HDL Compiler™ for Verilog User Guide

Version U-2022.12-SP3, April 2023

SYNOPSYS®

Feedback

Contents

About This Manual

The HDL Compiler tool translates a Verilog hardware language description into a generic technology (GTECH) netlist that is used by the Synopsys synthesis tools to create an optimized netlist. This manual describes the following:

- Modeling combinational logic, synchronous logic, three-state buffers, and multibit cells with the HDL Compiler tool for Verilog
- Sharing resources
- Using directives in the RTL

Audience

The *HDL Compiler for Verilog User Guide* is written for logic designers and electronic engineers who are familiar with the Design Compiler™ tool. Knowledge of the Verilog language is required, and knowledge of a high-level programming language is helpful.

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 Vision™
- Design Compiler®
- Fusion Compiler™
- DesignWare® components
- Library Compiler™
- Verilog Compiled Simulator® (VCS)

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.

Convention	Description
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
Verilog for Synthesis

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

Reading Verilog Designs

You can use either of these methods to read Verilog designs into the HDL Compiler tool.

- `read_verilog` or `read_file -format verilog`

  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_verilog  parameterized_interface.v
current_design top
link
compile
write -format verilog -hierarchy \
   -output gates.parameterized_interface_rd.v
```

-
  ```
  analyze -format verilog {files}
  elaborate topdesign
  ```

  For example,

  ```
  analyze -format verilog parameterized_interface.v
  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.

Note:

See Also

- Parameterized Designs
- Automatic Detection of Input Type

Specifying the Verilog Version

To specify which Verilog language version to use during the read process, set the `hdlin_vrlg_std` variable . The valid values for this variable are `1995`, `2001`, and `2005`, corresponding to the 1999, 2001, and 2005 Verilog LRM releases respectively. When you set the `hdlin_vrlg_std` variable to a valid version, the Verilog LRM features of this version are enabled when you read Verilog RTL into the tool. The default version 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 Verilog files other than the default (.v) and sets file source lists that exclude some directories.

```
dc_shell> set_app_var hdlin_autoread_verilog_extensions {.v .ver}
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 verilog -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
`hdlin_autoread_sverilog_extensions`	Specifies the file extension to analyze files as SystemVerilog files.	.sv .sveri log

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.

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 in unknown, the tool assumes the .ddc format. The file extensions in the following 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)
```

Coding for QoR

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

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

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

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

Reading 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"
```

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
```

Customizing Elaboration Reports

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

Table 1   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
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:

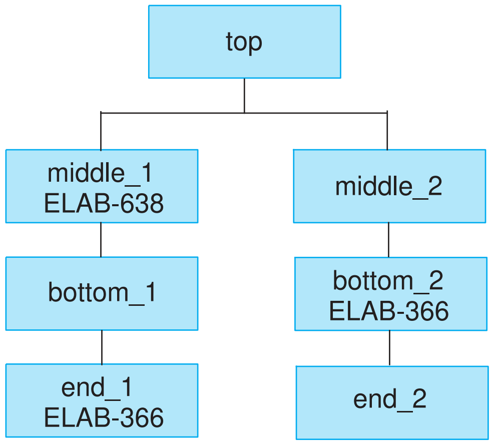
1. Identify and fix all syntax errors in the design.
2. Set the `hdlin_elab_errors_deep` variable to `true`.

   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 1, to report all elaboration errors in the hierarchy.

Figure 1 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 tool is not a linting tool, but you can use the `check_design` command in the HDL Compiler tool to perform basic structural checks.
- 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.
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)
...
```

Netlist Reader

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

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

The following table summarizes the recommended and alternative commands to read in your designs.

Type of input	Reading method

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

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

The tool automatically decrypts netlists that are encrypted according to the IEEE 1735 standard.

Automatic Detection of Input Type

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

Caution:

For best memory usage and runtime, do not mix RTL and netlist designs into a single read. The automatic detector chooses one reader—netlist or RTL—to read all files included in the command. Mixed files default to the RTL reader, because it can read both types; the netlist reader can read-only netlists.

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

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

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

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 1364-2005*, parameters without a default are not supported.

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 and INSIZE parameters with a default of eight:

Example 5 Declaring a Parameterized Design

```
module sub #(parameter SIZE=8, 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_verilog` command to build a design with parameters is not recommended because you can build a design only with the default of the parameters.

