
    OUT1 <= 1'bz;
  else
    OUT1 <= DATA;
end
endmodule

Inference reports

     =========================================================================
     |Register Name       |   Type   | Width | Bus  | AR | AS | SR | SS | ST |
     =========================================================================
     |OUT1_reg            |Flip-flop |   1   |  N   | N  | N  | N  | N   | N |
     |OUT1_tri_enable_reg |Flip-flop |   1   |  N   | N  | N  | N  | N   | N |
     =========================================================================

     ============================================
     | Register Name |       Type       | Width |
     ============================================
     |   OUT1_tri    | Tri-State Buffer |   1   |
     ============================================
```

Figure 8-2 Three-State Driver With Enable and Data Registered



Instantiating Three-State Drivers

The following gate types are supported:

- bufif0 (active-low enable line)
- bufif1 (active-high enable line)
- notif0 (active-low enable line, output inverted)
- notif1 (active-high enable line, output inverted)

Connection lists for bufif and notif gates use positional notation. Specify the order of the terminals as follows:

- The first terminal connects to the output of the gate.
- The second terminal connects to the input of the gate.
- The third terminal connects to the control line.

Example 8-7 shows a three-state gate instantiation with an active-high enable and no inverted output.

Example 8-7 Three-State Gate Instantiation

```
module three_state (in1,out1,cntrl1);
     input in1,cntrl1;
     output out1;

     bufif1 (out1,in1,cntrl1);
endmodule
```

Errors and Warnings

When you use the coding styles recommended in this chapter, you do not need to declare variables that drive multiply driven nets as tri data objects. But if you don’t use these coding styles, or you don’t declare the variable as a tri data object, HDL Compiler issues an ELAB-366 error message and terminates. To force HDL Compiler to warn for this condition (ELAB-365) but continue to create a netlist, set `hdlin_prohibit_nontri_multiple_drivers` to false (the default is true). With this variable false, HDL Compiler builds the generic netlist for all legal designs. If a design is illegal, such as when one of the drivers is a constant, HDL Compiler issues an error message.

The following code generates an ELAB-366 error message (OUT1 is a reg being driven by two always@ blocks):

```
 module three_state (ENABLE, IN1, RESET, OUT1);

 input IN1, ENABLE, RESET;
 output OUT1;
 reg OUT1;

always @(IN1 or ENABLE)
     if (ENABLE)
     OUT1 = IN1;

always@ (RESET)
    if (RESET)
    OUT1 = 1'b0;
endmodule
```

The ELAB-366 error message is

```
Error:  Net '/...v:14: OUT1' or a directly connected net is
driven by more than one source, and not all drivers are
three-state. (ELAB-366)
```

9

Other SystemVerilog Features

This section provides examples of supported SystemVerilog features.

- Variables
- The foreach Loop
- Functions and Tasks
- Binding Function and Task Arguments by Name
- Parameterized Functions and Tasks Using Virtual Classes
- Parameterized Data Types
- Bit-Level Support for Compiler Directives
- Structures
- Unions
- Multidimensional Arrays
- Configurations
- Implicit Port Connections
- Casting
- Macro Expansion and Parameter Substitution

- `begin_keywords and `end_keywords
- Predefined SYSTEMVERILOG Macro
- Matching Block Names
- Port Renaming
- Generic Wire Type
- General Verilog Coding Guidelines
- Guidelines for Interacting With Other Flows

Variables

In Verilog, you need to use different variables for parallel `for` loops. If loops in two or more parallel processes use the same control variable, there is a possibility that one loop is modifying the variable that other loops are still using. However, SystemVerilog allows you to use the same variable for multiple loops because a block creates a new hierarchical scope making the variable local to the loop scope.

Verilog for Loop Example

This example uses the j and k variables for the two `for` loops.

```
module varloop1 (
   input clk,
   input [3:0] in,
   output reg [3:0] out
);
integer j;
integer k;
reg [3:0] tmp;
always @(posedge clk) begin
   for(j=0;j<4;j=j+1) begin
   tmp[j] = !in[j];
end
end
always @(posedge clk) begin
   for(k=0;k<4;k=k+1) begin
      out[k] <= tmp[k];
   end
end
endmodule
```

SystemVerilog for Loop Example

This example uses only the j variable for the two `for` loops.

```
module varloop (
   input clk,
   input [7:0] in,
   output logic [3:0] out
):
logic [7:0] tmp;
always_ff @(posedge clk) begin
   for(int j=0;j<8;j=j+2) begin
      tmp[j]   <= !in[j];
      tmp[j+1] <= in[j+1];
   end
end
always_ff @(posedge clk) begin
   for(int j=0;j<4;j++) begin
      out[j] <= tmp[j];
   end
end
endmodule
```

Automatic Variable Initialization

By default, the tool initializes automatic variables to zero. This initialization applies to both two-state and four-state variables.

The foreach Loop

SystemVerilog provides the `foreach` construct for iterating over the elements of arrays. The iterators have a local scope and automatically match the type and range of the array bounds, so you do not need to hard-code the array bounds.

The following examples contrast the two different coding styles between the `for` loop and the `foreach` loop:

- The `for` loop

```
module for_loop (
   input [15:0] h_pixel,
   input [31:0] v_pixel,
   output logic [15:0][31:0] hv_xor_pixel
);

always_comb
for (int i=15; i>=0; i--) begin
   for (int j=31; j>=0; j--) begin
      hv_xor_pixel[i][j] = h_pixel[i] ^ v_pixel[j];
   end
end
endmodule
```

- The `foreach` loop

```
module foreach_loop (
   input [15:0] h_pixel,
   input [31:0] v_pixel,
   output logic [15:0][31:0] hv_xor_pixel
);

always_comb
foreach (hv_xor_pixel[i,j]) begin
   hv_xor_pixel[i][j] = h_pixel[i] ^ v_pixel[j];
end
endmodule
```

Functions and Tasks

This topic contains examples that use various function and task features:

- Function Before or Within a Module
- The logic Type
- The longint Type
- User-Defined Structure
- Output Argument and a Return Value
- SystemVerilog for Loop
- Sensitivity List Within a Function
- Memory Elements Outside a Function

Within a function block,

- If the `always`, `always_comb`, `always_latch`, or `always_ff` keyword is used, the tool issues an error message.
- Delay elements are ignored with a warning message.
- Combinational logic is allowed.

Function Before or Within a Module

You can place a function before or within a module, but not after a module. Placing a function after a module causes an error because the tool cannot see the function during the `analyze` step.

- Function before a module

```
typedef logic [3:0] ar4;
function automatic ar4 swap (
   input [3:0] value, switch, pack_hdr, vlan, status,
);
begin: myFunc
   unique case(1'b1)
   status[0]: swap = value;
   status[1]: swap = switch;
   status[2]: swap = pack_hdr;
   status[3]: swap = vlan;
   endcase
end
endfunction

module do_auto16_sv (
   input [3:0] value, switch, pack_hdr, vlan, status,
   output ar4 array
);
always_comb
array =  swap (value, switch, pack_hdr, vlan, status);
endmodule
```

- Function within a module

```
typedef logic [7:0] ar8_8 [3:0]; //supports multidimensional array
module do_auto10_sv (
   input integer i,
   input [7:0] value, switch, pack_hdr,
   output ar8_8 array
);
function ar8_8 swap (
   input [7:0] value, switch, pack_hdr,
   input integer i
);
localparam VLU =  0;
localparam SCH =  1;
localparam HDR =  2;
localparam NUM =  3;
begin: myFunc
   swap[VLU] = value;
   swap[SCH] = switch;
   swap[HDR] = pack_hdr;
   swap[NUM] = i;
end
endfunction
always_comb
array =  swap(value, switch, pack_hdr, i);
endmodule
```

The logic Type

This example uses the `logic` type to describe a 33-bit adder.

```
module add_fun_new (
   input logic [31:0] val1, val2,
   output logic [32:0] result
);
function logic [32:0] adder33 ([31:0] val1, val2); // default input logic
return (val1 + val2);
endfunction
assign result = adder33(val1, val2);
endmodule
```

The longint Type

This example defines the val1 and val2 inputs with the `longint` type as the arguments of the subtractor64 function, which returns a result of the `longint` type.

```
module subtractor_func_longint_new (
   input longint val1, val2,
   output longint result
);
function longint subtractor64 (longint val1, val2); // input direction
return( val1 - val2 );
endfunction
assign result = subtractor64 (val1, val2);
endmodule
```

User-Defined Structure

This example uses the `typedef` construct to create user-defined structures in module ports, task inputs, and output arguments. This code builds an adder and a subtractor.

```
typedef struct {
   reg [32:0] sum;
   reg [31:0] diff;
} addsub;

typedef struct {
   reg [31:0] val_1;
   reg [31:0] val_2;
} in_vals;

module struct_to_and_from_task (
   input in_vals  val_1_and_val_2,
   output addsub result
);                               );

task calc_values (input in_vals val_1_and_val_2, output addsub result);
addsub tmp;
tmp.sum  = val_1_and_val_2.val_1 +  val_1_and_val_2.val_2;
tmp.diff = val_1_and_val_2.val_1 -  val_1_and_val_2.val_2;
result   = tmp;
endtask
always_comb calc_values (val_1_and_val_2, result);
endmodule
```

Output Argument and a Return Value

This example shows that the prod_and_diff function returns a value and an output argument. This design has a subtractor and a multiplier in the function.

```
module function_with_output_arguments #(parameter N=8)(
   input logic [N-1:0] A, B,
   output logic [N:0] DIFF, logic [2*N-1:0] PROD
);
function automatic logic [N-1:0]  prod_and_diff (
   input logic [N-1:0] Aval, Bval,
   output logic [2*N-1:0] prod_val
);
prod_val = Aval * Bval;
return (Aval - Bval);
endfunction

always_comb
DIFF = prod_and_diff(A, B, PROD);
endmodule
```

SystemVerilog for Loop

This example counts the number of zeros in the input and outputs the result. The legal function checks whether the input is legal using a SystemVerilog `for` loop. The zeros function, which counts zeros using a SystemVerilog `for` loop, has an output argument and returns nothing. It is a pseudo void function.

```
function automatic logic legal (input [7:0] x);
reg seenZero, seenTrailing;
begin :_legal_block
   legal = 1; seenZero = 0; seenTrailing = 0;
   for(int i = 0; i <= 7; i++)
   if( seenTrailing && (x[i] == 1'b0) )
   begin
      return 0;
   end
   else if( seenZero && (x[i] == 1'b1) )
      seenTrailing = 1;
   else if( x[i] == 1'b0 )
      seenZero = 1;
end
endfunction

function automatic void zeros (
   input [7:0] data,
   output logic [3:0] num_zeros
);
logic [3:0] count;
count = 0;
for(int i = 0; i <= 7; i++)
if( data[i] == 1'b0)
   count++;
   num_zeros = count;
endfunction

module count_zeros (
   input logic [7:0] data,
   output logic [3:0] result, logic error
);
wire is_legal = legal(data);
logic [3:0] temp_result;
assign error =! is_legal;
always_comb zeros(data, temp_result);
assign result = is_legal ? temp_result : 1'b0;
endmodule
```

Sensitivity List Within a Function

The tool supports a sensitivity list in a function. To avoid synthesis and simulation mismatch, you must specify a sensitivity list if you use a `case` statement in the RTL. For example,

```
typedef logic [3:0] ar4;
module do_auto12_sv (
   input [3:0] value, switch, pack_hdr, vlan,
   input [1:0] status,
   output ar4 array
);
function automatic ar4 swap (
   input [3:0] value, switch, pack_hdr, vlan,
   input [1:0] status
);
begin: myFunc
   priority case(status)//supports sensitivity list
   0: swap = value;
   1: swap = switch;
   2: swap = pack_hdr;
   3: swap = vlan;
endcase
end
endfunction

always_comb
array =  swap(value, switch, pack_hdr, vlan, status);
endmodule
```

Memory Elements Outside a Function

Memory elements, such as registers, should be placed outside a function. For example,

```
module do_auto19_sv (
   input [31:0] stop_now,
   input clk,
   output logic [31:0] watchdog
);
function automatic [31:0] counter(input [31:0] stop_now);
automatic logic [31:0] temp = stop_now;
localparam [5:0] size = 32;
for (int i = 0; i < size; i++)
begin
   if (i == 0)
   begin
      if (!temp[0])
         counter = 0;
      else
         counter = 1;
   end
   else if (temp[i])
      counter++;
   end
endfunction

always_ff @(posedge clk)
watchdog <= counter(stop_now);
endmodule
```

Binding Function and Task Arguments by Name

You can pass function and task arguments by name with a port-like .name syntax, as shown in the following example. If both positional and named arguments are specified in a single subroutine call, all the positional arguments must come before the named arguments. For more information, see the *IEEE Std 1800-2012*.

```
module test (
   input integer value1, value2,
   output logic [32:0] result1, result2
);

function logic [32:0] adder (integer a, b);
return (a + b);
endfunction

// Pass arguments by order
assign result1 = adder(value1, value2);

// Pass arguments by name
assign result2 = adder(.b(value2), .a(value1));
endmodule
```

As shown in the following code, the tool does not allow default argument values for functions, and it issues a VER-721 error message.

```
function logic [32:0] adder33 (int a, b = 25);
return (a + b);
endfunction
```

Parameterized Functions and Tasks Using Virtual Classes

The tool supports parameterized functions and tasks via static methods of parameterized virtual classes.

Functions or tasks containing static methods that use parameters allow you to easily redefine the function or task behaviors based on the parameter definitions. You create and maintain only one parameter definition instead of multiple subroutines with different array sizes, data types, and variable widths.

The following example shows the declaration of the F1 function inside a virtual class where the S and T parameters are defined to characterize the behavior of the F1 function. The top module uses the class scope resolution operator (::) to access the F1 function and redefines the S and T parameters.

```
virtual class MyClass #(parameter S=4, parameter type T=bit);
static function T F1 (T [S-1:0] x);
return (&x);
endfunction
endclass

module top (x, y);
   input  logic [7 :0] x;
   output logic  y ;
assign y = MyClass#(.S(8), .T(logic))::F1(x);
endmodule
```

To use the default parameter values, use an empty #() parameter setting. For example,

```
assign y = MyClass#()::F1(x);
```

The following restrictions apply:

- Virtual class declaration is supported only in $unit.
- Only static methods can be declared inside virtual classes.
- Only the class scope resolution operator (::) can be used to access virtual class methods.
- Synthesis supports only virtual classes.

Parameterized Data Types

To modify parameters in modules or interfaces, set the parameter data types by using the `parameter type` keywords. If you do not specify a data type, the default is `logic`.

To learn how to parameterize data types, see

- Parameterized Standard Data Types
- Parameterized User-Defined Data Types
- Parameterized Data Types in Interfaces

Parameterized Standard Data Types

The following example sets the default data type of comparatortype to `int` in the comparator module using the `parameter type` keywords. The top module parameterizes comparatortype to `int`, `shortint`, and `longint` for the instantiated comparator modules comp16, comp32, and comp_fp respectively. Explicit redefinitions are needed for the comp32 and comp_fp instances (#(.comparatortype(shortint))and #(.comparatortype(longint))) because of the nondefault types.

```
module comparator #(parameter type comparatortype = int)(
   input comparatortype a, comparatortype b,
   output logic lt, logic gt, logic eq
);
always_comb
begin
   unique if (a < b)
   begin
      lt = 1'b1;
      gt = 1'b0;
      eq = 1'b0;
   end
   else if (a > b)
   begin
      lt = 1'b0;
      gt = 1'b1;
      eq = 1'b0;
   end
   else if ( a == b)
   begin
      eq = 1'b1;
      lt = 1'b0;
      gt = 1'b0;
   end
end
endmodule

module top (
   input int a1, b1,
   input shortint a2, b2,
   input longint a3, b3,
   output logic [2:0], less_than, greater_than, equal
);
//32-bit comparator
comparator comp32 (a1, b1, less_than[0], greater_than[0], equal[0]);

//16-bit comparator
comparator #(.comparatortype(shortint))
comp16 (a2, b2, less_than[1], greater_than[1], equal[1]);

//long comparator
comparator #(.comparatortype(longint))
comp_fp (a3, b3, less_than[2], greater_than[2], equal[2]);
endmodule
```

Parameterized User-Defined Data Types

The following example contains two user-defined data types: data_packet and big_data_packet. The test module sets the default data type of data_packet_type to data_packet using the `parameter type` keywords. The top module parameterizes data_packet_type to data_type and big_data_type for the u1 and u2 instances respectively. Explicit redefinition is needed for the u2 instance (#(.data_packet_type(big_data_packet))) because of the nondefault type.

```
typedef struct {
  logic [31:0] src_a;
  logic [31:0] dst_a;
  logic  [3:0] hdr;
} data_packet;

typedef struct {
  logic [63:0] src_a;
  logic [63:0] dst_a;
  logic  [9:0] hdr;
} big_data_packet;

module test #(parameter type data_packet_type = data_packet)(
   input data_packet_type a,
   output data_packet_type b
);
assign b = a;
endmodule

module top (
   input data_packet di1,
   input big_data_packet bdi1,
   output data_packet do1,
   output big_data_packet bdo1
);
test u1(di1, do1);
test #(.data_packet_type(big_data_packet)) u2(bdi1, bdo1);
endmodule
```

Parameterized Data Types in Interfaces

The following example sets the default data type of new_type to `bit` in the I interface using the `parameter type` keywords. The two_dff module parameterizes new_type to `bit` and `logic` for the instantiated interfaces inst1 and inst2 respectively. Explicit redefinition is needed for the inst2 instance (#(.new_type(logic))) because of the nondefault type.

```
interface I #(parameter type new_type = bit)(input new_type clk, rst, d);
modport MP(input clk, rst, d);
endinterface : I

module dff (
   I.MP a,
   output logic q
);

always_ff @ (posedge a.clk, negedge a.rst)
begin
   if(!a.rst) q <= ’0;
   else q <= a.d;
end
endmodule

module two_dff (
   input clk, rst, d [1:0],
   output logic q [1:0]
);

I inst1 (.clk, .rst, .d(d[1])); //two state
dff u1(.a(inst1.MP), .q(q[1]));

I # (.new_type(logic)) inst2 (.clk, .rst, .d(d[0])); //four state
dff u2 (.a(inst2.MP), .q(q[0]));
endmodule
```

Bit-Level Support for Compiler Directives

You can apply the `sync_set_reset`, `async_set_reset`, `one_hot`, `one_cold`, and `keep_signal_name` compiler directives at the bit level.

Example: sync_set_reset

```
module dff_sync (
   input clk, d, st, rst,
   output logic q
);
// synopsys sync_set_reset rst
always_ff @ (posedge clk)
begin
   if (rst) q <= 1'b0;
   else     q <= d;
end
endmodule
```

Example: async_set_reset

```
module dff_async (
   input clk, d, st, rst,
   output logic q
);
// synopsys async_set_reset "st, rst"
always_ff @ (posedge clk or posedge st or posedge rst)
begin
   if (st)       q <= 1'b1;
   else if (rst) q <= 1'b0;
   else          q <= d;
end
endmodule
```

Example: one_hot

```
module dff_async (input clk, d, st, rst, output logic q);
// synopsys one_hot "st, rst"
always_ff @...
```

Example: one_cold

```
module dff_async (input clk, d, st, rst, output logic q);
// synopsys one_cold "st, rst"
always_ff @...
```

Structures

The Synopsys synthesis tools support structure data types in SystemVerilog. You can use the `struct` type to group a collection of variables.

Structure Data Type

This example uses a `typedef` declaration to create a structure type for a CPU instruction that consists of an 8-bit opcode and a 32-bit address.

```
typedef struct {
byte opcode;     // 8 bits
int addr;        // 32 bits
} instruction;   // named structure type

module m;
instruction IR1, IR2, IR3; // define variable
...
endmodule
```

Packed Structure Data Type and the Initialization

The following example uses a packed structure, enums, and $unit to describe a 35-bit register that consists of asynchronous reset flip-flops.

```
typedef enum logic [1:0]{OFF = 2'd0, ON = 2'd3} SWITCH_VALUES;
// default - integer for enums
// RED = 0, GREEN = 1, BLUE = 2
typedef enum {RED, GREEN, BLUE} LIGHT_COLORS;

typedef struct packed {
SWITCH_VALUES switch;  //  2 bits
LIGHT_COLORS light;    // 32 bits
logic test_bit;        //  1 bit
} sw_lgt_pair;
module struct_default (
   input logic clock, reset,
   output sw_lgt_pair slp
);
always_ff @ (posedge clock, posedge reset)
begin
   if(reset)
   // Initialization: clears all 35 bits in the packed structure
      slp <= '0;
   else
   begin
      slp.switch <= ON;
      slp.light <= GREEN;
      slp.test_bit <= 1;
   end
end
endmodule
```

You do not need to initialize each member of the packed structure because all the members of the packed structure are initialized by the reset statement (if(reset) slp <= '0;). The tool treats the packed structure as a single vector, as shown in the following inference report:

```
=========================================================================
| Register Name |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST
=========================================================================
|    slp_reg    | Flip-flop |  35   |  Y  | N  | Y  | N  | N  | N  | N
=========================================================================
```

Unpacked Structure and the Initialization

You must initialize each member of an unpacked structure separately during reset, as shown in the following example. If you initialize an unpacked structure as a group using the reset statement (if(reset) slp <= '0;), the tool issues an ELAB-930 error message. This packed structure example contains the correct initialization statement.

```
typedef enum logic [1:0] {ON = 2'd3, OFF = 2'd0} SWITCH_VALUES;
typedef enum {RED, GREEN, BLUE} LIGHT_COLORS;//default- integer for enums
typedef struct {
SWITCH_VALUES switch; //2 bits
LIGHT_COLORS light;   //32 bits
logic test_bit;       // 1 bit
} sw_lgt_pair;

module struct_default(
   input logic clock, reset,
   output sw_lgt_pair slp
);
always_ff @ (posedge clock, posedge reset)
begin
   if(reset)
        // Initialize each member because it is an unpacked struct
      slp <= '{SWITCH_VALUES : ON, LIGHT_COLORS : RED, logic : 1'b0};
     else
   begin
      slp.switch <= OFF;
      slp.light <= GREEN;
      slp.test_bit <= 1;
     end
end
endmodule
```

The tool builds a 2-bit register using asynchronous set flip-flops and a 33-bit register using asynchronous reset flip-flops, as shown in the following inference report:

```
=========================================================================
| Register Name |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST
|========================================================================
|    slp_reg    | Flip-flop |   2   |  Y  | N  | N  | Y  | N  | N  | N
|    slp_reg    | Flip-flop |  33   |  Y  | N  | Y  | N  | N  | N  | N
=========================================================================
```

Unions

The synthesis tools support the `union` construct in SystemVerilog, as shown in the following example and inference report:

RTL Containing a Union Construct

```
typedef struct {
   union packed{
   logic [31:0] data;
   int i;
   }ff;
} my_struct;

module union_example (
   input clk,
   input my_struct d,
   output my_struct q
);
my_struct loop_index;
always_ff @(posedge clk)
begin
   for (loop_index.ff.i = 0; loop_index.ff.i <= 31; loop_index.ff.i++)
   q.ff.data[loop_index.ff.i] <= d.ff.data[loop_index.ff.i];
end
endmodule
```

Inference Report

```
=========================================================================
| Register Name |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
=========================================================================
|     q_reg     | Flip-flop |   32   |  Y  | N  | N  | N  | N  | N  | N |
=========================================================================
```

See Also

- Unsupported Constructs

Multidimensional Arrays

You can use multidimensional arrays as function arguments, module ports in packed and unpacked arrays, array slicing, and part-select operations:

- Multidimensional Arrays as Function Arguments
- Multidimensional Arrays as Unpacked Arrays
- Multidimensional Arrays as Unpacked Arrays Using $low and $high
- Multidimensional Arrays as Unpacked Arrays Using $left and $right
- Multidimensional Array Slicing
- Multidimensional Arrays Using Part-Select Addressing

Multidimensional Arrays as Function Arguments

This example uses multidimensional arrays as function arguments.

```
function logic test (
   input logic [10:1][2:1] packed_mda,
   input logic [10:1] unpacked_mda [2:1]
);
...
endfunction
```

Multidimensional Arrays as Unpacked Arrays

This example generates and checks the parity of 10 packets. It uses an unpacked array of unpacked structures to model the packets and uses multidimensional arrays as module ports.

```
// Generates and checks parity of ten packets. Uses unpacked array of
// unpacked structures to model all the bits of all the packets.
typedef struct {
logic [7:0] hdr1;
logic [7:0] hdr2;
logic null_flag;
logic [27:0] data_body;
} network_packet;

module packet_op_array #(parameter NUM_PACKETS = 10)(
   input network_packet packet1 [NUM_PACKETS -1:0],
   input network_packet packet2 [NUM_PACKETS -1:0],
   output logic packet_parity1 [NUM_PACKETS -1:0],
   output logic packet_parity2 [NUM_PACKETS -1:0],
   output logic packets_are_equal [NUM_PACKETS -1:0]
);

function logic parity_gen (network_packet packet);
return(^{packet.hdr1, packet.hdr2, packet.null_flag, packet.data_body});
endfunction

function logic compare_packets(network_packet packet1, packet2);
if ((packet1.hdr1 == packet2.hdr1)
  && (packet1.hdr2 == packet2.hdr2)
  && (packet1.null_flag == packet2.null_flag)
  && (packet1.data_body == packet2.data_body) )
   return (1'b1);
else
   return (1'b0);
endfunction

always_comb
begin
   for(int i = 0; i< NUM_PACKETS; i++)
   begin
      packet_parity1[i] = parity_gen(packet1[i]);
      packet_parity2[i] = parity_gen(packet2[i]);
      packets_are_equal[i] = compare_packets(packet1[i], packet2[i]);
   end
end
endmodule
```

Multidimensional Arrays as Unpacked Arrays Using $low and $high

This example uses an unpacked array as a module port and the `$low` and `$high` array query functions with SystemVerilog `for` loops.

```
module mda_array_query (
   input [7:0] a,
   output logic t [0:3][0:7], logic z
);
integer k;
always_comb
begin
   for (int j = $low(a, 1); j <= $high(a, 1); j++) begin
      t[0][j] = a[j];
   end
   for (int i = 1; i < 4; i++) begin
      k = 1 << (3-i);
      for (int j = 0; j < k; j++) begin
         t[i][j] = t[i-1][2*j] ^ t[i-1][2*j+1];
      end
   end
end
assign z = t[3][0];
endmodule
```

Multidimensional Arrays as Unpacked Arrays Using $left and $right

This example uses unpacked arrays as module ports, the `$left` and `$right` array query functions, and an enhanced `for` loop to describe the matrix_adder module.

```
function automatic logic signed [32:0] add_val1_and_val2
(logic signed [31:0] val1, val2);
return (val1 + val2);
endfunction

module matrix_adder (
   input logic signed [31:0]  a[0:2][0:2],
   logic signed [31:0] b[0:2][0:2],
   output logic signed [32:0] sum[0:2][0:2]
);
always_comb
begin
   for (int i=$left(a, 1); i<=$right(a, 1); i++)
   begin
      for (int j=$left(a, 2); j<=$right(a, 2); j++)
      begin
         sum[i][j] = add_val1_and_val2(a[i][j], b[i][j]);
      end
   end
end
endmodule
```

Multidimensional Array Slicing

This example shows that a multidimensional array is referenced in two array slices.

```
module mda_slicing (
   input logic[31:0] j[7:0],
   output int k [1:0]
);
assign k = j[7:6];
endmodule : mda_slicing
```

Multidimensional Arrays Using Part-Select Addressing

This example uses a `generate` statement and assigns values to the r_val, b_val, and g_val multidimensional arrays by using part-select addressing.

```
typedef logic [0:23] three_byte;
typedef logic [0:7] one_byte;

module mda_unpacked_psel (
   input three_byte pixel_array[0:3],
   output one_byte r_val[0:3], g_val[0:3], b_val[0:3]
);
genvar i;
generate
for ( i=0; i<4; i++) begin: outer_loop
   assign r_val[i] = pixel_array[i][0+:8];   // select all red
   assign g_val[i] = pixel_array[i][8+:8];   // select all green
   assign b_val[i] = pixel_array[i][16+:8];  // select all blue
end
endgenerate
endmodule : mda_unpacked_psel
```

For synthesis restrictions on multidimensional arrays, see Unsupported Constructs.

Configurations

You can use configurations to specify binding information of module instances down to the cell level in the design. The configurations can be analyzed and elaborated by the `analyze` and `elaborate` commands respectively. For example,

```
dc_shell> analyze -f sverilog {submodule.sv ...}
dc_shell> analyze -f sverilog top_module.sv
dc_shell> analyze -f sverilog config_file.sv
dc_shell> elaborate my_config_of_design
```

By default, the HDL Compiler tool resolves lower module instances by applying a design library search order taken from the dc_shell environment and the analyzed parent module. Alternatively, you can specify design library locations (bindings) for module or interface instances of a specific design in configurations by using the `config` element. All bindings in the design hierarchy are constrained by the configuration rules given in the config_rule subset in the configuration.

The following configuration syntax is supported for synthesis:

```
config config_id;
   design {[lib_id .] design_id};
   {config_rule}
endconfig [: config_id]
```

where

```
config_rule ::= default liblist {lib_id};
              | instance design_id {. inst_id} liblist lib_id;
```

```
design_id ::= module_id | interface_id
```

For more information about the syntax, see the *IEEE Std 1800-2012*.

The following limitations apply when you use configurations:

- Only one library is allowed for instances.
- Only one default rule is allowed.
- Library declarations are not allowed.

  To define libraries, use the `define_design_lib` command. Design library names in dc_shell are not case-sensitive.
- The `read_file` command and the `-autoread` option do not support configurations.
- Configuration rules do not affect the bindings of designs that are already elaborated or loaded in memory.

Configuration Examples

The following topics provide examples on how to use configuration rules and designs:

- Default Statement
- Instance Bindings
- Multiple Top-Level Designs

The examples use these low-level modules:

- sub1.v

  ```
  module sub1(
     input i1, i2,
     output o1
  );
  assign o1 = i1 & i2;
  endmodule
  ```

- sub2.v

```
module sub1(
   input i1, i2,
   output o1
);
assign o1 = i1 | i2;
endmodule
```

- sub3.v

```
module sub1(
   input i1, i2,
   output o1
);
assign o1 = i1 ^ i2;
endmodule
```

Note:
The three low-level files use the same sub1 module name, but they implement different functions. The sub1.v, sub2.v, and sub3.v files implement AND, OR, and XOR functions respectively.

Default Statement

The following example uses a configuration to direct the tool to choose the implementation of the instances in the top-level module. The configuration file specifies the default statement, but no binding information.

- Top-level top.v file

```
module top(
   input i1, i2, i3, i4,
   output o1, o2, o3
);
sub1 U1 (i1, i2, o1);
sub1 U2 (o1, i3, o2);
sub1 U3 (o2, i4, o3);
endmodule
```

- Configuration file

```
config cfg1;
design rtlLib.top;
default liblist rtlLib;
endconfig
```

- Design Compiler Tcl script

```
define_design_lib lib1 -path ./lib1
define_design_lib lib2 -path ./lib2
define_design_lib rtlLib -path ./rtlLib
analyze -f sverilog -library lib1 sub1.v
analyze -f sverilog -library lib2 sub2.v
analyze -f sverilog -library rtlLib sub3.v
analyze -f sverilog -library rtlLib top.v
analyze -f sverilog config.v
elaborate cfg1
```

- Netlist

  The output netlist shows that the sub1 module analyzed in the rtlLib library is chosen for the instantiations in the top module.

```
module sub1 ( i1, i2, o1 );
   input i1, i2;
   output o1;
GTECH_XOR2 C7 ( .A(i1), .B(i2), .Z(o1) );
endmodule

module top ( i1, i2, i3, i4, o1, o2, o3 );
   input i1, i2, i3, i4;
   output o1, o2, o3;
sub1 U1 ( .i1(i1), .i2(i2), .o1(o1) );
sub1 U2 ( .i1(o1), .i2(i3), .o1(o2) );
sub1 U3 ( .i1(o2), .i2(i4), .o1(o3) );
endmodule
```

Instance Bindings

The following example shows how to use instance bindings in configurations. The configuration file specifies the binding of each instance of the sub1 module, but no default statement.

- Top-level top.2 file

```
module top(
   input i1, i2, i3, i4,
   output o1, o2, o3
);
sub1 U1 (i1, i2, o1);
sub1 U2 (o1, i3, o2);
sub1 U3 (o2, i4, o3);
endmodule
```

- Configuration file

```
config cfg1;
design rtlLib.top;
instance top.U1 liblist lib1;
instance top.U2 liblist lib2;
instance top.U3 liblist lib3;
endconfig
```

- Design Compiler Tcl script

```
define_design_lib lib1 -path ./lib1
define_design_lib lib2 -path ./lib2
define_design_lib lib3 -path ./lib3
define_design_lib rtlLib -path ./rtlLib
analyze -f sverilog -library lib1 sub1.v
analyze -f sverilog -library lib2 sub2.v
analyze -f sverilog -library lib3 sub3.v
analyze -f sverilog -library rtlLib top.v
analyze -f sverilog config.v
elaborate cfg1
```

- Netlist

The output netlist shows that each instance of the sub1 module uses a different library specified in the configuration file. The U1 instance uses the sub1 module from the lib1 library to implement the AND function. The U2 instance uses the sub1 module from the lib2 library to implement the OR function. The U3 instance uses the sub1 module from the lib3 library to implement the XOR function.

