HDL Compiler™ for SystemVerilog User Guide

Version U-2022.12-SP3, April 2023

SYNOPSYS®

Feedback

Contents

About This Manual

The HDL Compiler tool translates a SystemVerilog hardware language description into a GTECH netlist that is used by the Synopsys synthesis tools to create an optimized netlist.

Audience

The *HDL Compiler for SystemVerilog User Guide* is written for logic designers and electronic engineers who are familiar with the HDL Compiler tool. Knowledge of the Verilog language is required, and knowledge of a high-level programming language is helpful. This document is not a standalone document but must be used in conjunction with the *IEEE Std 1800-2017*.

This preface includes the following sections:

- New in This Release
- Related Products, Publications, and Trademarks
- Conventions
- Customer Support
- Statement on Inclusivity and Diversity

New in This Release

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

Related Products, Publications, and Trademarks

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

https://solvnetplus.synopsys.com

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

- DC Explorer
- Design Compiler®
- Fusion Compiler™

- DesignWare® components
- Library Compiler™

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`**
`Purple`	• Within an example, indicates information of special interest. • Within a command-syntax section, indicates a default, such as `include_enclosing =` **`true`** `| false`
[ ]	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`
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Bold	Indicates a graphical user interface (GUI) element that has an action associated with it.
Edit > Copy	Indicates a path to a menu command, such as opening the **Edit** menu and choosing **Copy**.
Ctrl+C	Indicates a keyboard combination, such as holding down the Ctrl key and pressing C.

Customer Support

Customer support is available through SolvNetPlus.

Accessing SolvNetPlus

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

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

https://solvnetplus.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 SolvNetPlus site, click REGISTRATION HELP in the top-right menu bar.

Contacting Customer Support

To contact Customer Support, go to https://solvnetplus.synopsys.com.

Statement on Inclusivity and Diversity

Synopsys is committed to creating an inclusive environment where every employee, customer, and partner feels welcomed. We are reviewing and removing exclusionary language from our products and supporting customer-facing collateral. Our effort also includes internal initiatives to remove biased language from our engineering and working environment, including terms that are embedded in our software and IPs. At the same time, we are working to ensure that our web content and software applications are usable to people of varying abilities. You may still find examples of non-inclusive language in our software or documentation as our IPs implement industry-standard specifications that are currently under review to remove exclusionary language.

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 Designs Using the VCS Command-Line Options
- 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 Settings
- Parameterized Designs
- Defining Macros
- Using $display During RTL Elaboration
- System Functions and Tasks
- Elaboration System Tasks
- 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 1800-2017*. To download a copy, go to the following address:

https://en.wikipedia.org/wiki/IEEE_Xplore

Table 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`)	Functions and Tasks Data Type Declarations
	Structures (packed and unpacked)	Structures
	Enumerations	
	Enumeration methods	*IEEE Std 1800-2017* For restrictions, see .
	Range of `enum` labels	
	Unions (packed)	Unions
	Casting	Casting
	Void data types	*IEEE Std 1800-2017*
	Generic wire type	Generic Wire Type
Arrays	Packed arrays	Multidimensional Arrays Casting
	Packed array of enumerations	*IEEE Std 1800-2017*
	Unpacked arrays	Multidimensional Arrays
	Array querying (`$size`, `$left`, `$right`, `$low`, `$high`, `$increment`, `$dimensions`, `$unpacked_dimensions`)	Multidimensional Arrays

Table 1 Supported Constructs (Continued)

Category	Feature	Usage reference
Data declaration	Scoping	*IEEE Std 1800-2017*
	Constraints	*IEEE Std 1800-2017*
	Variables	Used in many examples
Operators	"." operator	Structures
	+=, -=, ++, --, &=, |=, ^=	Synthetic Operators
	Wildcard equality and inequality operators (==? and !=?)	*IEEE Std 1800-2017*
	Left-to-right streaming operator {>>{}}[1]	*IEEE Std 1800-2017*
	<<=, >>=, <<<=, >>>=	Synthetic Operators
	`inside` and `case-inside`	*IEEE Std 1800-2017*
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
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

1. *The destination must hold at least as many bits as the source (see the IEEE Std 1800-2017, section 11.4.14.3). Related tool messages are VER-553 and VER-554.*

Table 1 Supported Constructs (Continued)

Category	Feature	Usage reference
	Automatic variable initialization	Variables
	Argument binding using the `.name` syntax	*IEEE Std 1800-2017*
Assertions	Assertions	Unsupported Constructs Reading Designs With Assertion Checker Libraries
Hierarchy	All types as legal module ports	Multidimensional Arrays Functions and Tasks
	`$unit`	About the Global Name Space
	Implicit `.name` and .* port connections	Implicit Port Connections
Interfaces	Interface as a signal container and module port replacement	Example: Interface With Wires
	Interface modports	Example: Interface With Modports
	Interface ports	Ports in Interfaces Example
	Parameterized interfaces	Parameterized Interfaces Example
	Interface functions and tasks	Example: Interface With Functions and Tasks
	Generic interface	*IEEE Std 1800-2017*
	Array of interfaces	Arrays of Interfaces
Parameters	Default `logic` type	*IEEE Std 1800-2017*
	Data type parameter	Parameterized Data Types
System tasks and functions	Size system function (`$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 1800-2017*
	`do...while`	Synthesizable do...while Loops

Table 1 Supported Constructs (Continued)

Category	Feature	Usage reference
Packages	• Scope extraction using :: • Wildcard imports inside modules • Wildcard imports inside `$unit` • Specific imports	Packages

Coding for QoR

The HDL Compiler tool optimizes a design to provide the best quality of results (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 either of these methods to read SystemVerilog designs into the HDL Compiler 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  parameterized_interface.sv
current_design top
link
compile
write -format verilog -hierarchy \
   -output gates.parameterized_interface_rd.v
```

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

  For example,

  ```
  analyze -format sverilog parameterized_interface.sv
  elaborate top
  compile
  write -format verilog -hierarchy \
     -output gates.parameterized_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`.

Note:

The tool automatically supports designs that are encrypted according to the IEEE 1735 standard.

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

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 read SystemVerilog RTL into the tool. The default version 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`.

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 of 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> analyze {.} -autoread -recursive -top E1
```

The following example specifies the file extensions for SystemVerilog files other than the default (.sv and .sverilog) and sets file source lists that exclude some directories.

```
dc_shell> set_app_var hdlin_autoread_sverilog_extensions {.sve .SVE}
dc_shell> set my_sources {mod1/src mod2/src}
dc_shell> set my_excludes {mod1/src/incl_dir/ mod2/src/incl_dir/}
dc_shell> analyze $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_autoread_vhdl_extensions`	Specifies the file extension to analyze files as VHDL files.	.vhd .vhdl

Variable	Description	Default
`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 an 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 is eventually replaced by synthesizable RTL.

To enable this feature, the following two methods are available:

- Elaboration Command Based Interface-Only Method (Recommended)
- Analyze Command Based Interface-Only Method

Elaboration Command Based Interface-Only Method (Recommended)

During elaboration, the HDL Compiler tool creates a black box for the module body without netlisting the subblocks and other logic blocks inside the interface-only blocks. To enable this feature, set the following variables:

- `hdlin_elaborate_black_box`: Set the variable to ignore the module body listed during elaboration.
- `hdlin_elaborate_black_box_all_except`: Set the variable to ignore the body of all the modules except the modules that are listed during elaboration.

Note:

Use these options only if there are no syntax errors and non-synthesizable designs constructs in the RTL.

For example,

```
dc_shell> set_app_var \
          -name hdlin_elaborate_black_box \
          -value {my_module1 my_module2}
          dc_shell> analyze -format sverilog top.sv

dc_shell> set_app_var \
          -name hdlin_elaborate_black_box_all_except \
          -value {my_mod1 my_mod2}
          dc_shell> analyze -format sverilog top.sv
```

For more information about a specific variable, see the `hdlin_elaborate_black_box` and `hdlin_elaborate_black_box_all_except` man pages.

Analyze Command Based Interface-Only Method

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

```
dc_shell> set_app_var hdlin_sv_interface_only_modules \
          {my_module1 my_module2}

dc_shell> analyze -format 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 1800-2017* (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 the 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)

Caution:

This topic documents the legacy module black-boxing functionality. You should use the improved functionality described in Ignoring Modules During the Read Process on page 25.

For information on the difference between these methods, see SolvNetPlus article 000020764, “What is the Difference Between the hdlin_sv_interface_only_modules and hdlin_sv_blackbox_modules Variables?”

Note:
You must be logged in to SolvNetPlus for the link to connect directly to the article. If you are prompted to log in to SolvNetPlus upon clicking the link to the article, log in, then click the link again to reach the article.

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)
```

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"
```

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.

Parameterized Interface Ports

In the HDL Compiler tool, the interface parameter values for interface ports can be specified using the `elaborate` command. This removes the need for creating a separate wrapper module. The parameters of interface ports can be specified using the following syntax:

```
elaborate <design> -parameters "Portname1.param1=>param1.value1
 [Port_m.param_p=>param_p.value_s]"
```

For example, you can specify the value of `WIDTH` for the two interface ports `P1` and `P2` to be 8 and 16 using the following command:

```
dc_shell> elaborate block \
   -parameters "P1.WIDTH=>8,P2.WIDTH=>16"
```

You can also specify module parameters and interface parameters together in a single `elaborate` command. For example, you can specify the width of the two interface ports `P1` and `P2` to be 8 and 16 and also set the module parameter length to 5 using the following command:

```
dc_shell> elaborate block \
   -parameters "P1.WIDTH=>8,P2.WIDTH=>16,Length=>5"
```

Specifying name-based parameters in any order generates the same netlist.

Any positional module parameters must be specified before any name-based parameters and position-based specification is applicable only for module parameters and not for interface parameters. For example, you can use the following command to specify the values of 2, 30, and 4 for the first 3 module parameters, set the module parameter length to 5, and set the `WIDTH` value of interface ports `P1` and `P2` to be 8 and 16.

```
dc_shell> elaborate block \
   -parameters "2, 30, 4, P1.WIDTH=>8,P2.WIDTH=>16,Length=>5"
```

Arrays of interface ports must have the same parameter values. For interface arrays, specification of a parameter on the array name sets the value for the entire array. For example, set the `WIDTH` parameter for an array A[] using the following command:

```
dc_shell> elaborate block -parameters "A.WIDTH=10"
```

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 settings to avoid port name mismatch linking errors:

```
# default is false
set_app_var link_portname_allow_period_to_match_underscore true
```

```
# default is false
# (use only for matching modport arrays)
set_app_var link_portname_allow_square_bracket_to_match_underscore true
```

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_app_var 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

  ```
  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

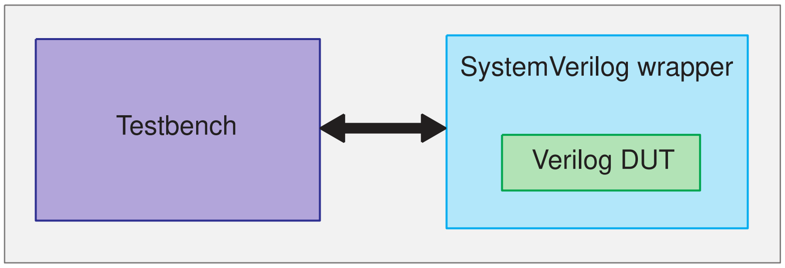
```
// child.sv
module child(reset_n, clk);
   input reset_n, clk;
   reg [7:0] count;

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
```

For more information, see 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 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 2. 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 2   Basic 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

Table 2 Basic Reporting Level Variable Settings (Continued)

Information displayed (information keyword)	basic (default)	none	comprehensive	verbose
Inferred sequential elements `inferred_modules`)	true	false	true	true
MUX_OPs (`mux_op`)	true	false	true	true
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 reporting level as follows is equivalent to setting a level of `comprehensive`.

```
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`.

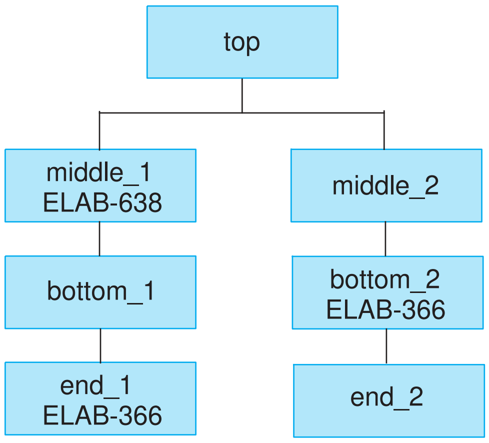
The tool runs in the RTL debug mode.

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 2, to report all elaboration errors in the hierarchy.

Figure 2 Hierarchical Design



Example 1 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 2 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 3   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 2, 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 3, 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 HDL 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 the following table for the type of information to be reported. The default is 0.

Setting	Information reported
0	No preprocessing information.

Setting	Information reported
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 -format 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 -format 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 Settings

To get a list of variables that affect RTL reading, use the following command:

```
dc_shell> report_app_var 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> report_app_var template*
dc_shell> report_app_var 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 1800-2017*, 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 4 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 5 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 6 overrides both parameters and instantiates U1, a 4-bit wide inverter block.

Example 6 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 7 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 7 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 8 does not override either parameter. Parameter SIZE is undefined (no default or override value) causing a compile error.

Example 8 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 sverilog 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 9 uses a parameter to determine the register bit-width; the default width is declared as 8.

Example 9 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_sverilog` 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.

Predefined Macros

You can also use the following predefined macros:

- `SYNTHESIS`—Used to specify simulation-only code, as shown in Example 10.

Example 10 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.

Persistent Macros

To save the SystemVerilog text macros (`` `-define``) definitions persistently across different `analyze` commands, set the `hdlin_enable_persistent_macros` variable to `true`. The default is `false`.

To change the default macro file name, use the `hdlin_persistent_macros_filename` variable. The default macro file name is `syn_auto_generated_macro_file.sv`.

Note:

The generated persistent macro file is encrypted with the synenc encryption.