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

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

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

Defining Macros

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

Note:

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

See Also

- `define

Predefined Macros

You can also use the following predefined macros:

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

Use of $display During RTL Elaboration

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

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

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

HDL Compiler produces output such as the following during elaboration:

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

Inputs and Outputs

This section contains the following topics:

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

Input Descriptions

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

The functional elements of a Verilog description for synthesis include

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

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

Functional and structural descriptions can be used in the same module, as shown in Example 12.

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 12   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_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_blocks` command. For information about how to use the `group` command with Verilog designs, see the man page.

Component Inference and Instantiation

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

- You can directly instantiate registers into a Verilog description, selecting from any element in your ASIC library, but the code is technology dependent and the description is difficult to write.
- You can use Verilog constructs to direct the HDL Compiler 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 HDL Compiler reads a design, it creates a generic netlist consisting of generic components, such as SEQGENs.

For example, after HDL Compiler reads the my_fsm design in Example 13, it creates the generic netlist shown in Example 14.

Example 13 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 14   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 15 shows the `report_cell` output for my_fsm design.

Example 15   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
```

Feedback

```
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
```

Inference Reports

The HDL Compiler tool generates inference reports for the following inferred components:

- Flip-flops and latches, described in Inference Reports for Registers on page 80.
- MUX_OP cells, described in Selection and Multiplexing Logic on page 61.
- Three-state devices, described in Three-State Driver Inference Report on page 101.
- Multibit devices, described in infer_multibit and dont_infer_multibit on page 113.

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}
```

Language Construct Support

HDL Compiler supports only those constructs that can be synthesized, that is, realized in logic. For example, you cannot use simulation time as a trigger, because time is an element of the simulation process and cannot be realized in logic. See Appendix B, Verilog Language Support.”

Licenses

Reading and writing license requirements are listed in the following table.

Reader	Reading license required		Writing license required	
	RTL	**Netlist**	**RTL**	**Netlist**
HDL Compiler	Yes	Yes	No	No
UNTI-Verilog (netlist reader)	Not applicable	No	Not applicable	No
Automatic detection (read_verilog)	Yes	Yes	Not applicable	Not applicable

Feedback

2

Coding Considerations

This chapter describes HDL Compiler synthesis coding considerations in the following sections:

- General Verilog Coding Guidelines
- Guidelines for Interacting With Other Flows

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 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 written 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 written values and reinitializes all static variables in functions and tasks to unknowns at the beginning of each call.

Verilog code that does not conform to this synthesis assumption can cause a synthesis and simulation mismatch. You should declare all functions and tasks by using the `automatic` keyword, which instructs the simulator to allocate new memory for local variables at the beginning of each function or task call.

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 Verilog Language Reference Manual.

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 Design 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 2 shows the variable settings.

Table 2 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 2 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 16, 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 16 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 3 shows how the settings of the variable and directive affect the preservation of signals test1, test2, and test3. An asterisk (*) indicates that the Design Compiler tool does not attempt to preserve the signal.

Table 3 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 16)	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 4 shows the coding styles that can cause simulation and synthesis mismatches and how to avoid the mismatches.

Table 4 Coding Styles Causing Synthesis and Simulation Mismatches

Synthesis and simulation mismatch	Coding technique
Using the `one_hot` and `one_cold` directives in a Verilog 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 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.
Selecting bits from an array that is not valid.	See Part-Select Addressing Operators ([+:] and [-:]).
Masking the set or reset signal with an unknown during initialization in simulation.	See sync_set_reset.
Using asynchronous design techniques.	The tool does not issue any warning for asynchronous designs. You must verify the design.
Using unknowns and high impedance in comparison.	See Unknowns and High Impedance in Comparison.
Including timing control information in the design.	See Timing Specifications.
Using incomplete sensitivity list.	See Sensitivity Lists.
Using local `reg` variables in functions or tasks.	See Initial States for Variables.