```
module sub1 ( i1, i2, o1 );
   input i1, i2;
   output o1;
GTECH_AND2 C7 ( .A(i1), .B(i2), .Z(o1) );
endmodule

module sub1_1 ( i1, i2, o1 );
   input i1, i2;
   output o1;
GTECH_OR2 C7 ( .A(i1), .B(i2), .Z(o1) );
endmodule

module sub1_2 ( i1, i2, o1 );
   input i1, i2;
   output o1;
GTECH_XOR2 C7 ( .A(i1), .B(i2), .Z(o1) );
endconfig

module top ( i1, i2, i3, i4, o1, o2, o3 );
   input i1, i2, i3, i4;
   output o1, o2, o3;
sub1 U1 ( .i1(i1), .i2(i2), .o1(o1) );
sub1_1 U2 ( .i1(o1), .i2(i3), .o1(o2) );
sub1_2 U3 ( .i1(o2), .i2(i4), .o1(o3) );
endmodule
```

Multiple Top-Level Designs

The following example shows that you can specify multiple top-level designs in configurations. The configuration file instantiates the top1 and top2 top-level designs.

- Top-level top1.v file

```
module top1(
   input i1, i2, i3, i4,
   output logic o1, o2, o3
);
sub1 U1 (i1, i2, o1);
endmodule
```

- Top-level top2.v file

```
module top2(
   input i1, i2, i3, i4,
   output logic o1, o2, o3
);
sub1 U2 (o1, i3, o2);
endmodule
```

- Configuration file

```
config cfg1;
design lib1.top1 lib2.top2;
instance top1.U1 liblist lib3;
instance top2.U2 liblist lib4;
endconfig
```

- Design Compiler Tcl script

```
define_design_lib lib1 -path ./lib1
define_design_lib lib2 -path ./lib2
define_design_lib lib3 -path ./lib3
define_design_lib lib4 -path ./lib4
define_design_lib lib5 -path ./lib5
analyze -f sverilog -library lib4 sub2.v
analyze -f sverilog -library lib5 sub3.v
analyze -f sverilog -library lib3 sub1.v
analyze -f sverilog -library lib1 top1.v
analyze -f sverilog -library lib2 top2.v
analyze -f sverilog config.v
elaborate cfg1
```

- Netlist of the top1.v file

  The top1netlist shows that the sub1_1 module from the lib3 library is used to implement the AND function, as specified in the configuration file.

```
module sub1_1 ( i1, i2, o1 );
   input i1, i2;
   output o1;
GTECH_AND2 C7 ( .A(i1), .B(i2), .Z(o1) );
endmodule

module top1 ( i1, i2, i3, i4, o1, o2, o3 );
   input i1, i2, i3, i4;
   output o1, o2, o3;
sub1_1 U1 ( .i1(i1), .i2(i2), .o1(o1) );
endmodule
```

- Netlist of the top2.v file

  The top2 netlist shows that the sub1 module from lib4 library is used to implement the OR function, as specified in the configuration file.

```
module sub1 ( i1, i2, o1 );
   input i1, i2;
   output o1;
GTECH_OR2 C7 ( .A(i1), .B(i2), .Z(o1) );
endmodule

module top2 ( i1, i2, i3, i4, o1, o2, o3 );
   input i1, i2, i3, i4;
   output o1, o2, o3;
sub1 U2 ( .i1(o1), .i2(i3), .o1(o2) );
endmodule
```

Implicit Port Connections

The Synopsys synthesis tools support the SystemVerilog .name and .* implicit port connections. The implicit port connections apply to both module ports and interface ports. Unlike Verilog named port connections (also called explicit port connections), the implicit port connections list each port name with a leading period for all the ports of the instantiated module.

Implicit .name Port Connections

In the following example, the dot_name module instantiates the dff module using the .name syntax (`dff U1(.in, .clk, .rst, .out);`), which is equivalent to the explicit port connections (`dff U1(.in(in), .clk(clk), .rst(rst), .out(out));`).

```
module dot_name (
   input in, clk, rst,
   output logic out
);
dff U1(.in, .clk, .rst, .out);
endmodule

module dff (
   input in, clk, rst,
   output logic out
);
always_ff @(posedge clk or negedge rst)
if (!rst) out <= '0;
else out      <= in;
endmodule
```

Mixed Implicit and Explicit Port Connections

You can mix the Verilog explicit port connections and SystemVerilog implicit port connections, as shown in the following example:

```
module dot_name (
   input in, clk, rst,
   output logic out
);
dff U1(.in, .clk, .reset(rst), .out );
endmodule

module dff (
   input in, clk, reset,
   output logic out
);
...
endmodule
```

Implicit .* Port Connections

In the previous example, the instance port name and module port name are identical for each port except the reset port. You can use the .* syntax, which connects the instance port and module port that have the same port name and port size, as shown in the following example:

```
module dot_star_test (
   input in, clk, rst,
   output logic out
);
dff U1(.*, .reset(rst));
endmodule

module dff (
   input in, clk, reset,
   output logic out
);
...
endmodule
```

You can also use the .* syntax to connect interface ports. The design instance contains the .* port connection must meet either one of the following requirements; otherwise, the tool issues an ELAB-197 error message.

- The module or interface being instantiated must have already been analyzed.
- The module design must be loaded into the link library.

Implicit .name Interface Port Connections

In the following example, both the M module and dot_name module instantiate the I interface as i1. You can connect the interface ports using the .name port connection (`M m1 (.i1);`), which is equivalent to the Verilog explicit port connection (`M m1 (.i1(i1));`).

```
interface I (
input logic clk, rst, logic [7:0] d,
output logic [7:0] q
);
endinterface

module M(I i1);
endmodule

module dot_name (
   input logic clk, rst, logic [7:0] d,
   output logic [7:0] q
);
I i1(.rst, .clk, .q, .d); //or I i1(.rst(rst), .clk(clk), .q(q), .d(d))
M m1(.i1);  //or  M m1(.inst1(i1)); if module declaration M had I
endmodule
```

Casting

The following example uses size, sign, and user-defined type casting. To determine the size of a packed array, the example uses the `$bits` system task to compute the total number of bits in the my_struct packed array.

Size, Sign, and User-Defined Type Casting

```
localparam VEC = 1;
localparam STRUCT_ARRAY_SIZE = 2;

typedef logic [3:0] nibble;
typedef enum nibble {A=1, B=2, C=4, D=8} one_hot_variable;

typedef struct packed{
one_hot_variable  [VEC:0] nibble_array; // 2 * 4 = 8 bits
logic b;      //1 bit
} my_struct;  // total 9 bits

module test (
   input my_struct [STRUCT_ARRAY_SIZE:0] struct_array_in,  // 9*3=27 bits
   output logic [$bits(struct_array_in)-1:0] packed_array, // 27 bits
   output my_struct single_struct,                         // 9 bits
   output logic [19:0] twenty_bits_of_packed_array,        // 20 bit
   output logic one_bit_of_packed_array_with_sign          //signed 1 bit
);

// assign the entire array of packed structures to a packed vector
assign packed_array = struct_array_in;

// casting to the my_struct user-defined type
assign single_struct = my_struct’(packed_array);

// size casting, assigning 20 bits of the packed array
assign twenty_bits_of_packed_array = 20’(packed_array);

// sign casting
assign one_bit_of_packed_array_with_sign =
signed’(twenty_bits_of_packed_array);

endmodule
```

Assignment Patterns

An *assignment pattern* specifies a correspondence between a collection of expressions, and structure fields or array elements of a data object or value.

The Base Assignment Pattern

An assignment pattern is constructed of a single quotation mark (’) followed by a collection of expressions enclosed in curly brackets ({}):

```
var1 = '{var2, var3};
```

An assignment pattern has no self-determined data type, but it can be used as one of the sides in an assignment-like context when the other side has a self-determined data type:

```
logic [2:0][3:0] dout;
dout <= '{3 {2'b11}};   // will get "0011_0011_0011"
```

Structures allow sparse assignments to named fields. The `default` key assigns a value to all fields not covered by other keys:

```
typedef struct { int f1; int f2; int f3 } threeFields;
localparam threeFields var2 = '{f2 : 2, default : 0};  // f1 and f3 get 0
```

Assignment patterns can be nested to support assignment to complex data types:

```
typedef struct {
  int field1 [3];
  int field2 [3];
}  twoFields [1:0];

localparam twoFields var1 = '{ '{field1: '{1,2,3}, field2: '{4,5,6}},
                               '{field1: '{7,8,9}, field2: '{10,11,12}}};
```

Assignment Patterns Versus Concatenation

The syntax difference between concatenation `{}` and an assignment pattern `'{}` is small, but the behavior can differ significantly in some cases:

- The assignment pattern behavior is based on the destination type and thus supports much more robust functionality (including implicit type casting, key-based assignment, default value assignment, assignment to unpacked arrays and structures).
- A concatenation does not change behavior based on the destination type; it simply puts all of the bits into a single vector and passes that to the assignment.

Note the difference in assignment behavior in the following example:

```
logic [2:0][3:0] dout;
dout <= '{3 {2'b10}};   // will get "0010_0010_0010"
dout <= {3 {2'b10}};    // will get "0000_0010_1010"
```

Assignment Pattern Expressions

Another form of assignment pattern is the *assignment pattern expression*. The syntax is similar to the base assignment pattern, but a type definition is provided before the single quotation mark ('):

```
typedef logic [7:0] twoChar [2];
var1 = myChar'{var2, var3};
```

An assignment pattern expression can be used to construct or deconstruct an array or structure. Unlike the base assignment pattern, an assignment pattern expression has a self-determined data type and is not restricted to being used in an assignment-like context:

```
logic [3:0] dout1, dout2, dout3;
typedef logic [3:0] unpackedLogic [3];

bot b1 (.dout(unpackedLogic'{dout1,dout2,dout3}));
```

Limitations

The following constructs are not supported by the tool:

- Assignment patterns or assignment pattern expressions on the left side of assignments
- Array pattern keys in assignment patterns

Because each module is analyzed in its own context, expressions that rely on types from both sides of a module boundary (such as base assignment patterns) are not supported. Thus, the following constructs are supported only with assignment pattern expressions:

- Parameter specification overrides
- Port connections of module or interface instantiations

Macro Expansion and Parameter Substitution

In SystemVerilog, macro expansion occurs before parameters get substituted. When you use parameters in macro calls, the parameter names remain the same even after the macro calls. As shown in the following example, the MAC macro is called by `` `MAC(N) ``. The macro is expanded to the PN parameter and then replaced by the value of the parameter, which is 4. Because the statement `(4 == 4)` is true, output test_bit is assigned a value of 1.

```
`define MAC(x) P``x
module test #(parameter N = 2, PN = 4, P2 = 8)(output test_bit);
assign test_bit = (`MAC(N) == PN);
endmodule
```

`begin_keywords and `end_keywords

To prevent compilation errors due to SystemVerilog keywords in legacy code, encapsulate the code between the `‘begin_keywords` directive followed by a version specifier and the `‘end_keywords` directive. You can set the version specifier to `1364-1995`, `1364-2001`, `1364-2001-noconfig`, `1364-2005`, `1800-2005`, or `1800-2012`.

You use the directive pair outside a design element, such as a module, primitive, configuration, interface, program, or package. The directive pair affects all source code that is encapsulated, even across source code file boundaries.

For example, the following code assigns the logic name to the output; this coding style is not permitted in SystemVerilog. When you convert this code to SystemVerilog, you must encapsulate the code between the directive pair because `logic` is a keyword in SystemVerilog; otherwise, the tool reports an error.

```
`begin_keywords "1364-2005"
module test (input a, input b , output logic);
   assign logic = a | b;
endmodule
`end_keywords
```

Predefined SYSTEMVERILOG Macro

The Synopsys synthesis tools support the predefined SYSTEMVERILOG macro. You can include SystemVerilog constructs in your existing Verilog code by using this macro. All the predefined macros for Verilog 2005 are defined in SystemVerilog. For example,

```
`ifdef SYSTEMVERILOG
module M (input logic i, output int o);
`else
module M (input i, output signed [31:0] o);
`endif
//...
endmodule
```

Matching Block Names

You can append a matching block name proceeded by a colon to the block end keyword. Using matching block names is optional, but this coding style enhances code legibility. You can apply matching block names to `endinterface`, `endmodule`, `endtask`, `endfunction`, and named `begin-end` blocks.

Matching Block Names for State Machines

This example uses a matching block name for each design element:

- seq_block and count_block for the `begin-end` blocks in the sequential `always_ff` block
- comb_block for the `begin-end` block in the combinational `always_comb` block
- up_block and down_block for the two cases in the `case` statement
- counter for the `endmodule` keyword

```
module counter (
   input rst, clk,
   output logic [3:0] cnt
);

localparam DOWN=0, UP=1;
logic crt_ste, nxt_ste;
logic [3:0] int_cnt;

always_ff @ (posedge clk, negedge rst)
begin : seq_block
   if (!rst) crt_ste <= UP;
   else      crt_ste <= nxt_ste;
end : seq_block

always_ff @ (posedge clk, negedge rst)
begin : count_block
   if (!rst) cnt <= '0;
   else      cnt <= int_cnt;
end : count_block

always_comb
begin : comb_block
   nxt_ste = 'bx;
   int_cnt = 0;
   case ( crt_ste )
   UP  : begin: up_block
         if (cnt==14) nxt_ste = DOWN;
         else         nxt_ste = UP;
         int_cnt = cnt + 1;
         end : up_block
   DOWN: begin: down_block
         if (cnt==1) nxt_ste = UP;
         else        nxt_ste = DOWN;
         int_cnt = cnt - 1;
         end : down_block
   endcase
end : comb_block
endmodule : counter
```

Matching Block Names Interfaces and Modules

This example uses the I, loop_iterations, and test matching block names for the interface, `always_comb` block, and module respectively.

```
interface I;
logic [31:0] i;
logic [31:0] o;
modport MP(input i, output o);
endinterface : I

module test ( I.MP a ) ;
always_comb
begin: loop_iterations
   for(int iter = 0; iter <32; iter++)
   a.o[iter] = a.i[iter] ;
end : loop_iterations
endmodule : test
```

Port Renaming

When structures, unions, and multidimensional arrays are used as ports in the RTL, the tool renames the ports in the GTECH netlist, as shown in the following examples:

- Structures
- Unions
- Multidimensional Arrays

Structures

When structures are used as module ports in the RTL, the tool renames the ports in the GTECH netlist. For example,

- RTL of the structure

```
typedef struct {
logic [1:0] field;    // 2 bits
logic flag;           // 1 bit
} packet;             // total 3 bits

module test(input packet p1, output packet p2);
assign p2 = p1;
endmodule
```

- GTECH netlist of the structure

```
module test ( .p1({\p1[field][1] , \p1[field][0] , \p1[flag] }),
              .p2({\p2[field][1] , \p2[field][0] , \p2[flag] }) );

  input  \p1[field][1] , \p1[field][0] , \p1[flag] ;
  output \p2[field][1] , \p2[field][0] , \p2[flag] ;
  wire   \p2[field][1] , \p2[field][0] , \p2[flag] ;

  assign \p2[field][1]  = \p1[field][1] ;
  assign \p2[field][0]  = \p1[field][0] ;
  assign \p2[flag]  = \p1[flag] ;
endmodule
```

Unions

When packed unions with members of the same size are used in module ports, the tool renames the ports in the GTECH netlist. As shown in the following RTL and GTECH netlist, the tool renames the field1 and field2 signals in the RTL to the p1 and p2 vectors in the netlist.

- RTL of the packed union

```
typedef union packed {
logic [7:0] field1;
byte field2;
} packet;

module test(input packet p1, output packet p2);
   assign p2.field2 = p1.field1;
endmodule
```

- GTECH netlist of the packed union

```
module test ( p1, p2 );
  input [7:0] p1;
  output [7:0] p2;

  assign p2[7] = p1[7];
  assign p2[6] = p1[6];
  assign p2[5] = p1[5];
  assign p2[4] = p1[4];
  assign p2[3] = p1[3];
  assign p2[2] = p1[2];
  assign p2[1] = p1[1];
  assign p2[0] = p1[0];
endmodule
```

Multidimensional Arrays

When multidimensional arrays are used in module ports, the tool renames the ports in the GTECH netlist. For example,

- RTL of the multidimensional arrays

```
typedef  logic [0:2] array;

module test (
   input  array A1 [0:1],
   output array A2 [0:1]
);
assign A2[0] = A1[0];
assign A2[1] = A1[1];
endmodule
```

- GTECH netlist of the multidimensional arrays

```
module test ( .A1({\A1[0][0] , \A1[0][1] , \A1[0][2] , \A1[1][0] ,
                   \A1[1][1] , \A1[1][2] }),
              .A2({\A2[0][0] , \A2[0][1] , \A2[0][2] , \A2[1][0] ,
                   \A2[1][1] , \A2[1][2] }) );

  input  \A1[0][0] , \A1[0][1] , \A1[0][2] ,
         \A1[1][0] , \A1[1][1] , \A1[1][2] ;
  output \A2[0][0] , \A2[0][1] , \A2[0][2] ,
         \A2[1][0] , \A2[1][1] , \A2[1][2] ;
  wire   \A2[0][0] , \A2[0][1] , \A2[0][2] ,
         \A2[1][0] , \A2[1][1] , \A2[1][2] ;

  assign \A2[0][0]  = \A1[0][0] ;
  assign \A2[0][1]  = \A1[0][1] ;
  assign \A2[0][2]  = \A1[1][2] ;
  assign \A2[1][0]  = \A1[1][0] ;
  assign \A2[1][1]  = \A1[0][1] ;
  assign \A2[1][2]  = \A1[0][2] ;
endmodule
```

Generic Wire Type

The *IEEE Std 1800-2012* specifies a generic wire type, `interconnect`, to model generic netlists with different types of nets. During synthesis, the Design Compiler tool maps it to a wire, port, or pin just like any data type.

If your design contains this wire type, the tool issues a VER-709 warning similar to the following:

```
Warning:  xxx.sv:2: The interconnect net will be treated as a wire net in
synthesis. (VER-709)
```

In the following example, both signals clk and din are created as input ports with one and two bits respectively:

```
module test (
   input interconnect clk,
   input interconnect [1:0] din,
   output logic [1:0] dout
);

always @(posedge clk) dout <= din;
endmodule
```

General Verilog Coding Guidelines

This topic describes the general Verilog coding guidelines.

- Persistent Variable Values Across Functions and Tasks
- defparam

Persistent Variable Values Across Functions and Tasks

During Verilog or SystemVerilog simulation, a local variable in a function or task has a static lifetime by default. The tool allocates memory for the variable only at the beginning of the simulation, and the recent value of the variable is preserved from one call to another. During synthesis, the HDL Compiler tool assumes that functions and tasks do not depend on the previous values and reinitializes all static variables in functions and tasks to unknowns at the beginning of each call.

The code that does not conform to this synthesis assumption can cause synthesis and simulation mismatches. You should declare all functions and tasks by using the `automatic` keyword, which instructs the simulator to allocate memory for local variables at the beginning of each function or task call.

Note:

> Static variables inside automatic functions or tasks are not allowed in the $unit name space. For more information about static variables, see Synthesis Restrictions for $unit.

defparam

You should not use the `defparam` statements in synthesis because of ambiguity problems. Because of these problems, the `defparam` statements are not supported in the `generate` blocks. For more information, see the *IEEE Std 1800-2012*.

Guidelines for Interacting With Other Flows

The design structure created by the HDL Compiler tool can affect commands applied to the design during the downstream design flows. The following topics provide guidelines for interacting with these flows during the `analyze` and `elaborate` steps:

- Synthesis Flows
- Low-Power Flows
- Verification Flows

Synthesis Flows

The HDL Compiler tool can infer multibit components. If your logic library supports multibit components, they can offer several benefits, such as reduced area and power or a more regular structure for place and route. For more information about inferring multibit components, see infer_multibit and dont_infer_multibit.

Low-Power Flows

This topic provides guidelines to keep signal names in low-power flows:

- Keeping Signal Names
- Using Same Naming Convention Between Tools

Keeping Signal Names

During optimization, the HDL Compiler tool removes nets defined in the RTL, such as dead code and unconnected logic. If your downstream flow needs these nets, you can direct the tool to keep the nets by using the `hdlin_keep_signal_name` variable and the `keep_signal_name` directive. Table 9-1 shows the variable settings.

Table 9-1 hdlin_keep_signal_name Variable Settings

Setting	Description
`all`	The tool preserves a signal if the signal is preserved during optimization. Both dangling and driving nets are considered. Note: This setting might cause the `check_design` command to issue LINT-2 and LINT-3 warning messages.

Table 9-1 hdlin_keep_signal_name Variable Settings (Continued)

Setting	Description
`all_driving` (default)	The tool preserves a signal if the signal is preserved during optimization and is in an output path. Only driving nets are considered.
`user`	The tool preserves a signal if the signal is preserved during optimization and is marked with the `keep_signal_name` directive. Both dangling and driving nets are considered. This setting works with the `keep_signal_name` directive.
`user_driving`	The tool preserves a signal if the signal is preserved during optimization, is in an output path, and is marked with the `keep_signal_name` directive. Only driving nets are considered.
`none`	The tool does not preserve any signal. This setting overrides the `keep_signal_name` directive.

Note:
When a signal has no driver, the tool assumes logic 0 (ground) for the driver.

When you set the `enable_keep_signal` variable to `true`, the tool preserves the nets and issues a warning about the preserved nets during compilation. The tool sets an implicit `size_only` attribute on the logic connected to the nets to be preserved. To mark a net to preserve, label the net with the `keep_signal_name` directive in the RTL and set the `hdlin_keep_signal_name` variable to `user` or `user_driving`. Preserving nets might cause QoR degradation.

In Example 9-1, the tool preserves signals test1 and test2 because they are in the output paths, but it does not preserve signal test3 because it is not in an output path. The tool removes nets syn1 and syn2 during optimization.

Example 9-1 Original RTL

```
module test12 (
   input [3:0] in1,
   input [7:0] in2,
   input in3,
   input in4,
   output logic  [7:0] out1, out2
);
wire test1,test2, test3, syn1, syn2;
//synopsys async_set_reset "in4"
assign test1 = ( in1[3] & ~in1[2] & in1[1] & ~in1[0] );
//test1 signal is in an input and output path
assign test2 = syn1+ syn2;
//test2 signal is in an output path, but not in an input path
assign test3 = in1 + in2;
//test3 signal is in an input path, but not in an output path
always @(in3 or in2 or in4 or test1)
   out2 = test2 + out1;
always @(in3 or in2 or in4 or test1)
   if (in4) out1 = 8'h0;
   else
      if (in3 & test1) out1 = in2;
endmodule
```

To preserve signal test3,

1. Enable the tool to preserve nets by setting the `enable_keep_signal` variable to `true`.
2. Set the `hdlin_keep_signal_name` variable to `user`.
3. Place the `keep_signal_name` directive on signal test3 after the signal declaration in the RTL. For example,

   ```
   wire test1,test2, test3, syn1, syn2;
   //synopsys keep_signal_name "test1 test2 test3"
   ```

Table 9-2 shows how the settings of the variable and directive affect the preservation of signals test1, test2, and test3. An asterisk (*) indicates that the HDL Compiler tool does not attempt to preserve the signal.

Table 9-2 Variable and Directive Matrix for Signals test1, test2, and test3

keep_signal_name	**hdlin_keep_signal_name setting**				
set or not set	**all**	**all_driving**	**user**	**user_driving**	**none**
not set on test1	attempts to keep	attempts to keep	*	*	*
set on test1	attempts to keep	attempts to keep	attempts to keep	attempts to keep	*

Table 9-2 Variable and Directive Matrix for Signals test1, test2, and test3 (Continued)

keep_signal_name	**hdlin_keep_signal_name setting**				
set or not set	**all**	**all_driving**	**user**	**user_driving**	**none**
not set on test2	attempts to keep	attempts to keep	*	*	*
set on test2	attempts to keep	attempts to keep	attempts to keep	attempts to keep	*
not set on test3 (Example 9-1)	attempts to keep	*	*	*	*
set on test3	attempts to keep	*	attempts to keep	*	*

Using Same Naming Convention Between Tools

In some cases, switching activity annotation from a SAIF file might be rejected because of naming differences across multiple tools. To ensure synthesis object names follow the same naming convention used by simulation tools, set the following variable to improve the SAIF annotation:

```
dc_shell> set_app_var hdlin_enable_upf_compatible_naming true
```

Verification Flows

To prevent simulation and synthesis mismatches, follow the guidelines described in this section. Table 9-3 shows the coding styles that can cause simulation and synthesis mismatches and how to avoid the mismatches.

Table 9-3 Coding Styles Causing Synthesis and Simulation Mismatches

Synthesis and simulation mismatch	**Coding technique**
Using the `one_hot` and `one_cold` directives in a Verilog or SystemVerilog design that does not meet the requirements of the directives.	See one_hot and one_cold.
Using the `full_case` and `parallel_case` directives in a Verilog or SystemVerilog design that does not meet the requirements of the directives.	See full_case and parallel_case.

Table 9-3 Coding Styles Causing Synthesis and Simulation Mismatches (Continued)

Synthesis and simulation mismatch	Coding technique
Inferring D flip-flops with synchronous and asynchronous loads.	See D Flip-Flop With Synchronous and Asynchronous Load.
Selecting bits from an array that is not valid.	See Part-Select Addressing Operators ([+:] and [-:]).
Masking the set or reset signal with an unknown during initialization in simulation.	See sync_set_reset.
Using asynchronous design techniques.	The tool does not issue any warning for asynchronous designs. You must verify the design.
Using unknowns and high impedance in comparison.	See Unknowns and High Impedance in Comparison.
Including timing control information in the design.	See Timing Specifications.
Using incomplete sensitivity list.	See Sensitivity Lists.
Using local `reg` variables in functions or tasks.	See Initial States for Variables.

Unknowns and High Impedance in Comparison

A simulator evaluates an unknown (x) or high impedance (z) as a distinct value different from 0 or 1; however, an x or z value becomes a 0 or 1 during synthesis. In HDL Compiler, these values in comparison are always evaluated to false. This behavior difference can cause simulation and synthesis mismatches. To prevent such mismatches, do not use don’t care values in comparison.

In the following example, simulators match 2'b1x to 2’b11 or 2’b10 and 2’b0x to 2’b01 or 2’b00, but both 2'b1x and 2’b0x are evaluated to false in the HDL Compiler tool. Because of the simulation and synthesis mismatches, the HDL Compiler tool issues an ELAB-310 warning.

```
case (A)
   2'b1x:... //  You want 2'b1x to match 11 and 10 but
             //  HDL Compiler always evaluates this comparison to false
   2'b0x:... //  you want 2'b0x to match 00 and 01 but
             //  HDL Compiler always evaluates this comparison to false
   default: ...
endcase
```

In the following example, because `if (A == 1'bx)` is evaluated to false, the tool assigns 1 to reg B and issues an ELAB-310 warning.

```
module test (
   input A,
   output logic B
);
always
begin
   if (A == 1'bx) B = 0;
   else           B = 1;
end
endmodule
```

SystemVerilog provides additional two constructs, `casez` and `casex`, to handle don't care conditions:

- The casez construct for z value
- The `casex` construct for z and x values or for branches that are treated as don't care conditions during comparison

Timing Specifications

The HDL Compiler tool ignores all timing controls because these signals cannot be synthesized. You can include timing control information in the description if it does not change the value clocked into a flip-flop. In other words, the delay must be less than the clock period to avoid synthesis and simulation mismatches.

You can assign a delay to a `wire` or `wand` declaration, and you can use the `scalared` and `vectored` Verilog keywords for simulation. The tool supports the syntax of these constructs, but they are ignored during synthesis.

Sensitivity Lists

When you run the HDL Compiler tool, a module is affected by all the signals in the module including those not listed in the sensitivity list. However, simulation relies only on the signals listed in the sensitivity list. To prevent synthesis and simulation mismatches, follow these guidelines to specify the sensitivity list:

- For sequential logic, include a clock signal and all asynchronous control signals in the sensitivity list.
- For combinational logic, ensure that all inputs are listed in the sensitivity list. Use the `always_comb` construct in SystemVerilog and the `always @*` construct in Verilog.

The tool ignores sensitivity lists that do not contain an edge expression and builds the logic as if all variables within the always block are listed in the sensitivity list. You cannot mix edge expressions and ordinary variables in the sensitivity list. If you do so, the tool issues an error message. When the sensitivity list does not contain an edge expression, combinational logic is usually generated. Latches might be generated if the variable is not fully specified; that is, the variable is not assigned to any path in the block. When you use a SystemVerilog `always_comb` construct that infers a latch, the tool issues an ELAB-974 warning (see The always_comb and always Constructs). When you use a SystemVerilog `always_latch` construct that infers no sequential logic, the tool issues an ELAB-983 warning (see Unintended Logic Inferred Using always_latch).

Note:

> The statements `@(posedge clock)` and `@(negedge clock)` are not supported in functions or tasks.

Initial States for Variables

For functions and tasks, any local variable is initialized to logic 0 and output port values are not preserved across function and task calls. However, values are typically preserved during simulation. This behavior difference often causes synthesis and simulation mismatches. For more information, see Persistent Variable Values Across Functions and Tasks.

See Also

- The *IEEE Std 1800-2012*

10

HDL Compiler Synthesis Directives

HDL Compiler allows you to annotate your SystemVerilog RTL with directives for synthesis. Pragmas are RTL comments that control how synthesis interprets and implements logic construct. SystemVerilog attributes are named values defined in the RTL that can be accessed by your synthesis scripts.

These are described in more detail in the following sections:

- RTL Pragmas
- SystemVerilog Attributes

RTL Pragmas

HDL Compiler synthesis directives are special comments that affect the actions of HDL Compiler and Design Compiler tools. These comments are ignored by other tools.

These synthesis directives begin as a Verilog comment (`//` or `/*`) followed by a *pragma prefix* (`pragma`, `synopsys`, or `synthesis`) and then the directive. The `//$s` or `//$S` prefix can be used as a shortcut for `//synopsys`. The simulator ignores these directives. Whitespace is permitted (but not required) before and after the Verilog comment prefix.

Note:
Not all directives support all pragma prefixes; see “Directive Support by Pragma Prefix” on page 10-20 for details.

The following sections describe the HDL Compiler synthesis pragmas:

- async_set_reset
- async_set_reset_local
- async_set_reset_local_all
- dc_tcl_script_begin and dc_tcl_script_end
- enum
- full_case
- infer_multibit and dont_infer_multibit
- infer_mux
- infer_mux_override
- infer_onehot_mux
- keep_signal_name
- one_cold
- one_hot
- parallel_case
- preserve_sequential
- sync_set_reset
- sync_set_reset_local
- sync_set_reset_local_all

- template
- Directive Support by Pragma Prefix

async_set_reset

When you set the `async_set_reset` directive on a single-bit signal, HDL Compiler searches for a branch that uses the signal as a condition and then checks whether the branch contains an assignment to a constant value. If the branch does, the signal becomes an asynchronous reset or set. Use this directive on single-bit signals.

The syntax is

```
// synopsys async_set_reset "signal_name_list"
```

See Also

- Inferring Latches

async_set_reset_local

When you set the `async_set_reset_local` directive, HDL Compiler treats listed signals in the specified block as if they have the `async_set_reset` directive set.

Attach the `async_set_reset_local` directive to a block label using the following syntax:

```
// synopsys async_set_reset_local block_label "signal_name_list"
```

async_set_reset_local_all

When you set the `async_set_reset_local_all` directive, HDL Compiler treats all listed signals in the specified blocks as if they have the `async_set_reset` directive set. Attach the `async_set_reset_local_all` directive to a block label using the following syntax:

```
// synopsys async_set_reset_local_all "block_label_list"
```

To enable the `async_set_reset_local_all` behavior, you must set `hdlin_ff_always_async_set_reset` to false and use the coding style shown in Example 10-1.

Example 10-1 Coding Style

```
// To enable the async_set_reset_local_all behavior, you must set
// hdlin_ff_always_async_set_reset to false in addition to coding per the
following template.