As shown in the following example, the tool saves the text macros defined in different `analyze` commands:

```
dc_shell> set_app_var hdlin_enable_persistent_macros true
dc_shell> set_app_var hdlin_persistent_macros_filename my_macros.tmp
dc_shell> analyze -format sverilog package.sv
// The my_macros.tmp text definitions are saved in the first analyze
 command package.sv file.

// The following analyze command gets translated to include
 the my_macros.tmp automatically as follows:
dc_shell> analyze -format sverilog "my_macros.tmp file2.sv"
```

For more information about a specific variable, see the `hdlin_enable_persistent_macros` and `hdlin_persistent_macros_filename` man pages.

Using $display During RTL Elaboration

The $display system task is usually used to report simulation progress. In synthesis, the HDL Compiler tool 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 the tool 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 11.

Example 11 $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
```

The tool produces output such as the following during elaboration:

```
dc_shell> elaborate TOP
Presto 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
Presto compilation completed successfully.
Information: Building the design 'F' instantiated from design 'TOP' with
        the parameters "32". (HDL-193)
$display output: Instantiating F, SIZE=32
Presto compilation completed successfully.
```

System Functions and Tasks

System functions and tasks are special functions and tasks with names that start with a dollar sign ($). The *IEEE Std 1800-2012* defines a variety of built-in system functions and tasks. Users can also define their own system functions and tasks.

The HDL Compiler tool supports the following built-in system functions and tasks.

Table 3 Supported SystemVerilog System Functions and Tasks

Context or usage	Supported system functions and tasks
General system tasks evaluated during elaboration See Using $display During RTL Elaboration on page 49 for more information.	`$display()`

Elaboration system tasks
See Elaboration System Tasks on page 51 for more information.
	`$info()`
	`$warning()`
	`$error()`
	`$fatal()`

Functions supported in constant context	`$bits()`
	`$clog2()`[2]
	`$dimensions()`
	`$high()`
	`$increment()`
	`$left()`
	`$low()`
	`$right()`
	`$size()`
	`$unpacked_dimensions()`

Functions synthesizable to hardware	`$countones()`

Other system functions and tasks are ignored or result in errors.

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

2. *clog2() is supported in constant expressions, but not in always blocks or continuous assignments.*

SystemVerilog syntax	Tool output format	Message number
`$warning(user_message);`	`Warning: file name: line number:` `$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 1800-2017* (Sections 20.10 and 20.11).

Example 12   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 13   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

SystemVerilog code input to the HDL Compiler tool can contain both structural and functional (RTL) descriptions. A SystemVerilog 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 SystemVerilog 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 14.

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 14 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 HDL 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_blockgroup_cells -hdl_block` command after reading in a Verilog design. The 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_blocksget_groups -hdl_of_module` 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 the tool 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 the HDL Compiler tool 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 the HDL Compiler tool reads a design, it creates a generic netlist consisting of generic components, such as SEQGENs.

For example, after the tool reads the my_fsm design in Example 15, it creates the generic netlist shown in Example 16.

Example 15   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
```

Example 16   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 17 shows the `report_cell` output for my_fsm design.

Example 17   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 HDL Compiler session. Warnings are errors that do not stop the read from completing, but the results might not be as expected.

You can use the `suppress_message` command to suppress particular warning messages when reading SystemVerilog source files. By default, the tool does not suppress any warnings. This command has no effect on error messages that stop the reading process.

To use it, specify the list of warning message ID codes that you want to suppress. For example, to suppress the following message:

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

then issue the following command:

```
dc_shell> suppress_message {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

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 18 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 18 $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 19, 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 19

```
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 20, the tool issues a VER-518 error message.

Example 20

```
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)
*** Presto 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 21, 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 21

```
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 reads the designwithout errors.

However, if the global file is not read before each design file, as shown in Example 22, 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 22

```
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)
*** Presto compilation terminated with 1 error. ***
0
```

- The `` `include `` construct

  You can use this method to fix the problem described in Example 22. Example 23 shows how to include the global file.

Example 23

```
'include "global.sv"

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

Example 24 shows the script for the `` `include `` method. The tool reads all the files and generates the netlist without errors.

Example 24

```
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 25, 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 25, 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 25

```
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

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 26 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 26, 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 26 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 27 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 27 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 28, 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 28   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 29 shows the code compaction benefit of using wildcard imports.

Example 29

```
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 30 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 31.

Example 30

```
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 31

```
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

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 3.

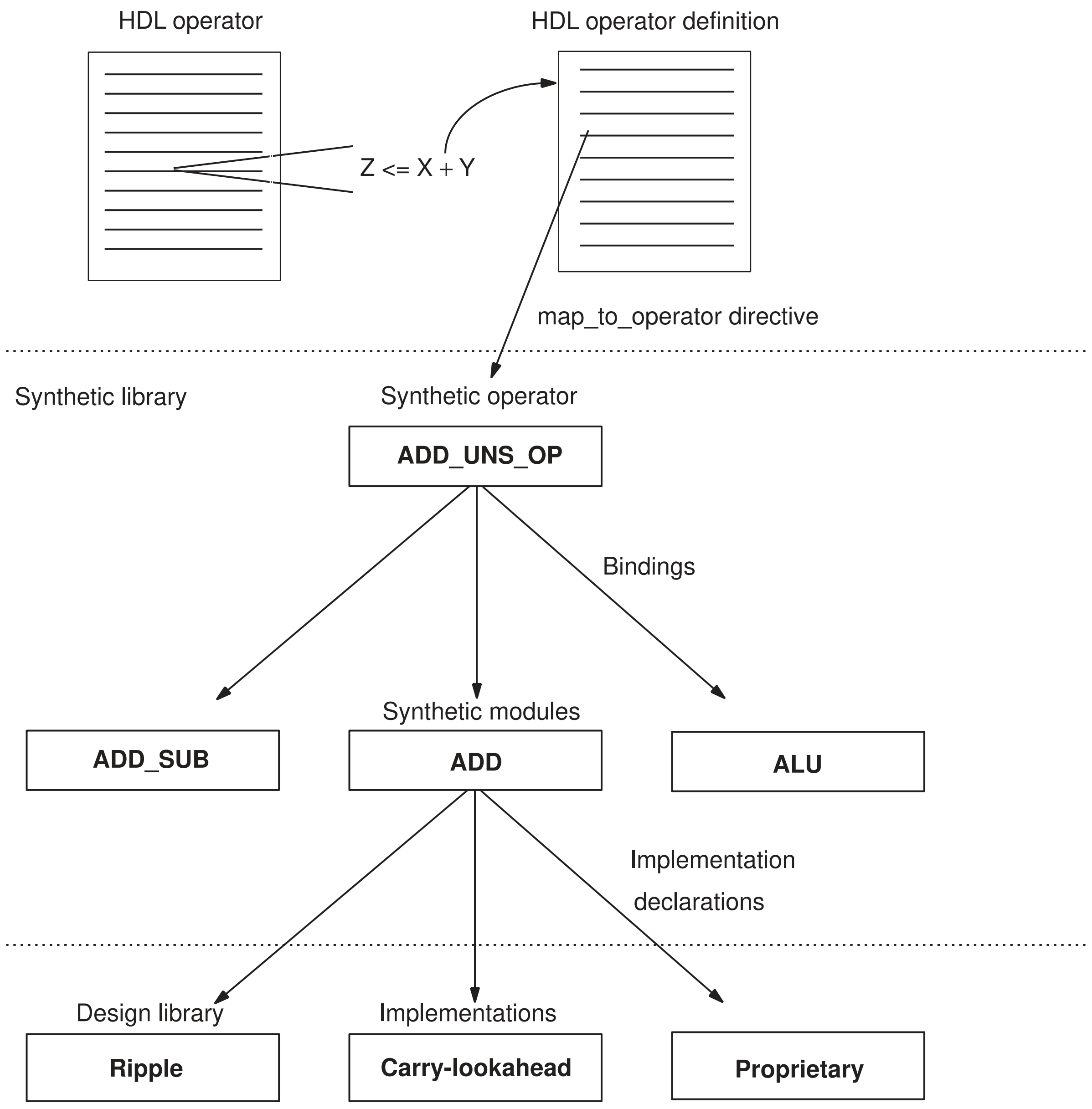
A synthetic library contains definitions for synthetic operators, synthetic modules, and bindings. It also contains declarations that associate synthetic modules with their implementations. To display information about the standard synthetic library that is included with the HDL Compiler license, use 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 3 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 HDL 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 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 shows the complete list of the standard synthetic operators. For more information, see the DesignWare Library documentation.

Table 4 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

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

HDL operator(s)	Synthetic operator(s)
<<, >> (logic)<<<, >>> (arith)	ASH_UNS_UNS_OP, ASH_UNS_TC_OP, ASH_TC_UNS_OP, ASH_TC_TC_OPASHR_UNS_UNS_OP, ASHR_UNS_TC_OP, ASHR_TC_UNS_OP, ASHR_TC_TC_OP
Barrel Shiftror, rol	BSH_UNS_OP, BSH_TC_OP, BSHL_TC_OPBSHR_UNS_OP, BSHR_TC_OP
Shift and Addsrl, sll, sra, sla	SLA_UNS_OP, SLA_TC_OPSRA_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 HDL 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.

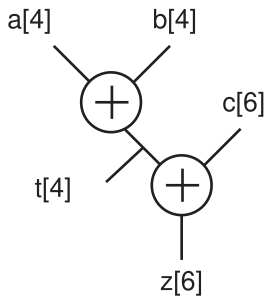
4-Bit Temporary Variable

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 32, 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 shows how the tool determines the default structure.

Example 32 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
```

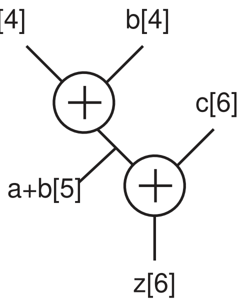
Figure 4 Default Structure for a 4-Bit Temporary Variable



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 5. 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.

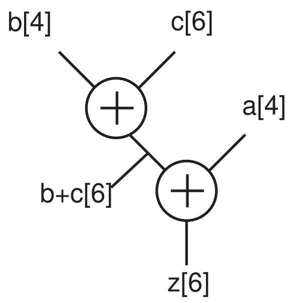
Figure 5 Structure for a 5-Bit Intermediate Result



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 6 shows the structure for this example.

Figure 6 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 4'b1010 after the assignment. A VER-318 warning is issued.

```
module t (/*...*/)
logic unsigned [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 33 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 33   Latch Inference

```
always @ (DATA or GATE) begin
   if (GATE) begin
```

```
      Q = DATA;
   end
end
```

In Example 34 and Example 35, 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 34 and Example 35 are not equivalent to Example 33, in which Q holds its previous value when GATE equals 1’b0.

Example 34   Avoiding Latch Inference—Method 1

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

Example 35   Avoiding Latch Inference—Method 2

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

Example 36 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 36   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 37 shows how to direct the tool to infer a multiplexer by using the `priority if` construct.

Example 37 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 38 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 38 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 39 shows how to direct the tool to infer a multiplexer by using the `unique if` construct.

Example 39   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 40 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 40   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

Selection and Multiplexing Logic

The HDL Compiler tool infers SELECT_OP and MUX_OP cells for logic that selects data signals based on control signals. SELECT_OP cells are mapped to combinational logic, while MUX_OP cells are mapped to structured trees of multiplexer cells. By default, the tool infers the cell that generally fits the needs of the RTL logic, but you can also control the inference yourself.

The following topics describe SELECT_OP and MUX_OP inference:

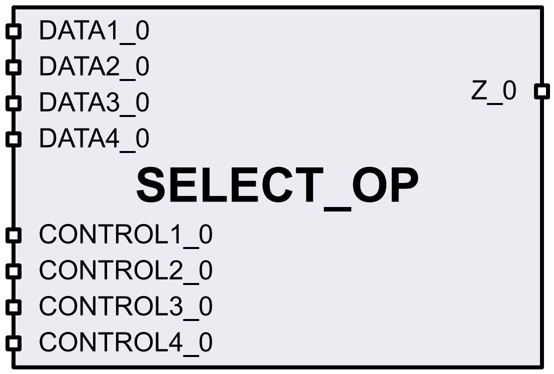
- The SELECT_OP Cell
- The MUX_OP Cell
- Default SELECT_OP and MUX_OP Inference Behavior
- Controlling Selection Statement Inference
- Controlling Array Read Inference
- Inferring One-Hot Multiplexer Logic

The SELECT_OP Cell

A SELECT_OP cell is a generic unmapped cell that uses *N* selection signals to select from *N* data signals. Because only one select signal can (and must) be asserted at a time, they are called *one-hot* selection signals.

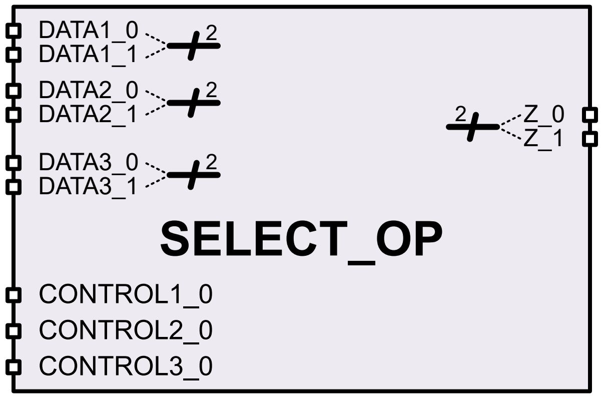
Figure 7 shows a SELECT_OP cell that selects one of four data input bits.

Figure 7 Single-Bit-Wide SELECT_OP Cell With Four Selectable Data Inputs



A SELECT_OP cell can have single-bit or multiple-bit data paths. The number of data inputs can be any practical integer number. Figure 8 shows a SELECT_OP cell that can select one of three two-bit-wide data inputs.

Figure 8 Two-Bit-Wide SELECT_OP Cell With Three Selectable Data Inputs



During elaboration, the tool creates GTECH control logic to drive the selection inputs according to the RTL functionality. (This logic, by construction, meets the one-hot signal requirement.)