Unknowns and High Impedance in Comparison

A simulator evaluates an unknown (x) or high impedance (z) as a distinct value different from 0 or 1; however, an x or z value becomes a 0 or 1 during synthesis. In HDL Compiler, these values in comparison are always evaluated to false. This behavior difference can cause simulation and synthesis mismatches. To prevent such mismatches, do not use don’t care values in comparison.

In the following example, simulators match 2'b1x to 2’b11 or 2’b10 and 2’b0x to 2’b01 or 2’b00, but both 2'b1x and 2’b0x are evaluated to false in the HDL Compiler tool. Because of the simulation and synthesis mismatches, the HDL Compiler tool issues an ELAB-310 warning.

```
case (A)
   2'b1x:... //  You want 2'b1x to match 11 and 10 but
             //  HDL Compiler always evaluates this comparison to false
   2'b0x:... //  you want 2'b0x to match 00 and 01 but
             //  HDL Compiler always evaluates this comparison to false
   default: ...
endcase
```

In the following example, because `if (A == 1'bx)` is evaluated to false, the tool assigns 1 to reg B and issues an ELAB-310 warning.

```
module test (
   input A,
   output reg 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 or use the `always @*` construct.

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.

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 `reg` 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 1364-2005*.

3

Modeling Combinational Logic

These topics describe how to model combinational logic using HDL operators, MUX_OP cells, and other Verilog constructs.

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 2.

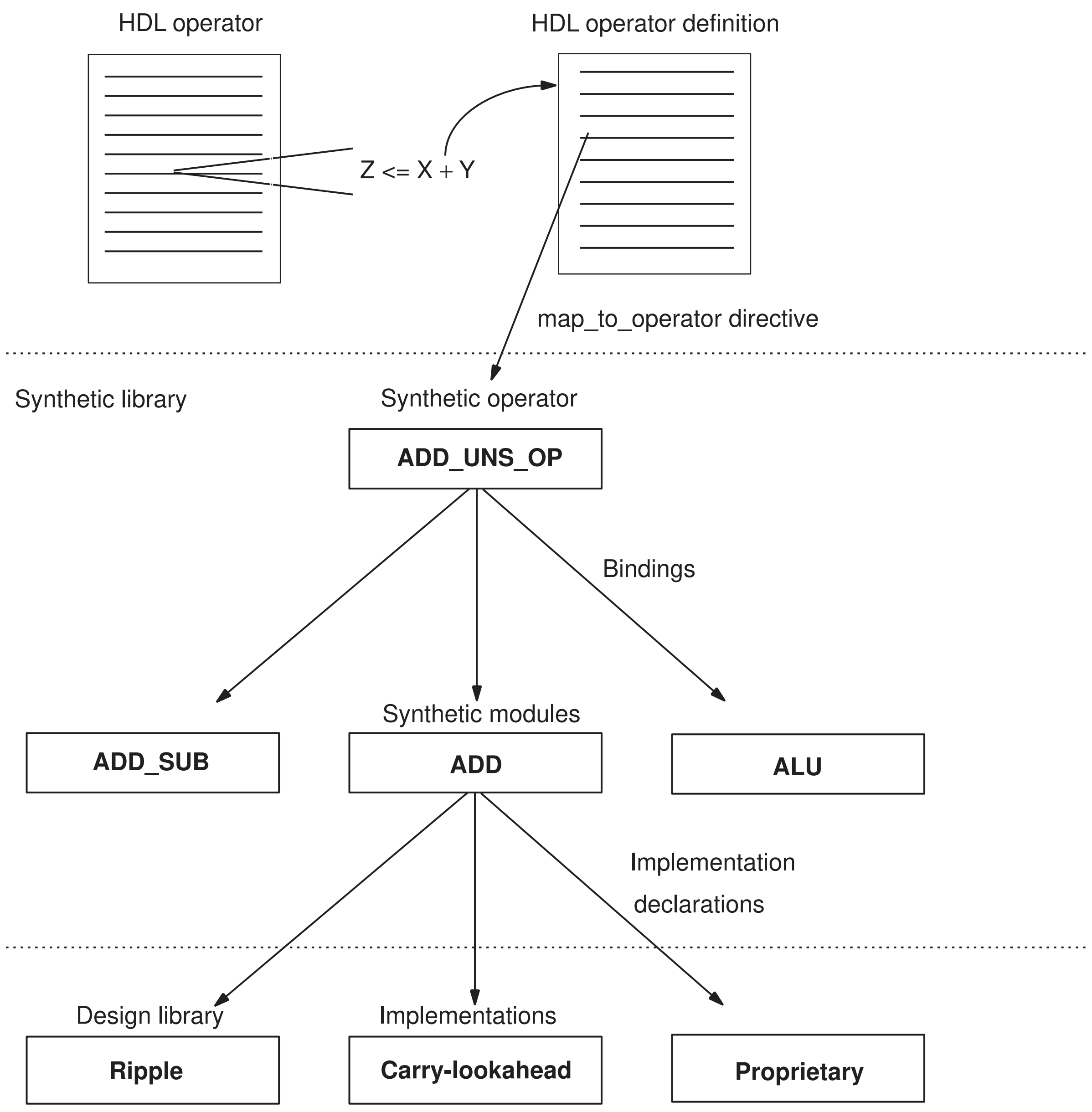
A synthetic library contains definitions for synthetic operators, synthetic modules, and bindings. It also contains declarations that associate synthetic modules with their implementations. To display information about the standard synthetic library that is included with the HDL Compiler license, use 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 2 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 HDL Compiler tool maps HDL operators, such as comparison (> or <), addition (+), decrement (-), and multiplication (*), to synthetic operators from the Synopsys standard synthetic library, standard.sldb. Table 5 shows the complete list of the standard synthetic operators. For more information, see the DesignWare Library documentation.

Table 5 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 5 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.

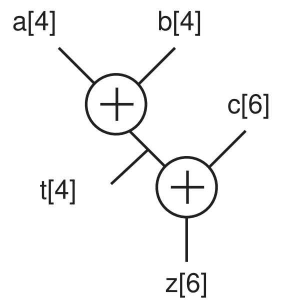
4-Bit Temporary Variable

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 17, 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 3 shows how the HDL Compiler tool determines the default structure.

Example 17 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
```