module m1 (input rst,set,d,d1,clk,clk1, output reg q,q1);
```

```
// synopsys async_set_reset_local_all "sync_rst"
 always @(posedge clk or posedge rst or posedge set) begin :sync_rst
  if (rst)
    q <= 1'b0;
  else if (set)
    q <= 1'b1;
  else q <= d;
end

  always @(posedge clk1 or posedge rst or posedge set)  begin :
default_rst
  if (rst)
    q1 <= 1'b0;
  else if (set)
    q1 <= 1'b1;
  else
    q1 <= d1;
end
endmodule
```

dc_tcl_script_begin and dc_tcl_script_end

You can embed Tcl commands that set design constraints and attributes within the RTL by using the `dc_tcl_script_begin` and `dc_tcl_script_end` directives, as shown in Example 10-2 and Example 10-3.

Example 10-2 Embedding Constraints With // Delimiters

```
...
// synopsys dc_tcl_script_begin
// set_max_area 0.0
// set_max_delay 0.0 port_z
// synopsys dc_tcl_script_end
...
```

Example 10-3 Embedding Constraints With / and */ Delimiters*

```
/* synopsys dc_tcl_script_begin
   set_max_area 10.0
   set_max_delay 5.0 port_z
*/
```

Design Compiler interprets the statements embedded between the `dc_tcl_script_begin` and the `dc_tcl_script_end` directives. If you want to comment out part of your script, use the # comment character.

The following items are not supported in embedded Tcl scripts:

- Hierarchical constraints
- Wildcards

- List commands
- Multiple line commands

Observe the following guidelines when using embedded Tcl scripts:

- Constraints and attributes declared outside a module apply to all subsequent modules declared in the file.
- Constraints and attributes declared inside a module apply only to the enclosing module.
- Any dc_shell scripts embedded in functions apply to the whole module.
- Include only commands that set constraints and attributes. Do not use action commands such as `compile`, `gen`, and `report`. The tool ignores these commands and issues a warning or error message.
- The constraints or attributes set in the embedded script go into effect after the read command is executed. Therefore, variables that affect the read process itself are not in effect before the read.
- Error checking is done after the `read` command finishes. Syntactic and semantic errors in dc_shell strings are reported at this time.
- You can have more than one dc_tcl_script_begin / dc_tcl_script_end pair per file or module. The compiler does not issue an error or warning when it sees more than one pair. Each pair is evaluated and set on the applicable code.
- An embedded dc_shell script does not produce any information or status messages unless there is an error in the script.
- If you use embedded Tcl scripts while running in dc_shell, Design Compiler issues the following error message:

```
Error: Design 'MID' has embedded Tcl commands which are
ignored in dcsh mode. (UIO-162)
```

- Usage of built-in Tcl commands is not recommended.
- Usage of output redirection commands is not recommended.

enum

Use the `enum` directive with the Verilog parameter definition statement to specify state machine encodings.

The syntax of the `enum` directive is

```
// synopsys enum enum_name
```

Example 10-4 shows the declaration of an enumeration of type colors that is 3 bits wide and has the enumeration literals red, green, blue, and cyan with the values shown.

Example 10-4 Enumeration of Type Colors

```
parameter [2:0] // synopsys enum colors
red = 3'b000, green = 3'b001, blue = 3'b010, cyan = 3'b011;
```

The enumeration must include a size (bit-width) specification. Example 10-5 shows an invalid `enum` declaration.

Example 10-5 Invalid enum Declaration

```
parameter /* synopsys enum colors */
red = 3'b000, green = 1;
// [2:0] required
```

Example 10-6 shows a register, a wire, and an input port with the declared type of colors. In each of the following declarations, the array bounds must match those of the enumeration declaration. If you use different bounds, synthesis might not agree with simulation behavior.

Example 10-6 enum Type Declarations

```
reg   [2:0]  /* synopsys enum colors */ counter;
wire  [2:0]  /* synopsys enum colors */ peri_bus;
input [2:0]  /* synopsys enum colors */ input_port;
```

Even though you declare a variable to be of type `enum`, it can still be assigned a bit value that is not one of the enumeration values in the definition. Example 10-7 relates to Example 10-6 and shows an invalid encoding for colors.

Example 10-7 Invalid Bit Value Encoding for Colors

```
counter = 3'b111;
```

Because 111 is not in the definition for colors, it is not a valid encoding. HDL Compiler accepts this encoding, but issues a warning for this assignment.

You can use enumeration literals just like constants, as shown in Example 10-8.

Example 10-8 Enumeration Literals Used as Constants

```
if (input_port == blue)
    counter = red;
```

If you declare a port as a reg and as an enumerated type, you must declare the enumeration when you declare the port. Example 10-9 shows the declaration of the enumeration.

Example 10-9 Enumerated Type Declaration for a Port

```
module good_example (a,b);
  parameter [1:0] /* synopsys enum colors */
     green = 2'b00, white = 2'b11;
  input a;
  output [1:0] /* synopsys enum colors */ b;
  reg [1:0] b;
...
endmodule
```

Example 10-10 declares a port as an enumerated type incorrectly because the enumerated type declaration appears with the reg declaration instead of with the output declaration.

Example 10-10 Incorrect Enumerated Type Declaration for a Port

```
module bad_example (a,b);
  parameter [1:0] /* synopsys enum colors */
     green = 2'b00, white = 2'b11;
  input a;
  output [1:0] b;
  reg [1:0] /* synopsys enum colors */ b;
...
endmodule
```

full_case

This directive prevents HDL Compiler from generating logic to test for any value that is not covered by the case branches and creating an implicit default branch. Set the `full_case` directive on a case statement when you know that all possible branches of the case statement are listed within the case statement. When a variable is assigned in a case statement that is not full, the variable is conditionally assigned and requires a latch.

Warning:

> Marking a case statement as full when it actually is not full can cause the simulation to behave differently from the logic HDL Compiler synthesizes because HDL Compiler does not generate a latch to handle the implicit default condition.

The syntax for the `full_case` directive is

```
// synopsys full_case
```

In Example 10-11, `full_case` is set on the first case statement and `parallel_case` and `full_case` directives are set on the second case statement.

Example 10-11 // synopsys full_case Directives

```
module test (in, out, current_state, next_state);
  input [1:0] in;
  output reg [1:0] out;
  input [3:0] current_state;
  output reg [3:0] next_state;

  parameter state1 = 4'b0001, state2 = 4'b0010,state3 = 4'b0100, state4 =
                    4'b1000;
always @* begin
case (in) // synopsys full_case
0: out = 2;
1: out = 3;
2: out = 0;
endcase
case (1) // synopsys parallel_case full_case
current_state[0] : next_state = state2;
current_state[1] : next_state = state3;
current_state[2] : next_state = state4;
current_state[3] : next_state = state1;
endcase
end
endmodule
```

In the first case statement, the condition in == 3 is not covered. However, the designer knows that in == 3 will never occur and therefore sets the `full_case` directive on the case statement.

In the second case statement, not all 16 possible branch conditions are covered; for example, current_state == 4’b0101 is not covered. However,

- The designer knows that these states will never occur and therefore sets the `full_case` directive on the case statement.
- The designer also knows that only one branch is true at a time and therefore sets the `parallel_case` directive on the case statement.

In the following example, at least one branch will be taken because all possible values of sel are covered, that is, 00, 01, 10, and 11:

```
module mux(a, b,c,d,sel,y);
  input a,b,c,d;
  input [1:0] sel;
  output y;
  reg y;
  always @ (a or b or c or d or sel)
  begin
    case (sel)
    2'b00 : y=a;
    2'b01 : y=b;
    2'b10 : y=c;
    2'b11 : y=d;
    endcase
  end
endmodule
```

In the following example, the case statement is not full:

```
module mux(a, b,c,d,sel,y);
  input a,b,c,d;
  input [1:0] sel;
  output y;
  reg y;
  always @ (a or b or c or d or sel)
  begin
    case (sel)
    2'b00 : y=a;
    2'b11 : y=d;
    endcase
  end
endmodule
```

It is unknown what happens when sel equals 01 and 10. In this case, HDL Compiler generates logic to test for any value that is not covered by the case branches and creates an implicit “default” branch that contains no actions. When a variable is assigned in a case statement that is not full, the variable is conditionally assigned and requires a latch.

infer_multibit and dont_infer_multibit

The HDL Compiler tool can infer registers that have identical structures as multibit components.

The following sections describe how to use the multibit inference directives:

- Using the infer_multibit Directive
- Using the dont_infer_multibit Directive
- Reporting Multibit Components

Multibit sequential mapping does not pull in as many levels of logic as single-bit sequential mapping. Therefore, Design Compiler might not infer complex multibit sequential cells, such as a JK flip-flop.

For more information, see the *Design Compiler Optimization Reference Manual.*

Note:

> The term multibit *component* refers, for example, to an x-bit register in your HDL description. The term multibit library cell refers to a library macro cell, such as a flip-flop cell.

Using the infer_multibit Directive

By default, the `hdlin_infer_multibit` variable is set to the `default_none` value and no multibit cells are inferred unless you set the `infer_multibit` directive on specific components in the Verilog code. This directive gives you control over individual wire and register signals. Example 10-12 shows usage.

Example 10-12 Inferring a Multibit Flip-Flop With the infer_multibit Directive

```
module test (d0, d1, d2, rst, clk, q0, q1, q2);
  parameter d_width = 8;

  input [d_width-1:0] d0, d1, d2;
  input clk, rst;
  output [d_width-1:0] q0, q1, q2;
  reg [d_width-1:0] q0, q1, q2;

  //synopsys infer_multibit "q0"
  always @(posedge clk)begin
    if (!rst) q0 <= 0;
    else q0 <= d0;
  end

  always @(posedge clk or negedge rst)begin
    if (!rst) q1 <= 0;
    else q1 <= d1;
  end

  always @(posedge clk or negedge rst)begin
    if (!rst)  q2 <= 0;
    else q2 <= d2;
  end

endmodule
```

Example 10-13 shows the inference report.

Example 10-13 Multibit Inference Report

```
Inferred memory devices in process
```

```
        in routine test line 10 in file
                '/.../test.v'.
===============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|       q0_reg        | Flip-flop |   8   |  Y  | Y  | N  | N  | N  | N  | N  |
===============================================================================

Inferred memory devices in process
        in routine test line 16 in file
                '/.../test.v'.
===============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|       q1_reg        | Flip-flop |   8   |  Y  | N  | Y  | N  | N  | N  | N  |
===============================================================================
Inferred memory devices in process
        in routine test line 21 in file
                '/.../test.v'.
===============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|       q2_reg        | Flip-flop |   8   |  Y  | N  | Y  | N  | N  | N  | N  |
===============================================================================
Compilation completed successfully.
```

The MB column of the inference report indicates if a component is inferred as a multibit component. This report shows the q0_reg register is inferred as a multibit component. The q1_reg and q2_reg registers are not inferred as multibit components.

Using the dont_infer_multibit Directive

If you set the `hdlin_infer_multibit` variable to the `default_all` value, all bused registers are inferred as multibit components. Use the `dont_infer_multibit` directive to prevent multibit inference.

Example 10-14 Using the dont_infer_multibit Directive

```
// the hdlin_infer_multibit variable is set to the default_all value
module test (d0, d1, d2, rst, clk, q0, q1, q2);
  parameter d_width = 8;

  input [d_width-1:0] d0, d1, d2;
  input clk, rst;
  output [d_width-1:0] q0, q1, q2;
  reg [d_width-1:0] q0, q1, q2;

  always @(posedge clk)begin
    if (!rst) q0 <= 0;
    else q0 <= d0;
  end

  //synopsys dont_infer_multibit "q1"
  always @(posedge clk or negedge rst)begin
    if (!rst) q1 <= 0;
    else q1 <= d1;
  end

  always @(posedge clk or negedge rst)begin
    if (!rst)  q2 <= 0;
    else q2 <= d2;
  end

endmodule
```

Example 10-15 shows the multibit inference report.

Example 10-15 Multibit Inference Report

```
Inferred memory devices in process
        in routine test line 10 in file
                '/.../test.v'.
===============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|       q0_reg        | Flip-flop |   8   |  Y  | Y  | N  | N  | N  | N  | N  |
===============================================================================

Inferred memory devices in process
        in routine test line 16 in file
                '/.../test.v'.
===============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|       q1_reg        | Flip-flop |   8   |  Y  | N  | Y  | N  | N  | N  | N  |
===============================================================================

Inferred memory devices in process
        in routine test line 21 in file
                '/.../test.v'.
===============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|       q2_reg        | Flip-flop |   8   |  Y  | Y  | Y  | N  | N  | N  | N  |
===============================================================================
Presto compilation completed successfully.
```

Reporting Multibit Components

The `report_multibit` command reports all multibit components in the current design. The report, viewable before and after compile, shows the multibit group name and what cells implement each bit.

Example 10-16 shows a multibit component report.

Example 10-16 Multibit Component Report

```
****************************************
Report : multibit
Design : test
Version: F-2011.09
Date   : Thu Aug  4 21:42:30 2011
****************************************

Attributes:
    b - black box (unknown)
    h - hierarchical
    n - noncombinational
    r - removable
    u - contains unmapped logic

Multibit Component : q0_reg
Cell                    Reference      Library         Area   Width  Attributes
--------------------------------------------------------------------------------
q0_reg[7]               **SEQGEN**                     0.00   1      n, u
q0_reg[6]               **SEQGEN**                     0.00   1      n, u
q0_reg[5]               **SEQGEN**                     0.00   1      n, u
q0_reg[4]               **SEQGEN**                     0.00   1      n, u
q0_reg[3]               **SEQGEN**                     0.00   1      n, u
q0_reg[2]               **SEQGEN**                     0.00   1      n, u
q0_reg[1]               **SEQGEN**                     0.00   1      n, u
q0_reg[0]               **SEQGEN**                     0.00   1      n, u
--------------------------------------------------------------------------------
Total 8 cells                                          0.00   8
```

The multibit group name for registers is set to the name of the bus. In the cell names of the multibit registers with consecutive bits, a colon separates the outlying bits.

If the colon conflicts with the naming requirements of your place-and-route tool, you can change the colon to another delimiter by using the `bus_range_separator_style` variable.

For multibit library cells with nonconsecutive bits, a comma separates the nonconsecutive bits. This delimiter is controlled by the `bus_multiple_separator_style` variable. For example, a 4-bit banked register that implements bits 0, 1, 2, and 5 of bus data_reg is named data_reg [0:2,5].

infer_mux

Use the infer_mux directive to infer MUX_OP cells for a specific case or if statement, as shown in the following RTL code:

```
always@(SEL) begin
case (SEL)  // synopsys infer_mux
   2'b00: DOUT <= DIN[0];
   2'b01: DOUT <= DIN[1];
   2'b10: DOUT <= DIN[2];
   2'b11: DOUT <= DIN[3];
endcase
```

You must use a simple variable as the control expression; for example, you can use the input "A" but not the negation of input "A". If statements have special coding considerations. For more information, see MUX_OP Inference and Using the case Statement for MUX_OP Inference.

infer_mux_override

Use the infer_mux_override directive to infer MUX_OP cells for a specific case or if statement regardless of the settings of the following variables:

- hdlin_infer_mux
- hdlin_mux_oversize_ratio
- hdlin_mux_size_limit
- hdlin_mux_size_min

The tool marks the MUX_OP cells inferred by this directive with the size_only attribute to prevent logic decomposition during optimization. This directive infers MUX_OP cells even if the cells cause loss of resource sharing.

For example,

```
module test (input [1:0] SEL,
             input [3:0] DIN,
             output logic DOUT);
always@(SEL or DIN) begin
case (SEL)  // synopsys infer_mux_override
   2'b00: DOUT <= DIN[0];
   2'b01: DOUT <= DIN[1];
   2'b10: DOUT <= DIN[2];
   2'b11: DOUT <= DIN[3];
endcase
end
endmodule
```

infer_onehot_mux

Use the `infer_onehot_mux` directive to map combinational logic to one-hot multiplexers in the logic library. For details, see One-Hot Multiplexer Inference.

keep_signal_name

Use the `keep_signal_name` directive to provide HDL Compiler with guidelines for preserving signal names.

The syntax is

```
// synopsys keep_signal_name "signal_name_list"
```

Set the `keep_signal_name` directive on a signal before any reference is made to that signal; for example, one methodology is to put the directive immediately after the declaration of the signal.

See Also

- Keeping Signal Names

one_cold

A one-cold implementation indicates that all signals in a group are active-low and that only one signal can be active at a given time. Synthesis implements the `one_cold` directive by omitting a priority circuit in front of the flip-flop. Simulation ignores the directive. The `one_cold` directive prevents Design Compiler from implementing priority-encoding logic for the set and reset signals. Attach this directive to set or reset signals on sequential devices, using the following syntax:

```
// synopsys one_cold signal_name_list
```

See Also

- D Latch With Asynchronous Set and Reset: Use hdlin_latch_always_async_set_reset

one_hot

A one-hot implementation indicates that all signals in a group are active-high and that only one signal can be active at a given time. Synthesis implements the `one_hot` directive by omitting a priority circuit in front of a flip-flop. Simulation ignores the directive. The `one_hot` directive prevents Design Compiler from implementing priority-encoding logic for the set and reset signals. Attach this directive to set or reset signals on sequential devices, using the following syntax:

```
// synopsys one_hot signal_name_list
```

See Also

- D Flip-Flop With Asynchronous Set and Reset
- JK Flip-Flop With Synchronous Set and Reset Using sync_set_reset

parallel_case

Set the `parallel_case` directive on a case statement when you know that only one branch of the case statement will be true at a time. This directive prevents HDL Compiler from building additional logic to ensure the first occurrence of a true branch is executed if more than one branch were true at one time.

Warning:

> Marking a case statement as parallel when it actually is not parallel can cause the simulation to behave differently from the logic HDL Compiler synthesizes because HDL Compiler does not generate priority encoding logic to make sure that the branch listed first in the case statement takes effect.

The syntax for the `parallel_case` directive is

```
// synopsys parallel_case
```

Use the `parallel_case` directive immediately after the case expression. In Example 10-17, the states of a state machine are encoded as a one-hot signal; the designer knows that only one branch is true at a time and therefore sets the `synopsys parallel_case` directive on the case statement.

Example 10-17 parallel_case Directives

```
reg [3:0] current_state, next_state;
parameter state1 = 4’b0001, state2 = 4’b0010,
      state3 = 4’b0100, state4 = 4’b1000;
case (1)//synopsys parallel_case
      current_state[0] : next_state = state2;
      current_state[1] : next_state = state3;
      current_state[2] : next_state = state4;
      current_state[3] : next_state = state1;
endcase
```

When a case statement is not parallel (more than one branch evaluates to true), priority encoding is needed to ensure that the branch listed first in the case statement takes effect.

The following table summarizes the types of case statements.

Case statement description	Additional logic
Full and parallel	No additional logic is generated.
Full but not parallel	Priority-encoded logic: HDL Compiler generates logic to ensure that the branch listed first in the case statement takes effect.
Parallel but not full	Latches created: HDL Compiler generates logic to test for any value that is not covered by the case branches and creates an implicit “default” branch that requires a latch.
Not parallel and not full	Priority-encoded logic: HDL Compiler generates logic to make sure that the branch listed first in the case statement takes effect. Latches created: HDL Compiler generates logic to test for any value that is not covered by the case branches and creates an implicit “default” branch that requires a latch.

preserve_sequential

The `preserve_sequential` directive allows you to preserve specific cells that would otherwise be optimized away by HDL Compiler. See Keeping Unloaded Registers.

sync_set_reset

Use the `sync_set_reset` directive to infer a D flip-flop with a synchronous set/reset. When you compile your design, the SEQGEN inferred by HDL Compiler will be mapped to a flip-flop in the logic library with a synchronous set/reset pin, or Design Compiler will use a regular D flip-flop and build synchronous set/reset logic in front of the D pin. The choice depends on which method provides a better optimization result.

It is important to use the `sync_set_reset` directive to label the set/reset signal because it tells Design Compiler that the signal should be kept as close to the register as possible during mapping, preventing a simulation/synthesis mismatch which can occur if the set/reset signal is masked by an X during initialization in simulation.

When a single-bit signal has this directive set to `true`, HDL Compiler checks the signal to determine whether it synchronously sets or resets a register in the design. Attach this directive to single-bit signals. Use the following syntax:

```
//synopsys sync_set_reset "signal_name_list"
```

For an example of a D flip-flop with a synchronous set signal that uses the `sync_set_reset` directive, see D Flip-Flop With Synchronous Set: Use sync_set_reset. For an example of a JK flip-flop with synchronous set and reset signals that uses the `sync_set_reset` directive, see JK Flip-Flop With Synchronous Set and Reset Using sync_set_reset.

For an example of a D flip-flop with a synchronous reset signal that uses the `sync_set_reset` directive, see D Flip-Flop With Synchronous Reset: Use sync_set_reset. For an example of multiple flip-flops with asynchronous and synchronous controls, see Multiple Flip-Flops With Asynchronous and Synchronous Controls.

sync_set_reset_local

The `sync_set_reset_local` directive instructs HDL Compiler to treat signals listed in a specified block as if they have the `sync_set_reset` directive set to true.

Attach this directive to a block label, using the following syntax:

```
 //synopsys sync_set_reset_local block_label "signal_name_list"
```

Example 10-18 shows the usage.

Example 10-18 sync_set_reset_local Usage

```
module m1 (input d1,d2,clk, set1, set2, rst1, rst2, output reg q1,q2);

// synopsys sync_set_reset_local sync_rst "rst1"
//always@(posedge clk or negedge rst1)
  always@(posedge clk )
    begin: sync_rst
      if(~rst1)
        q1 <= 1'b0;
      else if (set1)
        q1 <= 1'b1;
      else
      q1 <= d1;
    end

  always@(posedge clk)
    begin: default_rst
      if(~rst2)
        q2 <= 1'b0;
      else if (set2)
        q2 <= 1'b1;
      else
       q2 <= d2;
    end

endmodule
```

sync_set_reset_local_all

The `sync_set_reset_local_all` directive instructs HDL Compiler to treat all signals listed in the specified blocks as if they have the `sync_set_reset` directive set to true.

Attach this directive to a block label, using the following syntax:

```
// synopsys sync_set_reset_local_all "block_label_list"
```

Example 10-19 shows usage.

Example 10-19 sync_set_reset_local_all Usage

```
module m2 (input d1,d2,clk, set1, set2, rst1, rst2, output reg q1,q2);

// synopsys sync_set_reset_local_all sync_rst
//always@(posedge clk or negedge rst1)
  always@(posedge clk )
    begin: sync_rst
      if(~rst1)
        q1 <= 1'b0;
      else if (set1)
        q1 <= 1'b1;
      else
        q1 <= d1;
    end

  always@(posedge clk)
    begin: default_rst
      if(~rst2)
        q2 <= 1'b0;
      else if (set2)
        q2 <= 1'b1;
      else
        q2 <= d2;
      end

endmodule
```

template

The `template` directive saves an analyzed file and does not elaborate it. Without this directive, the analyzed file is saved and elaborated. If you use this directive and your design contains parameters, the design is saved as a template. Example 10-20 shows usage.

Example 10-20 template Directive

```
module template (a, b, c);
  input a, b, c;
  // synopsys template
  parameter width = 8;
```

```
.
.
.
endmodule
```

See Also

- Parameterized Designs

Directive Support by Pragma Prefix

Not all pragma prefixes support all directives:

- The `synopsys` prefix is intended for directives specific to HDL Compiler. The tool issues an error message if an unknown directive is encountered.
- The `pragma` and `synthesis` prefixes are intended for industry-standard directives. The tool ignores any unsupported directives to allow for directives intended for other tools. Directives specific to HDL Compiler are not supported.

Table 10-1 shows how each directive is handled by each pragma prefix.

Table 10-1 Directive Support by Pragma Prefix

Directive	// synopsys, // $s	// pragma	// synthesis
`translate_off` `translate_on`	Used	Used	Used

`dc_tcl_script_begin`
`dc_tcl_script_end`
`dc_script_begin`
`dc_script_end`
	Used	Ignored	Ignored

`async_set_reset`
`async_set_reset_local`
`async_set_reset_local_all`
	Used	Ignored	Ignored

`enum`	Used	Ignored	Ignored
`full_case` `parallel_case`	Used	Ignored	Ignored
`infer_multibit` `dont_infer_multibit`	Used	Ignored	Ignored

Table 10-1 Directive Support by Pragma Prefix (Continued)

Directive	// synopsys, // $s	// pragma	// synthesis
`infer_mux` `infer_mux_override`	Used	Ignored	Ignored
`infer_onehot_mux`	Used	Ignored	Ignored
`keep_signal_name`	Used	Ignored	Ignored
`one_cold` `one_hot`	Used	Ignored	Ignored
`preserve_sequential`	Used	Ignored	Ignored

`sync_set_reset`
`sync_set_reset_local`
`sync_set_reset_local_all`
	Used	Ignored	Ignored

`template`	Used	Ignored	Ignored
Any unknown directive	Error	Ignored	Ignored

SystemVerilog Attributes

In SystemVerilog, *attributes* allow properties about objects, statements, and groups of statements in the RTL to be communicated to tools reading the RTL. HDL Compiler supports SystemVerilog attributes by re-applying them as synthesis attributes, so they become accessible as if set by the `set_attribute` command.

The following sections describe SystemVerilog attribute support in HDL Compiler:

- Using SystemVerilog Attributes in Synthesis
- Supported Attributes
- Supported RTL Constructs

See Also

- *IEEE Std 1800-2012* section 5.12 for details on SystemVerilog attributes

Using SystemVerilog Attributes in Synthesis

SystemVerilog attributes are defined in the RTL as prefixes preceding the RTL construct they are attached to. Their value type is inferred as Boolean, integer, or string, based on the value provided:

```
(* attr_name *)  // no value is an implicit Boolean value of true
(* attr_name = integer_value *)
(* attr_name = "string_value" *)
```

By default, HDL Compiler ignores SystemVerilog attributes. The contents of the attributes are not parsed or checked for syntax.

To instruct the tool to read and re-apply them as synthesis attributes to the design objects created during RTL read, set the following variable:

```
dc_shell> set_app_var hdlin_sv_enable_rtl_attributes true
```

When this feature is enabled, SystemVerilog attributes are parsed by HDL Compiler and applied as synthesis attributes to the corresponding design objects. Attributes with incorrect syntax are flagged as errors.

Supported Attributes

You can set application (built-in) attributes (such as `dont_touch` or `size_only`) as well as user-defined attributes:

- If the named attribute is an application attribute, that attribute is set to the specified value:

  ```
  (* dont_touch *)    // application attribute
  core UCORE (.CLK(CLK), ...)
  ```

  The value type (Boolean, integer, or string) must match that of the application attribute, and the object class must be supported by that attribute.

- If the named attribute is not an application attribute, the tool applies its specified value as a user-defined attribute:

  ```
  (* my_interface_type = "TX" *)    // user-defined attribute
  tx UTXBLOCK1 (.CLK(CLK), ...)
  ```

  You do not need to predefine the attributes using the `define_user_attribute` command; HDL Compiler defines them as needed using the object class and value type derived from the RTL.

  The first definition of a user-defined attribute for an object class defines its value type. Subsequent applications of that attribute for that object class must be of the same type.

Note that Boolean SystemVerilog attributes always have an implicit value of `true`. There is no way to set a Boolean synthesis attribute to a value of `false` using SystemVerilog attributes.

Supported RTL Constructs

SystemVerilog attributes can be defined on a variety of language elements. HDL Compiler supports the following subset described in this section.

When SystemVerilog attribute support is enabled, the tool warns of ignored attributes applied to unsupported RTL constructs. For example,

```
Warning:  ./rtl/top.sv:17: The construct 'statement attribute' is not
supported in synthesis; it is ignored. (VER-708)
```

The supported RTL constructs are:

- Designs (Modules)
- Ports
- Cells (Instantiations)
- Pins
- Inferred Register Cells (Sequential Processes)

Designs (Modules)

A SystemVerilog attribute on a module in the RTL sets that attribute on the corresponding design in the Design Compiler database.

RTL:

```
(* my_design_type = "IP", dont_touch *)
module IP_block (
  ...
endmodule
```

Synthesis:

```
dc_shell> report_attribute [get_designs IP_block]
...

Design      Object         Type      Attribute Name        Value
--------------------------------------------------------------------
IP_block    sub            design    dont_touch            true
IP_block    sub            design    my_design_type        IP
```

Ports

A SystemVerilog attribute on a module port in the RTL sets that attribute on the corresponding design port in the Design Compiler database.

RTL:

```
module top (RXCLK, RXDATA, TXCLK, TXDATA, ...);
  (* my_port_type = "RX" *)   input RXCLK;
  (* my_port_type = "RX" *)   input [31:0]  RXDATA;

  (* my_port_type = "TX" *)   input TXCLK;
  (* my_port_type = "TX" *)   input [31:0]  TXDATA;

  (* my_port_type = "test_data" *)
                              input [2:0]   scanin;
  (* my_port_type = "test_data" *)
                              output [2:0]  scanout;
...
endmodule
```

Synthesis:

```
dc_shell> get_ports * -filter {my_port_type == "test_data"}
{scanin[2] scanin[1] scanin[0] scanout[2] scanout[1] scanout[0]}
dc_shell> get_ports *CLK* -filter {my_port_type == "TX"}
{TXCLK}
```

Cells (Instantiations)

A SystemVerilog attribute on a cell instantiation in the RTL sets that attribute on the corresponding port in the Design Compiler database. All cell types (hierarchical, logic library, macro, and black-box) are supported.

RTL:

```
(* dont_touch *)
  spare_cells USPARE (.CLK);

(* my_bank_num = 0 *) mem16x32 UMEM16x32_0 (...);
(* my_bank_num = 1 *) mem16x32 UMEM16x32_1 (...);
(* my_bank_num = 2 *) mem16x32 UMEM16x32_2 (...);
(* my_bank_num = 3 *) mem16x32 UMEM16x32_3 (...);
```

Synthesis:

```
dc_shell> get_attribute [get_cells USPARE] dont_touch
true
dc_shell> get_cells * -filter {my_bank_num >= 2 && my_bank_num <= 3}
{UMEM16x32_2 UMEM16x32_3}
```

Pins

A SystemVerilog attribute on a pin within a cell instantiation in the RTL sets that attribute on the corresponding instance pin in the Design Compiler database.

RTL:

```
PLL UPLL1 (.REFCLK(CLK1), .FDBCK(CLK1_FDBCK),
  (* my_pll_mult = 2 *) .CLKOUT2(PLLCLK1_X2),
  (* my_pll_mult = 4 *) .CLKOUT4(PLLCLK1_X4));
PLL UPLL2 (.REFCLK(CLK2), .FDBCK(CLK2_FDBCK),
  (* my_pll_mult = 2 *) .CLKOUT2(PLLCLK2_X2),
  (* my_pll_mult = 4 *) .CLKOUT4(PLLCLK2_X4));
```

Synthesis:

```
dc_shell> get_pins {*PLL*/*} -filter {my_pll_mult == 2}
{UPLL1/CLKOUT2 UPLL2/CLKOUT2}
dc_shell> get_pins {*PLL*/*} -filter {my_pll_mult == 4}
{UPLL1/CLKOUT4 UPLL2/CLKOUT4}
```

Inferred Register Cells (Sequential Processes)

A SystemVerilog attribute on a sequential process in the RTL sets that attribute on the corresponding GTECH sequential cells inferred by that process in the Design Compiler database. Combinational logic associated with the process is not affected.

Note:

During compile, only attributes kept persistent by the tool (such as `dont_touch` or `size_only`) will exist on the resulting mapped sequential cells.

RTL:

```
  (* size_only *)
  always @(posedge clk)
    counter <= (counter + write - read);
```

Synthesis:

```
dc_shell> report_attribute [get_cells * -filter {size_only == true}]
...