During compile, the tool maps SELECT_OP cells to the logic library using the available combinational cells: simple Boolean gates, complex multiple-input gates, multiplexer cells, or any mix of these types.

Example SELECT_OP: RTL, Inference, and Synthesis

Example 41 shows an example RTL statement that selects from three data signals using an if/else statement.

Example 41 RTL Statement That Infers a SELECT_OP Cell

```
always_comb
  if (A && !B)
   ZZ = D1;
  else if (!A && B)
   ZZ = D2;
  else
   ZZ = D3;
```

Figure 9 shows the elaborated result. The SELECT_OP selection signals are driven by GTECH logic gates that implement the if/else conditions.

Figure 9 Elaborated GTECH Logic With Inferred SELECT_OP Cell

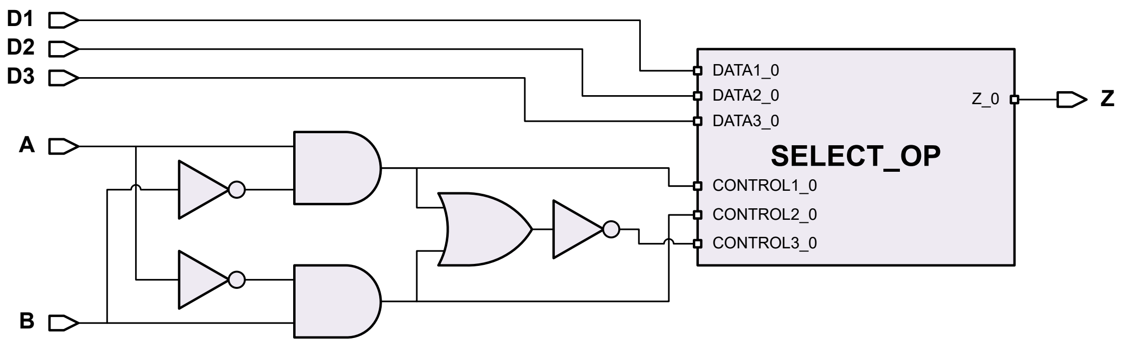
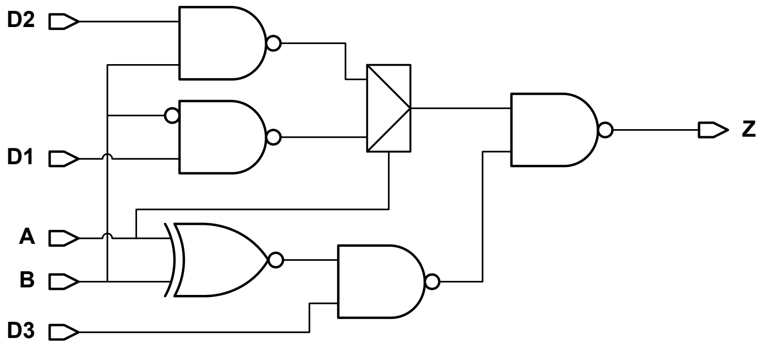


Figure 10 shows the compiled, mapped logic for the previous example. The control logic and the SELECT_OP cell are mapped together into an optimal gate structure.

Figure 10 SELECT_OP Cell and Selection Logic Mapped to Target Library

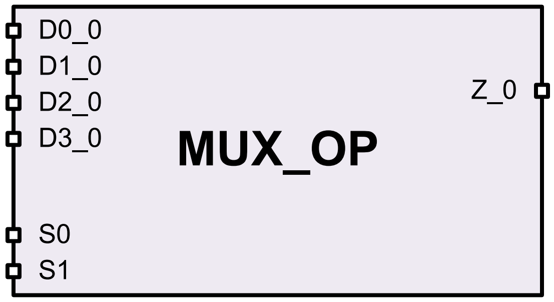


The MUX_OP Cell

A MUX_OP cell is a generic unmapped cell that uses log2(*N*) binary-encoded selection signals (rounded up) to select from *N* data signals.

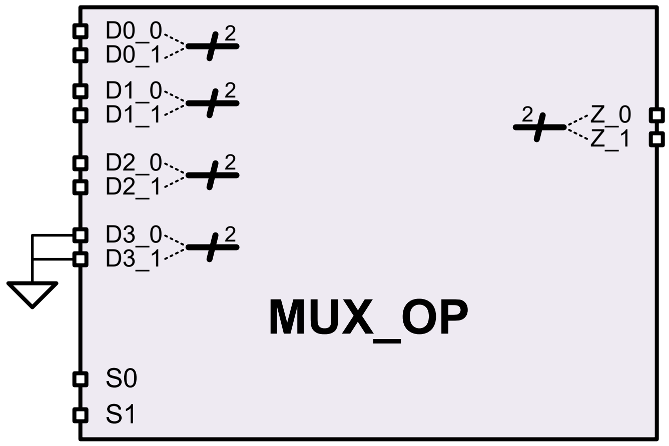
Figure 11 shows a MUX_OP that selects one of four data input bits.

Figure 11   Single-Bit-Wide MUX_OP With Four Selectable Data Inputs



A MUX_OP cell can have single-bit or multiple-bit data paths. For *S* selection inputs, the number of data inputs is 2^S, although the number of *selectable* data inputs can be less (with the excess tied to ground). Figure 12 shows a MUX_OP cell that can select one of three two-bit-wide data inputs.

Figure 12   Two-Bit-Wide MUX_OP With Three Selectable Data Inputs



During elaboration, the tool drives the MUX_OP selection inputs with the RTL selection signals. (These signals, by construction, meet the binary-encoded requirement.)

During compile, the tool maps MUX_OP cells to the logic library, strongly preferring a tree of multiplexer cells if possible.

Example MUX_OP: RTL, Inference, and Synthesis

Example 42 shows an example RTL statement that selects from seven data signals using an array read operation.

Example 42 RTL Statement That Infers a MUX_OP Cell

```
wire [6:0] DAT;  // 7 bits (not quite 2^3)
wire [2:0] SEL;
assign Z = DAT[SEL];  // synopsys infer_mux_override
```

Figure 13 shows the elaborated result. The MUX_OP selection signals are driven directly by the array index signals, which are binary-encoded by construction. The eighth data input of the MUX_OP cell is unused and thus tied to logic 0.

Figure 13 Elaborated Inferred MUX_OP Cell

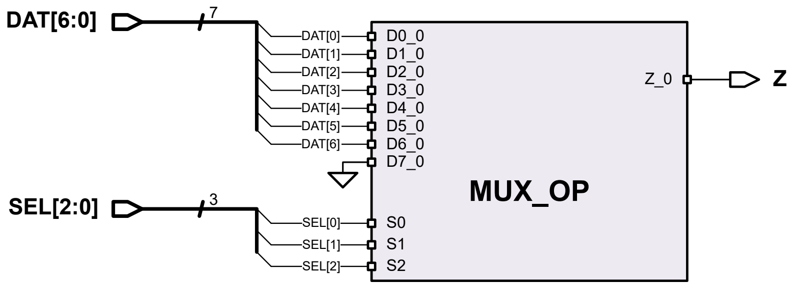
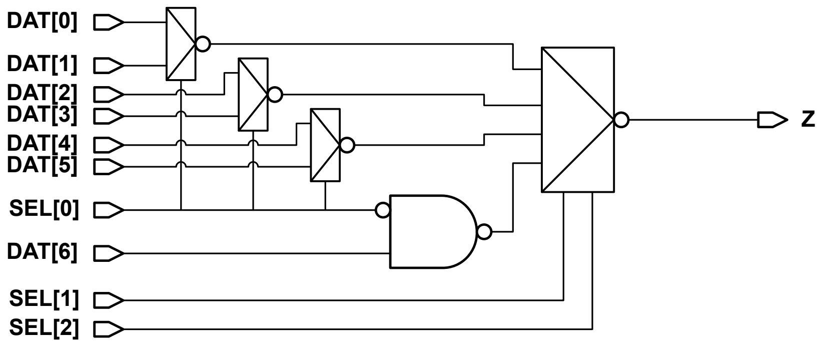


Figure 14 shows the compiled, mapped logic for the previous example. The MUX_OP is implemented using a compact inverting multiplexer tree structure, along with a logic gate that results from optimizing the unused MUX_OP input data bit.

Figure 14 MUX_OP Cell and Selection Logic Mapped to Target Library



Although MUX_OP cells are faster than SELECT_OP cells, they might increase congestion because of their pin density.

Default SELECT_OP and MUX_OP Inference Behavior

By default, the HDL Compiler tool infers SELECT_OP and MUX_OP cells using heuristics designed to fit most RTL use cases.

Table 5 shows the default inference behavior (when no RTL pragmas are applied).

Table 5 Default SELECT_OP and MUX_OP Inference Behavior

RTL Operator	**Default inference behavior**
`if` statement	SELECT_OP
`case` statement	SELECT_OP
The conditional operator (?:)	SELECT_OP
Array read (such as `DAT[ADR]`)	MUX_OP if `hdlin_mux_for_array_read_sparseness_limit` is met, SELECT_OP if not met

`if` and `case` statements and the conditional operator (?:) follow the same inference rules. Therefore, in this documentation they are collectively referred to as RTL *selection statements*.

Controlling Selection Statement Inference

By default, the tool infers SELECT_OP cells to implement selection statements: `case` and `if` statements and the selection operator (?:). However, you can configure the tool to infer MUX_OP cells instead.

There are two methods to control the inference behavior for selection statements:

- Globally, using application variables
- Locally, using RTL pragmas

These methods are interdependent in that either can take precedence over the other, depending on the settings used.

Controlling Selection Statement Inference Locally

You can locally infer MUX_OP cells for selection statements—`if` and `case` statements and the conditional operator (?:)—by placing the following pragmas in your RTL:

- `// synopsys infer_mux`

  Infer a MUX_OP for the selection statement, but only if permitted by global variable settings.

- `// synopsys infer_mux_override`

  Force a MUX_OP for the selection statement—regardless of any global variable settings—and force the tool to map to a tree of multiplexer cells.

The following sections describe the placement requirements for each RTL statement type. The requirements apply equally to both the `infer_mux` and `infer_mux_override` pragma. These requirements are for parsing order, with spaces and linefeeds ignored.

RTL Pragma Placement for if Statements

The inference pragma for an `if` statement must be placed directly after the closing parenthesis of the first conditional expression:

```
always_comb
  if (SEL1 == 2'b00)  // synopsys infer_mux_override
   Z = D1;
  else if (SEL1 == 2'b01)
   Z = D2;
  else
   Z = D3;
```

The pragma requirements and restrictions for `if` statements are:

- Each `if` expression must be an equality comparison of a simple variable to a constant value. Implicit single-bit Boolean tests are supported, such as `if (var)`.
- The `if` expressions cannot use any other operators, including negation (~) or array indexing.
- All comparisons must be of the same variable, although the last `else` branch can omit the `if` expression.
- All assignments must be to the same variable, although the values assigned can be arbitrarily complex and unique.

RTL Pragma Placement for case Statements

The inference pragma for a `case` statement must be placed directly after the closing parenthesis of the case selection expression:

```
always_comb
 case (SEL1)  // synopsys infer_mux
  2'b00: PARITY = ^{DAT[7:0]};
  2'b01: PARITY = ^{DAT[15:8]};
  2'b10: PARITY = ^{DAT[23:16]};
  2'b11: PARITY = ^{DAT[31:24]};
 endcase
```

The pragma requirements and restrictions for `case` statements are:

- The conditional expression must be a simple variable; it cannot use any operators, including negation (~).
- All assignments must be to the same variable, although the values assigned can be arbitrarily complex and unique.

RTL Pragma Placement for the :? Operator

The inference pragma for the conditional operator (?:) must be placed directly after the `?` (question mark) character:

```
assign ZCMP = SEL2 ? /* synopsys infer_mux */ (V1 < V2) : (V3 > V4);
```

The pragma requirements and restrictions for the conditional operator (?:) are:

- The selection expression before the `?` (question mark) character must be an equality comparison of a simple variable to a constant value. Implicit single-bit Boolean tests are supported, such as `(var) ?`.
- The selection expression cannot use any other operators, including negation (~) or array indexing.
- Multiple conditional operators in the same parent expression are not supported.

RTL Pragma Placement for Always Blocks

You can apply the `infer_mux` pragma to a named `always` or `always_comb` block to apply to all inferenceable `if` and `case` statements inside it. The pragma must be placed before the block and reference the block by name:

```
// synopsys infer_mux "this_block_name"
always_comb
begin: this_block_name
  ...
end
```

The pragma requirements and restrictions for `always` and `always_comb` blocks are:

- The `infer_mux` pragma supports block-based specification; the `infer_mux_override` pragma does not.
- `if` and `case` statements in the block are considered; conditional operators (?:) are not.
- `if` and `case` statements must each meet their own particular pragma criteria.

Controlling Selection Statement Inference Globally

To globally control the default inference behavior for selection statements, use the `hdlin_infer_mux` application variable.

Table 6 shows the valid values and resulting inference behaviors that apply.

Table 6 hdlin_infer_mux Application Variable Inference Behaviors

hdlin_infer_mux variable value	Default cell inference for selection statement	RTL MUX inference pragmas considered
`default` (default)	SELECT_OP	`infer_mux` `infer_mux_override`
`all`	MUX_OP	
`none`	SELECT_OP	`infer_mux_override`

For a MUX_OP cell to be inferred, the following global application variable criteria must also be met (unless the `infer_mux_override` pragma is applied):

- `hdlin_mux_size_limit` (default 32)

  This variable sets the upper limit for MUX_OP data input width.
- `hdlin_mux_size_min` (default 2)

  This variable sets the lower limit for MUX_OP data input width.
- `hdlin_mux_oversize_ratio` (default 100)

  This variable sets a limit for how many duplicated data signals are allowed, specified as the ratio of MUX_OP data inputs to unique data signals.

MUX_OP Inference and Resource Sharing

If you attempt to infer a MUX_OP cell for a selection statement that involves multiple synthetic operators, resource sharing could be degraded.