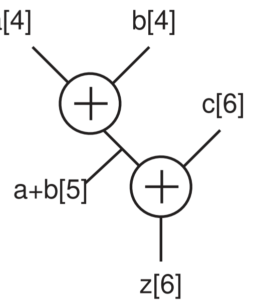
Figure 3 Default Structure for a 4-Bit Temporary Variable



5-Bit Intermediate Result

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

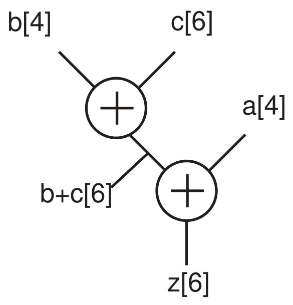
Figure 4 Structure for a 5-Bit Intermediate Result



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 5 shows the structure for this example.

Figure 5 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 *IEEE Std 1364-2005*.

```
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
```

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:

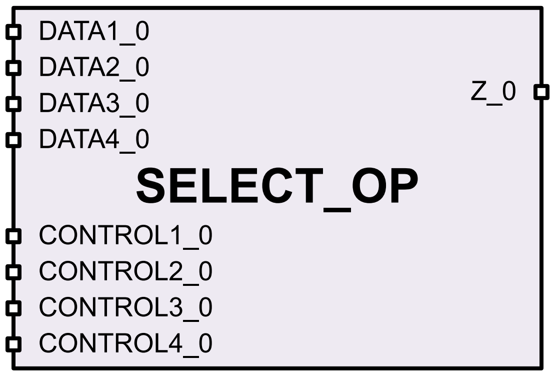
- The SELECT_OP Cell
- The MUX_OP Cell
- Default SELECT_OP and MUX_OP Inference Behavior
- Controlling Selection Statement Inference
- Controlling Array Read Inference
- Inferring One-Hot Multiplexer Logic

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.