Design    Object          Type     Attribute Name       Value
-------------------------------------------------------------------
top       counter_reg[3]  cell     size_only            true
top       counter_reg[2]  cell     size_only            true
top       counter_reg[1]  cell     size_only            true
top       counter_reg[0]  cell     size_only            true
```

11

Troubleshooting Guidelines

To troubleshoot your designs, you can use the basic guidelines described in this section.

- Code Expansion for Macros and Conditional Directives
- Minimizing Mismatches Between Simulation and Synthesis
- Data Type Declarations
- Synthesizable do...while Loops
- Troubleshooting generate Loops
- Assertions in Synthesis
- Other Troubleshooting Guidelines

Code Expansion for Macros and Conditional Directives

You can use macros and conditional compilation directives to automate complex tasks and reduce coding time. However, using macros and the directives makes the code complex and difficult to debug. To help debug such SystemVerilog designs, you can generate an expanded version of the original RTL by using the code expansion feature. When this feature is enabled, the tool processes all conditional compilation directives and expands all macro invocations.

The following topics describe the guidelines for code expansion and the RTL examples:

- Code Expansion Guidelines
- Code Expansion Example 1
- Code Expansion Example 2

See Also

- Querying Information about RTL Preprocessing

Code Expansion Guidelines

To enable code expansion, set the `hdlin_sv_tokens` variable to `true`. The default is `false`. When the feature is enabled, the tool generates expanded files, also called tokens files, by capturing the exact token stream seen by the parser after preprocessing. The tokens files are named tokens.1.sv, tokens.2.sv, tokens.3.sv, and so forth in the order they are written out. All tokens files are written in the current working directory. When encountering an error during parsing, the tool can create an incomplete or empty tokens file.

Follow these guidelines when you use code expansion:

- When the tool detects errors, it generates no output but reports the errors by default. When you set the `hdlin_sv_tokens` variable to `true`, the tool generates an output in spite of errors.
- In the expanded output file, the `` `line `` directive specifies the line number. For more information about the `` `line `` directive, see the *IEEE Std 1364-2005*.
- If the tool can read the original RTL, it can read the expanded file.
- You can write out multiple tokens files in one Design Compiler session.

The following limitations apply:

- The code expansion feature applies to SystemVerilog only; that is, it works with the `analyze -format sverilog`, `read_file -format sverilog`, and `read_sverilog` commands.
- If an input file is encrypted (including an `` `include `` file), the tool does not write out the tokens file.

As shown in the following example, the tool produces a tokens file so that you can see the stream of tokens that are parsed before an error occurs. This information can help you debug erroneous RTL code. Because q` in the first line of the msf_in_lib module causes an error, the corresponding tokens file is incomplete.

- Erroneous RTL code

```
`define MSFF(q,i,clk,rst)    \
msf_in_lib q`_reg (.o(q),    \
                   .clk(clk),\
                   .d(i),    \
                   .rst(rst));
module test (output o1, input i1,clk,rst);
   `MSFF(o1,i1,clk,rst)
endmodule
```

- Incomplete tokens file

```
`line 6  "err.v" 0

`line 7  "err.v" 0
                      module test (output o1, input i1,clk,rst);
`line 8  "err.v" 0
                      msf_in_lib o1
```

Code Expansion Example 1

This example shows an RTL design that contains macros, a script that uses the `hdlin_sv_tokens` variable to generate an expanded output file, and the contents of the expanded file. This design contains no errors.

- RTL design

```
`ifndef SYNTHESIS
 module my_testbench ();
/* Testbench goes in here. */
 endmodule
`endif

`ifndef GATES
module TOP_syn (a,clk, o1);
input a, clk;
`ifdef NOT
   output o1;
   o1=(!a);
`elsif FF
   output logic o1;
   always_ff @(posedge clk)
   o1= a;
`else
   output o1;
   logic temp;
   assign temp = a;
   assign o1 = temp;
 `endif
endmodule
`else
   `include "netlist_wrap.sv"
   `include "compiled_gates.v"
`endif

`define DUT(mod) \
`ifndef GATES \
   mod``_syn \
`else \
   mod``_svsim \
`endif
```

- Script

```
dc_shell> set hdlin_sv_tokens true
dc_shell> analyze -format sverilog ex1.v
```

- Excerpt of the expended file

```
`line 1   "ex1.sv" 0
`line 6   "ex1.sv" 0
`line 7   "ex1.sv" 0
                                module TOP_syn (a,clk, o1);
`line 8   "ex1.sv" 0
                                input a, clk;
`line 9   "ex1.sv" 0
`line 12  "ex1.sv" 0
`line 17  "ex1.sv" 0
                                output o1;
`line 18  "ex1.sv" 0
                                logic temp;
`line 19  "ex1.sv" 0
                                assign temp = a;
`line 20  "ex1.sv" 0
                                assign o1 = temp;
`line 22  "ex1.sv" 0
                                endmodule
```

Code Expansion Example 2

In this example, a large macro definition spans multiple lines. Using the code expansion feature not only enhances code legibility but also simplifies RTL debugging. This design contains no errors.

- RTL design

```
`define make_reg(q,i,clk,en,rst,rstd) \
logic i_``q ; \
logic en_``q ;\
always_comb   \
   if (rst) i_``q = rstd;  \
   else     i_``q = i;     \
assign en_``q = rst | en ; \
my_lat myreg``q (.o(q),    \
                 .clk(clk),\
                 .d(i_``q),\
                 .en(en_``q));
module test(
   output logic out1,
   input  logic in1,
   input  logic clk, en, rst
);
`make_reg(out1,in1,clk,en,rst,'b0)
endmodule
```

- Tokens file

In the tokens file, the descriptions of the `always_comb` block, the `assign` statement, the `make_reg` macro, and more design elements are in line 15. When the tool detects an error in the macro, it points to line 15 rather than the exact code that causes the error. To simplify RTL debugging, the tool breaks up the single-line macro description into many lines, as shown in the following tokens file:

```
`line 12  "ex2.sv" 0
                                 module test(output logic out1,
`line 13  "ex2.sv" 0
                                 input logic in1,
`line 14  "ex2.sv" 0
                                 input logic clk, en,rst);
`line 15  "ex2.sv" 0
                                 logic i_out1 ;
`line 15  "ex2.sv" 0
                                 logic en_out1 ;
`line 15  "ex2.sv" 0
                                 always_comb
`line 15  "ex2.sv" 0
                                 if (rst) i_out1 = 'b0;
`line 15  "ex2.sv" 0
                                 else i_out1 = in1;
`line 15  "ex2.sv" 0
                                 assign en_out1 = rst | en ;
`line 15  "ex2.sv" 0
                                 my_lat myregout1 (.o(out1),
`line 15  "ex2.sv" 0
                                 .clk(clk),
`line 15  "ex2.sv" 0
                                 .d(i_out1),
`line 15  "ex2.sv" 0
                                 .en(en_out1));
`line 16  "ex2.sv" 0
                                 endmodule
```

Minimizing Mismatches Between Simulation and Synthesis

You can use the coding styles described in these topics to minimize mismatches between synthesis and simulation:

- Preventing case Mismatches
- Using Void Functions Instead of Tasks Inside always_comb
- Conversion Between Two-State and Four-State Variables

Preventing case Mismatches

Table 11-1 shows the SystemVerilog `unique` and `priority` constructs and the Verilog equivalency `full_case` and `parallel_case` compiler directives.

Table 11-1 SystemVerilog and Verilog Equivalency

SystemVerilog	Verilog equivalency
`unique case` without the default	`full_case` and `parallel_case`
`priority case` without the default	`full_case`
`unique case` with the default	`parallel_case`
`priority case` with the default	No compiler directive

To prevent `case` mismatches between simulation and synthesis, you should follow these guidelines:

- Using unique Instead of full_case and parallel_case
- Using priority Instead of full_case

Using unique Instead of full_case and parallel_case

In SystemVerilog, a `case` statement qualified with the `unique` keyword without the default is the same as the Synopsys `full_case` and `parallel_case` compiler directives. You should use the `unique` keyword instead of the compiler directives to avoid simulation and synthesis mismatches. If you mix both the compiler directives and the `unique case` construct, the tool issues a VER-517 error message.

- Example—SystemVerilog `case` statement qualified with the `unique` keyword

```
typedef struct {
   logic a_sel;
   logic b_sel;
} priority_sel;

module unique_case_without_default_struct (
   input priority_sel one_hot_sel,
   output logic a_hi, logic b_hi
);
always_comb
unique case (1'b1)
   one_hot_sel.a_sel : begin a_hi = '1; b_hi = '0; end
   one_hot_sel.b_sel : begin a_hi = '0; b_hi = '1; end
endcase
endmodule
```

- Example—SystemVerilog `full_case` and `parallel_case` directives

```
typedef struct {
   logic a_sel;
   logic b_sel;
} priority_sel;

module full_case_parallel_case_struct(
   input priority_sel one_hot_sel,
   output logic a_hi, b_hi
);

always_comb
case (1'b1)    // synopsys full_case parallel_case
   one_hot_sel.a_sel : begin a_hi = '1; b_hi = '0; end
   one_hot_sel.b_sel : begin a_hi = '0; b_hi = '1; end
endcase
endmodule
```

- Example—Verilog 2001 `full_case` and `parallel_case` directives

```
module full_case_parallel_case_struct(
   input a_sel, b_sel,
   output reg a_hi, b_hi
);

always@(*)
case (1'b1)    // synopsys full_case parallel_case
   a_sel : begin a_hi = 1'b1; b_hi = 1'b0; end
   b_sel : begin a_hi = 1'b0; b_hi = 1'b1; end
endcase
endmodule
```

The preceding examples generate the same netlist and statistics for the `case` statement:

- Netlist

```
module full_case_parallel_case_struct ( a_sel, b_sel, a_hi, b_hi );
   input a_sel, b_sel;
   output a_hi, b_hi;
   wire   a_hi, b_hi;
   assign a_hi = a_sel;
   assign b_hi = b_sel;
endmodule
```

- Statistics

```
===============================================
|           Line          |  full/ parallel  |
===============================================
|            9            |    user/user     |
===============================================
HDLC compilation completed successfully.
```

Using priority Instead of full_case

In SystemVerilog, a `case` statement qualified with the `priority` keyword without the default is the same as the Synopsys `full_case` compiler directive. You should use the `priority` keyword instead of the compiler directive to avoid simulation and synthesis mismatches. If you mix the compiler directive and the `priority case` construct, the tool issues an ELAB-909 warning message.

- Example—SystemVerilog `case` statement qualified with the `priority` keyword

```
typedef struct {
   logic a_sel;
   logic b_sel;
} priority_sel;

module priority_case_without_default_struct(
   input priority_sel one_hot_sel,
   output logic a_hi, b_hi
);

always_comb
priority case (1'b1)
   one_hot_sel.a_sel : begin a_hi = '1; b_hi = '0; end
   one_hot_sel.b_sel : begin a_hi = '0; b_hi = '1; end
endcase
endmodule
```

- Example—SystemVerilog `full_case` directive

```
typedef struct {
   logic a_sel;
   logic b_sel;
} priority_sel;

module full_case_struct (
   input priority_sel one_hot_sel,
   output logic a_hi, b_hi
);

always_comb
case (1'b1)    // Synopsys full_case
   one_hot_sel.a_sel : begin a_hi = '1; b_hi = '0; end
   one_hot_sel.b_sel : begin a_hi = '0; b_hi = '1; end
endcase
endmodule
```

- Example—Verilog 2001 `full_case` directive

```
module full_case_struct(
   input a_sel, b_sel,
   output reg a_hi, b_hi
);

always@(*)
case (1'b1)    // Synopsys full_case
   a_sel : begin a_hi = 1'b1; b_hi = 1'b0; end
   b_sel : begin a_hi = 1'b0; b_hi = 1'b1; end
endcase
endmodule
```

The preceding examples generate the same netlist and statistics for the `case` statement:

- Netlist

```
module full_case_struct ( a_sel, b_sel, a_hi, b_hi );
   input a_sel, b_sel;
   output a_hi, b_hi;
   wire   a_hi;
   assign a_hi = a_sel;
   IV U4 ( .A(a_hi), .Z(b_hi) );
endmodule
```

- Statistics

```
===============================================
|           Line           |  full/ parallel  |
===============================================
|            10            |    user/user     |
===============================================
HDLC compilation completed successfully.
```

Using Void Functions Instead of Tasks Inside always_comb

The *IEEE Std 1800-2012* states that `always_comb` is sensitive to changes within the contents of a function, whereas `always @*` is only sensitive to changes to the arguments of a function. It does not define the behavior of a task inside an `always_comb` block or the sensitivity list. This can cause a mismatch between simulation and synthesis. To prevent such mismatches, use void functions instead of tasks inside an `always_comb` block.

The following example shows a design containing a task in an `always_comb` block, the testbench for the design, the GTECH netlist, and the simulation log:

- RTL containing a task in the `always_comb` block

```
module comb1(
   input logic a, b ,c,
   output logic [1:0] y
);

always_comb orf1(a);
function void orf1 (a);
   y[0] = a | b | c;
endfunction

always_comb ort1 (a);
task ort1 (a);
   y[1] = a | b | c;
endtask
endmodule
```

- Testbench

```
module comb1_tb(
   output logic a, b, c
);

initial
begin
       a = 0; b = 0; c = 0;
   #10 a = 0; b = 0; c = 1;
   #10 a = 0; b = 1; c = 0;
   #10 a = 0; b = 1; c = 1;
   #10 a = 1; b = 0; c = 0;
   #10 a = 1; b = 0; c = 1;
   #10 a = 1; b = 1; c = 0;
   #10 a = 1; b = 1; c = 1;
end
endmodule

module top;
wire a_w, b_w, c_w;
wire y1_w, y0_w ;
comb1 u1(a_w, b_w, c_w, {y1_w, y0_w});
comb1_tb u2(a_w, b_w, c_w);

initial
begin
   $display("\t\tTime A B C Y1 Y0\n");
   $monitor($time,,,,a_w,,,,b_w,,,,c_w,,,,y1_w,,,,y0_w);
end
endmodule
```

- GTECH netlist

```
module comb1 ( a, b, c, y );
   output [1:0] y;
   input a, b, c;
   wire   N0, N1;
   GTECH_OR2 C7  ( .A(N0), .B(c), .Z(y[0]) );
   GTECH_OR2 C8  ( .A(a), .B(b), .Z(N0) );
   GTECH_OR2 C9  ( .A(N1), .B(c), .Z(y[1]) );
   GTECH_OR2 C10 ( .A(a), .B(b), .Z(N1) );
endmodule
```

- VCS simulation log

```
   ...
                   Time   A   B   C  Y1  Y0
                      0   0   0   0   0   0
                     10   0   0   1   0   1
                     20   0   1   0   0   1
                     30   0   1   1   0   1
                     40   1   0   0   1   1
                     50   1   0   1   1   1
                     60   1   1   0   1   1
                     70   1   1   1   1   1
              V C S   S i m u l a t i o n   R e p o r t
   ...
```

As shown in the netlist, y[0] and y[1] are outputs of the C7 and C9 OR gates. The simulation log shows that

- y[0] changes to logic 1 when any of the inputs changes to logic 1.
- y[1] changes to logic 1 only when the A input changes to logic 1, not sensitive to changes of the B and C inputs.

Notice that y[0] is the output of the void function and y[1] is the output of the ort1 task inside the `always_comb` block. A mismatch between simulation and synthesis occurs, and the synthesis tool issues the following VER-520 warning:

```
Running HDLC
Compiling source file …/comb.1.sv
Warning:  …/comb.1.sv:6: Task enable in always_comb block. (VER-520)
```

To avoid the mismatch, use a void function inside the `always_comb` block, as shown in the following example:

```
module comb1(
   input logic a, b , c,
   output logic [1:0] y
);
always_comb orf1(a);
function void orf1 (a);
   y[0] = a | b | c;
   y[1] = a | b | c;
endfunction
endmodule
```

Conversion Between Two-State and Four-State Variables

The HDL Compiler tool treats two-state values as four-state values (see Unsupported Constructs). When a four-state variable is converted to a two-state variable or vice versa, a mismatch between synthesis and simulation can occur. The simulation tool considers an x value as an unknown, whereas the synthesis tool considers an x value as a don't care value. To avoid such mismatches, use either two-state or four-state variables and avoid conversion between them.

In the following RTL, the a four-state input of the `logic` type makes a continuous assignment to the b two-state output of the `bit` type. The testbench module feeds the a signal through the a_driver variable of the `logic` type that is uninitialized. Because the `bit` type defaults to logic 0 when uninitialized, the `assign b = a` statement causes a mismatch at time 0 as shown in the simulation log.

- RTL

```
// logic_bit_test.sv
module logic_bit_test(
   input logic a,
   output bit b
);
assign b = a;
endmodule

module logic_bit_testbench(output logic a_driver);
initial begin // no initial value
   #10 a_driver = ’1;
   #10 a_driver = ’0;
   #10 $finish;
end
endmodule

module top;
wire a_con, b_con;
logic_bit_test u1(a_con, b_con);
logic_bit_testbench u2(a_con);
initial
begin
   $display("\t\tTime A  B\n");
   $monitor($time,,,,a_con,,,,b_con);
end
endmodule
```

- VCS simulation log

```
...
             Time   A   B
                   0   x   0
                  10   1   1
                  20   0   0
$finish called from file
"redu.sim.syn.mismatch_state.conver.2state.4state.sv", line 8.
$finish at simulation time 30
```

Data Type Declarations

Before you use a data type, you must first declare the data type by using the `typedef` construct. As shown in the following example, the mytype `logic` type is declared before it is used in the my_design module. If you use a data type without declaring it first, the tool issues a syntax error message.

```
typedef logic mytype;
module my_design(
   input logic clock,
   input mytype in,
   output mytype out
);

always_ff @(posedge clock)
   out <= in;
endmodule
```

Synthesizable do...while Loops

A `do...while` loop is synthesizable if the tool can determine the exit condition. The tool does not handle an unknown initial value in the loop when the number of iterations is still bounded. The VCS tool does not have this restriction. The following examples show that one `do...while` loop is synthesizable and the other is not. Because the loop that is not synthesizable has an unknown initial value, the tool issues an error message.

- Example—synthesizable `do...while` loop

```
module do_while_test2(
   input logic [3:0] count1,
   output logic [3:0] z
);
logic [3:0] x, count;
always_comb
begin
   x = 4'd2;
   count = count1;
   do
   begin
      count++;
      x++;
   end
   while(x < 4'd15);
   z = count;
end
endmodule
```

- Example—`do...while` loop not synthesizable

```
module do_while_test2(
   input logic [3:0] count1,
   output logic[3:0] z
);
logic [3:0]  count, x;
always_comb
begin
   count = count1;
   do
   begin
      count++;
      x++;
   end
   while(x < 4'd15);
   z = count;
end
endmodule
```

Troubleshooting generate Loops

To debug `generate` loops, use the `$display()` system task. For example,

```
/* The `ifdef SYNTHESIS is mandatory.
The $display() task does not affect the netlist but causes additional
messages to be written out during elaboration. */

module test #(N=32) (
output [N-1:0] out,
input [N-1:0] in
);
genvar I;
generate
for (I = $left(out); I >= $right(out); I--) begin:GEN
`ifdef SYNTHESIS
always $display("Instantiating: mod GEN[%d].inst ( .out(out[%d]),
.in(in[%d]) )", I, I, I);
`endif
mod inst( .out(out[I]), .in(in[I]) );
end:GEN
endgenerate
endmodule:test
```

Assertions in Synthesis

The following SystemVerilog keywords are parsed and ignored during synthesis: `assert`, `assume`, `before`, `bind`, `bins`, `binsof`, `clocking`, `constraint`, `cover`, `coverpoint`, `covergroup`, `cross`, `endclocking`, `endgroup`, `endprogram`, `endproperty`, `endsequence`, `extends`, `final`, `first_match`, `intersect`, `ignore_bins`, `illegal_bins`, `local`, `program`, `property`, `protected`, `sequence`, `super`, `this`, `throughout`, and `within`. If an assertion-related keyword is not parsed and ignored, it is considered unsupported. For these unsupported keywords, see Unsupported Constructs.

As shown in the following RTL and inference report, the synthesis tool ignores the `assert` keyword and correctly infers a flip-flop:

- RTL containing an `assert` keyword

```
module dff_with_imm_assert(
   input DATA, CLK, RESET,
   output logic Q
);
// Synopsys sync_set_reset "RESET"
always_ff @(posedge CLK)
if (~RESET)
begin
   Q <= 1'b0;
   assert (Q == 1'b0)
   $display("%m PASS:Flip Flop got reset");
else
   $display("%m FAIL:Flip Flop got reset");
end
else
   Q <= DATA;
endmodule
```

- Inference report

```
======================================================================
=======
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR |
SS | ST |
======================================================================
=======
|        Q_reg        | Flip-flop |   1   |  N  | N  | N  | N  | Y  |
N  | N  |
======================================================================
=======
```

Other Troubleshooting Guidelines

Table 11-2 Other Troubleshooting Guidelines

For guideline on	See
Designs containing checker libraries	Reading Designs With Assertion Checker Libraries
Issues with the global name space (`$unit`)	Global Name Space ($unit)

Table 11-2 Other Troubleshooting Guidelines (Continued)

For guideline on	See
Module renaming issues	Renamed Modules Example 3
Interfaces	Interfaces
Designs containing interfaces	Reading SystemVerilog Designs Note: You cannot use the `elaborate` command to instantiate a parameterized design.
Unsupported SystemVerilog constructs	Unsupported Constructs
Casting	The tool supports nonvoid function calls as statements, but it generates a warning.

A

SystemVerilog Design Examples

This section contains examples that use various SystemVerilog constructs.

- FIFO Example
- Bus Fabric Design
- Coding for Late-Arriving Signals
- Master-Slave Latch Inferences

You can find more examples in the `$DC_HOME_DIR`/doc/syn/examples/verilog directory. The `$DC_HOME_DIR` variable specifies the location of the Design Compiler installation.

FIFO Example

Example A-1 uses a variety of SystemVerilog features to build a FIFO.

Example A-1 FIFO

```
// Synchronous FIFO. 4 x 16 bit words.
typedef logic [7:0] ubyte;

typedef struct {
    ubyte src;
    ubyte dst;
    ubyte [0:3] data;
} packet_t;

// Use interface and modport to declare data in and out
interface port;
logic enable;
logic stall;

packet_t packet;
modport sendm(input enable, packet, output stall);
modport recvm(input enable, output packet, stall);
endinterface : port

module fifo #(DEPTH = 2, MAX_COUNT = (1<<DEPTH)) (
   input clk, rstp,
   port.sendm in,
   port.recvm out
);

// Define the FIFO pointers. A FIFO is essentially a circular queue.
reg [(DEPTH-1):0] tail;
reg [(DEPTH-1):0] head;

// Define the FIFO counter. Count the number of entries in the FIFO
// to figure out things like Empty and Full.
reg [(DEPTH):0] count;

// Define the register bank. Array of structures
packet_t fifomem[0:MAX_COUNT];

// Dout is registered and gets the value that tail points to RIGHT NOW.
always_ff @(posedge clk)
begin
   if (rstp == 1)
      out.packet <= '{default:0};
   else
      out.packet <= fifomem[tail];
end

// Update FIFO memory.
always_ff @(posedge clk)
begin
   if (rstp == 1'b0 && in.enable == 1'b1 && in.stall == 1'b0)
      fifomem[head] <= in.packet;
end
```

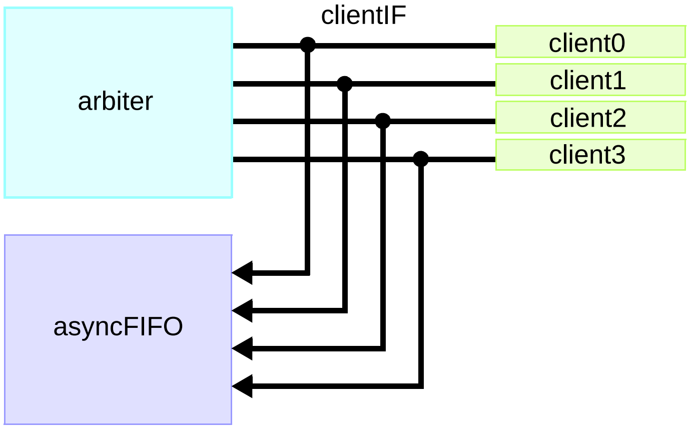
```
// Update the head register.
always_ff @(posedge clk)
begin
   if (rstp == 1'b1)
      head <= 0;
   else
   if (in.enable == 1'b1 && in.stall == 1'b0)
      head <= head + 1; // WRITE
end

// Update the tail register.
always_ff @(posedge clk)
begin
   if (rstp == 1'b1)
      tail <= 0;
   else
   if (out.enable == 1'b1 && out.stall == 1'b0)
      tail <= tail + 1; // READ
end

// Update the count register.
always_ff @(posedge clk)
begin
   if (rstp == 1'b1)
   begin
           count <= 0;
   end
   else
   begin
      case ({out.enable, in.enable})
         2'b00: count <= count;
         2'b01: // WRITE
                if (!in.stall)
                count <= count + 1;
         2'b10: // READ
                if (!out.stall)
                count <= count - 1;
         2'b11: // Concurrent read and write. No change in count
                count <= count;
      endcase
   end
end

// First, update the empty flag.
always_comb
begin
   if (count == 0)
      out.stall = 1'b1;
   else
      out.stall = 1'b0;
end

// Update the full flag
always_comb
begin
  if (count < MAX_COUNT)
     in.stall = 1'b0;
  else
     in.stall = 1'b1;
end
endmodule
```

Bus Fabric Design

This example shows a bus fabric structure that is commonly used in networking designs.

As shown in Figure A-1, the client sends a request to the arbiter to get permission to transfer data. To avoid bus contention, the arbiter handles the request in a weighted round-robin fashion and issues a unique grant signal to the client. After the client receives the permission, it sends data and a write enable signal to the FIFO. When the FIFO is almost full (not shown in the design), the client stops sending data.

Figure A-1 Bus Fabric Structure



To implement this bus fabric design, the following examples use unpacked arrays, packed arrays, interface arrays, casting, modport instantiations, and other SystemVerilog features. You can implement a more complex architecture based on this design.

To encapsulate the complex interconnection between the arbiter and clients, this design uses the `interface` construct to create the clientIF module, as shown in Example A-2, and an array of interfaces with modports. Using the `interface` construct reduces the design complexity and increases reusability of the clientIF interface between modules. Furthermore, you can reconfigure this type of bus fabric structure using parameters.

Example A-2 clientIF Interface Module

```
interface clientIF #(parameter lengthOfId = 8,parameter dataWidth = 128);
logic  grant;
logic  req;
logic [lengthOfId-1:0] priorityID;
logic  wrEn;
logic [dataWidth-1:0] cData;
logic  wrFifoFull;

modport clientMod  (input grant, wrFifoFull, output req, priorityID,
wrEn, cData);
modport arbiterMod (output grant, input req, priorityID);
modport fifoMod    (input grant, cData, wrEn, output wrFifoFull);
endinterface
```

Because this interface is a bundle of wires, it contains no sequential logic. To create combinational logic in an interface, use the `function-endfunction`, `task-endtask`, or `generate-endgenerate` keyword pairs. The logic can be point-to-point connections or one output net driving multiple nets of the same name, such as the grant signal in Example A-2. The example uses one-dimensional array; you can also use two-dimensional arrays.

As shown Example A-3, the arbiterMod module grants bus access in a round-robin fashion using a state machine. If you choose one-hot state encoding, use the `one_hot` pragma for better QoR. You should use one coding style for the state machine for easy debugging. The arbiterMod module

- Contains one `always_ff` block for the sequential logic and one `always_comb` block for the combinational logic.
- Uses enumerations to describe the state variables and provides defaults for the state variables to reduce repetitive logic.
- Checks the token value (`cIF[j].priorityID`) to determine to which client to grant access.
- Specifies the arbiterMod modport of the clientIF interface in the module port declarations (`clientIF.arbiterMod cIF[4]`) to connect to the interface.
- Uses an unpacked array in the interface array to connect all nets, for example, `cIF[k].grant = grant[k]`.

Example A-3 arbiterMod Module

```
module arbiterMod #(parameter lengthOfId =8)(
   input logicclk,
   input logicrst,
   clientIF.arbiterMod cIF[4]
);

// Using little endian
logic [lengthOfId-1:0] priorityID[4];
logic [$clog2(4)-1:0]  tokenValue;
logic [lengthOfId-1:0] tmpValue;
logic [4:0]            tokenRR;  // Concatenate idle state in 1st bit
logic                  req[4];
logic                  grant [4];

typedef enum logic [4:0] {IDLE = 5'b00001, GNT0ST = 5'b00010,
GNT1ST = 5'b00100, GNT2ST = 5'b01000, GNT3ST = 5'b10000} tState;
tState current_state, next_state;

for (genvar j = 0; j < 4; j++)
begin : signalsFromInterfaceBus
   assign req[j]        = cIF[j].req;
   assign priorityID[j] = cIF[j].priorityID;
end

always @*
begin : tokenGenerate
   tmpValue   = '0;
   tokenValue = '0;
   for (int i = 0; i < 4; i++)
   if (priorityID[i] > tmpValue)
   begin : linearSearchToServeBiggestValue
      tokenValue = i;
      tmpValue = priorityID[i];
   end
end

always_comb
begin : tokenRR
   case (tokenValue)
      2'b00: tokenRR = 5'b00010;
      2'b01: tokenRR = 5'b00100;
      2'b10: tokenRR = 5'b01000;
      2'b11: tokenRR = 5'b10000;
   endcase
end
always_ff @(posedge clk, negedge rst)
begin : roundRobinStateMachine
   if (!rst)
      current_state <= IDLE;
   else
      current_state <= next_state;
```

```
end

always_comb
begin : roundRobinDecoder
   grant[0] = 0;
   grant[1] = 0;
   grant[2] = 0;
   grant[3] = 0;
   next_state = current_state;
   case (current_state)
      IDLE  : if (req[3] | req[2] | req[1] | req[0])
              /* Using casting method to convert type. You should
              ensure correct connection and syntax because the tool does
              not check the casting. Alternatively, you can use
              localparam and logic. */
                 next_state = tState'(tokenRR);
      GNT0ST: if (req[0])
                 grant[0] = 1'b1;
              else if (!req[0] & req[3])
                 next_state   = GNT3ST;
              else if (!req[0] & !req[3] & req[2])
                 next_state   = GNT2ST;
              else if (!req[0] & !req[3] & !req[2] & req[1])
                 next_state = GNT1ST;
              else
                 next_state = IDLE;
      GNT1ST: if (req[1])
                 grant[1] = 1'b1;
              else if (!req[1] & req[0])
                 next_state = GNT0ST;
              else if (!req[1] & !req[0] & req[3])
                 next_state   = GNT3ST;
              else if (!req[1] & !req[0] & !req[3] & req[2])
                 next_state = GNT2ST;
              else
                 next_state = IDLE;
      GNT2ST: if (req[2])
                 grant[2] = 1'b1;
              else if (!req[2] & req[1])
                 next_state = GNT1ST;
              else if (!req[2] & !req[1] & req[0])
                 next_state = GNT0ST;
              else if (!req[2] & !req[1] & !req[0] & req[3])
                 next_state = GNT3ST;
              else
                 next_state = IDLE;
      GNT3ST: if (req[3])
                 grant[3] = 1'b1;
              else if (!req[3] &  req[2])
                 next_state = GNT2ST;
              else if (!req[3] & !req[2] &  req[1])
                 next_state = GNT1ST;
              else if (!req[3] & !req[2] & !req[1] & req[0])
```

```
                  next_state = GNT0ST;
               else
                  next_state = IDLE;
   endcase
end

for (genvar k = 0; k < 4; k++)
begin : sendGrandToInterfaceBus
   assign cIF[k].grant = grant[k];
end
endmodule
```

This bus fabric design shows the client module communicates with two modules, the arbiter and FIFO, through the interface array. Example A-4 shows how to connect the modules to the interface. The asynchronous FIFO module

- Uses the `default` keyword to set the array element values.
- Instantiates the DesignWare memory cell to save area and speed up the runtime.
- Concatenates packed arrays to prioritize the decoding by using a `casex` statement.
- Uses functions for common portions of the design to reduce the code size.
- Uses the `genvar` keyword to create parallel combinational logic and the `assign` statement to assign logic declarations in the `begin-end` block.

For more information about the Synopsys DesignWare Flip-Flop-Based Asynchronous Dual-Port RAM used in this example, see the datasheet for the DW_ram_r_w_a_dff block in the DesignWare Library documentation.

Example A-4 Asynchronous FIFO Module

```
module fifoMod #(
   parameter dataWidth = 128,
   parameter addrWidth = 4, //exclude status bit
   parameter ramDepth  = (1 << addrWidth),
   parameter dwRstMode = 0 //0: ram reset active low
)(
   input logic         clkA,
   input logic         clkB,
   input logic         rst,
   input logic         wrCsA,
   input logic         rdCsB,
   input logic         rdEnB,

   /* DesignWare RAM control signal using scan chain from test mode
    When dwTestMode is high, the test clk will capture data.
    If dwTestMode is low, it is in normal mode */
   input logic  dwTestClk,
   input logic         dwTestMode,   //1: enable test clk to be testMode

   output logic        rdFifoEmptyB,
   output logic        fifoDataVldB, //make data valid from fifo data
   output logic        [dataWidth-1:0]dataOut,

   clientIF.fifoMod    cIF [4]
);

typedef logic [addrWidth:0] syncT;
syncT rdPtrBb2gSync1;
syncT rdPtrBb2gSync;
syncT wrPtrAb2gSync1;
syncT wrPtrAb2gSync;

logic [dataWidth-1:0]clientToFifoDataTmp [4];
logic [3:0] clientwrEn;
logic [3:0] clientwrEnD;
logic [3:0] tmpClientwrEn;
logic [3:0] clientGrant;
logic [dataWidth-1:0]fifoRam [ramDepth];
logic [dataWidth-1:0]dataIn;
logic wrEnA;
// First bit used as fifo status bit, so it does not minus 1
logic [addrWidth:0]wrPtrA;
logic [addrWidth:0]wrPtrAb2g;
logic wrFifoFullA;
logic wrFifoFullASync;
logic wrFifoFullGray;
logic rdFifoEmptyGray;
logic [addrWidth:0]rdPtrB;
logic [addrWidth:0]rdPtrBb2g;
logic [addrWidth:0]rdPtrBb2gSame;
logic fifoCs;
logic wrRamA;
logic [dataWidth-1:0]dataOutP;