To prevent this, the tool issues the following warning message and infers a SELECT_OP cell instead:

```
Warning:  /proj/rtl/case.sv:30: No MUX_OP inferred for the case because
it might lose the benefit of resource sharing. (ELAB-370)
```

In this case, you can still force a MUX_OP cell by adding the `infer_mux_override` pragma to your RTL.

Controlling Array Read Inference

By default, the tool infers a MUX_OP cell when you access an array value (single bit or word) using a nonconstant index value. For example,

```
assign Z = DAT[SEL];
```

There are two methods to control the inference behavior for array reads:

- Globally, using application variables
- Locally, using RTL pragmas

These methods are interdependent in that either can take precedence over the other, depending on the settings used.

Controlling Array Read Inference Globally

To globally control the default inference behavior for selection statements, use the `hdlin_infer_mux` application variable.

Table 7 shows the valid values and resulting inference behaviors that apply.

Table 7 hdlin_infer_mux Application Variable Inference Behaviors

hdlin_infer_mux variable value	**Default cell inference for array read**	**RTL MUX inference pragmas considered**
`default` (default)	MUX_OP	`infer_mux_override`
`all`	MUX_OP (sparseness limit ignored)	
`none`	SELECT_OP	`infer_mux_override`

For a MUX_OP cell to be inferred, the following global application variable criteria must also be met (unless the `hdlin_infer_mux` application variable is set to `all`):

- `hdlin_mux_for_array_read_sparseness_limit` (default 90)

  When the width of the array being indexed is not a power of two, this variable specifies a percentage requirement for how many MUX_OP data inputs must be connected.

Controlling Array Read Inference Locally

You can unconditionally force a MUX_OP cell for an array read, regardless of the global inference or sparseness variable settings, by placing the `infer_mux_override` pragma in your RTL.

For example,

```
assign mask_bit = mask[idx];  // synopsys infer_mux_override
```

The `infer_mux_override` pragma also forces the tool to map to a tree of multiplexer cells.

Array reads do not use or support the `infer_mux` pragma, as they are already the default when the `hdlin_infer_mux` variable is set to `default`.

RTL Pragma Placement for Array Reads

To apply the pragma to all array reads of an RTL statement, place it at the end of the line after the semicolon:

```
assign selected_bit =
 mem1[addr1][idx1] ||
 mem2[addr2][idx2];  // synopsys infer_mux_override
```

To apply the pragma to specific array reads, place it directly before the closing bus bracket as an inline comment:

```
assign selected_bit =
 mem1[addr1 /*synopsys infer_mux_override*/][idx1] ||
 mem2[addr2 /*synopsys infer_mux_override*/][idx2];
```

For nested array reads, an inline pragma applies to all reads nested within that level:

```
// inference applies to addr1 (directly applied) *and* X (nested inside)
assign selected_bit =
 mem1[addr1[X] /*synopsys infer_mux_override*/][idx1[Y]];
```

Inferring One-Hot Multiplexer Logic

Some technology libraries contain fast *one-hot* multiplexer cells. Logically, these cells are similar to AND-OR or AND-OR-INVERT cells, but electrically they require the selection inputs to be one-hot. Because of this requirement, synthesis cannot automatically make use of them. However, you can use the `infer_onehot_mux` RTL pragma to take advantage of them.

To do this, use the `unique case` SystemVerilog statement, which indicates that the branches are mutually exclusive. Then, place the `infer_onehot_mux` pragma directly after the closing parenthesis of the case selection expression.

The one-hot selection signals can be used together as the case selection expression (Example 44) or individually as the case item expressions (Example 43).

Example 43   One-Hot Multiplexer Using One-Hot Case Selection Signals

```
module onehot_1 (
  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;
  default:  out = 1'bX;
 endcase
end
endmodule
```

Example 44   One-Hot Multiplexer Using One-Hot Case Item Signals

```
module onehot_2 (
  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;
  default: out = 1'bX;
 endcase
end
endmodule
```

The `infer_onehot_mux` pragma infers a SELECT_OP cell with an internal attribute that marks it for one-hot MUX cell mapping.

The number of selection signals is important: a one-hot MUX cell must exist in the technology library that is as least as wide as the number of branches in the `case` statement. The tool cannot compose wider one-hot MUX logic from smaller one-hot MUX cells.

The `infer_onehot_mux` is independent of the `infer_mux` and `infer_mux_override` pragmas and is not affected by any of the MUX inference application variables.

For details on one-hot MUX library requirements and logic synthesis, see the “Mapping to One-Hot Multiplexers” topic in the *Design Compiler User Guide*.

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 HDL Compiler documentation.

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 the following 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];`

Truncation type	Description

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 45 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 45   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 46 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 46   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 47 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 47 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 48 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 48 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:

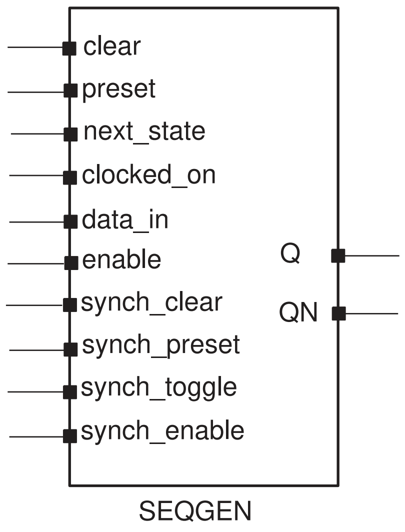
- Generic Sequential Cell SEQGEN
- Inference Reports for Registers
- Register Inference Guidelines
- Register Inference Examples

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 15 to represent an inferred flip-flop or latch.

Figure 15 SEQGEN Cell and Pin Assignments



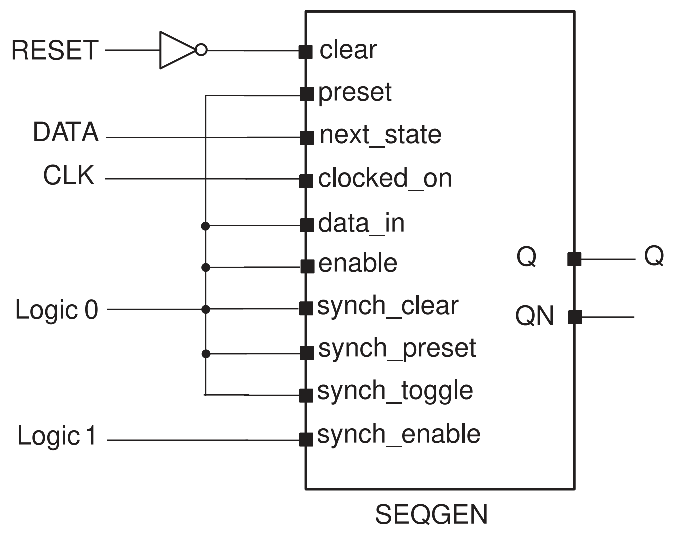
Example 49 shows how to direct the HDL Compiler tool to use a SEQGEN cell to implement a D flip-flop with an asynchronous reset.

Example 49 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 16 shows the SEQGEN implementation.

Figure 16 SEQGEN Implementation



Example 50 shows the `report_cell` output, where the inferred Q_reg flip-flop is mapped to a SEQGEN cell.

Example 50 report_cell Output

```
****************************************
Report : cell
Design : dff_async_set
Version: P-2019.03
Date   : Tue May 14 14:42:54 2019
****************************************

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
```

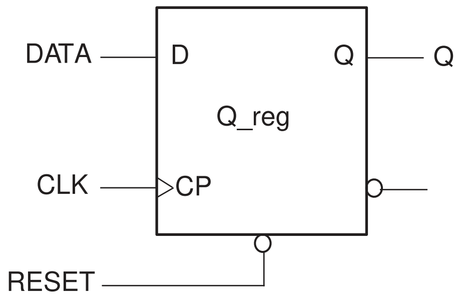
Example 51 shows the GTECH netlist.

Example 51 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 the HDL Compiler tool synthesizes the design, the SEQGEN is mapped to the appropriate flip-flop in the logic library. Figure 17 shows an example of an implementation after compile.

Figure 17 Gate-Level Implementation



Note:

If the logic library does not contain the inferred flip-flop or latch, the HDL 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

The HDL Compiler tool 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, the level is set to `basic`. When the level is set to `basic` or `comprehensive`, the tool generates a report similar to Example 52. This basic inference report shows only which type of register was inferred.

Example 52 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 53 shows a verbose inference report.

Example 53 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 the HDL Compiler tool 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

The HDL Compiler tool supports multiple events in a single `always` block, as shown in Example 54.

Example 54   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 55 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 55 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 56 shows the inference report which contains the six inferred flip-flops.

Example 56 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 57 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 57 creates a more area-efficient design.

Example 57 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 58 shows the inference report, which contains three inferred flip-flops.

Example 58 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 it to `all`.
- To preserve all unloaded flip-flops only, set it to `ff`.
- To preserve all unloaded latches only, set it to `latch`.
- To preserve all unloaded sequential cells, including unloaded sequential cells that are used solely as loop variables, set it to `all+loop_variables`.

- To preserve flip-flop cells only, including unloaded sequential cells that are used solely as loop variables, set it to `ff+loop_variables`.
- To preserve unloaded latch cells only, including unloaded sequential cells that are used solely as loop variables, set it to `latch+loop_variables`.

If you want to preserve specific registers, use the `preserve_sequential` directive as shown in Example 59 and Example 60.

Caution:

To preserve unloaded cells through compile, you must set the `compile_delete_unloaded_sequential_cells` variable to `false`. Otherwise, the HDL Compiler tool removes them during optimization.

Example 59 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 60.

Example 59   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 60 preserves all combinational logic before reg save.

Example 60   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 the tool. 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 61, sum2 is not inferred, so `preserve_sequential` does not save sum2.

Example 61   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 it 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 62 for a feedthrough.

Example 62

```
module test (
   input clk,in,
   output logic out
);
logic tmp1;
always_ff @ (posedge clk)
begin
   tmp1 <= in;
   out  <= tmp1;
end
endmodule
```

With the `hdlin_preserve_sequential` variable 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 63   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).

To avoid unwanted latches, use the SystemVerilog `always_comb` construct to model combinational logic. For these blocks, the tool issues an ELAB-974 warning if inferred latches are detected.

In addition, you can set the `hdlin_check_no_latch` variable, which causes ELAB-395 warnings to be issued for latches inferred in `always` blocks.

Latch inference warnings are not issued for `always_latch` blocks.

As shown in Example 64, one branch of the `case` statement is commented out, so output DOUT is not fully specified and the tool infers a latch.

Example 64

```
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 in the HDL Compiler tool:

- 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 tool cannot infer flip-flops and latches with three-state outputs. You must instantiate these components in your Verilog description.
- The tool cannot infer flip-flops with bidirectional pins. You must instantiate these components in the RTL.
- The tool cannot infer flip-flops with multiple clock inputs. You must instantiate these components in the RTL.
- The tool cannot infer multiport latches. You must instantiate these components in the RTL.
- The 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 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 65 shows that a D latch is inferred for the `always_latch` and `always@` constructs.

Example 65   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 tool generates the inference report shown in Example 66.

Example 66   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 67 shows the recommended coding style for an asynchronously set latch using the `async_set_reset` directive.

Example 67   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 68.

Example 68   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 69 shows the recommended coding style for an asynchronously reset latch using the `async_set_reset` directive.

Example 69   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 70.

Example 70 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 71.

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 71 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 72 shows the inference report.

Example 72 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 73, 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 73   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 18, or you can synthesize a flip-flop with built-in reset functionality, as shown in Figure 19.

Figure 18 Flip-Flop With External Combinational Logic to Represent Reset

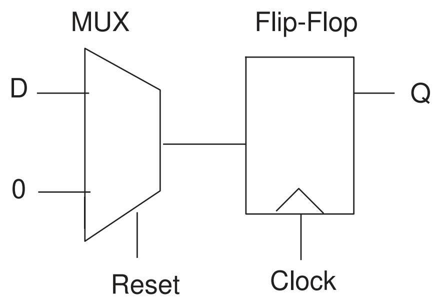
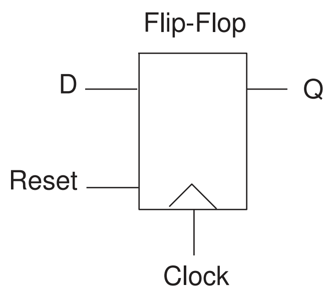


Figure 19 Flip-Flop With Built-In Reset Functionality



The intended implementation is not apparent from the RTL code. You should specify HDL Compiler synthesis directives or 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 74 infers a basic D flip-flop.

Example 74   Basic D Flip-Flop

```
module dff_pos (
   input DATA, CLK,
   output logic Q
);
always_ff @(posedge CLK)
   Q <= DATA;
endmodule
```

The tool generates the inference report shown in Example 75.

Example 75   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 76 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 76   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
```

The tool generates the inference report shown in Example 77.

Example 77   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 78 infers a D flip-flop with an asynchronous reset.

Example 78   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
```

The tool generates the inference report shown in Example 79.

Example 79   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 80 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 80   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 81 shows the inference report.

Example 81   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 HDL 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 82 infers a D flip-flop with synchronous reset. The `sync_set_reset` directive is applied to the RESET signal.

Example 82   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
```

The tool generates the inference report shown in Example 83.

Example 83   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 84 to infer a D flip-flop with both synchronous and asynchronous load signals.