// clock A domain
for (genvar j = 0; j < 4; j++)
begin : combineIndividula4chennelIntoTwoDementionalArray
   assign clientToFifoDataTmp[j] = cIF[j].cData;
   assign clientwrEn[j]          = cIF[j].wrEn;
   assign clientGrant[j]         = cIF[j].grant;
end

always_ff @(posedge clkA, negedge rst)
begin : fromClientInterfacePorts
   if (!rst)
      dataIn <= '0;
   else
   begin : decodeClientChannelDataWithWriteEnable
      for (int k = 0; k < 4; k++)
      if (clientGrant[k])
      dataIn  <= clientToFifoDataTmp[k];
   end
end

always_ff @(posedge clkA, negedge rst)
if (!rst)
   clientwrEnD <= '{default:0};
else
   clientwrEnD <=  clientwrEn;
   for (genvar m = 0; m < 4; m++)
   begin
      assign tmpClientwrEn[m] = clientwrEn[m] & !clientwrEnD[m];
   end

always_ff @(posedge clkA, negedge rst)
begin: fromClientInterfacePortsEn
   if (!rst)
      wrEnA  <= '0;
   else
   begin
      casex ({clientGrant[0], clientGrant[1], clientGrant[2],
      clientGrant[3]})
         // This RTL style can get priority decoding in some designs.
         4'b1xxx: wrEnA <= tmpClientwrEn[0];
         4'b01xx: wrEnA <= tmpClientwrEn[1];
         4'b001x: wrEnA <= tmpClientwrEn[2];
         4'b0001: wrEnA <= tmpClientwrEn[3];
         default: wrEnA <= '0;
      endcase
   end
end

always_ff @ (posedge clkA, negedge rst)
if (!rst)
```

```
   wrPtrA <= '0;
else
   wrPtrA <= wrPtrA + (wrCsA & wrEnA & !wrFifoFullA);

always_ff @ (posedge clkA, negedge rst)
if (!rst)
   wrPtrAb2g <= '0;
else
   wrPtrAb2g <= binaryToGray (wrPtrA);

always_ff @ (posedge clkA, negedge rst)
if (!rst)
   {rdPtrBb2gSync1, rdPtrBb2gSync} <= '0;
else
   {rdPtrBb2gSync1, rdPtrBb2gSync} <= {rdPtrBb2gSync, rdPtrBb2g};

always_ff @ (posedge clkA, negedge rst)
if (!rst)
   wrFifoFullGray <= '0;
else
   wrFifoFullGray <= (wrPtrAb2g[addrWidth-3:0] ==
   rdPtrBb2gSync1[addrWidth-3:0] ) &
   !(wrPtrAb2g[addrWidth-2] ^ rdPtrBb2gSync1[addrWidth-2]) &
   (wrPtrAb2g[addrWidth-1] ^ rdPtrBb2gSync1[addrWidth-1]);

always_ff @ (posedge clkA, negedge rst)
if (!rst)
begin
   wrFifoFullA <= '0;
   wrFifoFullASync <= '0;
end
else
begin
   wrFifoFullASync <= wrFifoFullGray;
   wrFifoFullA <= wrFifoFullASync;
end

assign fifoCs = !(wrCsA & wrEnA);
assign wrRamA = !(wrCsA & wrEnA & !wrFifoFullA);

// Instantiate DesignWare dual port async RAM
DW_ram_r_w_a_dff #(.data_width(dataWidth), .depth(ramDepth),
.rst_mode(dwRstMode))
dual_ram_u1 (.rst_n(rst), .cs_n(fifoCs), .wr_n(wrRamA),
.test_mode(dwTestMode), .test_clk(dwTestClk),
.rd_addr(rdPtrB[addrWidth-1:0]),
.wr_addr(wrPtrA[addrWidth-1:0]), .data_in(dataIn), .data_out(dataOutP));

// gray code
function automatic logic [addrWidth:0] binaryToGray (input [addrWidth:0]
binaryIn);
return (binaryIn >> 1) ^ binaryIn;
endfunction
```

```
// clock B domain
always_ff @ (posedge clkB, negedge rst)
if (!rst)
   fifoDataVldB <= 1'b0;
else
   fifoDataVldB <= rdCsB & rdEnB;

always_ff @ (posedge clkB, negedge rst)
if (!rst)
   dataOut <= '0;
else
   dataOut <= dataOutP;

always_ff @ (posedge clkB, negedge rst)
if (!rst)
   rdFifoEmptyB <= '1;
else
   rdFifoEmptyB <= rdFifoEmptyGray;

always_ff @ (posedge clkB, negedge rst)
if (!rst)
   {wrPtrAb2gSync1, wrPtrAb2gSync} <= '0;
else
   {wrPtrAb2gSync1, wrPtrAb2gSync} <= {wrPtrAb2gSync, wrPtrAb2g};

always_comb
begin
   rdFifoEmptyGray = (wrPtrAb2gSync1 == rdPtrBb2gSame);
end

assign rdPtrBb2gSame = binaryToGray (rdPtrB);

always_ff @ (posedge clkB, negedge rst)
if (!rst)
   rdPtrB <= '0;
else
   rdPtrB <= rdPtrB +(rdCsB & rdEnB & !(rdFifoEmptyB | rdFifoEmptyGray));

always_ff @ (posedge clkB, negedge rst)
if (!rst)
   rdPtrBb2g <= '0;
else
   rdPtrBb2g <= binaryToGray (rdPtrB);

for (genvar i = 0; i < 4; i++)
begin
   assign cIF[i].wrFifoFull = wrFifoFullGray;
end
endmodule
```

Each client module can have its own functions using the same generic interface from the interface module or have the same functions from the interface module. Example A-5 shows that an interface modport connects to one of the four client modules through a module port without using an interface array. You should use explicit interface declarations (`clientIF.clientMod`) in the module port list, but not generic declarations (`interface.clientMod`), so the tool can check the connections. For the arbiter and FIFO, interface arrays are used because of the four interfaces with one modport. The code for the input from the interface is “`assign tmpClientGrant = cIF.grant`”, whereas the code for the output to the interface is “`assign cIF.req = req`”. The example also uses unsized constants, ‘0 and ‘1, to assign all 0s or 1s to any bus width to implement the reset or set circuitry respectively.

Example A-5 clientMod Module

```
module clientMod #(parameter lengthOfId = 8, parameter dataWidth = 128)(
   input logic clk,
   input logic rst,
   input logic validIn,
   input logic [dataWidth-1:0] clientData,
   input logic [lengthOfId-1:0] priorityIDPgam,
   output logic done,
   /* interface.clientMod cIF, generic declaration.
   Use explicit declaration. */
   clientIF.clientMod cIF
);

logic clientGrant;
logic tmpClientGrant;
logic clientGrantD;
logic clientGrantD1;
logic clientGrantD2;
logic wrFifoFull;
logic [lengthOfId-1:0]tmpPriorityID;
logic wrEnable;
logic [dataWidth-1:0]tmpBuffer;
logic valid;
logic req;

assign tmpClientGrant= cIF.grant;
assign wrFifoFull= cIF.wrFifoFull;

always_ff @(posedge clk, negedge rst)
if (!rst)
begin
   clientGrantD  <= '0;
   clientGrantD1 <= '0;
   clientGrantD2 <= '0;
end
else
begin
   clientGrantD  <= tmpClientGrant;
   clientGrantD1 <= clientGrantD;
   clientGrantD2 <= clientGrant;
end

assign clientGrant = !clientGrantD1 & tmpClientGrant;

always_ff @(posedge clk, negedge rst)
if (!rst)
   tmpPriorityID <= '0;
else if (validIn)
   tmpPriorityID <= priorityIDPgam;

always_ff @(posedge clk, negedge rst)
if (!rst)
```

```
   tmpBuffer <= '0;
else if (validIn)
   tmpBuffer <= clientData;
else if (wrEnable)
   tmpBuffer <= '0;

assign done = clientGrant & valid & !wrFifoFull;

always_ff @(posedge clk, negedge rst)
if (!rst)
   valid <= '0;
else if (validIn)
   valid <= '1;
else if (clientGrantD2)
   valid <= '0;

always_ff @(posedge clk, negedge rst)
if (!rst)
   wrEnable <= 1'b0;
else if (clientGrant & valid)
   wrEnable <= 1'b1;
else
   wrEnable <= 1'b0;

always_ff @(posedge clk, negedge rst)
if (!rst)
   req <= 1'b0;
else if (validIn)
   req <= 1'b1;
else if (wrEnable)
   req <= 1'b0;

assign cIF.priorityID = tmpPriorityID;
assign cIF.req = req;
assign cIF.wrEn = wrEnable;
assign cIF.cData = tmpBuffer;
endmodule
```

For complex and large designs, Example A-6 shows a quick way to configure the client channels using port parameters during the top-level module instantiations. To avoid mismatches and reduce errors during design changes, you should specify modports of a hierarchically referenced interface (`cIFArray[0].clientMod`) instead of just an interface (`cIFArray[0]`) during module instantiation, so the tool can match the RTL.

Example A-6 topMod Module

```
module topMod (
   // DesignWare test control signal
   input logic dwTestClk,
   input logic dwTestMode,
   // System signal
   input logic clkA, clkB,
   input logic rst,
   // clock A domain
   input logic wrCsA,
   input logic validIn0,
   input logic [127:0]clientData0,
   input logic [7:0]priorityIDPgam0, //from register map
   input logic validIn1,
   input logic [127:0]clientData1,
   input logic [7:0]priorityIDPgam1,
   input logic validIn2,
   input logic [127:0]clientData2,
   input logic [7:0]priorityIDPgam2,
   input logic validIn3,
   input logic [127:0]clientData3,
   input logic [7:0]priorityIDPgam3,

   output logic done0,
   output logic done1,
   output logic done2,
   output logic done3,

   // clock B domain
   input logic rdEnB,
   input logic rdCsB,

   output logic rdFifoEmptyB,
   output logic fifoDataVldB,
   output logic [127:0]dataOut
);

localparam lengthOfId = 8;
localparam dataWidth = 128;
localparam addrWidth = 4;         //exclude status bit
localparam ramDepth = (1 << addrWidth);
localparam dwRstMode = 0;         //0: ram reset active low

clientIF #(.lengthOfId(lengthOfId), .dataWidth(dataWidth)) cIFArray[4]();

arbiterMod #(
.lengthOfId(lengthOfId)
) arbiterModU0 (
.clk(clkA),
.rst(rst),
.cIF(cIFArray.arbiterMod)
);
```

```
fifoMod #(
.dataWidth(dataWidth),
.addrWidth(addrWidth),
.ramDepth(ramDepth),
.dwRstMode(dwRstMode)
) fifoModU0 (
.clkA(clkA),
.clkB(clkB),
.rst(rst),
.wrCsA(wrCsA),
.rdCsB(rdCsB),
.rdEnB(rdEnB),
.dwTestClk(dwTestClk),
.dwTestMode(dwTestMode),
.rdFifoEmptyB(rdFifoEmptyB),
.fifoDataVldB(fifoDataVldB),
.dataOut(dataOut),
.cIF(cIFArray.fifoMod)
);

clientMod #(
.lengthOfId(lengthOfId),
.dataWidth(dataWidth)
) clientModU0 (
.clk(clkA),
.rst(rst),
.validIn(validIn0),
.clientData(clientData0),
.priorityIDPgam(priorityIDPgam0),
.done(done0),
//.cIF(cIFArray[0])               // pass the interface
.cIF(cIFArray[0].clientMod)       // Passing the modport is recommended
);

clientMod #(
.lengthOfId(lengthOfId),
.dataWidth(dataWidth)
) clientModU1 (
.clk(clkA),
.rst(rst),
.validIn(validIn1),
.clientData(clientData1),
.priorityIDPgam(priorityIDPgam1),
.done(done1),
.cIF(cIFArray[1].clientMod)
);

clientMod #(
.lengthOfId(lengthOfId),
.dataWidth(dataWidth)
) clientModU2 (
.clk(clkA),
.rst(rst),
.validIn(validIn2),
.clientData(clientData2),
.priorityIDPgam(priorityIDPgam2),
.done(done2),
.cIF(cIFArray[2].clientMod)
);

clientMod #(
.lengthOfId(lengthOfId),
.dataWidth(dataWidth)
) clientModU3 (
.clk(clkA),
.rst(rst),
.validIn(validIn3),
.clientData(clientData3),
.priorityIDPgam(priorityIDPgam3),
.done(done3),
.cIF(cIFArray[3].clientMod)
);
endmodule
```

See Also

- State Machines
- Interfaces

Coding for Late-Arriving Signals

The following topics describe coding techniques for late-arriving signals:

- Duplicating Datapaths
- Moving Late-Arriving Signals Close to Output

Note:

> These techniques apply to the HDL Compiler output. When this output is constrained and optimized by the Design Compiler tool, the structure might be changed depending on the design constraints and option settings. For more information, see the *Design Compiler User Guide*.

Duplicating Datapaths

To improve the timing of late-arriving signals, you can duplicate datapaths, but at the expense of more area and increased input loads.

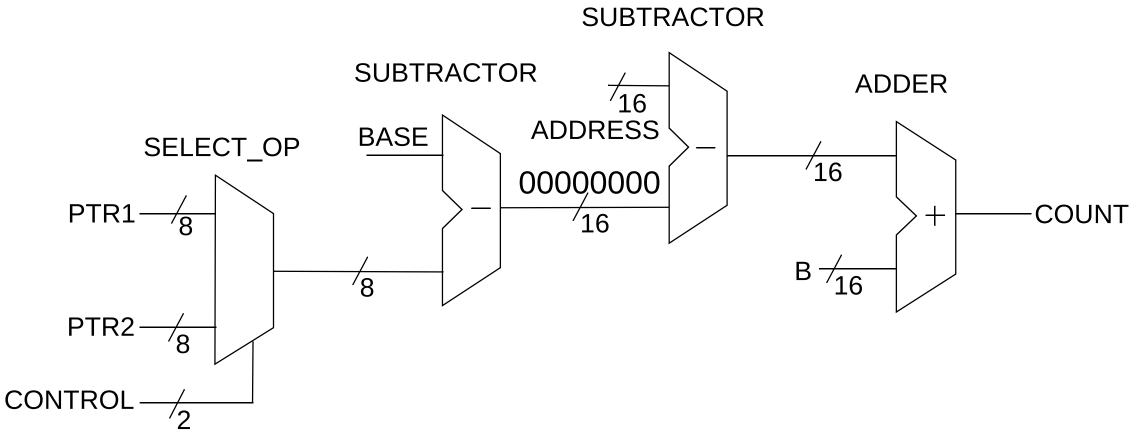
Original RTL

In Example A-7, the late-arriving CONTROL signal selects either the PTR1 or PTR2 input, and then the selected input drives a chain of arithmetic operations ending at output COUNT. As shown in Figure A-2, a SELECT_OP is next to a subtractor. When you see a SELECT_OP next to an operator, you should duplicate the conditional logic of the SELECT_OP and move the SELECT_OP to the end of the operation, as shown in Example A-8.

Example A-7 Original RTL

```
module BEFORE #(parameter [7:0] BASE = 8'b10000000)(
   input [7:0] PTR1,PTR2,
   input [15:0] ADDRESS, B,
   input CONTROL, //CONTROL is late arriving
   output [15:0] COUNT
);
   wire [7:0] PTR, OFFSET;
   wire [15:0] ADDR;
assign PTR = (CONTROL == 1'b1) ? PTR1 : PTR2;
assign OFFSET = BASE - PTR; // Could be any function of f(BASE,PTR)
assign ADDR = ADDRESS - {8'h00, OFFSET};
assign COUNT = ADDR + B;
endmodule
```

Figure A-2 Schematic of the Original RTL



Modified RTL With the Duplicate Datapath

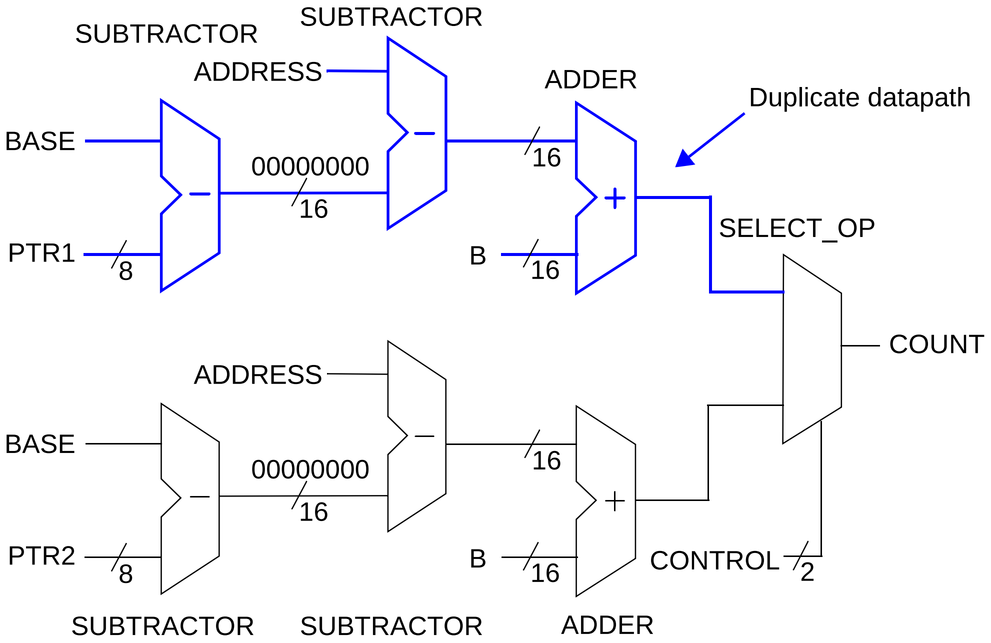
In the modified RTL, the entire datapath is duplicated because signal CONTROL arrives late. The resulting output COUNT becomes a conditional selection between two parallel datapaths based on input PTR1 or PTR2 and controlled by signal CONTROL. The path from signal CONTROL to output COUNT is no longer a critical path. The timing is improved, but at the expense of more area and more loads on the input pins. In general, the amount of datapath duplication is proportional to the number of conditional statements of the

SELECT_OP. For example, if you have four input signals to the SELECT_OP, you duplicate three datapaths. To minimize the area of duplicate logic, you can design signal CONTROL to arrive early.

Example A-8 Modified RTL With the Duplicate Datapath

```
module PRECOMPUTED #(parameter [7:0] BASE = 8'b10000000)(
   input [7:0] PTR1, PTR2,
   input [15:0] ADDRESS, B,
   input CONTROL,
   output [15:0] COUNT
);
   wire [7:0] OFFSET1,OFFSET2;
   wire [15:0] ADDR1,ADDR2,COUNT1,COUNT2;
assign OFFSET1 = BASE - PTR1;  // Could be f(BASE,PTR)
assign OFFSET2 = BASE - PTR2;  // Could be f(BASE,PTR)
assign ADDR1 = ADDRESS - {8'h00 , OFFSET1};
assign ADDR2 = ADDRESS - {8'h00 , OFFSET2};
assign COUNT1 = ADDR1 + B;
assign COUNT2 = ADDR2 + B;
assign COUNT = (CONTROL == 1'b1) ? COUNT1 : COUNT2;
endmodule
```

Figure A-3 Schematic of the Modified RTL



See Also

- SELECT_OP Inference

Moving Late-Arriving Signals Close to Output

If you know which signals in your design are late-arriving, you can structure the code so that the late-arriving signals are close to the output.

The following examples show the coding techniques of using the `if` and `case` statements for late-arriving signals:

- Overview
- Late-Arriving Data Signal Example 1
- Late-Arriving Data Signal Example 2
- Late-Arriving Data Signal Example 3
- Late-Arriving Control Signal Example 1
- Late-Arriving Control Signal Example 2

Overview

To better handle late-arriving signals, use sequential `if` statements to create a priority-encoded implementation. You assign priority in descending order; that is, the last `if` statement corresponds to the data signal of the last SELECT_OP cell in the chain.

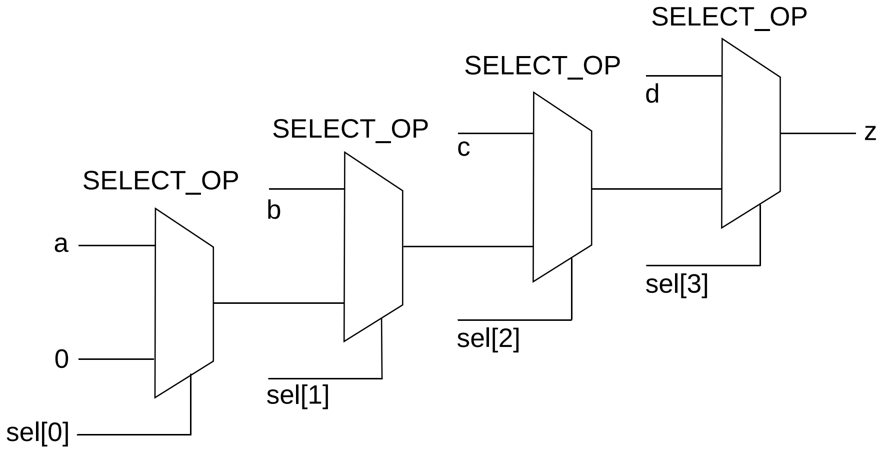
RTL With Sequential if Statements

The a and sel[0] signals have the longest delays to the z output, while the d and sel[3] signals have the shortest delays to the z output.

Example A-9 RTL With Sequential if Statements

```
module mult_if (
   input a, b, c, d,
   input [3:0] sel,
   output logic z
);
always_comb
begin
   z = 0;
   if (sel[0]) z = a;
   if (sel[1]) z = b;
   if (sel[2]) z = c;
   if (sel[3]) z = d;
end
endmodule
```

Figure A-4 Schematic of the RTL



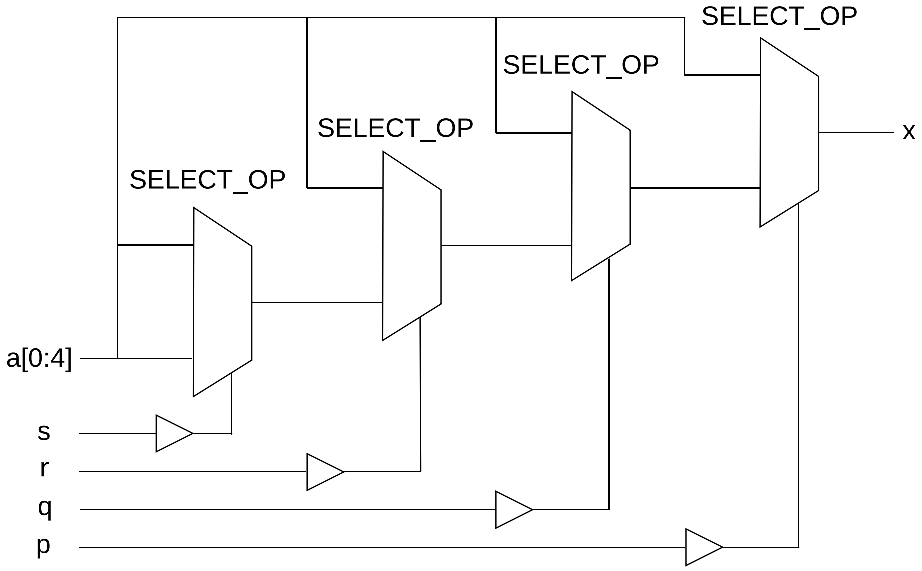
Modified RTL With Named begin-end Blocks

If you use the `if-else` construct with the `begin-end` blocks to build a priority encoded MUX, you must use the named `begin-end` blocks.

Example A-10 Modified RTL With Named begin-end Blocks

```
module m1 (
   input p, q, r, s,
   input [0:4] a,
   output logic x
);
always_comb
if ( p )
   x = a[0];
else begin :b1
   if ( q )
      x = a[1];
   else begin :b2
      if ( r )
         x = a[2];
      else begin :b3
         if ( s )
            x = a[3];
         else
            x = a[4];
      end :b3
   end :b2
end :b1
endmodule
```

Figure A-5 Schematic of the Modified RTL



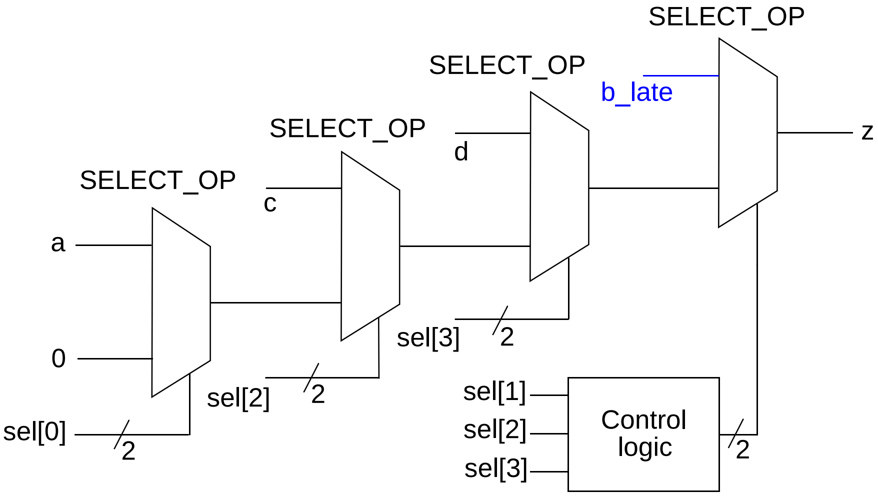
Late-Arriving Data Signal Example 1

This example shows how to place the late-arriving b_late signal close to the z output.

Example A-11 RTL Containing a Late-Arriving Data Signal

```
module mult_if_improved(
   input a, b_late, c, d,
   input [3:0] sel,
   output logic z
);
logic z1;
always_comb
begin
   z1 = 0;
   if (sel[0]) z1 = a;
   if (sel[2]) z1 = c;
   if (sel[3]) z1 = d;
   if (sel[1] & ~(sel[2]|sel[3])) z = b_late;
   else        z = z1;
end
endmodule
```

Figure A-6 Schematic of the RTL



Late-Arriving Data Signal Example 2

This example contains operators in the conditional expression of an `if` statement. The A signal in the conditional expression is a late-arriving signal, so you should move the signal close to the output.

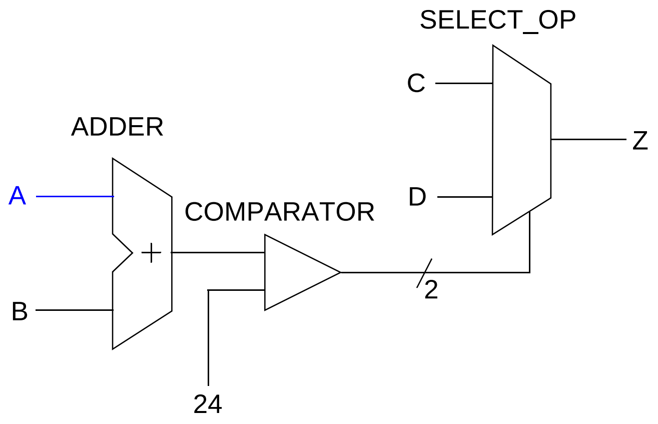
Original RTL Containing the Late-Arriving Input A

The original RTL contains input A that is late arriving.

Example A-12 Original RTL

```
module cond_oper #(parameter N = 8)(
   input [N-1:0] A, B, C, D, // A is late arriving
   output logic [N-1:0] Z
);
always_comb
begin
   if (A + B < 24) Z = C;
   else            Z = D;
end
endmodule
```

Figure A-7 Schematic of the Original RTL



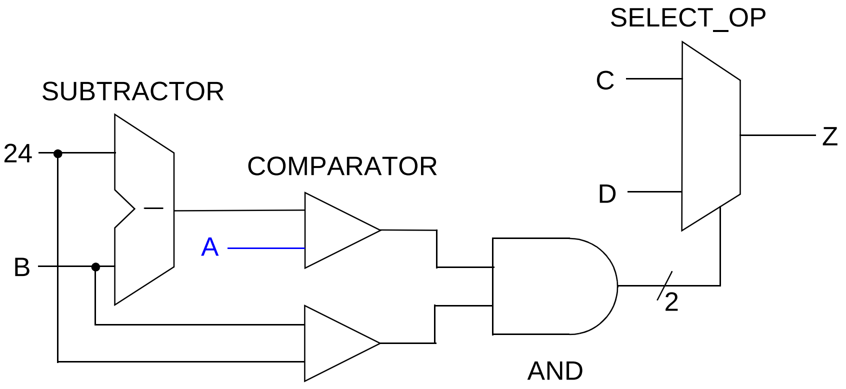
Modified RTL

The following RTL restructures the code to move signal A closer to the output.

Example A-13 Modified RTL

```
module cond_oper_improved #(parameter N = 8)(
   input [N-1:0] A, B, C, D, // A is late arriving
   output logic [N-1:0] Z
);
always_comb
begin
   if ( B < 24 && A < 24 - B) Z = C;
   else                       Z = D;
end
```

Figure A-8 Schematic of the Modified RTL



Late-Arriving Data Signal Example 3

This example shows a `case` statement nested in an `if` statement. The Data_late data signal is late-arriving.

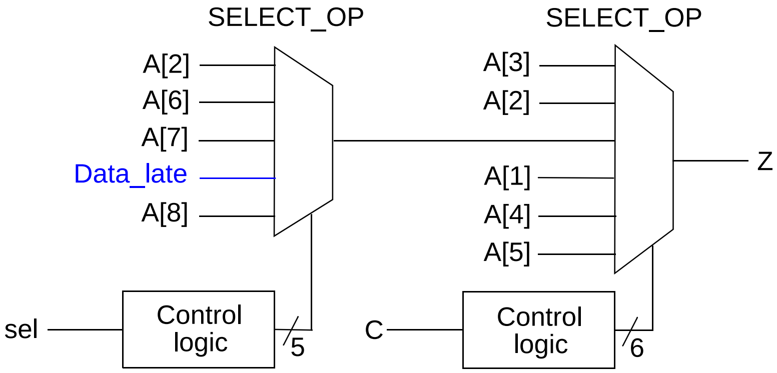
Original RTL Containing a Late-Arriving Input Data_late

The original RTL contains input Data_late that is late arriving.

Example A-14 Original RTL

```
module case_in_if_01 (
   input [8:1] A,
   input Data_late,
   input [2:0] sel,
   input [5:1] C,
   output logic Z
);
always_comb
begin
if (C[1])
   Z = A[5];
else if (C[2] == 1'b0)
   Z = A[4];
else if (C[3])
   Z = A[1];
else if (C[4])
   case (sel)
      3'b010: Z = A[8];
      3'b011: Z = Data_late;
      3'b101: Z = A[7];
      3'b110: Z = A[6];
      default:Z = A[2];
   endcase
else if (C[5] == 1'b0)
   Z = A[2];
else
   Z = A[3];
end
endmodule
```

Figure A-9 Schematic of the Original RTL



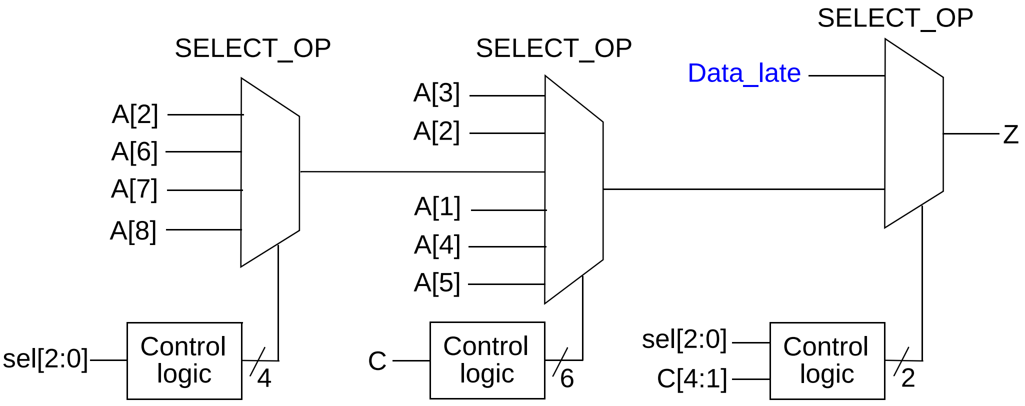
Modified RTL for the Late-Arriving Signal

The late-arriving signal, Data_late, is an input to the first SELECT_OP in the path. You can improve the startpoint for synthesis by moving signal Data_late close to output Z. To do this, move the Data_late assignment from the nested `case` statement to a separate `if` statement. As a result, signal Data_late is an input to the SELECT_OP that is closer to output Z.

Example A-15 Modified RTL