Example 84   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
```

The tool generates the inference report shown in Example 85.

Example 85 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 the HDL Compiler tool 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 86, 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 86   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 87 shows the inference report.

Example 87 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 88, 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 88

```
module empty_alway_ff (
   input logic clk, in,
   output logic out
);

logic tmp;
always_ff @(posedge clk)
begin
   tmp <= in;
end
endmodule
```

6 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:

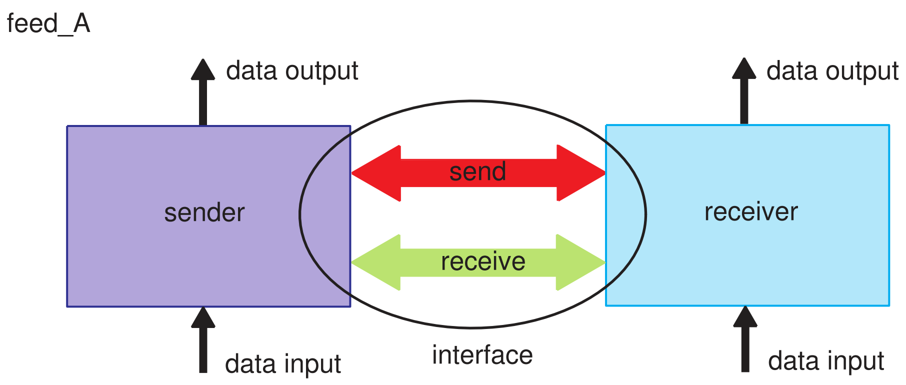
- Example: Interface With Wires
- Example: Interface With Modports
- Example: Interface With Modport Expressions
- Example: Interface With Functions
- Example: Interface With Functions and Tasks
- Example: Interface With always Blocks

Example: Interface With Wires

The feed_A design uses an interface for the send and receive buses as shown in Figure 20. 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.

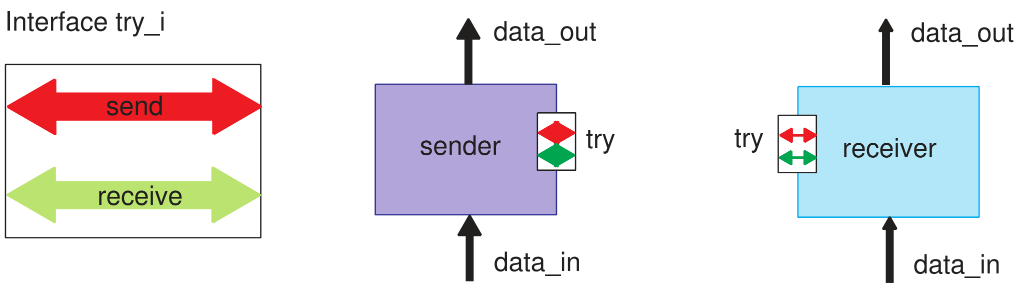
Figure 20 Design feed_A



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 21 Interface With Wires Only



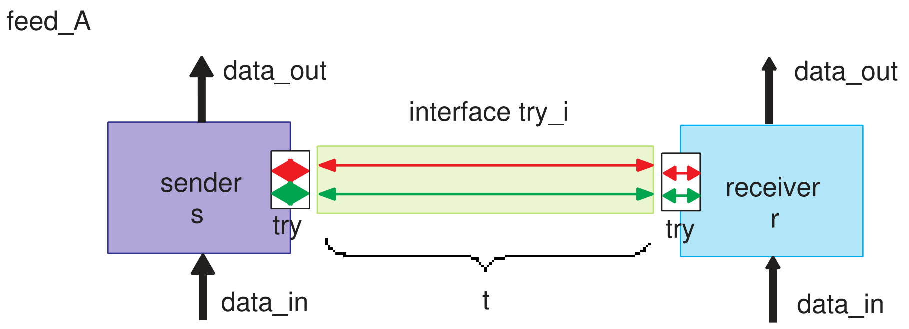
This example shows the RTL for the interface with wires only.

```
// 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.

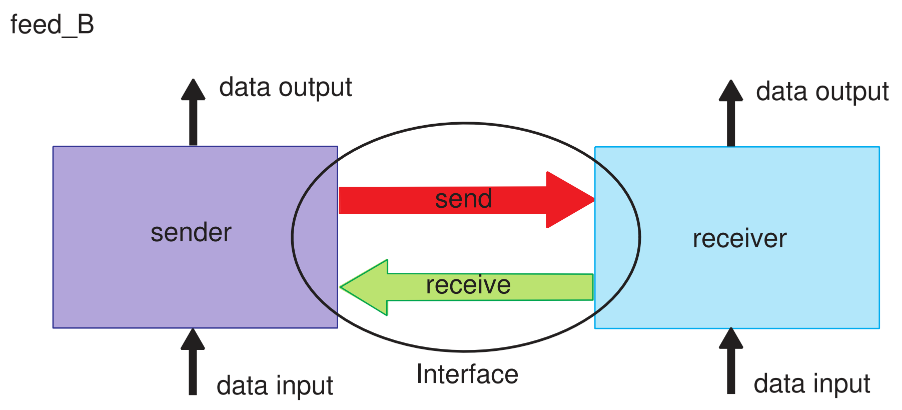
Figure 22 Block Diagram of the feed_A Design



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 23 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

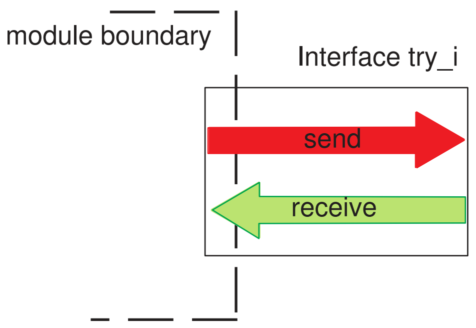
The try_i Interface

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
```

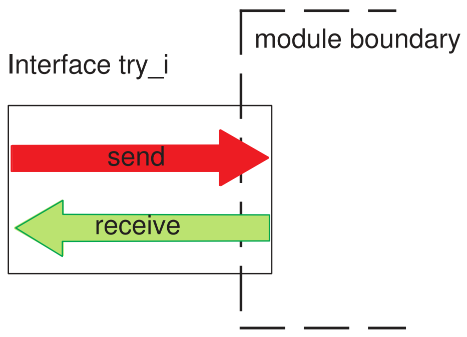
The following figure shows the block diagram of the sendmode modport in the try_i interface used by a module. The send bus is the output from the module, and the receive bus is the input to the module.

Figure 24 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 25 try_i Interface With the receivemode Modport



The sender Module

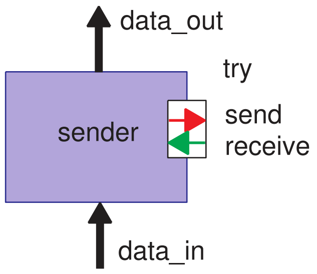
The following code shows that the sender module uses the sendmode modport of the try_i interface as a port named try. In the sendmode modport, the send bus is an output, and the receive bus is an input.

```
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 26 The sender Module With the sendmode Modport



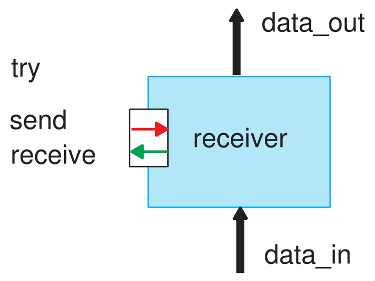
The receiver Module

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.

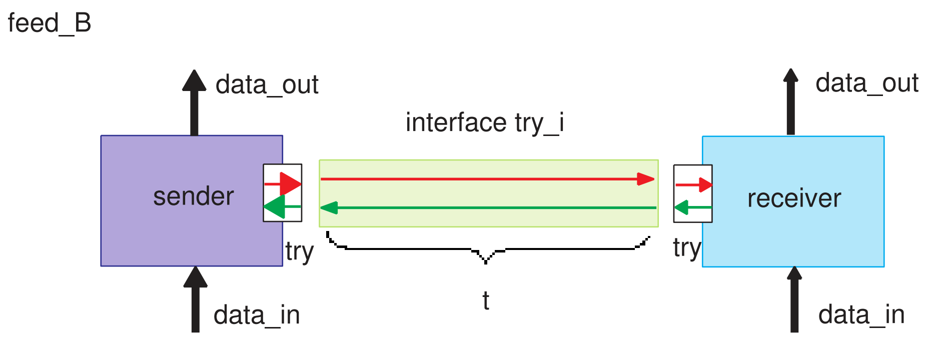
Figure 27 The receiver Module With the receivemode Modport



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 28 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 89 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 90   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:

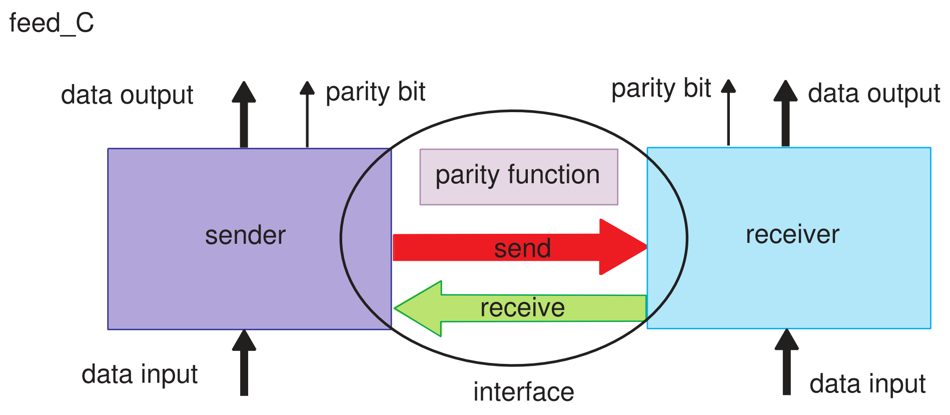
```
module regBlock (myIf iPort);
always @(posedge iPort.clk)
   iPort.dout <= iPort.din;
endmodule
```

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 29 Design feed_C: Interface With Modports and a Function



Complete RTL of the feed_C Design

As shown in Example 91, 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.

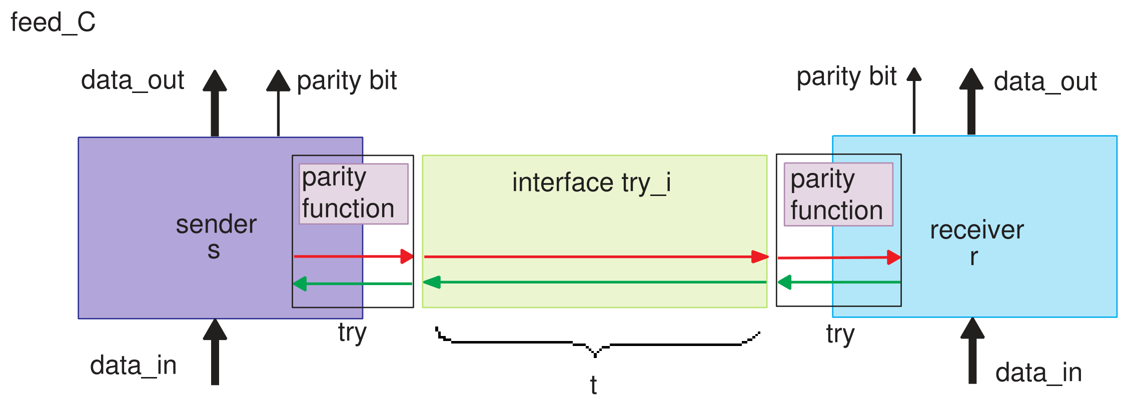
Example 91 Design feed_C: Interface With Modports and a Function

```
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 30   Block Diagram of the feed_C Design

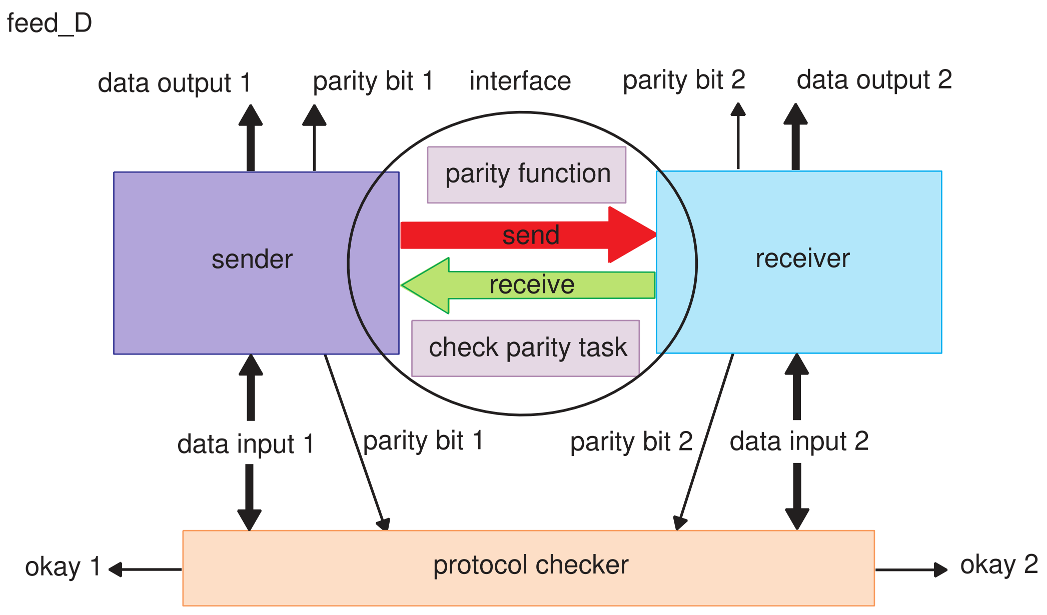


Example: Interface With Functions and Tasks

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 31 Design feed_D: Interface With Modports, a Function, and a Task

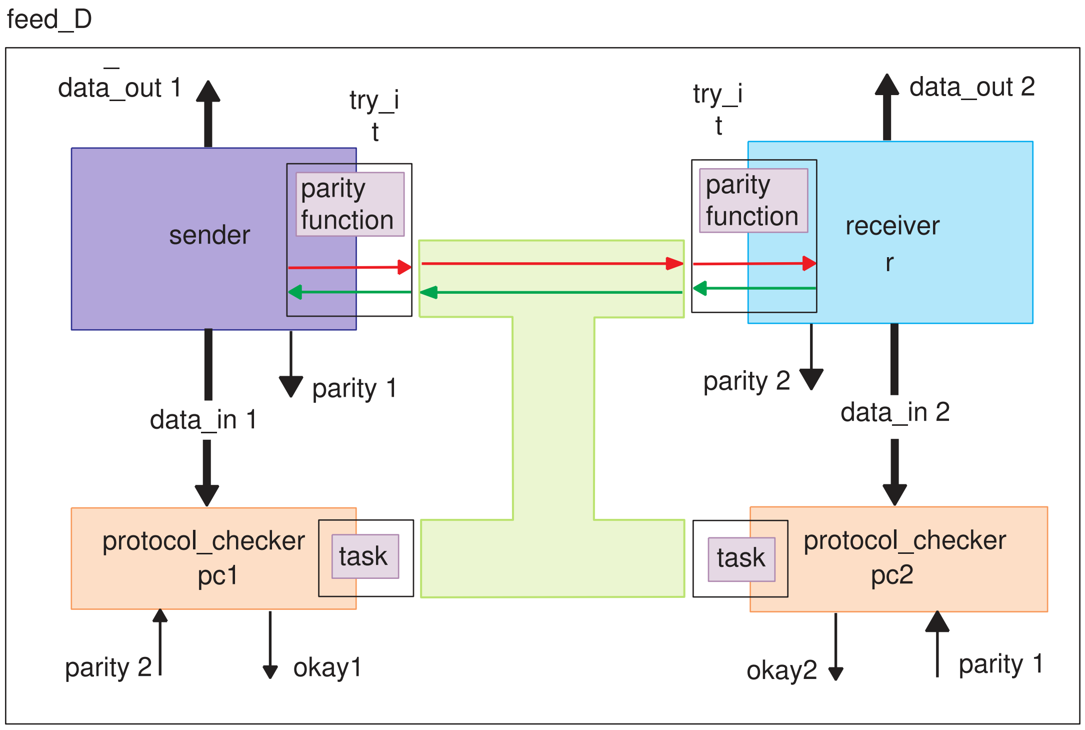


Block Diagram of the feed_D Design

As shown in Figure 32, 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 32 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 92 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 93 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 (
```

Feedback

```
  .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 33, the ALU design uses an interface to connect all the top-level inputs to the subdesigns.

The ALU design contains the following interface ports:

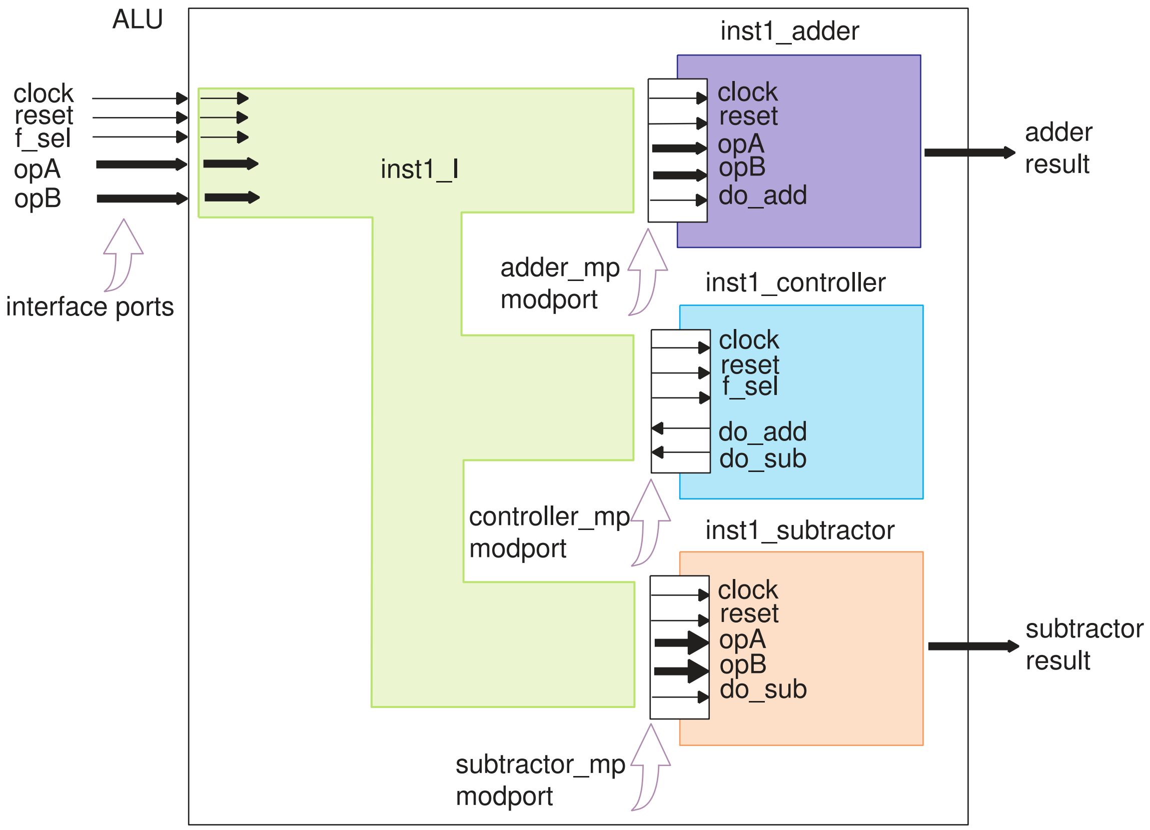
- Input ports: clock, reset, f_sel, opA, and opB
- Output ports: adder_result and subtractor_result

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 33 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 94   Complete RTL of the ALU Design

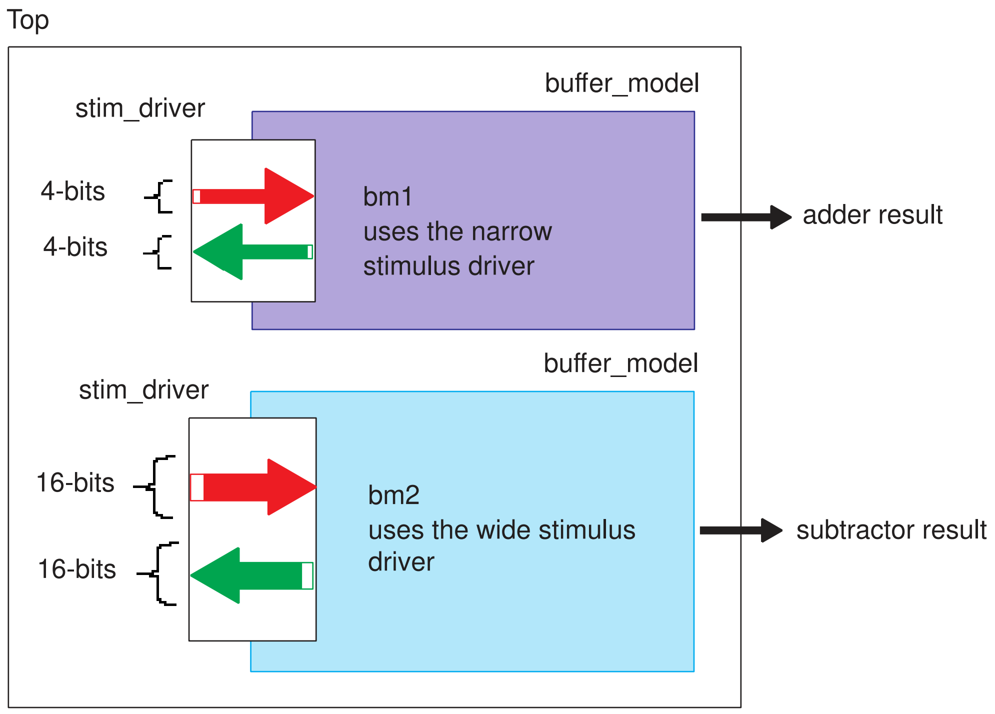
```
interface I (
   input logic clock, reset, f_sel,
   logic [7:0] opA, opB
);
logic do_add;
logic do_sub;
modport adder_mp (input clock, reset, do_add, opA, opB);
modport subtractor_mp (input clock, reset, do_sub, opA, opB);
modport controller_mp (input clock, reset, f_sel, output do_add, do_sub);
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 34   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 95 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:

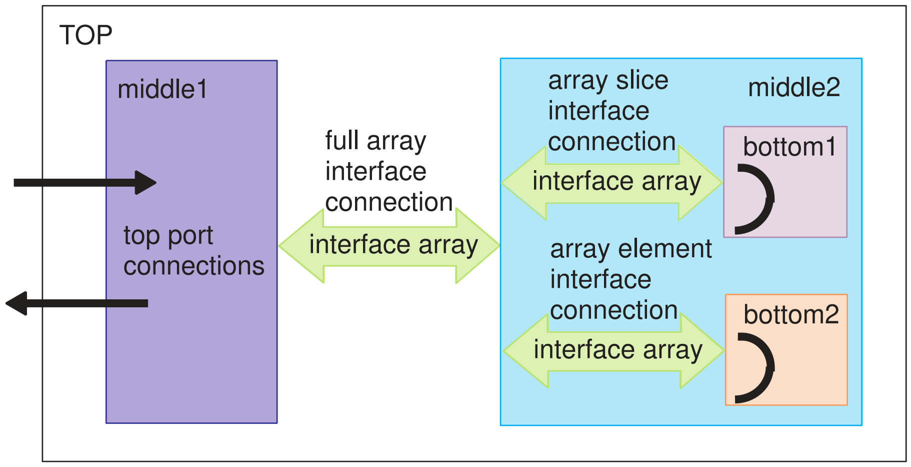
- Full array interface connection
- Array slice connection for interfaces with modports
- Array element connection for interfaces with modports

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 35 Arrays of Interfaces


TOP
middle1
top port connections
full array interface connection
interface array
array slice interface connection
middle2
interface array
bottom1
array element interface connection
interface array
bottom2

Coding Styles for Interface Arrays

The following examples show the supported coding styles for the interface arrays shown in Figure 35:

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
```

Interface Array Coding Style Recommendations

When using array interfaces, try to implement the following suggested coding styles:

- Standardize on a convention [*N*-1:0] or [0:*N*-1] for arrayed interface ports and use it consistently throughout the design. The bounds must end or start at zero, respectively.
- If the standard direction is [*N*-1:0], then set the following application variable before reading the RTL:

  ```
  set_app_var hdlin_interface_port_downto true
  ```

  This should eliminate any ELAB-123 messages.
- Mention a modport whenever passing down interface instances, even a slice of an arrayed instance.

  This should eliminate any VER-735 messages.

The following example has comments that explain how the coding style suggestions are implemented.

```
module downto(IFC.mp ifc_port[3:0]);
// Set hdlin_interface_port_downto to true
// because interface array module ports are declared N-1 down to 0;
// if you don't, you'll see ELAB-123, and GTECH port names that
// look swapped
endmodule

module test;
  IFC ifc_inst[15:0]();
  // Use a modport when you pass down interface instances, even slices
  // If you don't, you'll see VER-735 and extra unconnected GTECH ports
  downto U3(ifc_inst[15:12].mp);
  downto U2(ifc_inst[11:8].mp);
  downto U1(ifc_inst[7:4].mp);
  downto U0(ifc_inst[3:0].mp);
endmodule

interface IFC();
  logic x, y, z, w;
  modport mp (input x, output y);
endinterface
```

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
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 and slice connections for interfaces with modports, i1[2].modB and i1[0:1].modB, is supported.

```
module top();
   Inf i1[0:7];
   middle1 M1 (i1[0].modB);    // supported
   middle2 M2 (i1[1:2].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 96 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 97   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 98   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, the HDL Compiler tool 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 before 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 parameters without default values are not allowed at the top level of elaboration because there is not enough information available to resolve the interface. However, such modules can be elaborated as described in Parameterized Interface Ports.

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.

Feedback

7

Modeling Three-State Buffers

The HDL Compiler tool infers a three-state driver when you assign the value z (high impedance) to a variable. The tool 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

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. The HDL Compiler tool generates an error for this expression.

Three-State Driver Inference Report

The `hdlin_reporting_level` variable determines whether the HDL Compiler tool generates a three-state inference report. If you do not want inference reports, set the level to `none`. The default is `basic`, which indicates to generate a report. Example 99 shows a three-state inference report:

Example 99   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. The tool generates the same report for the default and verbose reports for three-state inference. For more information about the `hdlin_reporting_level` variable to `basic+fsm`, see Customizing Elaboration Reports.

Assigning a Single Three-State Driver to a Single Variable

Example 100 infers a single three-state driver and shows the associated inference report.

Example 100 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
```

Example 101 Inference Report

```
============================================
| Register Name |       Type       | Width |
============================================
|   OUT1_tri    | Tri-State Buffer |   1   |
============================================
```

Example 102 infers a single three-state driver with MUXed inputs and shows the associated inference report.

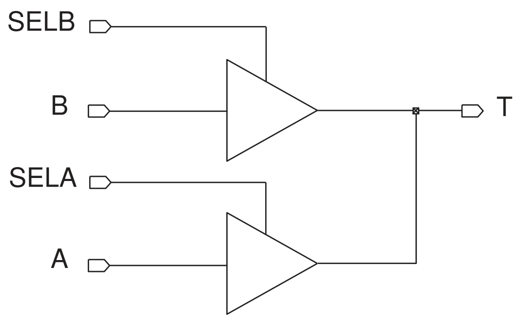
Example 102 Single Three-State Driver With MUXed Inputs

```
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 36, always use assign statements, as shown in Example 103.

Figure 36 Two Three-State Drivers Assigned to a Single Variable



Example 103 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 104). 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 104 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, the HDL Compiler tool also registers the driver's enable signal, as shown in Example 105. Figure 37 shows the compiled gates and the associated inference report.