```
module case_in_if_01_improved (
   input [8:1] A,
   input Data_late,
   input [2:0] sel,
   input [5:1] C,
   output logic Z
);
logic Z1, FIRST_IF;

always_comb
begin
   if (C[1])
      Z1 = A[5];
   else if (C[2] == 1'b0)
      Z1= A[4];
   else if (C[3])
      Z1 = A[1];
   else if (C[4])
      case (sel)
         3'b010: Z1 = A[8];
         //3'b011: Z1 = Data_late;
         3'b101: Z1 = A[7];
         3'b110: Z1 = A[6];
         default: Z1 = A[2];
      endcase
   else if (C[5] == 1'b0)
      Z1 = A[2];
   else
      Z1 = A[3];

FIRST_IF = (C[1] == 1'b1) || (C[2] == 1'b0) || (C[3] == 1'b1);

if (!FIRST_IF && C[4] && (sel == 3'b011))
   Z = Data_late;
else
   Z = Z1;
end
endmodule
```

Figure A-10 Schematic of the Modified RTL



Late-Arriving Control Signal Example 1

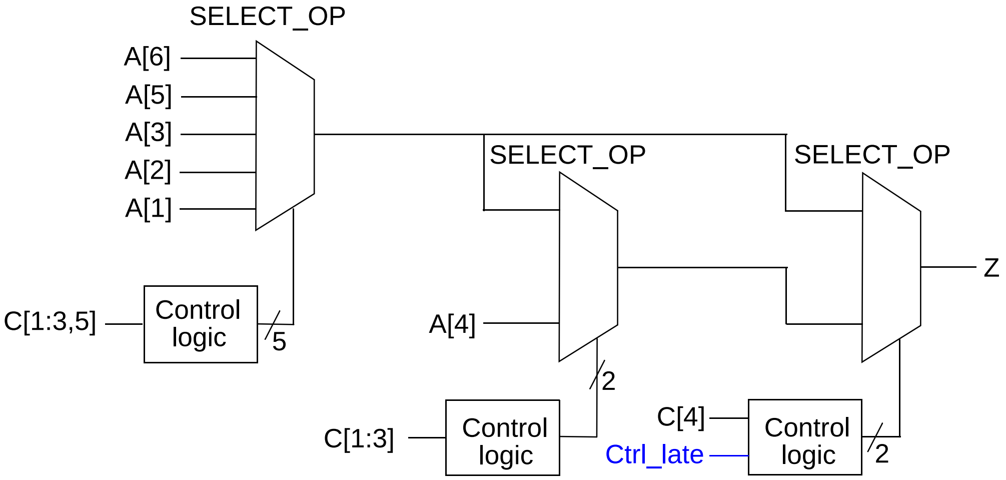
If you have a late-arriving control signal in the design, you should place it close to the output.

In this example, input Ctrl_late is a late-arriving control signal and is placed close to output Z.

Example A-16 RTL with a Late-Arriving Control Signal

```
module single_if_improved (
   input [6:1] A,
   input [5:1] C,
   input Ctrl_late,
   output logic Z
);
logic Z1;
wire Z2, prev_cond;
always_comb
begin
   // remove the branch with the late-arriving control signal
   if (C[1] == 1'b1) Z1 = A[1];
   else if (C[2] == 1'b0) Z1 = A[2];
   else if (C[3] == 1'b1) Z1 = A[3];
   else if (C[5] == 1'b0) Z1 = A[5];
   else                   Z1 = A[6];
end

assign Z2 = A[4];
assign prev_cond = (C[1] == 1'b1) || (C[2] == 1'b0) || (C[3] == 1'b1);
always_comb
begin
   if (C[4] == 1'b1 && Ctrl_late == 1'b0)
      if (prev_cond) Z = Z1;
      else           Z = Z2;
   else
      Z = Z1;
end
endmodule
```

Figure A-11 Schematic of the RTL



Late-Arriving Control Signal Example 2

If you know your design has a late-arriving control signal, you should place the signal close to the output.

Original RTL

This example shows an `if` statement nested in a `case` statement and contains a late-arriving control signal, sel[1].

Example A-17 Original RTL

```
module if_in_case (
   input [2:0] sel,  // sel[1] is late arriving
   input X, A, B, C, D,
   output logic Z
);

always_comb
begin
   case (sel)
      3'b000:  Z = A;
      3'b001:  Z = B;
      3'b010:  if (X) Z = C;
               else   Z = D;
      3'b100:  Z = A ^ B;
      3'b101:  Z = !(A && B);
      3'b111:  Z = !A;
      default: Z = !B;
   endcase
end
endmodule
```

Modified RTL

Because signal sel[1] is a late-arriving input, you should restructure the code to get the best startpoint for synthesis. As shown in the modified RTL, the nested `if` statement is placed outside the `case` statement so that signal sel[1] is closer to output Z. Output Z takes either value Z1 or Z2 depending on whether signal sel[1] is 0 or 1. When signal sel[1] is late arriving, placing it closer to output Z improves the timing.

Example A-18 Modified RTL

```
module if_in_case_improved (
   input [2:0] sel,  // sel[1] is late arriving
   input X, A, B, C, D,
   output logic Z
);
logic Z1, Z2;
logic [1:0] i_sel;
always_comb
begin
   i_sel = {sel[2],sel[0]};
   case (i_sel) // For sel[1]=0
      2'b00:   Z1 = A;
      2'b01:   Z1 = B;
      2'b10:   Z1 = A ^ B;
      2'b11:   Z1 = !(A && B);
      default: Z1 = !B;
   endcase

   case (i_sel) // For sel[1]=1
      2'b00:   if (X) Z2 = C;
               else   Z2 = D;
      2'b11:   Z2 = !A;
      default: Z2 = !B;
   endcase

   if (sel[1]) Z = Z2;
   else Z = Z1;
end
endmodule
```

Master-Slave Latch Inferences

These topics provide information about how to direct the tool to infer various types of master-slave latches.

- Overview for Inferring Master-Slave Latches
- Master-Slave Latch With One Master-Slave Clock Pair
- Master-Slave Latch With Multiple Master-Slave Clock Pairs

- Master-Slave Latch With Discrete Components
- JK Flip-Flop With Synchronous Set and Reset Using sync_set_reset

Overview for Inferring Master-Slave Latches

The Design Compiler tool infers master-slave latches through the `clocked_on_also` attribute. You attach this signal-type attribute to the clocks using an embedded dc_shell script.

Follow these coding guidelines to describe a master-slave latch:

- Specify the master-slave latch as a flip-flop by using only the slave clock.
- Specify the master clock as an input port, but do not connect it.
- Attach the `clocked_on_also` attribute to the master clock port.

This coding style requires that cells in the target library contain slave clocks marked with the `clocked_on_also` attribute. The `clocked_on_also` attribute defines the slave clocks in the cell state declaration. For more information about defining slave clocks in the target library, see the *Library Compiler User Guide*.

The Design Compiler tool does not use D flip-flops to implement the equivalent functionality of a master-slave latch.

Note:

> Although the vendor's component behaves as a master-slave latch, the Library Compiler tool supports only the description of a master-slave flip-flop.

Master-Slave Latch With One Master-Slave Clock Pair

This example shows a basic master-slave latch with one master-slave clock pair using the `dc_tcl_script_begin` and `dc_tcl_script_end` compiler directives.

Example A-19 Master-Slave Latch

```
module mslatch (
   input SCK, MCK, DATA,
   output logic Q
);
// synopsys dc_tcl_script_begin
// set_attribute -type string MCK signal_type clocked_on_also
// set_attribute -type boolean MCK level_sensitive true
// synopsys dc_tcl_script_end

always_ff Q <= DATA;
endmodule
```

Example A-20 Inference Report

```
========================================================================
| Register Name |    Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
========================================================================
|     Q_reg     | Flip-flop |   1   |  N  | N  | N  | N  | N  | N  | N  |
========================================================================
```

See Also

- dc_tcl_script_begin and dc_tcl_script_end

Master-Slave Latch With Multiple Master-Slave Clock Pairs

If the design requires more than one master-slave clock pair, you must specify the associated slave clock in addition to the `clocked_on_also` attribute. This example shows how to use the `clocked_on_also` attribute with the `associated_clock` option.

Example A-21 RTL for Inferring Master-Slave Latches With Two Pairs of Clocks

```
module mslatch2 (
   input SCK1, SCK2, MCK1, MCK2, D1, D2,
   output logic Q1, Q2,
);
// synopsys dc_tcl_script_begin
// set_attribute -type string MCK1 signal_type clocked_on_also
// set_attribute -type boolean MCK1 level_sensitive true
// set_attribute -type string MCK1 associated_clock SCK1
// set_attribute -type string MCK2 signal_type clocked_on_also
// set_attribute -type boolean MCK2 level_sensitive true
// set_attribute -type string MCK2 associated_clock SCK2
// synopsys dc_tcl_script_end
always_ff Q1 <= D1;
always_ff Q2 <= D2;
endmodule
```

Example A-22 Inference reports

```
=========================================================================
| Register Name |    Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
=========================================================================
|    Q1_reg     | Flip-flop |   1   |  N  | N  | N  | N  | N  | N  | N  |
=========================================================================

=========================================================================
| Register Name |    Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
=========================================================================
|    Q2_reg     | Flip-flop |   1   |  N  | N  | N  | N  | N  | N  | N  |
=========================================================================
```

Master-Slave Latch With Discrete Components

If your target library does not contain master-slave latch components, you can direct the tool to infer two-phase systems by using D latches.

This example shows a simple two-phase system with clocks MCK and SCK.

Example A-23 RTL for Two-Phase Clocks

```
module latch_verilog (
   input DATA, MCK, SCK,
   output logic Q
);
logic TEMP;
always_latch
   if (MCK) TEMP <= DATA;
always_latch
   if (SCK) Q <= TEMP;

endmodule
```

Example A-24 Inference Reports

```
========================================================================
|   Register Name   | Type  | Width | Bus | MB | AR | AS | SR | SS | ST |
========================================================================
|     TEMP_reg      | Latch |   1   |  N  | N  | N  | N  | -  | -  | -  |
========================================================================

========================================================================
|   Register Name   | Type  | Width | Bus | MB | AR | AS | SR | SS | ST |
========================================================================
|       Q_reg       | Latch |   1   |  N  | N  | N  | N  | -  | -  | -  |
========================================================================
```

JK Flip-Flop With Synchronous Set and Reset Using sync_set_reset

For the tool to infer JK flip-flops properly, you should code the J, K, and clock signals at the top-level design ports so that simulation can initialize the design.

The following Verilog design infers the JK flip-flop described in Table A-1. The design uses the `sync_set_reset` directive to specify the J and K signals as the synchronous set and reset signals (the JK function) and the `one_hot` directive to prevent priority encoding of the J and K signals.

Table A-1 Truth Table for JK Flip-Flop

J	**K**	**CLK**	**Qn+1**
0	0	Rising	Qn
0	1	Rising	0
1	0	Rising	1
1	1	Rising	QnB
X	X	Falling	Qn

JK Flip-Flop Design

```
module JK (
   input J, K,
   input CLK,
   output logic Q
);
// synopsys sync_set_reset "J, K"
// synopsys one_hot "J, K"

always_ff @ (posedge CLK)
case ({J, K})
   2'b01 : Q <= 0;
   2'b10 : Q <= 1;
   2'b11 : Q <= ~Q;
endcase
endmodule
```

Inference Report

```
=========================================================================
| Register Name |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
=========================================================================
|    Q_reg      | Flip-flop |   1   |  N  | N  | N  | N  | Y  | Y  | N  |
=========================================================================
```

See Also

- Unintended Logic Inferred Using always_ff

B

Verilog Language Support

The following sections describe the Verilog language as supported by HDL Compiler:

- Syntax
- Verilog Keywords
- Unsupported Verilog Language Constructs
- Construct Restrictions and Comments
- Verilog 2001 and 2005 Supported Constructs
- Ignored Constructs
- Verilog 2001 Feature Examples
- Verilog 2005 Feature Example
- Configurations

Syntax

Synopsys supports the Verilog syntax as described in the *IEEE Std 1364-2001*.

The lexical conventions HDL Compiler uses are described in the following sections:

- Comments
- Numbers

Comments

You can enter comments anywhere in a Verilog description, in two forms:

- Beginning with two slashes //

  HDL Compiler ignores all text between these characters and the end of the current line.

- Beginning with the two characters /* and ending with */

  HDL Compiler ignores all text between these characters, so you can continue comments over more than one line.

  Note:
  You cannot nest comments.

Numbers

You can declare numbers in several different radices and bit-widths. A radix is the base number on which a numbering system is built. For example, the binary numbering system has a radix of 2, octal has a radix of 8, and decimal has a radix of 10.

You can use these three number formats:

- A simple decimal number that is a sequence of digits in the range of 0 to 9. All constants declared this way are assumed to be 32-bit numbers.
- A number that specifies the bit-width as well as the radix. These numbers are the same as those in the previous format, except that they are preceded by a decimal number that specifies the bit-width.
- A number followed by a two-character sequence prefix that specifies the number's size and radix. The radix determines which symbols you can include in the number. Constants declared this way are assumed to be 32-bit numbers. Any of these numbers can include underscores (_), which improve readability and do not affect the value of

the number. Table B-1 summarizes the available radices and valid characters for the number.

Table B-1 Verilog Radices

Name	Character prefix	Valid characters
Binary	’b	0 1 x X z Z _ ?
Octal	’o	0–7 x X z Z _ ?
Decimal	’d	0–9 _
Hexadecimal	’h	0–9 a–f A–F x X z Z _ ?

Example B-1 shows some valid number declarations.

Example B-1 Valid Verilog Number Declarations

```
391                 //  32-bit decimal number
’h3a13              //  32-bit hexadecimal number
10’o1567            //  10-bit octal number
3’b010              //  3-bit binary number
4’d9                //  4-bit decimal number
40’hFF_FFFF_FFFF    //  40-bit hexadecimal number
2’bxx               //  2-bits don’t care
3’bzzz              //  3-bits high-impedance
```

Verilog Keywords

Table B-2 lists the Verilog keywords. You cannot use these words as user variable names unless you use an escape identifier.

Important:

Configuration-related keywords are not treated as keywords outside of configurations. HDL Compiler does not support configurations at this time.

Table B-2 Verilog Keywords

always	and	assign	automatic	begin	buf
bufif0	bufif1	case	casex	casez	cell
cmos	config	deassign	default	defparam	design
disable	edge	else	end	endcase	endconfig

Table B-2 Verilog Keywords (Continued)

endfunction	endgenerate	endmodule	endprimitive	endspecify	endtable
endtask	event	for	force	forever	fork
function	generate	genvar	highz0	highz1	if
ifnone	incdir	include	initial	inout	input
instance	integer	join	large	liblist	library
localparam	macromodule	medium	module	nand	negedge
nmos	nor	noshowcancelled	not	notif0	notif1
or	output	parameter	pmos	posedge	primitive
pull0	pull1	pulldown	pullup	pulsestyle_onevent	pulsestyle_ondetect
rcmos	real	realtime	reg	release	repeat
rnmos	rpmos	rtran	rtranif0	rtranif1	scalared
showcancelled	signed	small	specify	specparam	strong0
strong1	supply0	supply1	table	task	time
tran	tranif0	tranif1	tri	tri0	tri1
triand	trior	trireg	unsigned	use	vectored
wait	wand	weak0	weak1	while	wire
wor	xnor	xor			

Unsupported Verilog Language Constructs

HDL Compiler does not support the following constructs:

- Configurations
- Unsupported definitions and declarations
  - primitive definition

    - time declaration
    - event declaration
    - triand, trior, tri1, tri0, and trireg net types
    - Ranges for integers
- Unsupported statements
    - initial statement
    - repeat statement
    - delay control
    - event control
    - forever statement (The forever loop is only supported if it has an associated disable condition, making the exit condition deterministic.)
    - fork statement
    - deassign statement
    - force statement
    - release statement
- Unsupported operators
    - Case equality and inequality operators (=== and !==)
- Unsupported gate-level constructs
    - nmos, pmos, cmos, rnmos, rpmos, rcmos
    - pullup, pulldown, tranif0, tranif1, rtran, rtrainf0, and rtrainf1 gate types
- Unsupported miscellaneous constructs
    - hierarchical names within a module

If you use an unsupported construct, HDL Compiler issues a syntax error such as

```
event is not supported
```

Construct Restrictions and Comments

Construct restrictions and guidelines are described in the following sections:

- always Blocks

- generate Statements
- Conditional Expressions (?:) Resource Sharing
- Case
- defparam
- disable
- Blocking and Nonblocking Assignments
- Macromodule
- inout Port Declaration
- tri Data Type
- HDL Compiler Directives
- reg Types
- Types in Busing
- Combinational while Loops

always Blocks

The tool does not support more than one independent `if` block when asynchronous behavior is modeled within an `always` block. If the `always` block is purely synchronous, the tool supports multiple independent `if` blocks. In addition, the tool does not support more than one conditional operator (?:) inside an `always` block.

Note:

> If an `always` block is very small, the tool might move the logic inside the block during synthesis.

generate Statements

Synopsys support of the `generate` statement is described in the following sections:

- Generate Overview
- Restrictions

Generate Overview

HDL Compiler supports both the 2001 and the 2005 `standards for the generate` statement. The default is the 2005 standard; to enable the 2001 standard, set the `hdlin_vrlg_std` variable to `2001`. The following subsections describe the naming-style differences between these two standards.

Types of generate Blocks

Standalone generate Blocks

Standalone generate blocks are blocks using the `begin` statement that are not associated with a *conditional generate* or l*oop generate* block. These are legal under the 2001 standard, but are illegal according to the Verilog 2005 LRM, as illustrated in the following example.

Example B-2 Standalone generate Block

```
module top ( input in1, output out1 );
  generate
begin : b1
  mod1 U1(in1, out1);
end
endgenerate
endmodule

module mod1( input in1, output out1 );
endmodule
```

When you use the 2001 standard, HDL Compiler creates the name b1.U1 for mod 1:

```
Cell           Reference       Library            Area  Attributes
-------------------------------------------------------------------
b1.U1          mod1                           0.000000  b
-------------------------------------------------------------------
Total 1 cells                                 0.000000
```

When you use the 2005 standard, HDL Compiler issues a VER-946 error message:

```
Compiling source file RTL/t1.v
Error:  RTL/t1.v:3: Syntax error on an obsolete Verilog 2001 construct
 standalone generate block 'b1'. (VER-946)
*** HDLC compilation terminated with 1 errors. ***
```

Anonymous generate Blocks

Anonymous generate blocks are `generate` blocks that do not have a user-defined label. They are also referred to as unnamed blocks.

According to the 2001 Verilog LRM, anonymous blocks do not create their own scope, but the 2005 standard has an implicit naming convention that allows scope creation. The Verilog

2005 standard assigns a number to every `generate` construct in a given scope. The number is 1 for the first construct and is incremented by 1 for each subsequent `generate` construct in the scope. All unnamed `generate` blocks are given the name genblk`n`, where `n` is the number assigned to the enclosing `generate` construct. If the name conflicts with an explicitly declared name, leading zeros are added in front of the number until the conflict is resolved.

Example B-7 shows the difference between the two standards.

Example B-3 Anonymous generate Block

```
module top( input [0:3] in1, output [0:3] out1 );
genvar I;
generate
for( I = 0; I < 3; I = I+1 ) begin: b1
  if( 1 ) begin : b2
    if( 1 )
      if( 1 )
        if( 1 )
          mod1 U1(in1[I], out1[I]);
   end
end
endgenerate
endmodule

module mod1( input in1, output out1 );
endmodule
```

When you use the Verilog 2001 standard, HDL Compiler creates the names b1[0].b2.U1, b1[1].b2.U1, and b1[2].b2.U1 for the instantiated subblocks:

```
Cell                      Reference       Library       Area  Attributes
-------------------------------------------------------------------------
b1[0].b2.U1               mod1                      0.000000  b
b1[1].b2.U1               mod1                      0.000000  b
b1[2].b2.U1               mod1                      0.000000  b
-------------------------------------------------------------------------
Total 3 cells                                       0.000000
```

When you use the Verilog 2005 standard, HDL Compiler creates the names b1[0].b2.genblk1.U1, b1[1].b2.genblk1.U1, and b1[2].b2.genblk1.U1. Note that there are no multiple genblk1's for the nested anonymous `if` blocks:

```
Cell                      Reference       Library       Area  Attributes
-------------------------------------------------------------------------
b1[0].b2.genblk1.U1        mod1                      0.000000  b
b1[1].b2.genblk1.U1        mod1                      0.000000  b
b1[2].b2.genblk1.U1        mod1                      0.000000  b
-------------------------------------------------------------------------
Total 3 cells                                       0.000000
```

Another type of anonymous `generate` block is created when the block does not have a label, but each block has a `begin` ...`end` statement:

Example B-4 Anonymous generate Block with begin...end

```
module top( input [0:3] in1, output [0:3] out1 );
genvar I;
generate
for( I = 0; I < 3; I = I+1 ) begin: b1
  if( 1 ) begin : b2
    if( 1 ) begin
      if( 1 ) begin
        if( 1 ) begin
          mod1 U1(in1[I], out1[I]);
        end
      end
    end
  end
end
endgenerate
endmodule

module mod1( input in1, output out1 );
endmodule
```

When you use the 2001 standard, HDL Compiler creates the names b1[0].b2.U1, b1[1].b2.U1, and b1[2].b2.U1 for the instantiated subblocks:

```
Cell                      Reference       Library       Area  Attributes
-------------------------------------------------------------------------
b1[0].b2.U1               mod1                      0.000000  b
b1[1].b2.U1               mod1                      0.000000  b
b1[2].b2.U1               mod1                      0.000000  b
-------------------------------------------------------------------------
Total 3 cells                                       0.000000
```

When you use the 2005 standard, the tool creates the names b1[0].b2.genblk1.genblk1.genblk1.U1, b1[1].b2.genblk1.genblk1.genblk1.U1, b1[2].b2.genblk1.genblk1.genblk1.U1:

```
Cell                    Reference       Library         Area  Attributes
-------------------------------------------------------------------------
b1[0].b2.genblk1.genblk1.genblk1.U1
                          mod1                      0.000000  b
b1[1].b2.genblk1.genblk1.genblk1.U1
                          mod1                      0.000000  b
b1[2].b2.genblk1.genblk1.genblk1.U1
                          mod1                      0.000000  b
-------------------------------------------------------------------------
Total 3 cells                                       0.000000
```

Note that there is a genblk1 for each of the nested `begin`...`end if` blocks that creates a new scope.

The following example illustrates how scope creation can produce an error under the Verilog 2005 standard from code that compiles cleanly under the Verilog 2001 standard:

Example B-5 Scope Creation

```
module top(input in, output out);
generate if(1) begin
  wire w = in;
end endgenerate
assign out = w;
endmodule
```

Under the Verilog 2001 standard, `w` is visible in the `assign` statement, but under the Verilog 2005 standard, scope creation makes `w` invisible outside the `generate` block, and HDL Compiler issues an error message:

```
Error:  RTL/t5.v:5: The symbol 'w' is not defined. (VER-956)
```

Loop Generate Blocks and Conditional Generate Blocks

Loop generate blocks are `generate` blocks that contain a `for` loop. *Conditional generate blocks* are `generate` blocks that contain an `if` statement. Loop generate blocks and conditional generate blocks can be nested, as shown in the following example.

Example B-6 Loop and Conditional generates

```
module top( input D1, input clk, output Q1 );
genvar i, j;
parameter param1 = 0;
parameter param2 = 1;

generate
for (i=0; i < 3; i=i+1) begin : loop1
  for (j=0; j < 2; j=j+1) begin : loop2
    if (j == param1) begin : if1_label
    memory U_00 (D1,clk,Q1);
    end
    if (j == param2) begin : if2_label
    memory U_00 (D1,clk,Q1);
    end
  end //loop2
end //loop1
endgenerate
endmodule

module memory( input D1, input clk, output Q1 );
endmodule
```

In this case, the instance name is the same under both standards:

```
Cell                      Reference       Library        Area  Attributes
-------------------------------------------------------------------------
loop1[0].loop2[0].if1_label.U_00
```

```
                          memory                     0.000000  b
loop1[0].loop2[1].if2_label.U_00
                          memory                     0.000000  b
loop1[1].loop2[0].if1_label.U_00
                          memory                     0.000000  b
loop1[1].loop2[1].if2_label.U_00
                          memory                     0.000000  b
loop1[2].loop2[0].if1_label.U_00
                          memory                     0.000000  b
loop1[2].loop2[1].if2_label.U_00
                          memory                     0.000000  b
--------------------------------------------------------------------------
Total 6 cells
```

Restrictions

- Hierarchical Names (Cross Module Reference)

  HDL Compiler supports hierarchical names or cross-module references, if the hierarchical name remains inside the module that contains the name and each item on the hierarchical path is part of the module containing the reference.

  In the following code, the item is not part of the module and is not supported.

```
    module top ();
      wire x;
      down d ();
    endmodule

    module down ();
      wire y, z;
      assign t = top.d.z;
  // not supported:
  // hier. ref. starts outside current module
    endmodule
```

- Parameter Override (defparam)

  The use of defparam is highly discouraged in synthesis because of ambiguity problems. Because of these problems, defparam is not supported inside generate blocks. For details, see the Verilog 1800 LRM.

Conditional Expressions (?:) Resource Sharing

HDL Compiler supports resource sharing in conditional expressions such as

```
 dout = sel ? (a + b) : (a + c);
```

In such cases, HDL Compiler marks the adders as sharable; Design Compiler determines the final implementation during timing-drive resource sharing.

The tool does not support more than one ?: operator inside an always block. For more information, see “always Blocks” on page B-6.

Case

The case construct is discussed in the following sections:

- casez and casex
- Full Case and Parallel Case

casez and casex

HDL Compiler allows ? and z bits in casez items but not in expressions; that is, the z bits are allowed in the branches of the case statement but not in the expression immediately following the casez keyword.

```
casez (y)   // y is referred to as the case expression

2’b1z:    //2’b1z is referred to as the item
```

Example B-7 shows an invalid expression in a casez statement.

Example B-7 Invalid casez Expression

```
casez (1’bz)  //illegal testing of an expression
    ...
endcase
```

The same holds true for casex statements using x, ?, and z. The code

```
casex (a)
2’b1x : // matches 2’b10 and 2’b11
endcase
```

does not equal the following code:

```
b = 2’b1x;
casex (a)
b:    // in this case, 2’b1x only matches 2’b10
endcase
```

When x is assigned to a variable and the variable is used in a casex item, the x does not match both 0 and 1 as it would for a literal x listed in the case item.

Full Case and Parallel Case

Case statements can be full or parallel. HDL Compiler can usually determine automatically whether a case statement is full or parallel. Example B-8 shows a case statement that is both full and parallel.

Example B-8 A case Statement That Is Both Full and Parallel

```
input [1:0] a;
always @(a or w or x or y or z) begin
   case (a)
      2’b11:
          b = w ;
      2’b10:
          b = x ;
      2’b01:
          b = y ;
      2’b00:
          b = z ;
   endcase
end
```

In Example B-9, the case statement is not parallel or full, because the values of inputs w and x cannot be determined.

Example B-9 A case Statement That Is Not Full and Not Parallel

```
always @(w or x) begin
   case (2’b11)
      w:
          b = 10 ;
      x:
          b = 01 ;
   endcase
end
```

However, if you know that only one of the inputs equals 2’b11 at a given time, you can use the `parallel_case` directive to avoid synthesizing an unnecessary priority encoder.

If you know that either w or x always equals 2’b11 (a situation known as a one-branch tree), you can use the `full_case` directive to avoid synthesizing an unnecessary latch. A latch is necessary whenever a variable is conditionally assigned. Marking a case as full tells the compiler that some branch will be taken, so there is no need for an implicit default branch. If a variable is assigned in all branches of the case, HDL Compiler then knows that the variable is not conditionally assigned in that case, and, therefore, that particular case statement does not result in a latch for that variable.

However, if the variable is assigned in only some branches of the case statement, a latch is still required as shown in Example B-10. In addition, other case statements might cause a latch to be inferred for the same variable.

Example B-10 Latch Result When Variable Is Not Fully Assigned

```
reg a, b;
reg [1:0] c;
case (c)   // synopsys full_case
    0: begin a = 1; b = 0; end
    1: begin a = 0; b = 0; end
    2: begin a = 1; b = 1; end
    3: b = 1;  // a is not assigned here
endcase
```

For more information, see parallel_case and full_case.

defparam

Use of defparam is highly discouraged in synthesis because of ambiguity problems. Because of these problems, defparam is not supported inside generate blocks. For details, see the Verilog LRM.

disable

HDL Compiler supports the disable statement when you use it in named blocks and when it is used to disable an enclosing block. When a disable statement is executed, it causes the named block to terminate. You cannot disable a block that is not in the same always block or task as the disable statement. A comparator description that uses disable is shown in Example B-11.

Example B-11 Comparator Using disable

```
begin : compare
   for (i = 7; i >= 0; i = i - 1) begin
    if (a[i] != b[i]) begin
         greater_than = a[i];
         less_than = ~a[i];
         equal_to = 0;
         //comparison is done so stop looping
         disable compare;
      end
   end

// If you get here a == b
// If the disable statement is executed, the next three
// lines will not be executed
   greater_than = 0;
   less_than = 0;
   equal_to = 1;
end
```

You can also use a disable statement to implement a synchronous reset, as shown in Example B-12.

Example B-12 Synchronous Reset of State Register Using disable in a forever Loop

```
always
   begin: test
      @ (posedge clk)
      if (Reset)
         begin
            z <= 1’b0;
            disable test;
         end
            z <= a;
   end
```

The disable statement in Example B-12 causes the test block to terminate immediately and return to the beginning of the block.

Blocking and Nonblocking Assignments

HDL Compiler does not allow both blocking and nonblocking assignments to the same variable within an always block.

The following code applies both blocking and nonblocking assignments to the same variable in one always block.