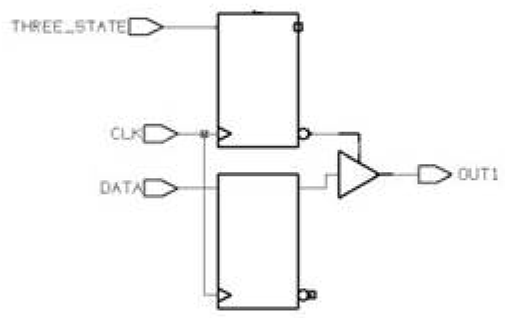
Example 105 Three-State Driver With Enable and Data Registered

```
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
```

Example 106 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 37 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 107 shows a three-state gate instantiation with an active-high enable and no inverted output.

Example 107 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, the HDL Compiler tool issues an ELAB-366 error message and terminates. To force the tool to warn for this condition (ELAB-365) but continue to create a netlist, set the `hdlin_prohibit_nontri_multiple_drivers` variable to `false` (the default is `true`). With this variable false, the tool builds the generic netlist for all legal designs. If a design is illegal, such as when one of the drivers is a constant, the tool 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)
```

8 Other SystemVerilog Features

This section provides examples of supported SystemVerilog features.

Feedback

- 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:

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 subtracter 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
```

Real Math Functions

In the declarations of local parameters, the tool supports all the standard unary system functions that have equivalent C language real math library functions as listed in Table 8.

Table 8 Unary System Functions to C Language Real Math Functions Cross-Listing

Unary System Function	Equivalent C Language Function	Description
$ln (x)	log (x)	Natural logarithm
$log10 (x)	log10 (x)	Decimal logarithm
$exp (x)	exp (x)	Exponential
$sqrt (x)	sqrt (x)	Square root
$floor (x)	floor (x)	Floor
$ceil (x)	ceil (x)	Ceiling
$sin (x)	sin (x)	Sine
$cos (x)	cos (x)	Cosine
$tan (x)	tan (x)	Tangent
$asin (x)	asin (x)	Arc-sine

Table 8 Unary System Functions to C Language Real Math Functions Cross-Listing (Continued)

Unary System Function	Equivalent C Language Function	Description
$acos (x)	acos (x)	Arc-cosine
$atan (x)	atan (x)	Arc-tangent
$sinh (x)	sinh (x)	Hyperbolic sine
$cosh (x)	cosh (x)	Hyperbolic cosine
$tanh (x)	tanh (x)	Hyperbolic tangent
$asinh (x)	asinh (x)	Arc-hyperbolic sine
$acosh (x)	acosh (x)	Arc-hyperbolic cosine
$atanh (x)	atanh (x)	Arc-hyperbolic tangent

Restrictions

HDL Compiler does not support the following binary system functions:

- $pow
- $atan2
- $hypot

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 1800-2017*.

```
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.
- The parameterized function must be called from within a module.

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

For synthesis restrictions on multidimensional arrays, see Unsupported Constructs.

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
```

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 -format sverilog {submodule.sv ...}
dc_shell> analyze -format sverilog top_module.sv
dc_shell> analyze -format 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 1800-2017*.

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
```

- HDL Compiler Tcl script

```
define_design_lib lib1 -path ./lib1
define_design_lib lib2 -path ./lib2
define_design_lib rtlLib -path ./rtlLib
analyze -format sverilog -library lib1 sub1.v
analyze -format sverilog -library lib2 sub2.v
analyze -format sverilog -library rtlLib sub3.v
analyze -format sverilog -library rtlLib top.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
  ```

- HDL 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 -format sverilog -library lib1 sub1.v
  analyze -format sverilog -library lib2 sub2.v
  analyze -format sverilog -library lib3 sub3.v
  analyze -format sverilog -library rtlLib top.v
  analyze -format 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
```

- HDL 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 -format sverilog -library lib4 sub2.v
analyze -format sverilog -library lib5 sub3.v
analyze -format sverilog -library lib3 sub1.v
analyze -format sverilog -library lib1 top1.v
analyze -format sverilog -library lib2 top2.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:
- Assignment patterns on parameter specification overrides
- Assignment patterns on port connections 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));
  ```

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 1800-2017* specifies a generic wire type, `interconnect`, to model generic netlists with different types of nets. During synthesis, the HDL 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 1800-2017*.

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 shows the variable settings.

Table 9 Settings for Keeping Signal Names

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.
`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.

Table 9 Settings for Keeping Signal Names (Continued)

Setting	Description
`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 `hdlin_keep_signal_name` variable 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 108, 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 108 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 10 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 10 Variable and Directive Matrix for Signals test1, test2, and test3

keep_signal_name	**hdlin_keep_signal_name variable 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	*
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 108)	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, use the following setting 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 11 shows the coding styles that can cause simulation and synthesis mismatches and how to avoid the mismatches.

Table 11 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.
Inferring D flip-flops with synchronous and asynchronous loads.	See D Flip-Flop With Synchronous and Asynchronous Load.
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 the HDL Compiler tool, 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 tool. Because of the simulation and synthesis mismatches, the 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.

For more information, see *IEEE Std 1800-2017*.

9

HDL Synthesis Directives

The HDL Compiler tool allows you to annotate your SystemVerilog RTL with directives for synthesis. Pragmas are RTL comments that control how synthesis processes the RTL. 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 synthesis directives are special comments that affect the actions of the 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 235 for details.

The following sections describe the HDL 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, the HDL Compiler tool 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, the HDL Compiler tool 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, the HDL Compiler tool 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 109.

Example 109 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 110 and Example 111.

Example 110 Embedding Constraints With // Delimiters

```
...
// synopsys dc_tcl_script_begin
// set_max_area 0.0
// set_max_delay 0.0 -to port_z
// synopsys dc_tcl_script_end
...
```

Example 111 Embedding Constraints With / and */ Delimiters*

```
/* synopsys dc_tcl_script_begin
   set_max_area 10.0
   set_max_delay 5.0 port_z
   # no end needed for this form
*/
```

The HDL Compiler tool 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 Tcl # comment character within the RTL comments.

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 completes. 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.
- 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 112 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 112 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 113 shows an invalid `enum` declaration.

Example 113 Invalid enum Declaration

```
parameter /* synopsys enum colors */
red = 3'b000, green = 1;
// [2:0] required
```

Example 114 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 114 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 115 relates to Example 114 and shows an invalid encoding for colors.

Example 115 Invalid Bit Value Encoding for Colors

```
counter = 3'b111;
```

Because 111 is not in the definition for colors, it is not a valid encoding. The HDL Compiler tool accepts this encoding, but issues a warning for this assignment.

You can use enumeration literals just like constants, as shown in Example 116.

Example 116 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 117 shows the declaration of the enumeration.

Example 117 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 118 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 118 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 the HDL Compiler tool 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.

Caution:

Marking a case statement as full when it actually is not full can cause the simulation to behave differently from the synthesized logic because the HDL Compiler tool does not generate a latch to handle the implicit default condition.

The syntax for the `full_case` directive is

```
// synopsys full_case
```

In Example 119, `full_case` is set on the first case statement and `parallel_case` and `full_case` directives are set on the second case statement.

Example 119 // 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 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 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 is 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, the tool 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, the HDL Compiler tool might not infer complex multibit sequential cells, such as a JK flip-flop.

For more information, see the HDL Compiler documentation.

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 120 shows usage.

Example 120 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 121 shows the inference report.

Example 121 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 122 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 123 shows the multibit inference report.

Example 123 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 124 shows a multibit component report.

Example 124 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 Controlling Selection Statement 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 Inferring One-Hot Multiplexer Logic on page 99.

keep_signal_name

Use the `keep_signal_name` directive to provide the HDL Compiler tool 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 the HDL Compiler tool 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 the HDL Compiler tool 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 is true at a time. This directive prevents the HDL Compiler tool 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.

Caution:

> Marking a case statement as parallel when it actually is not parallel can cause the simulation to behave differently from the synthesized logic because the HDL Compiler tool 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 125, 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 125 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: Synthesis generates logic to ensure that the branch listed first in the case statement takes effect.
Parallel but not full	Latches created: Synthesis 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: Synthesis generates logic to make sure that the branch listed first in the case statement takes effect.Latches created: Synthesis 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 otherwise are optimized away by the HDL Compiler tool. 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 the HDL Compiler tool is mapped to a flip-flop in the logic library with a synchronous set/reset pin, or the tool uses 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 the tool 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`, the

HDL Compiler tool 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 the HDL Compiler tool 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 126 shows the usage.

Example 126 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 the HDL Compiler tool 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 127 shows usage.

Example 127 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 128 shows usage.

Example 128 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 the HDL Compiler tool. 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 the HDL Compiler tool are not supported.

Table 12 shows how each directive is handled by each pragma prefix.

Table 12 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 12 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. The HDL Compiler tool supports SystemVerilog attributes by reapplying them as synthesis attributes, so they become accessible as if set by the `set_attribute` command.

The following sections describe SystemVerilog attribute support in the HDL Compiler tool:

- Using SystemVerilog Attributes in Synthesis
- Supported Attributes
- Supported RTL Constructs

For more information, see *IEEE Std 1800-2017* 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, the HDL Compiler tool 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, use the following setting:

```
dc_shell> set_app_var hdlin_sv_enable_rtl_attributes true
```

When this feature is enabled, SystemVerilog attributes are parsed by the tool 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; the HDL Compiler tool 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. The HDL Compiler tool 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 HDL Compiler database.

RTL:

```
(* my_design_type = "IP", dont_touch *)
module IP_block (
  ...
endmodule
```

Synthesis:

```
dc_shell> report_attributes [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 HDL 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 HDL 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 HDL 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 HDL 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`) exists on the resulting mapped sequential cells.

RTL:

```
  (* size_only *)
  always @(posedge clk)
    counter <= (counter + write - read);
```

Synthesis:

```
dc_shell> report_attributes \
          [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
```

10

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 HDL 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 13 shows the SystemVerilog `unique` and `priority` constructs and the Verilog equivalency `full_case` and `parallel_case` compiler directives.

Table 13   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     |
===============================================
Presto 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     |
===============================================
Presto compilation completed successfully.
```

Using Void Functions Instead of Tasks Inside always_comb

The *IEEE Std 1800-2017* 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 14 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)
Module renaming issues	Renamed Modules Example 3

Table 14 Other Troubleshooting Guidelines (Continued)

For guideline on	See
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 HDL Compiler installation.

FIFO Example

Example 129 uses a variety of SystemVerilog features to build a FIFO.

Example 129 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
```

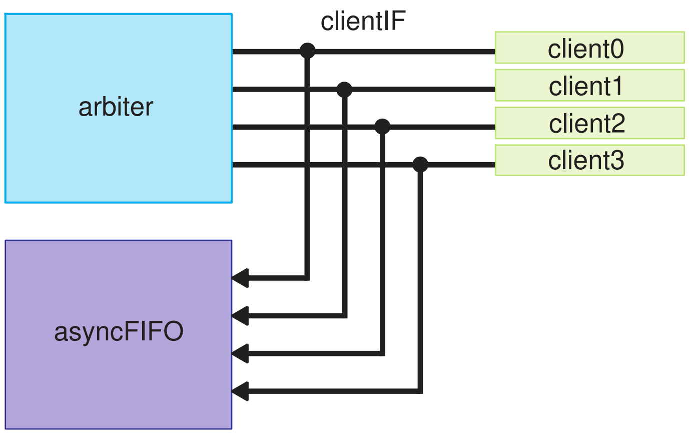
```
                if (!out.stall)
                count <= count - 1;
         2'b11: // Concurrent read and write. No change in count
                count <= count;
      endcase
   end
end

// 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 38, 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 38 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 130, 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 130 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 130. The example uses one-dimensional array; you can also use two-dimensional arrays.

As shown Example 131, 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 131 arbiterMod Module

```
module arbiterMod #(parameter lengthOfId = 8)(
   input logic  clk,
   input logic  rst,
   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 132 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 132 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 133 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 133 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 134 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 134 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

- 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 synthesizes performed by the HDL Compiler tool. When this output is constrained and optimized, the structure might be changed depending on the design constraints and option settings. For more information, see the HDL Compiler documentation.

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.

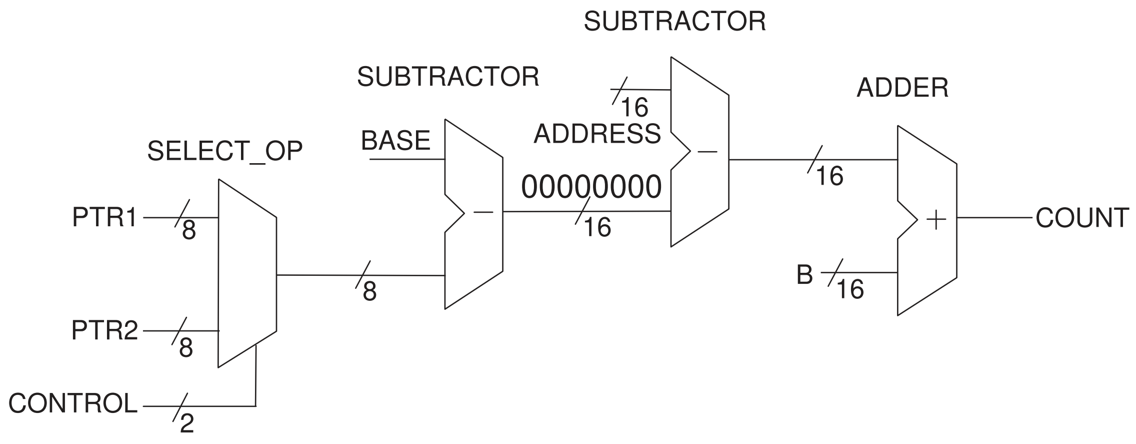
Original RTL

In Example 135, 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 39, 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 136.