```
always @(posedge clk or negedge reset) begin
  if (~ reset)
    q = 1’b0;
  else
    q <= d;
end
```

HDL Compiler does not permit this and generates an error message.

During simulation, race conditions can result from blocking assignments, as shown in Example B-13. In this example, the value of x is indeterminate, because multiple procedural blocks run concurrently, causing y to be loaded into x at the same time z is loading into y. The value of x after the first @ (posedge clk) is indeterminate. Use of nonblocking assignments solves this race condition, as shown in Example B-14.

In Example B-13 and Example B-14, HDL Compiler creates the gates shown in Figure B-1.

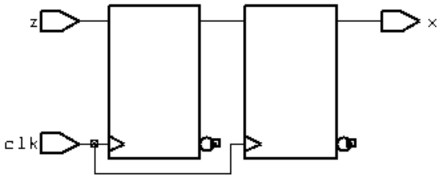
Example B-13 Race Condition Using Blocking Assignments

```
always @(posedge clk)
  x = y;
always @(posedge clk)
  y = z;
```

Example B-14 Race Solved With Nonblocking Assignments

```
always @(posedge clk)
  x <= y;
always @(posedge clk)
  y <= x;
```

Figure B-1 Simulator Race Condition—Synthesis Gates



If you want to switch register values, use nonblocking assignments, because blocking assignments will not accomplish the switch. For example, in Example B-15, the desired outcome is a swap of the x and y register values. However, after the positive clock edge, y does not end up with the value of x; y ends up with the original value of y. This happens because blocking statements are order dependent and each statement within the procedural block is executed before the next statement is evaluated and executed. In Example B-16, the swap is accomplished with nonblocking assignments.

Example B-15 Swap Problem Using Blocking Assignments

```
always @(posedge clk)
begin
  x = y;
  y = x;
end
```

Example B-16 Swap Accomplished With Nonblocking Assignments

```
always @(posedge clk)
  x <= y;
  y <= z;
```

Macromodule

HDL Compiler treats the macromodule construct as a module construct. Whether you use module or macromodule, the synthesis results are the same.

inout Port Declaration

HDL Compiler allows you to connect inout ports only to module or gate instantiations. You must declare an inout before you use it.

tri Data Type

The tri data type allows multiple three-state devices to drive a wire. When inferring three-state devices, you need to ensure that all the drivers are inferred as three-state devices and that all inputs to a device are z, except the one variable driving the three-state device which will have a 1.

HDL Compiler Directives

HDL compiler directives are discussed in the following sections:

- `define
- `include
- `ifdef, `else, `endif, `ifndef, and `elsif
- `rp_group and `rp_endgroup
- `rp_place
- `rp_fill
- `rp_array_dir
- rp_align
- rp_orient
- rp_ignore and rp_endignore
- `undef

`define

The `` `define `` directive can specify macros that take arguments. For example,

```
`define BYTE_TO_BITS(arg)((arg) << 3)
```

The `` `define `` directive can do more than simple text substitution. It can also take arguments and substitute their values in its replacement text.

Macro substitution assigns a string of text to a macro variable. The string of text is inserted into the code where the macro is encountered. The definition begins with the back quotation mark (`), followed by the keyword `define`, followed by the name of the macro variable. All text from the macro variable until the end of the line is assigned to the macro variable.

You can declare and use macro variables anywhere in the description. The definitions can carry across several files that are read into Design Compiler at the same time. To make a macro substitution, type a back quotation mark (`) followed by the macro variable name.

Some example macro variable declarations are shown in Example B-17.

Example B-17 Macro Variable Declarations

```
`define highbits       31:29
`define bitlist        {first, second, third}
wire [31:0] bus;
`bitlist = bus[‘highbits];
```

The `analyze -define` command allows macro definition on the command line. Only one `-define` per `analyze` command is allowed but the argument can be a list of macros, as shown in Example B-18.

Note:
When using the `-define` option with multiple analyze commands, you must remove any designs in memory before analyzing the design again. To remove the designs, use `remove_design -all`. Because elaborated designs in memory have no timestamps, the tool cannot determine whether the analyzed file has been updated or not. The tool might assume that the previously elaborated design is up-to-date and reuse it.

Curly brackets are not required to enclose one macro, as shown in Example B-19. However, if the argument is a list of macros, curly brackets are required.

Example B-18 analyze Command With List of Defines

```
analyze -f verilog -define { RIPPLE, SIMPLE } mydesign.v
```

Example B-19 analyze Command With One Define

```
analyze -f verilog -define ONLY_ONE mydesign.v
```

Note:
In dctcl mode, the `read_verilog` command does not accept the `-define` option.

See Also

- Predefined SYSTEMVERILOG Macro

`include

The `` `include `` construct in Verilog is similar to the `#include` directive in the C language. You can use this construct to include Verilog code, such as type declarations and functions, from one module in another module. Example B-20 shows an application of the `` `include `` construct.

Example B-20 Including a File Within a File

```
Contents of file1.v
`define WORDSIZE 8

function [`WORDSIZE-1:0] fastadder;
  input [`WORDSIZE-1:0] fin1, fin2;
  fastadder = fin1 + fin2;
endfunction

Contents of file2.v
module secondfile (clk, in1, in2, out);

`include "file1.v"
. . .
wire [`WORDSIZE-1:0] temp;
assign temp = fastadder (in1,in2);
. . .
endmodule
```

Included files can include other files, with up to 24 levels of nesting. You cannot use the `` `include`` construct recursively.

When your design contains multiple files for multiple subblocks and include files for subblocks, in their respective sub directories, you can elaborate the top-level design without making any changes to the search path. The tool will automatically find the include files. For example, if your structure is as follows:

```
Rtl/top.v
Rtl/sub_module1/sub_module1.v
Rtl/sub_module2/sub_module2.v
Rtl/sub_module1/sub_module1_inc.v
Rtl/sub_module2/sub_module2_inc.v
```

You do not need to add Rtl/sub_module1/ and Rtl/sub_module2/ to your search path to enable the tool to find the include files sub_module1_inc.v and sub_module2_inc.v when you elaborate top.v.

`ifdef, `else, `endif, `ifndef, and `elsif

These directives allow the conditional inclusion of code.

- The `` `ifdef`` directive executes the statements following it if the indicated macro is defined; if the macro is not defined, the statements after `` `else`` are executed.
- The`` `ifndef`` directive executes the statements following it if the indicated macro is not defined; if the macro is defined, the statements after `` `else`` are executed.
- The `` `elsif`` directive allows one level of nesting and is equivalent to the `` `else `ifdef ... `endif`` directive sequence.

Example B-21 illustrates usage. Use the `` `define `` directive to define the macros that are arguments to the `` `ifdef `` directive; see `define.

Example B-21 Design Using `ifdef...`else...`endif Directives

```
`ifdef SELECT_XOR_DESIGN
module selective_design(a,b,c);
  input a, b;
  output c;
    assign c = a ^ b;
endmodule

`else

module selective_design(a,b,c);
  input a, b;
  output c;
    assign c = a | b;
endmodule
`endif
```

`rp_group and `rp_endgroup

The `` `rp_group `` and `` `rp_endgroup `` directives allow you to specify a relative placement group. All cell instances declared between the directives are members of the specified group. These directives are available for RTL designs and netlist designs.

The Verilog syntax for RTL designs is as follows:

```
`rp_group ( group_name {num_cols num_rows} )
`rp_endgroup ( {group_name} )
```

Use the following syntax for netlist designs:

```
//synopsys rp_group ( group_name {num_cols num_rows} )
//synopsys rp_endgroup ( {group_name} )
```

For more information and an example, see Specifying Relative Placement Groups.

`rp_place

The `` `rp_place `` directive allows you to specify a subgroup at a specific hierarchy, a keepout region, or an instance to be placed in the current relative placement group. When you use the `` `rp_place `` directive to specify a subgroup at a specific hierarchy, you must instantiate the subgroup's instances outside of any group declarations in the module. This directive is available for RTL designs and netlist designs.

The Verilog syntax for RTL designs is as follows:

```
‘rp_place ( hier group_name col row )
‘rp_place ( keep keepout_name col row width height )
‘rp_place ({leaf} [inst_name] col row )
```

Use the following syntax for netlist designs:

```
//synopsys rp_place ( hier group_name col row )
//synopsys rp_place ( hier group_name [inst_name] col row )
//synopsys rp_place ({leaf} [inst_name] col row )
//synopsys rp_place ( keep keepout_name col row width height )
```

For more information and examples, see Specifying Subgroups, Keepouts, and Instances.

`rp_fill

The `` `rp_fill `` directive automatically places the cells at the location specified by a pointer. Each time a new instance is declared that is not explicitly placed, it is inserted into the grid at the location indicated by the current value of the pointer. After the instance is placed, the pointer is updated incrementally and the process is ready to be repeated. This directive is available for RTL designs and netlist designs.

The `` `rp_fill `` arguments define how the pointer is updated. The *`col`* and *`row`* parameters specify the initial coordinates of the pointer. These parameters can represent absolute row or column locations in the group's grid or locations that are relative to the current pointer value. To represent locations relative to the current pointer, enclose the column and row values in angle brackets (<>). For example, assume the current pointer location is (3,4). In this case, specifying `rp_fill <1> 0` initializes the pointer to (4,0) and that is where the next instance is placed. Absolute coordinates must be nonnegative integers; relative coordinates can be any integer.

The Verilog syntax for RTL designs is as follows:

```
‘rp_fill ( {col row} {pattern pat} )
```

Use the following syntax for netlist designs:

```
//synopsys rp_fill ( col row} {pattern pat} )
```

For more information and an example, see Enabling Automatic Cell Placement.

`rp_array_dir

Note:

> This directive is available for creating relative placement in RTL designs but not in netlist designs.

The `` `rp_array_dir `` directive specifies whether the elements of an array are placed upward, from the least significant bit to the most significant bit, or downward, from the most significant bit to the least significant bit.

The Verilog syntax for RTL designs is as follows:

```
`rp_array_dir ( up|down )
```

For more information and an example, see Specifying Placement for Array Elements.

rp_align

The `rp_align` directive explicitly specifies the alignment of the placed instance within the grid cell when the instance is smaller than the cell. If you specify the optional *`inst`* instance name argument, the alignment applies only to that instance; however, if you do not specify an instance, the new alignment applies to all subsequent instantiations within the group until HDL Compiler encounters another `rp_align` directive. If the instance straddles cells, the alignment takes place within the straddled region. The alignment value is `sw` (southwest) by default. The instance is snapped to legal row and routing grid coordinates.

Use the following syntax for netlist designs:

```
//synopsys rp_align ( n|s|e|w|nw|sw|ne|se|pin=name { inst })
```

Note:
This directive is available for creating relative placement in netlist designs only.

For more information and an example, see Specifying Cell Alignment.

rp_orient

Note:
This directive is available for creating relative placement in netlist designs only.

The `rp_orient` directive allows you to control the orientation of library cells placed in the current group. When you specify a list of possible orientations, HDL Compiler chooses the first legal orientation for the cell.

Use the following syntax for netlist designs:

```
//synopsys rp_orient ( {N|W|S|E|FN|FW|FS|FE}* { inst } )
//synopsys
rp_orient ( {N|W|S|E|FN|FW|FS|FE}* { group_name inst } ))
```

For more information and an example, see Specifying Cell Orientation.

rp_ignore and rp_endignore

Note:
This directive is available for creating relative placement in netlist designs only.

The `rp_ignore` and `rp_endignore` directives allow you to ignore specified lines in the input file. Any lines between the two directives are omitted from relative placement. The `include` and `define` directives, variable substitution, and cell mapping are not ignored.

The rp_ignore and rp_endignore directives allow you to include the instantiation of submodules in a relative placement group close to the rp_place hier group(inst) location to place relative placement array.

Use the following syntax for netlist designs:

```
//synopsys rp_ignore
//synopsys rp_endignore
```

For more information and an example, see Ignoring Relative Placement.

`undef

The `undef directive resets the macro immediately following it.

reg Types

The Verilog language requires that any value assigned inside an always statement must be declared as a reg type. HDL Compiler returns an error if any value assigned inside an always block is not declared as a reg type.

Types in Busing

Design Compiler maintains types throughout a design, including types for buses (vectors). Example B-22 shows a Verilog design read into HDL Compiler containing a bit vector that is NOTed into another bit vector.

Example B-22 Bit Vector in Verilog

```
module test_busing_1 ( a, b );
  input  [3:0] a;
  output [3:0] b;

  assign b = ~a;
endmodule
```

Example B-23 shows the same description written out by HDL Compiler. The description contains the original Verilog types of ports. Internal nets do not maintain their original bus types. Also, the NOT operation is instantiated as single bits.

Example B-23 Bit Blasting

```
module test_busing_2 ( a, b );
  input  [3:0] a;
  output [3:0] b;
    assign b[0] = ~a[0];
    assign b[1] = ~a[1];
    assign b[2] = ~a[2];
    assign b[3] = ~a[3];
endmodule
```

Combinational while Loops

To create a combinational while loop, write the code so that an upper bound on the number of loop iterations can be determined. The loop iterative bound must be statically determinable; otherwise an error is reported.

HDL Compiler needs to be able to determine an upper bound on the number of trips through the loop at compile time. In HDL Compiler, there are no syntax restrictions on the loops; while loops that have no events within them, such as in the following example, are supported.

```
input [9:0] a;
// ....
i = 0;
while ( i < 10 && !a[i] ) begin
  i = i + 1;
  // loop body
end
```

To support this loop, HDL Compiler interprets it like a simulator. The tool stops when the loop termination condition is known to be false. Because HDL Compiler can’t determine when a loop is infinite, it stops and reports an error after an arbitrary (but user-defined) number of iterations (the default is 1024).

To exit the loop, HDL Compiler allows additional conditions in the loop condition that permit more concise descriptions.

```
for (i = 0; i < 10 && a[i]; i = i+1) begin
  // loop body
end
```

A loop must unconditionally make progress toward termination in each trip through the loop, or it cannot be compiled. The following example makes progress (that is, increments i) only when !done is true and does not terminate.

```
while ( i < 10 ) begin
  if ( ! done )
    done = a[i];
    // loop body
    i = i + 1;
  end
end
```

The following modified version, which unconditionally increments i, will terminate. This code creates the desired logic.

```
while ( i < 10 ) begin
  if ( ! done ) begin
    done = a[i];
    end// loop body
    i = i + 1;
end
```

In the next example, loop termination depends on reading values stored in x. If the value is unknown (as in the first and third iterations), HDL Compiler assumes it might be true and generates logic to test it.

```
x[0] = v;       // Value unknown: implies “if(v)”
x[1] = 1;       // Known TRUE: no guard on 2nd trip
x[2] = w;       // Not known: implies “if(w)”
x[3] = 0;       // Known FALSE: stop the loop

i = 0;
while( x[i] ) begin
  // loop body
  i = i + 1;
end
```

This code terminates after three iterations when the loop tests x[3], which contains 0.

In Example B-24, a supported combinational while loop, the code produces gates, and an event control signal is not necessary.

Example B-24 Supported while Loop Code

```
module modified_s2 (a, b, z);
parameter N = 3;
input [N:0] a, b;
output [N:1] z;
reg [N:1] z;
integer i;
always @(a or b or z)
    begin
        i = N;
        while (i)
            begin
                z[i] = b[i] + a[i-1];
                i = i - 1;
            end
      end
endmodule
```

In Example B-25, a supported combinational while loop, no matter what *x* is, the loop will run for 16 iterations at most because HDL Compiler can keep track of which bits of *x* are constant. Even though it doesn't know the initial value of *x*, it does know that *x* >> 1 has a zero in the most significant bit (MSB). The next time *x* is shifted right, it knows that x has two zeros in the MSB, and so on. HDL Compiler can determine when *x* becomes all zeros.

Example B-25 Supported Combinational while Loop

```
module while_loop_comb1(x, count);
  input [7:0] x;
  output [2:0] count;
  reg [7:0] temp;
  reg [2:0] count;
  always @ (x)
  begin
      temp = x;
      count = 0;
      while (temp != 0)
      begin
      count = count + 1;
      temp = temp >> 1;
      end
  end
endmodule
```

In Example B-26, a supported combinational while loop, HDL Compiler knows the initial value of *x* and can determine *x+1* and all subsequent values of *x*.

Example B-26 Supported Combinational while Loop

```
module while_loop_comb2(y, count1, z);
  input [3:0] y, count1;output [3:0] z;
  reg [3:0] x, z, count;
  always @ (y, count1)
  begin
    x = 2;
    count = count1;
    while (x < 15)
      begin
        count = count + 1;
        x = x + 1;
      end
    z = count;
  end
endmodule
```

In Example B-27, HDL Compiler cannot detect the initial value of i and so cannot support this while loop. Example B-28 is supported because i is determinable.

Example B-27 Unsupported Combinational while Loop

```
module my_loop1 #(parameter N=4) (input [N:0] in, output reg [2*N:0] out);
  reg [N:0] i;
  always @* begin
  i = in;
  out = 0 ;
  while (i>0) begin
    out = out + i;
    i = i - 1;
  end
end
endmodule
```

Example B-28 Supported Combinational while Loop

```
module my_loop2 #(parameter N=4) (input [N:0] in, output reg [2*N:0] out);
  reg [N:0] i;
  reg [N+1:0] j;
  always @*
  for (j = 0 ; j < (2<<N) ; j = j+1 )
    if (j==in) begin
      i = j;
      out = 0 ;
      while (i>0) begin
        out = out + i;
        i = i - 1;
      end
    end
endmodule
```

Verilog 2001 and 2005 Supported Constructs

Table B-3 lists the Verilog 2001 and 2005 features implemented by HDL Compiler. For additional information about these features, see the *IEEE Std 1364-2001*.

Table B-3 Supported Verilog 2001 and 2005 Constructs

Feature	Description
Automatic tasks and functions	Fully supported
Constant functions	Fully supported
Local parameter	Fully supported
generate statement	See generate Statements.
SYNTHESIS macro	Fully supported
Implicit net declarations for continuous assignments	Fully supported
`line directive	Fully supported
ANSI-C-style port declarations	Fully supported
Casting operators	Fully supported
Parameter passing by name (IEEE 12.2.2.2)	Fully supported
Implicit event expression list (IEEE 9.7.5)	Fully supported
ANSI-C-style port declaration (IEEE 12.3.3)	Fully supported
Signed/unsigned parameters (IEEE 3.11)	Fully supported
Signed/unsigned nets and registers (IEEE 3.2, 4.3)	Fully supported
Signed/unsigned sized and based constants (IEEE 3.2)	Fully supported

Table B-3 Supported Verilog 2001 and 2005 Constructs (Continued)

Feature	Description
Multidimensional arrays and arrays of nets (IEEE 3.10)	Fully supported
Part select addressing ([+:] and [-:] operators) (IEEE 4.2.1)	Fully supported
Power operator (**) (IEEE 4.1.5)	Fully supported
Arithmetic shift operators (<<< and >>>) (IEEE 4.1.12)	Fully supported
Sized parameters (IEEE 3.11.1)	Fully supported
`ifndef, `elsif, `undef (IEEE 19.4,19.3.2)	Fully supported
`ifdef VERILOG_2001 and `ifdef VERILOG_1995	Fully supported
Comma-separated sensitivity lists (IEEE 4.1.15 and 9.7.4)	Fully supported

Ignored Constructs

The following sections include directives that HDL Compiler accepts but ignores.

Simulation Directives

The following directives are special commands that affect the operation of the Verilog HDL simulator:

```
'accelerate
'celldefine
'default_nettype
'endcelldefine
'endprotect
'expand_vectornets
'noaccelerate
'noexpand_vectornets
'noremove_netnames
'nounconnected_drive
'protect
```

```
'remove_netnames
'resetall
'timescale
'unconnected_drive
```

You can include these directives in your design description; HDL Compiler accepts but ignores them.

Verilog System Functions

Verilog system functions are special functions that Verilog HDL simulators implement. Their names start with a dollar sign ($). All of these functions are accepted but ignored by HDL Compiler with the exception of $display, which can be useful during synthesis elaboration. See Using $display During RTL Elaboration.

Verilog 2001 Feature Examples

This section provides examples for Verilog 2001 features in the following sections:

- Multidimensional Arrays and Arrays of Nets
- Signed Quantities
- Comparisons With Signed Types
- Controlling Signs With Casting Operators
- Part-Select Addressing Operators ([+:] and [-:])
- Power Operator (**)
- Arithmetic Shift Operators (<<< and >>>)

Multidimensional Arrays and Arrays of Nets

HDL Compiler supports multidimensional arrays of any variable or net data type. This added functionality is shown in the following examples:

Example B-29 Multidimensional Arrays

```
module m (a, z);
  input [7:0] a;
  output z;
  reg t [0:3][0:7];
  integer i, j;
  integer k;
  always @(a)
    begin
     for (j = 0; j < 8; j = j + 1)
      begin
        t[0][j] = a[j];
      end
     for (i = 1; i < 4; i = i + 1)
      begin
        k = 1 << (3-i);
       for (j = 0; j < k; j = j + 1)
         begin
           t[i][j] = t[i-1][2*j] ^ t[i-1][2*j+1];
         end
      end
  end
  assign z = t[3][0];
endmodule
```

Example B-30 Arrays of Nets

```
module m (a, z);
  input [0:3] a;
  output z;
  wire x [0:2] ;
  assign x[0] = a[0] ^ a[1];
  assign x[1] = a[2] ^ a[3];
  assign x[2] = x[0] ^ x[1];
  assign z = x[2];
endmodule
```

Example B-31 Multidimensional Array Variable Subscripting

```
reg [7:0] X [0:7][0:7][0:7];

assign out = X[a][b][c][d+:4];
```

Verilog 2001 allows more than one level of subscripting on a variable, without use of a temporary variable.

Example B-32 Multidimensional Array

```
module test(in, en, out, addr_in, addr_out_reg, addr_out_bit, clk);

  input [7:0] in;
  input en, clk;
  input [2:0] addr_in, addr_out_reg, addr_out_bit;
  reg [7:0] MEM [0:7];
  output out;

  assign out = MEM[addr_out_reg][addr_out_bit];

  always @(posedge clk) if (en) MEM[addr_in] = in;
endmodule
```

Signed Quantities

HDL Compiler supports signed arithmetic extensions. Function returns and reg and net data types can be declared as signed. This added functionality is shown in examples B-33 through B-38.

Example B-33 results in a sign extension, that is, z[0] connects to a[0].

Example B-33 Signed I/O Ports

```
module m1 (a, z);
  input signed [0:3] a;
  output signed [0:4] z;
  assign z = a;
endmodule
```

In Example B-34, because 3’sb111 is signed, the tool infers a signed adder. In the generic netlist, the ADD_TC_OP cell denotes a 2’s complement adder and z[0] will not be logic 0.

Example B-34 Signed Constants: Code and GTECH Gates

```
module m2 (a, z);
  input signed [0:2] a;
  output [0:4] z;
  assign z = a + 3’sb111;
endmodule
```

In Example B-35, because 4’sd5 is signed, a signed comparator (LT_TC_OP) is inferred.

Example B-35 Signed Registers: Code and GTECH Gates

```
 module m3 (a, z);
  input [0:3] a;
  output z;
  reg signed [0:3] x;
  reg z;
  always begin
    x = a;
    z = x < 4’sd5;
  end
endmodule
```

In Example B-36, because in1, in2, and out are signed, a signed multiplier (MULT_TC_OP_8_8_8) is inferred.

Example B-36 Signed Types: Code and Gates

```
module m4 (in1, in2, out);
   input  signed [7:0] in1, in2;
   output signed [7:0] out;
   assign out = in1 * in2;
endmodule
```

The code in Example B-37 results in a signed subtractor (SUB_TC_OP).

Example B-37 Signed Nets: Code and Gates

```
module m5 (a, b, z);
  input  [1:0] a, b;
  output [2:0] z;
  wire signed [1:0] x = a;
  wire signed [1:0] y = b;
  assign z = x - y;
endmodule
```

In Example B-38, because 4’sd5 is signed, a signed comparator (LT_TC_OP) is inferred.

Example B-38 Signed Values

```
module m6 (a, z);
  input [3:0] a;
  output z;
  reg signed [3:0] x;
  wire z;
  always @(a) begin
    x = a;
  end
  assign z = x < -4’sd5;
endmodule
```

Verilog 2001 adds the signed keyword in declarations:

```
reg signed [7:0] x;
```

It also adds support for signed, sized constants. For example, 8'sb11111111 is an 8-bit signed quantity representing -1. If you are assigning it to a variable that is 8 bits or less, 8'sb11111111 is the same as the unsigned 8'b11111111. A behavior difference arises when the variable being assigned to is larger than the constant. This difference occurs because signed quantities are extended with the high-order bit of the constant, whereas unsigned quantities are extended with 0s. When used in expressions, the sign of the constant helps determine whether the operation is performed as signed or unsigned.

HDL Compiler enables signed types by default.

Note:

If you use the `signed` keyword, any signed constant in your code, or explicit type casting between signed and unsigned types, HDL Compiler issues a warning.

Comparisons With Signed Types

Verilog sign rules are tricky. All inputs to an expression must be signed to obtain a signed operator. If one is signed and one unsigned, both are treated as unsigned. Any unsigned quantity in an expression makes the whole expression unsigned; the result doesn’t depend on the sign of the left side. Some expressions always produce an unsigned result; these include bit and part-select and concatenation. See IEEE P1364/P5 Section 4.5.1.

You need to control the sign of the inputs yourself if you want to compare a signed quantity against an unsigned one. The same is true for other kinds of expressions. See Example B-39 and Example B-40.

Example B-39 Unsigned Comparison Results When Signs Are Mismatched

```
module m8 (in1, in2, lt);
// in1 is signed but in2 is unsigned
   input signed [7:0] in1;
   input        [7:0] in2;
   output lt;
   wire uns_lt, uns_in1_lt_64;
/* comparison is unsigned because of the sign mismatch, in1
is signed but in2 is unsigned */
   assign uns_lt = in1 < in2;
/* Unsigned constant causes unsigned comparison; so negative
values of in1 would compare as larger than 8’d64 */
   assign uns_in1_lt_64 = in1 < 8'd64;
   assign lt = uns_lt + uns_in1_lt_64;
endmodule
```

Example B-40 Signed Values

```
module m7 (in1, in2, lt, in1_lt_64);
   input  signed [7:0] in1, in2;  // two signed inputs
   output lt, in1_lt_64;
   assign lt  =  in1 < in2;     // comparison is signed
   // using a signed constant results in a signed comparison
   assign in1_lt_64 = in1 < 8'sd64;
endmodule
```

Controlling Signs With Casting Operators

Use the Verilog 2001 casting operators, $signed() and $unsigned(), to convert an unsigned expression to a signed expression. In Example B-41, the casting operator is used to obtain a signed comparator. Note that simply marking an expression as signed might give undesirable results because the unsigned value might be interpreted as a negative number. To avoid this problem, zero-extend unsigned quantities, as shown in Example B-41.

Example B-41 Casting Operators

```
module m9 (in1, in2, lt);
   input signed [7:0] in1;
   input        [7:0] in2;
   output lt;
   assign lt = in1 < $signed ({1'b0, in2});
  //Cast to get signed comparator.
  //Zero-extend to preserve interpretation of unsigned value as positive
number.
```

Part-Select Addressing Operators ([+:] and [-:])

Verilog 2001 introduced variable part-select operators. These operators allow you to use variables to select a group of bits from a vector. In some designs, coding with part-select operators improves elaboration time and memory usage.

Variable part-select operators are discussed in the following sections:

- Variable Part-Select Overview
- Example—Ascending Array and +:
- Example—Ascending Array and -:
- Example—Descending Array and the -: Operator
- Example—Descending Array and the +: Operator

Variable Part-Select Overview

A Verilog 1995 part-select operator requires that both upper and lower indexes be constant: a[2:3] or a[value1:value2].

The variable part-select operator permits selection of a fixed-width group of bits at a variable base address and takes the following form:

- [base_expr +: width_expr] for a positive offset
- [base_expr -: width_expr] for a negative offset

The syntax specifies a variable base address and a known constant number of bits to be extracted. The base address is always written on the left, regardless of the declared direction of the array. The language allows variable part-select on the left side and the right side of an expression. All of the following expressions are allowed:

- data_out = array_expn[index_var +: 3]
  (part select is on the right side)
- data_out = array_expn[index_var -: 3]
  (part select is on the right side)
- array_expn[index_var +: 3] = data_in
  (part select is on the left side)
- array_expn[index_var -: 3] = data_in
  (part select is on the left side)

This table shows examples of Verilog 2001 syntax and the equivalent Verilog 1995 syntax.

Verilog 2001 syntax	Equivalent Verilog 1995 syntax	
a[x +: 3] for a descending array	{ a[x+2], a[x+1], a[x] }	a[x+2 : x]
a[x -: 3] for a descending array	{ a[x], a[x-1], a[x-2] }	a[x : x-2]
a[x +: 3] for an ascending array	{ a[x], a[x+1], a[x+2] }	a[x : x+2]
a[x -: 3] for an ascending array	{ a[x-2], a[x-1], a[x] }	a[x-2 : x]

The original HDL Compiler tool allows nonconstant part-selects if the width is constant; HDL Compiler permits only the new syntax.

Example—Ascending Array and -:

The following Verilog code uses the -: operator to select bits from Ascending_Array.

```
reg [0:7] Ascending_Array;
...
   Data_Out = Ascending_Array[Index_Var -: 3];
```

The value of Index_Var determines the starting point for the bits selected. In the following table, the bits selected are shown as a function of Index_Var.

Ascending_Array	**[**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7]**
Index_Var = 0	not valid, synthesis/simulation mismatch								
Index_Var = 1	not valid, synthesis/simulation mismatch								
Index_Var = 2		**•**	**•**	**•**	•	•	•	•	•
Index_Var = 3		•	**•**	**•**	**•**	•	•	•	•
Index_Var = 4		•	•	**•**	**•**	**•**	•	•	•
Index_Var = 5		•	•	•	**•**	**•**	**•**	•	•
Index_Var = 6		•	•	•	•	**•**	**•**	**•**	•
Index_Var = 7		•	•	•	•	•	**•**	**•**	**•**

Ascending_Array[Index_Var -: 3] is functionally equivalent to the following part-select that is not computable:

Ascending_Array[Index_Var - 2 : Index_Var]

Example—Ascending Array and +:

The following Verilog code uses the +: operator to select bits from Ascending_Array.

```
reg [0:7] Ascending_Array;
...
   Data_Out = Ascending_Array[Index_Var +: 3];
```

The value of Index_Var determines the starting point for the bits selected. In the following table, the bits selected are shown as a function of Index_Var.

Ascending_Array	**[**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7]**
Index_Var = 0		**•**	**•**	**•**	•	•	•	•	•
Index_Var = 1		•	**•**	**•**	**•**	•	•	•	•

Ascending_Array	**[**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7]**
Index_Var = 2		•	•	▨	▨	▨	•	•	•
Index_Var = 3		•	•	•	▨	▨	▨	•	•
Index_Var = 4		•	•	•	•	▨	▨	▨	•
Index_Var = 5		•	•	•	•	•	▨	▨	▨
Index_Var = 6	not valid, synthesis/simulation mismatch; see the following note.								
Index_Var = 7	not valid, synthesis/simulation mismatch; see the following note.								

Note:

- Ascending_Array[Index_Var +: 3] is functionally equivalent to the following part-select that is not computable: Ascending_Array[Index_Var : Index_Var + 2]
- Noncomputable part-selects are not supported by the Verilog language. Ascending_Array[7 +:3] corresponds to elements Ascending_Array[7 : 9] but elements Ascending_Array[8] and Ascending_Array[9] do not exist. A variable part-select must always compute to a valid index; otherwise, a synthesis elaborate error and a runtime simulation error will result.

Example—Descending Array and the -: Operator

The following code uses the -: operator to select bits from Descending_Array.

```
reg [7:0] Descending_Array;
...
   Data_Out = Descending_Array[Index_Var -: 3];
```

The value of Index_Var determines the starting point for the bits selected. In the following table, the bits selected are shown as a function of Index_Var.

Descending_Array	**[7**	**6**	**5**	**4**	**3**	**2**	**1**	**0]**
Index_Var = 0	not valid, synthesis/simulation mismatch							
Index_Var = 1	not valid, synthesis/simulation mismatch							
Index_Var = 2	•	•	•	•	•	▨	▨	▨
Index_Var = 3	•	•	•	•	▨	▨	▨	•

Descending_Array	**[7**	**6**	**5**	**4**	**3**	**2**	**1**	**0]**
Index_Var = 4	•	•	•	▨	▨	▨	•	•
Index_Var = 5	•	•	▨	▨	▨	•	•	•
Index_Var = 6	•	▨	▨	▨	•	•	•	•
Index_Var = 7	▨	▨	▨	•	•	•	•	•

Descending_Array[Index_Var -: 3] is functionally equivalent to the following noncomputable part-select:

Descending_Array[Index_Var : Index_Var - 2]

Example—Descending Array and the +: Operator

The following Verilog code uses the +: operator to select bits from Descending_Array.

```
reg [7:0] Descending_Array;
...
   Data_Out = Descending_Array[Index_Var +: 3];
```

The value of Index_Var determines the starting point for the bits selected. In the following table, the bits selected are shown as a function of Index_Var.

Descending_Array	**[7**	**6**	**5**	**4**	**3**	**2**	**1**	**0]**
Index_Var = 0	•	•	•	•	•	▨	▨	▨
Index_Var = 1	•	•	•	•	▨	▨	▨	•
Index_Var = 2	•	•	•	▨	▨	▨	•	•
Index_Var = 3	•	•	▨	▨	▨	•	•	•
Index_Var = 4	•	▨	▨	▨	•	•	•	•
Index_Var = 5	▨	▨	▨	•	•	•	•	•
Index_Var = 6	not valid, synthesis/simulation mismatch							

Descending_Array		**[7**	**6**	**5**	**4**	**3**	**2**	**1**	**0]**
Index_Var = 7	not valid, synthesis/simulation mismatch								

Descending_Array[Index_Var +: 3] is functionally equivalent to the following noncomputable part-select:

Descending_Array[Index_Var + 2 : Index_Var]

Noncomputable part-selects are not supported by the Verilog language. Descending_Array[7 +:3] corresponds to elements Descending_Array[9 : 7] but elements Descending_Array[9] and Descending_Array[8] do not exist. A variable part-select must always compute to a valid index; otherwise, a synthesis elaborate error and a runtime simulation error will result.

Power Operator (**)

This operator performs y^x, as shown in Example B-42.

Example B-42 Power Operators

```
module m #(parameter b=2, c=4) (a, x, y, z);
  input [3:0] a;
  output [7:0] x, y, z;

  assign z = 2 ** a;
  assign x = a ** 2;
  assign y = b ** c; // where b and c are constants

endmodule
```

Arithmetic Shift Operators (<<< and >>>)

The arithmetic shift operators allow you to shift an expression and still maintain the sign of a value, as shown in Example B-43. When the type of the result is signed, the arithmetic shift operator (>>>) shifts in the sign bit; otherwise it shifts in zeros.

Example B-43 Shift Operator Code and Gates

```
module s1 (A, S, Q);
  input signed [3:0] A;
  input [1:0] S;
  output [3:0] Q;
  reg [3:0] Q;
  always @(A or S)
  begin

// arithmetic shift right,
// shifts in sign-bit from left

    Q = A >>> S;
  end
endmodule
```

Verilog 2005 Feature Example

Zero Replication

According to the Verilog 2005 LRM, a replication operation with a zero replication constant is considered to have a size of zero and is ignored. Such an operation can appear only within a concatenation in which at least one of the operands of the concatenation has a positive size.

Zero replication can be useful for parameterized designs. In the following example, the valid values for parameter P are 1 to 32.