Example 135 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 39 Schematic of the Original RTL



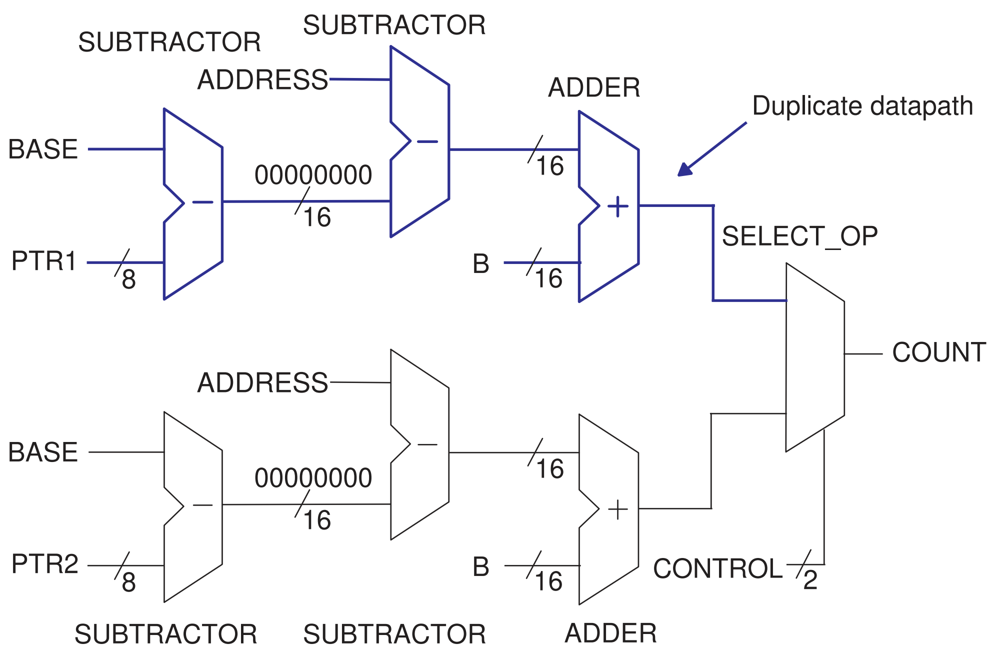
Modified RTL With the Duplicate Datapath

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 136 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 40 Schematic of the Modified RTL



See Also

- Selection and Multiplexing Logic

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.

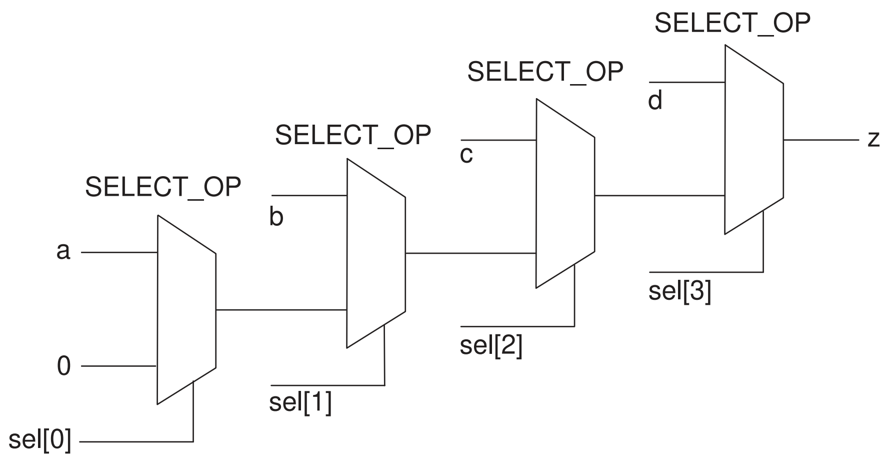
RTL With Sequential if Statements

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 137 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 41 Schematic of the RTL



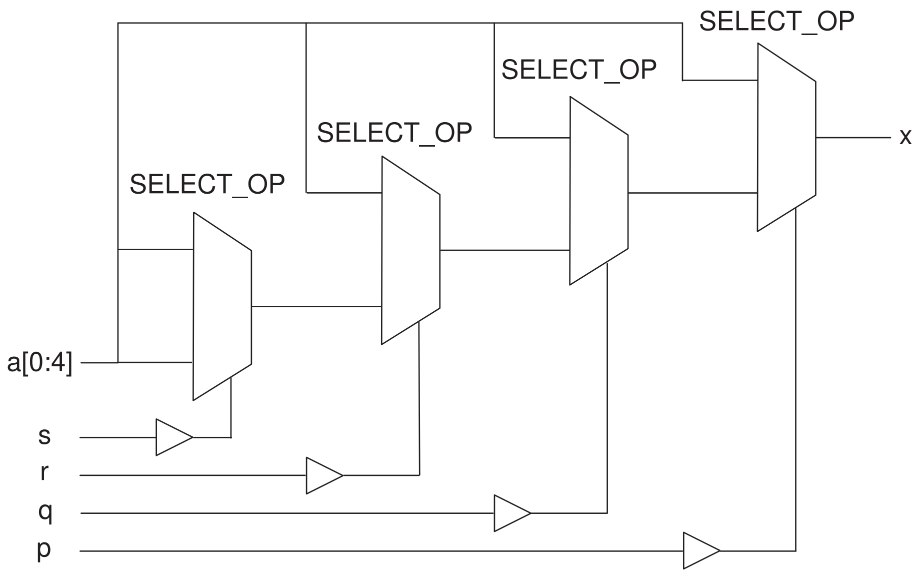
Modified RTL With Named begin-end Blocks

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 138 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 42 Schematic of the Modified RTL



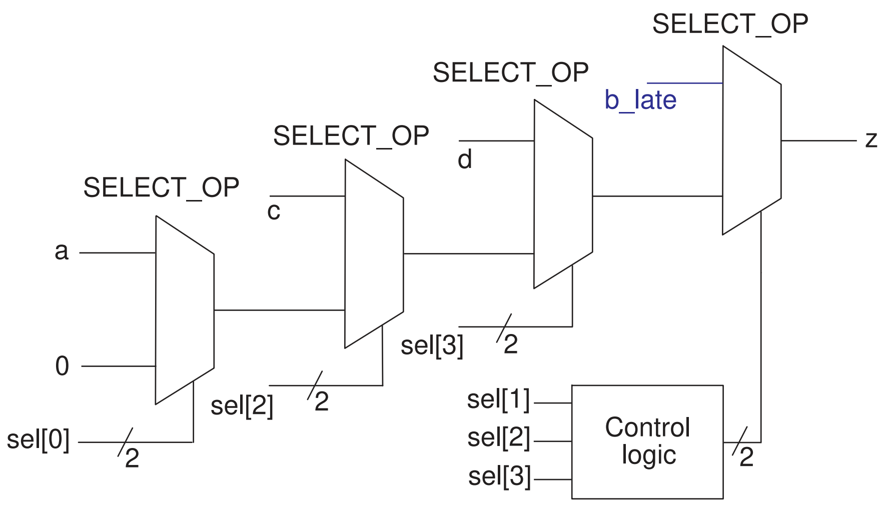
Late-Arriving Data Signal Example 1

This example shows how to place the late-arriving b_late signal close to the z output.

Example 139 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 43   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.

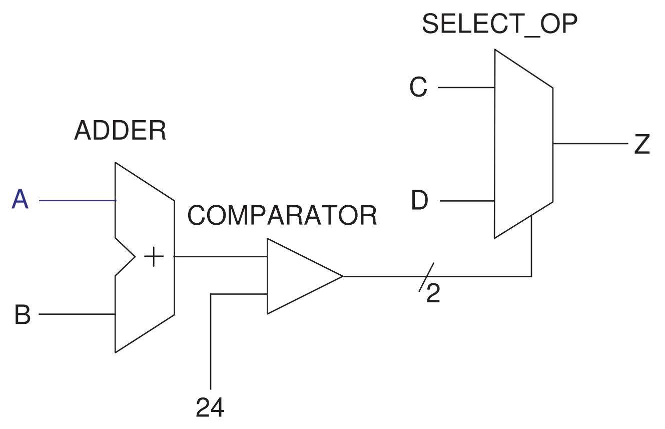
Original RTL Containing the Late-Arriving Input A

The original RTL contains input A that is late arriving.

Example 140 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 44 Schematic of the Original RTL



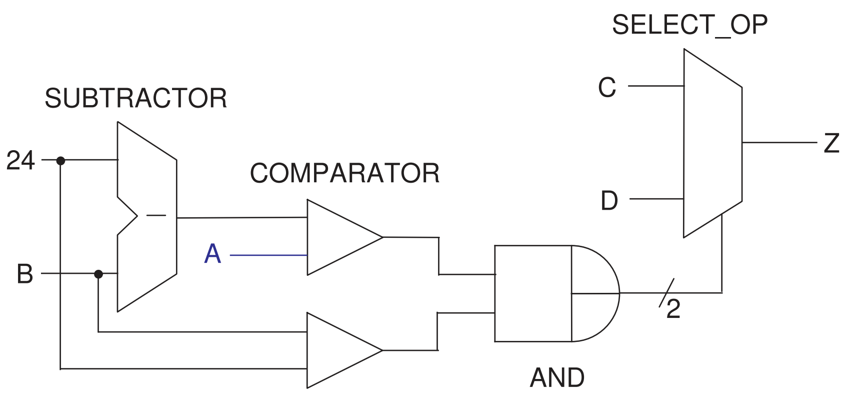
Modified RTL

The following RTL restructures the code to move signal A closer to the output.

Example 141 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 45 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

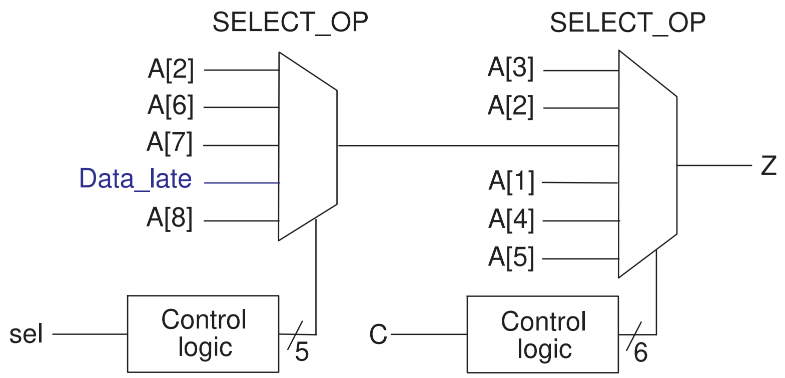
The original RTL contains input Data_late that is late arriving.

Example 142 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 46 Schematic of the Original RTL



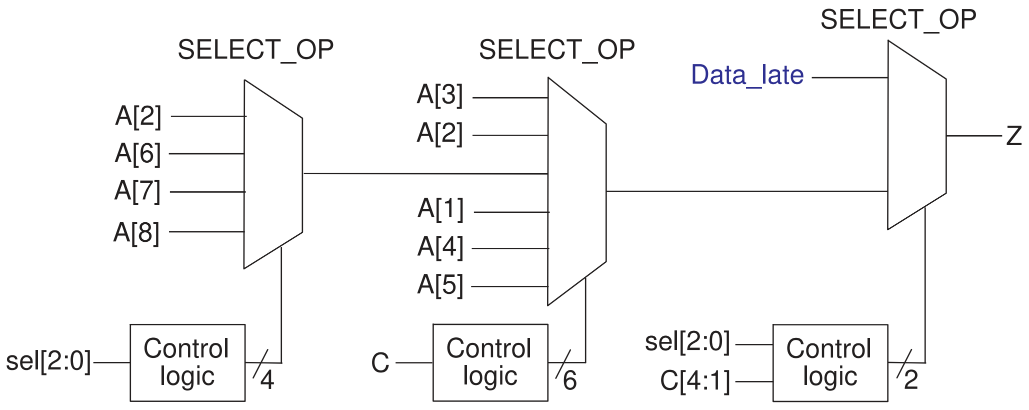
Modified RTL for the Late-Arriving Signal

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 143 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 47   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.

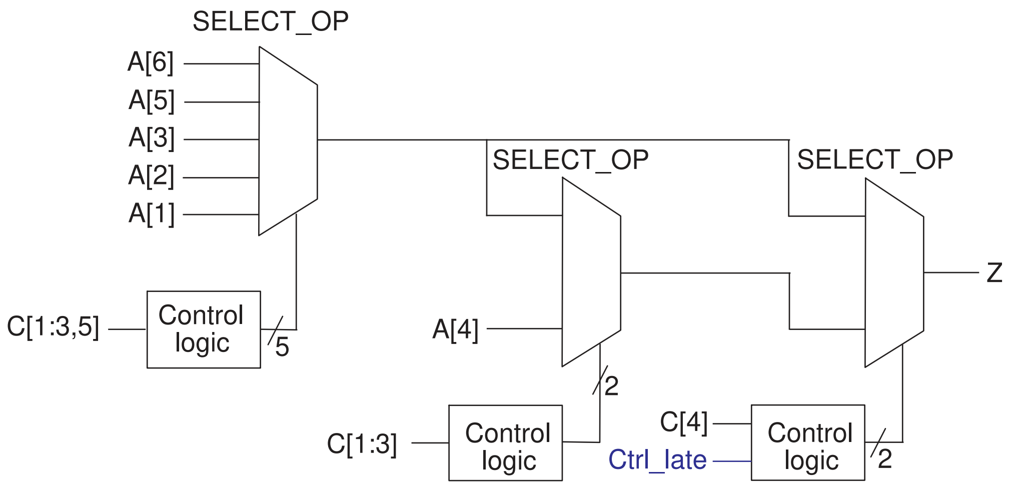
In this example, input Ctrl_late is a late-arriving control signal and is placed close to output Z.

Example 144 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 48 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 145 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 146 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 HDL 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 HDL 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 147 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 148 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 149 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 150 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 151 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 152 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 15. 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 15 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
Unsupported Constructs

The Synopsys SystemVerilog tool does not support all the synthesis features described in the *IEEE Std 1800-2017*. 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:

For more information, see

- The `$onehot`, `$onehot0`, 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 HDL 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.
- 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.
- Default port values for input ports in module 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.
- The `let` construct declaration is 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 uwire net type is not supported.
- *IEEE Std 1800-2017*
- Conversion Between Two-State and Four-State Variables

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. HDL 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 constraints

See *constraints*.

flip-flop

An edge-sensitive memory device.

HDL
Hardware Description Language.

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. HDL 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 HDL 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.