```
module top  #(parameter P = 32)  ( input [32-1:0]a, output [32-1:0] b);
assign b = {{32-P{1'b1}}, a[P-1:0]};
endmodule
```

When `hdlin_vrlg_std` is set to `2005`, and you analyze replication operations whose elaboration-time constant is zero or negative, the repeated expressions elaborate once (for their side-effects). But they do not contribute result values to a surrounding concatenation or assignment pattern. The Verilog 2005 standard permits such empty replication results only within an otherwise nonempty concatenation

Note:

> Nonstandard replication operations that are analyzed when the `hdlin_vrlg_std` variable is set to `1995` or `2001` return 1'b0. This is compatible with an extension made by Synopsys Verilog products of that era.

Configurations

You can use configurations to specify binding information of module instances down to the cell level in the design. The configurations can be analyzed and elaborated by the `analyze` and `elaborate` commands respectively. For example,

```
dc_shell> analyze -f verilog {submodule.v ...}
dc_shell> analyze -f verilog top_module.v
dc_shell> analyze -f verilog config_file.v
dc_shell> elaborate my_config_of_design
```

By default, the HDL Compiler tool resolves lower module instances by applying a design library search order taken from the dc_shell environment and the analyzed parent module. Alternatively, you can specify design library locations (bindings) for module or interface instances of a specific design in configurations by using the `config` element. All bindings in the design hierarchy are constrained by the configuration rules given in the config_rule subset in the configuration.

The following configuration syntax is supported for synthesis:

```
config config_id;
   design {[lib_id .] design_id};
   {config_rule}
endconfig [: config_id]
```

where

```
config_rule ::= default liblist {lib_id};
              | instance design_id {. inst_id} liblist lib_id;

design_id ::= module_id | interface_id
```

For more information about the syntax, see the *IEEE Std 1800-2012*.

The following limitations apply when you use configurations:

- Only one library is allowed for instances.
- Only one default rule is allowed.
- Library declarations are not allowed.

  To define libraries, use the `define_design_lib` command. Design library names in dc_shell are not case-sensitive.
- The `read_file` command and the `-autoread` option do not support configurations.
- Configuration rules do not affect the bindings of designs that are already elaborated or loaded in memory.

Configuration Examples

The following topics provide examples on how to use configuration rules and designs:

- Default Statement
- Instance Bindings
- Multiple Top-Level Designs

The examples use these low-level modules:

- sub1.v

```
module sub1(
   input i1, i2,
   output o1
);
assign o1 = i1 & i2;
endmodule
```

- sub2.v

```
module sub1(
   input i1, i2,
   output o1
);
assign o1 = i1 | i2;
endmodule
```

- sub3.v

```
module sub1(
   input i1, i2,
   output o1
);
assign o1 = i1 ^ i2;
endmodule
```

Note:

The three low-level files use the same sub1 module name, but they implement different functions. The sub1.v, sbu2.v, and sub3.v files implement AND, OR, and XOR functions respectively.

Default Statement

The following example uses a configuration to direct the tool to choose the implementation of the instances in the top-level module. The configuration file specifies the default statement, but no binding information.

- Top-level top.v file

  ```
  module top(
     input i1, i2, i3, i4,
     output o1, o2, o3
  );
  sub1 U1 (i1, i2, o1);
  sub1 U2 (o1, i3, o2);
  sub1 U3 (o2, i4, o3);
  endmodule
  ```

- Configuration file

  ```
  config cfg1;
  design rtlLib.top;
  default liblist rtlLib;
  endconfig
  ```

- Design Compiler Tcl script

  ```
  define_design_lib lib1 -path ./lib1
  define_design_lib lib2 -path ./lib2
  define_design_lib rtlLib -path ./rtlLib
  analyze -f verilog -library lib1 sub1.v
  analyze -f verilog -library lib2 sub2.v
  analyze -f verilog -library rtlLib sub3.v
  analyze -f verilog -library rtlLib top.v
  analyze -f verilog config.v
  elaborate cfg1
  ```

- Netlist

  The output netlist shows that the sub1 module analyzed in the rtlLib library is chosen for the instantiations in the top module.

  ```
  module sub1 ( i1, i2, o1 );
     input i1, i2;
     output o1;
  GTECH_XOR2 C7 ( .A(i1), .B(i2), .Z(o1) );
  endmodule

  module top ( i1, i2, i3, i4, o1, o2, o3 );
     input i1, i2, i3, i4;
     output o1, o2, o3;
  sub1 U1 ( .i1(i1), .i2(i2), .o1(o1) );
  sub1 U2 ( .i1(o1), .i2(i3), .o1(o2) );
  sub1 U3 ( .i1(o2), .i2(i4), .o1(o3) );
  endmodule
  ```

Instance Bindings

The following example shows how to use instance bindings in configurations. The configuration file specifies the binding of each instance of the sub1 module, but no default statement.

- Top-level top.2 file

```
module top(
   input i1, i2, i3, i4,
   output o1, o2, o3
);
sub1 U1 (i1, i2, o1);
sub1 U2 (o1, i3, o2);
sub1 U3 (o2, i4, o3);
endmodule
```

- Configuration file

```
config cfg1;
design rtlLib.top;
instance top.U1 liblist lib1;
instance top.U2 liblist lib2;
instance top.U3 liblist lib3;
endconfig
```

- Design Compiler Tcl script

```
define_design_lib lib1 -path ./lib1
define_design_lib lib2 -path ./lib2
define_design_lib lib3 -path ./lib3
define_design_lib rtlLib -path ./rtlLib
analyze -f verilog -library lib1 sub1.v
analyze -f verilog -library lib2 sub2.v
analyze -f verilog -library lib3 sub3.v
analyze -f verilog -library rtlLib top.v
analyze -f verilog config.v
elaborate cfg1
```

- Netlist

  The output netlist shows that each instance of the sub1 module uses a different library specified in the configuration file. The U1 instance uses the sub1 module from the lib1 library to implement the AND function. The U2 instance uses the sub1 module from the lib2 library to implement the OR function. The U3 instance uses the sub1 module from the lib3 library to implement the XOR function.

```
module sub1 ( i1, i2, o1 );
   input i1, i2;
   output o1;
GTECH_AND2 C7 ( .A(i1), .B(i2), .Z(o1) );
endmodule

module sub1_1 ( i1, i2, o1 );
   input i1, i2;
   output o1;
GTECH_OR2 C7 ( .A(i1), .B(i2), .Z(o1) );
endmodule

module sub1_2 ( i1, i2, o1 );
   input i1, i2;
   output o1;
GTECH_XOR2 C7 ( .A(i1), .B(i2), .Z(o1) );
endconfig

module top ( i1, i2, i3, i4, o1, o2, o3 );
   input i1, i2, i3, i4;
   output o1, o2, o3;
sub1 U1 ( .i1(i1), .i2(i2), .o1(o1) );
sub1_1 U2 ( .i1(o1), .i2(i3), .o1(o2) );
sub1_2 U3 ( .i1(o2), .i2(i4), .o1(o3) );
endmodule
```

Multiple Top-Level Designs

The following example shows that you can specify multiple top-level designs in configurations. The configuration file instantiates the top1and top2 top-level designs.

- Top-level top1.v file

```
module top1(
   input i1, i2, i3, i4,
   output o1, o2, o3
);
sub1 U1 (i1, i2, o1);
endmodule
```

- Top-level top2.v file

```
module top2(
   input i1, i2, i3, i4,
   output o1, o2, o3
);
sub1 U2 (o1, i3, o2);
endmodule
```

- Configuration file

```
config cfg1;
design lib1.top1 lib2.top2;
instance top1.U1 liblist lib3;
instance top2.U2 liblist lib4;
endconfig
```

- Design Compiler Tcl script

```
define_design_lib lib1 -path ./lib1
define_design_lib lib2 -path ./lib2
define_design_lib lib3 -path ./lib3
define_design_lib lib4 -path ./lib4
define_design_lib lib5 -path ./lib5
analyze -f verilog -library lib4 sub2.v
analyze -f verilog -library lib5 sub3.v
analyze -f verilog -library lib3 sub1.v
analyze -f verilog -library lib1 top1.v
analyze -f verilog -library lib2 top2.v
analyze -f verilog config.v
elaborate cfg1
```

- Netlist of the top1.v file

  The top1netlist shows that the sub1_1 module from the lib3 library is used to implement the AND function, as specified in the configuration file.

```
module sub1_1 ( i1, i2, o1 );
   input i1, i2;
   output o1;
GTECH_AND2 C7 ( .A(i1), .B(i2), .Z(o1) );
endmodule

module top1 ( i1, i2, i3, i4, o1, o2, o3 );
   input i1, i2, i3, i4;
   output o1, o2, o3;
sub1_1 U1 ( .i1(i1), .i2(i2), .o1(o1) );
endmodule
```

- Netlist of the top2.v file

  The top2 netlist shows that the sub1 module from lib4 library is used to implement the OR function, as specified in the configuration file.

```
module sub1 ( i1, i2, o1 );
   input i1, i2;
   output o1;
GTECH_OR2 C7 ( .A(i1), .B(i2), .Z(o1) );
endmodule

module top2 ( i1, i2, i3, i4, o1, o2, o3 );
   input i1, i2, i3, i4;
   output o1, o2, o3;
sub1 U2 ( .i1(o1), .i2(i3), .o1(o2) );
endmodule
```

C

Unsupported Constructs

The Synopsys SystemVerilog tool does not support all the synthesis features described in the *IEEE Std 1800-2012*. Generally, all the restrictions for the HDL Compiler tool apply to the SystemVerilog tool.

The following topic shows the unsupported constructs:

- Unsupported SystemVerilog Constructs

Unsupported SystemVerilog Constructs

The following constructs are not supported:

- The `$onehot`, `$onehot0`, `$countones`, and `$isunknown` system functions are not supported.
- Clocking blocks, defined by a `clocking-endclocking` keyword pair, are not supported in synthesis; they are parsed and ignored.
- Global clocking blocks, which are defined by the `global clocking` and `endclocking` keyword pair, are not supported in synthesis.

  If you use this unsupported keyword pair, the tool issues an error message. To prevent this error, wrap the keyword pair as follows:

  ```
  `ifndef SYNTHESIS
     ...
  `endif
  ```
- The following SystemVerilog keywords are parsed and ignored:

  `assert`, `assume`, `before`, `bind`, `bins`, `binsof`, `clocking`, `constraint`, `cover`, `coverpoint`, `covergroup`, `cross`, `endclocking`, `endgroup`, `endprogram`, `endproperty`, `endsequence`, `extends`, `final`, `first_match`, `intersect`, `ignore_bins`, `illegal_bins`, `local`, `program`, `property`, `protected`, `sequence`, `super`, `this`, `throughout`, and `within`.

  Note:
  The `var` keyword is supported; however, the use of the `var` keyword with a type reference is not supported.
- The following SystemVerilog keywords are not supported:
  `alias`, `chandle`, `context`, `dist` (allowed only in testbenches), `expect`, `export`, `extern`, `new`, `null`, `pure`, `shortreal`, `solve`, `string`, `tagged`, `wait_order`, and `with`.

  If you use an unsupported keyword, the tool terminates with an error message. To prevent this error, wrap the keywords as follows:

  ```
  `ifndef SYNTHESIS
     ...
  `endif
  ```
- The following keywords are not supported: `forkjoin`, `join_any`, `join_none`, `rand`, `randc`, `ref`, `randcase`, `randsequence`.
- Casting on types, `$cast`, is not supported.
- Compiler directives, such as operator label, in interfaces are not supported.

  The following code is not supported:

```
a = b + /* synopsys label my_adder */ c;
```

- Attributes are not fully supported; they are treated as comments.

  You can specify comments in the following two ways:

  - (* comment *)
  - // comment

- Two-state values are not supported; they are treated as four-state values.

  This conversion can cause simulation and synthesis mismatches. According to the SystemVerilog LRM, the `int`, `bit`, `shortint`, `byte`, and `longint` types are two-state data types with legal values of zero and one. Static variables of two-state data types without explicit declaration initialization are initialized to zero instead of x. The Design Compiler tool initializes all static variables to x (including those with explicit declaration initialization) even if they are two-state data types.

- Automatic assignments as expressions are not supported.

  The following code is not supported:

  ```
  mask & ( in << i++ )
  ```

  For example,

  ```
  module m (input a, output b);
  int i;
  assign b = a + i++;
  endmodule
  ```

  When the tool reads the module m, it issues the following error message:

  ```
  Error:  i1.v:3: The construct "assignment expression" is
  not supported.  (VER-721)
  ```

- Generic interfaces are not supported.

  For example, a module with a generic interface port cannot be the top module for elaboration.

- Automatic variables in static tasks and functions are not supported.

- Static variable initialization is ignored in synthesis.

- Variables referred to an interface port type can be accessed like a `ref` port of a module. In computer memory, this is similar to a call by reference, where the last write wins. In hardware, this can only be modeled by semantics of wires. Therefore, the `ref` ports are not achievable in silicon hardware and not synthesizable. They are used only in simulation.

- The `timeunit` statement, `timeprecision` statement, and the delays are ignored in synthesis.
- Ports of the `real` and `time` types in the connection list of modules, interfaces, tasks, and functions are not supported. The `real` and `time` type declarations are not supported.
- Unpacked unions are not supported.

  The following code is not supported:

  ```
  union {
     logic    my_logic;
     logic    [63:0] my_logic;
     logic    [63:0] my_logic;
     longint  my_longint;
  } a;
  ```
- Forward declarations of the `typedef` construct are not supported.

  The following code is not allowed because you must specify with what `typedef` declaration you define mydesign.

  ```
  typedef mydesign;
  mydesign p;
  typedef int;
  ```
- Nested module declarations and nested interface declarations are not supported.
- Using the array slice operators (+:, &, and -:) with arrays of interfaces is not supported.

  If the array slice operators are used, the tool issues error messages similar to the following:

  ```
  Error: ./example.v:16: The construct 'Interface Array Slice Indexing'
  is not supported. (VER-721)
  ```
- The following types of assignment patterns are not supported:
  - Assignment patterns on the left side of assignments
  - Array pattern keys in assignment patterns
  - Assignment patterns on parameter specification overrides
  - Assignment patterns on the ports of module or interface instantiations

    However, the tool supports the assignment patterns of constant 0 settings on input ports by the `default` keyword. For example,

    ```
    sub1 U1 (.i1('{default:'0}), .i2(i2), .o1(o1));
    ```

See Also

- Conversion Between Two-State and Four-State Variables

- The *IEEE Std 1800-2012*

Glossary

anonymous type
A predefined or underlying type with no name, such as universal integers.

ASIC
Application-specific integrated circuit.

behavioral view
The set of Verilog statements that describe the behavior of a design by using sequential statements. These statements are similar in expressive capability to those found in many other programming languages. See also the *data flow view*, *sequential statement*, and *structural view* definitions.

bit-width
The width of a variable, signal, or expression in bits. For example, the bit-width of the constant 5 is 3 bits.

character literal
Any value of type CHARACTER, in single quotation marks.

computable
Any expression whose (constant) value HDL Compiler can determine during translation.

constraints
The designer’s specification of design performance goals. Design Compiler uses constraints to direct the optimization of a design to meet area and timing goals.

convert
To change one type to another. Only integer types and subtypes are convertible, along with same-size arrays of convertible element types.

data flow view

The set of Verilog statements that describe the behavior of a design by using concurrent statements. These descriptions are usually at the level of Boolean equations combined with other operators and function calls. See also the *behavioral view* and *structural view*.

Design Compiler

The Synopsys tool that synthesizes and optimizes ASIC designs from multiple input sources and formats.

design constraints

See *constraints*.

flip-flop

An edge-sensitive memory device.

HDL

Hardware Description Language.

HDL Compiler

The Synopsys Verilog synthesis product.

identifier

A sequence of letters, underscores, and numbers. An identifier cannot be a Verilog reserved word, such as *type* or *loop*. An identifier must begin with a letter or an underscore.

latch

A level-sensitive memory device.

netlist

A network of connected components that together define a design.

optimization

The modification of a design in an attempt to improve some performance aspect. Design Compiler optimizes designs and tries to meet specified design constraints for area and speed.

port

A signal declared in the interface list of an entity.

reduction operator

An operator that takes an array of bits and produces a single-bit result, namely the result of the operator applied to each successive pair of array elements.

register

A memory device containing one or more flip-flops or latches used to hold a value.

resource sharing

The assignment of a similar Verilog operation (for example, +) to a common netlist cell. Netlist cells are the resources—they are equivalent to built hardware.

RTL

Register transfer level, a set of structural and data flow statements.

sequential statement

A set of Verilog statements that execute in sequence.

signal

An electrical quantity that can be used to transmit information. A signal is declared with a type and receives its value from one or more drivers. Signals are created in Verilog through either wire or reg declarations.

signed value

A value that can be positive, zero, or negative.

structural view

The set of Verilog statements used to instantiate primitive and hierarchical components in a design. A Verilog design at the structural level is also called a netlist. See also *behavioral view* and *data flow view*.

subtype

A type declared as a constrained version of another type.

synthesis

The creation of optimized circuits from a high-level description. When Verilog is used, synthesis is a two-step process: translation from Verilog to gates by HDL Compiler and optimization of those gates for a specific ASIC library with Design Compiler.

translation

The mapping of high-level language constructs onto a lower-level form. HDL Compiler translates RTL Verilog descriptions to gates.

type

In Verilog, the mechanism by which objects are restricted in the values they are assigned and the operations that can be applied to them.

unsigned

A value that can be only positive or zero.

Index

Symbols

-- 1-4
. 1-4
-: (variable part-select operator) B-35
‘define B-17
‘else B-19
‘elsif B-19
‘endif B-19
‘ifdef VERILOG_1995 B-29
‘ifdef VERILOG_2000 B-29
‘ifdef, ‘else, ‘endif, ‘ifndef, and ‘elsif B-19
‘ifndef B-19
‘include B-18
‘undefineall 1-55
** (power operator) B-40
&= 1-4
^= 1-4
+: (variable part-select operator) B-35
++ 1-4
+= 1-4
<<< (arithmetic shift operator) B-40
<<<= 1-4
<<= 1-4
-= 1-4
>>= 1-4
>>> (arithmetic shift operator) B-40
>>>= 1-4
|= 1-4
$cast C-2
$display 1-55
$left 9-22
$right 9-22
$unit 2-1

Numerics

2-state values C-3
2-state variables 11-13

A

adders 1-7
  carry bit overflow 4-5
  for loop to synthesize matrix adders 9-22
  using typedef, example 9-7
always block
  edge expressions 9-46
always blocks
  always_comb 11-11
always construct 1-58, 8-1, 8-4, B-14, B-15
arithmetic shift operators B-40
arrays

array slicing 9-19
multidimensional arrays 9-19
multidimensional arrays unsupported constructs 9-23
packed 1-3, 9-31
querying 1-3
querying functions 9-22
real/time multidimensional arrays C-4
renaming ports in multidimensional arrays 9-38
unpacked 1-3

Arrays of nets B-29

assertions 11-16

assignments
always construct 1-58, 8-1, 8-4, B-14, B-15
blocking B-15, B-16
continuous B-28
initial B-4, B-5
nonblocking 1-58, B-15, B-16

attributes C-3

autoassignment as expressions C-3

B

binary numbers B-2

bit accesses 4-28

bit and memory accesses 4-28

bit-blasting B-23

bit-level support 9-15

bit-truncation
explicit 4-29

bit-width
prefix for numbers B-2
specifying in numbers B-2

blocking and nonblocking B-15

blocking assignments B-15, B-16

bus fabric design A-4

bus_multiple_separator_style 10-13

bus_naming_style variable 1-60

bus_range_separator_style 10-13

Busing B-23

C

case statements
casex,casez B-12
hdlin_mux_size_limit 4-21
SELECT_OP Inference 4-17
used in multiplexing logic 4-16

casex B-12

casez B-12

casting on types C-2

casting operators B-35

coding for QoR 1-7

coding guidelines 9-40

coding guidelines for DC Ultra datapath optimization
bit-truncation
implicit 4-29

Comma-separated sensitivity lists B-29

compiler directives B-17

compiler directives in interfaces C-2

conditional assignments
if-else A-22

conditional compilation directives 11-2

conditional inclusion of code
‘ifdef, ‘else, ‘endif, ‘ifndef, and ‘elsif Directives B-19

constructs
troubleshooting and problem solving 11-1

continuous assignments B-28

controlling signs B-35

conventions for documentation 2-3

customer support 2-4

D

Data-Path Duplication A-18

dc_script_end directive 10-4, 10-5

decimal numbers B-2

declaration requirements
  tri data type B-17
Design Compiler 1-16, 1-60, 4-5, 6-11, 10-15, 10-16
directives
  ‘define 2-2, B-17
  ‘else B-19
  ‘endif B-19
  ‘include 2-2, B-18
  ‘undef B-23
  ‘undefineall 1-55
  dc_script_begin 10-4
  dc_script_end 10-4
  full_case 4-14, 10-7
  in interfaces C-2
  infer_multibit 10-10
  infer_mux 4-20, 10-14
  infer_onehot_mux 4-18, 10-15
  multibit inference 10-9
  one_cold 5-17, 10-15
  one_hot 10-15
  parallel_case 4-15, 10-16
  parallel_case used with full_case 10-7
  rp_align 1-24, B-22
  rp_array_dir 1-23, B-21
  rp_endgroup 1-20, B-20
  rp_endignore 1-25, B-22
  rp_fill 1-22, B-21
  rp_group 1-20, B-20
  rp_ignore 1-25, B-22
  rp_orient 1-24, B-22
  rp_place 1-21, B-20
  see also hdlin_ for variables
  simulation B-29
  support in %unit 2-2
disable B-14
don’t care 4-27
don’t cares
  in case statements 4-27

E

ELAB-366 8-6, 8-7
elaboration reports 1-42
embedding constraints and attributes
  dc_script_begin 10-4
  dc_script_end 10-4
enum directive 10-5
enumerated type inference report 6-15
enumerated types 6-15
error messages
  VER-294 2-4
  VER-523 2-9
  VER-721 C-3
errors 1-55, 8-2, 8-7, 10-5, B-5, B-23, B-24
  ELAB-366 8-6
  ELAB-900 B-24
Explicit bit-truncation 4-29
expression tree
  optimized for delay 4-6

F

file formats, automatic detection of 1-15
finite state machine 6-11
  fsm_auto_inferring 6-11
  inference report 6-14
flip 5-4
flip-flop
  clocked_on_also attribute A-32
  D-flip-flop
    D flip-flop with a synchronous load and an asynchronous load 5-23
    D flip-flop with an asynchronous reset 5-21
    D flip-flop with an asynchronous set 5-21
    D flip-flop with synchronous reset 5-23
    D flip-flop with synchronous set 5-22
    rising-edge-triggered D flip-flop 5-20
  master-slave latches A-32
  SEQGENs 5-2
flows

- interacting with low-power flows 9-41
- interacting with other flows 9-41
- interacting with Synthesis flows 9-41
- interacting with verification flows 9-44

FSM inference variables 6-11

fsm_auto_inferring 6-11

full_case 10-7

functional description
- function declarations in 1-58

functions 1-60, 6-10, 8-2, 9-47, 10-5, B-4, B-18, B-30, B-32, GL-2
- automatic 2-9
- default 2-9
- querying 9-22
- static 2-9
- VER-523 error 2-9

G

gate-level constructs 1-58

generic interfaces C-3

H

hdlin_infer_mux 4-21

hdlin_keep_signal_name variable 9-41

hdlin_mux_oversize_ratio 4-21

hdlin_mux_size_limit 4-21

hdlin_mux_size_min 4-21

hdlin_preserve_sequential variable 5-8

hdlin_prohibit_nontri_multiple_drivers 8-6

hdlin_reporting_level directive 6-11

hdlin_reporting_level variable 1-42, 5-4, 8-2

hdlin_sverilog_std 1-15

hdlin_vrlg_std 1-16

hexadecimal numbers B-2

hierarchical
- boundaries 1-59
- constructs 1-58

hierarchy
- objects supported in $unit 2-2
- reading designs that use the $unit 2-3
- top-level name space 2-2

I

If 5-4

if statements
- hdlin_infer_mux 4-21
- in case statements 4-27

ifdef VERILOG_1995 B-29

ifdef VERILOG_2001 B-29

if-else A-22

ignored functions B-30

implicit bit-truncation 4-29

include B-18

infer_multibit 10-10

infer_mux 4-20

infer_mux directive 10-14

infer_onehot_mux directive 4-18, 10-15

inference report
- description 8-2

inference reports 5-4
- enumerated types 6-15
- finite state machine 6-14
- multibit components 10-12

inferring flip-flops 5-19

initial assignment B-4, B-5

inout
- connecting to gate B-16
- connecting to module B-16

instantiations 1-58, 1-60, 8-6, B-16

interface
- basic 7-3
- netlist naming conventions 7-29
- with directives C-2
- with functions and tasks 7-13
- with parameters 7-23

interface array examples 7-24

interface arrays 7-24

K

keywords C-2

L

latches
- avoiding unintended latches 4-12
- clocked_on_also A-32
- D latch 5-16
- D latch with an active-low asynchronous set and reset 5-17
- generic sequential cells (SEQGENs) 5-2
- master-slave latches A-32
- resulting from conditionally assigned variables 5-15

late-arriving signals
- datapath duplication solution A-18
- moving late-arriving signal close to output solution A-18

lexical conventions B-2

loops
- do...while 11-15
- enhanced for loop to synthesize matrix adders 9-22

M

macro substitution B-17

macromodule B-16

Macros 11-2

macros B-20
- global reset
  - ‘undefineall 1-55
- local reset
  - ‘undefineall B-23
- macro definition on the command line B-18
- predefined macro, SVA_STD_INTERFACE 1-37
- predefined SYSTEMVERILOG macro 9-35
- specifying macros
  - ‘define B-17
- specifying macros that take arguments B-17
- SYNTHESIS 1-54
- VERILOG_1995 1-54
- VERILOG_2001 1-54

memory accesses 4-28

mismatch 8-2
- full_case usage 10-7
- parallel_case usage 10-16
- simulator/synthesis 8-2
- three-states 8-4
- z value 8-2
- z value comparison 8-2

module
- connecting to inout B-16

multibit components
- benefits 8-1
- bus_multiple_separator_style 10-13
- bus_range_separator_style 10-13
- described 8-1
- multibit inference report 10-12
- report_multibit command 10-12

multibit inference directives 10-9

multidimensional arrays B-30
- array slicing 9-19
- in module ports 9-38
- real/time multidimensional arrays C-4
- renaming ports 9-38
- unsupported constructs 9-23

multiplexer
- cell size 4-24

multiplexing logic
- case statements that contain don’t care values 4-27
- case statements that have a missing case statement branch 4-27
- Design Compiler implementation 4-17
- for bit or memory accesses 4-28
- hdlin_infer_mux variable 4-20
- hdlin_mux_oversize_ ratio 4-21
- hdlin_mux_size_limit 4-21
- hdlin_mux_size_min 4-21
- if-else-begin-if constructs A-22

- implement conditional operations implied by if and case statements 4-17
- infer MUX_OP cells 4-19
- infer_mux 4-20
- MUX_OP cells 4-16
- MUX_OP Inference Limitations 4-27
- SELECT_OP cells 4-16
- sequential if statements A-21
- warning message 4-27
- with if and case statements 4-16

MUX_OP inference 4-19

N

nonblocking assignments 1-58, B-15, B-16

number
- binary B-2
- decimal B-2
- formats B-2
- hexadecimal B-2
- octal B-2
- specifying bit-width B-2

O

octal numbers B-2

one 10-15

one_cold directive 5-17, 10-15

one_hot directive 10-15

one-hot multiplexer 4-18

operators
- casting B-35
- part-select operator 9-23
- power B-40
- shift B-40
- variable part-select B-35

optimization
- fsm_auto_inferring 6-11

P

packages 3-1

parallel_case 10-7, 10-16

parameterized functions 9-11

parameterized tasks 9-11

parameters 1-7, 1-17, 1-50, 1-55, 10-5, 10-19, B-4, B-28
- in an interface 7-23

Part-Select Addressing B-35

part-select operations
- multidimensional arrays use in 9-19

ports
- implicit instantiation of, using .* 9-29
- implicit instantiation of, using .name 9-29
- inout port requirements B-16
- multidimensional arrays use in 9-19
- real/time ports C-4
- renaming in multidimensional arrays 9-38

Power B-40

power operator (**) B-40

processes
- asynchronous 9-46
- synchronous 9-46

R

radices B-2

read_file -format command 1-15

reading designs
- analyze -f verilog { files } elaborate 1-17
- automatic structure detector 1-16
- netlists 1-16
- parameterized designs 1-16
- read -f verilog -netlist { files } (dcsh) 1-17
- read_file -f verilog -netlist { files } (tcl) 1-17
- read_file -f verilog -rtl { files } 1-17
- read_verilog 1-17
- read_verilog -netlist { files } (tcl) 1-17
- read_verilog -rtl { files } (tcl) 1-17
- reading SystemVerilog Files 1-7

RTL 1-16
that contain assertions 11-16
with $unit 2-3
real/time multidimensional arrays C-4
real/time ports on modules/interfaces/tasks/ functions C-4
register inference examples 5-15
relative placement 1-18
compiler directives B-17
creating groups 1-20, B-20
examples 1-26
figure 1-27
ignoring 1-25, B-22
placing cells automatically 1-22, B-21
specifying cell alignment 1-24, B-22
specifying cell orientation 1-24, B-22
specifying placement 1-23, B-21
specifying subgroups, keepouts, and instances 1-21, B-20
remove_design 1-54
removing designs 1-54
report_multibit 10-12
report_multibit command 10-12
resets
global reset
‘undefineall 1-55
local reset
‘undef B-23

S

SELECT_OP 4-17
sensitivity list 9-46
Sequential if statements A-21
shift operators B-40
Sign Conversion Warnings 4-6
sign rules B-34
signal names, keeping 9-41
signed arithmetic extensions B-32
Signed Constants B-32
Signed I/O Ports B-32
signed keyword B-33
Signed Registers B-32
Signed Types B-33
signs
casting operator B-35
controlling signs B-35
sign conversion warnings B-35
simulation
directives B-29
simulator/synthesis mismatch
full_case usage 10-7
parallel_case usage 10-16
slow reads 1-16
SolvNet
accessing 2-4
documentation 2-2
standard macros
macro definition on the command line B-18
see also macros
SYNTHESIS 1-54
VERILOG_1995 1-54
VERILOG_2001 1-54
state machines
automatically defined state variables 6-6
default base type 6-3
define integral named constants to variables 6-1
explicit base type 6-3
using enumerations 6-1
structural description
elements of 1-58
synchronous
processes 9-46
SYNTHESIS macro B-28
synthesis/simulation mismatches 11-6
synthetic comments.*See* directives
system functions, Verilog B-30
SystemVerilog language version, controlling 1-15

T

tasks 1-55, 1-58, 1-60, 6-10, 9-46, 9-47, B-4, B-14
  in an interface 7-13
  inside an always_comb block 11-11
template
  directive 10-19
three-state buffer
  hdlin_prohibit_nontri_multiple_drivers 8-6
  tri data type B-17
tri Data Type declaration requirement B-17
troubleshooting and problem-solving guidelines 11-1
two-phase design A-34
typedef 11-15, C-4

U

unions
  usage example 9-18
unloaded registers, keeping 5-8
unsupported constructs
  for multidimensional arrays 9-23
  real/time multidimensional arrays C-4
  real/time ports on modules/interfaces/tasks/functions C-4
  troubleshooting and problem solving 11-1

V

variable
  conditionally assigned 5-15
  reading 5-15
variable part-select operators B-35
variables, (see hdlin_)
VER-318 4-7
Verilog
  keywords B-3
  system function B-30
Verilog 2001 features B-28
  ‘ifndef, ‘elsif, ‘undef B-29
  ANSI-C-style port declaration B-28
  arithmetic shift operators B-29
  casting operators B-28
  comma-separated sensitivity lists B-29
  disabling features 1-54
  hdlin_vrlg_std = 1995 1-54
  hdlin_vrlg_std = 2001 1-54
  implicit event expression list B-28
  multidimensional arrays B-29
  parameter passing by name B-28
  power operator (**) B-29
  signed/unsigned nets and registers B-28
  signed/unsigned parameters B-28
  signed/unsigned sized and based constants B-28
  sized parameters B-29
  SYNTHESIS macro B-28
Verilog language version, controlling 1-16

W

warnings
  encodings 10-6
  hdlin_unsigned_integers B-34
  sign conversion 4-7
  VER-318 4-7
while loop B-24, B-25, B-26, B-27