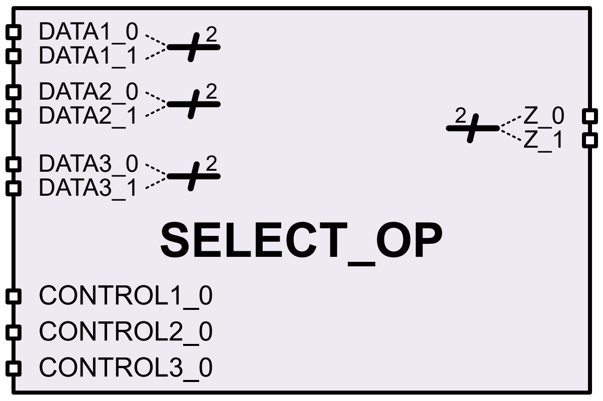
Figure 6 shows a SELECT_OP cell that selects one of four data input bits.

Figure 6 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 7 shows a SELECT_OP cell that can select one of three two-bit-wide data inputs.

Figure 7 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 18 shows an example RTL statement that selects from three data signals using an if/else statement.

Example 18 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 8 shows the elaborated result. The SELECT_OP selection signals are driven by GTECH logic gates that implement the if/else conditions.

Figure 8 Elaborated GTECH Logic With Inferred SELECT_OP Cell

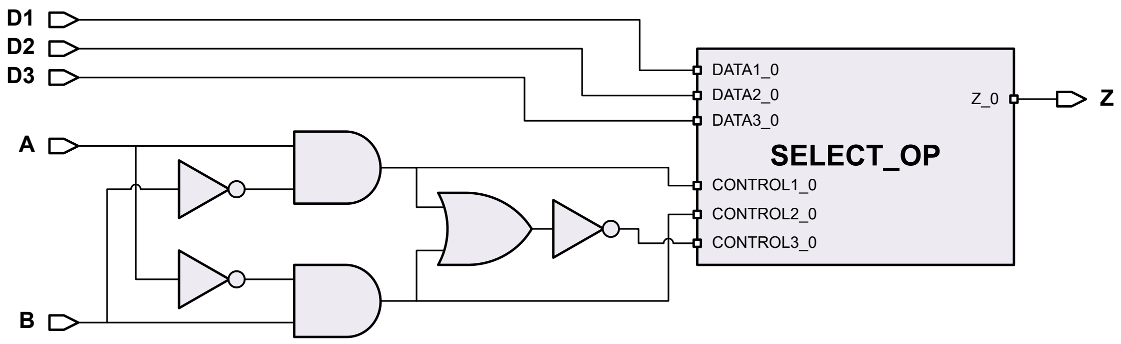
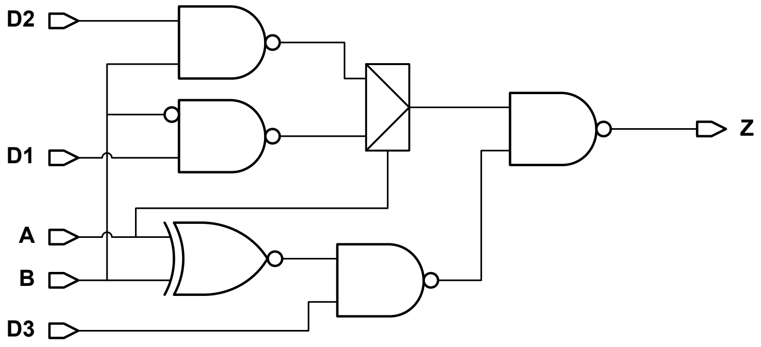


Figure 9 shows the compiled, mapped logic for the previous example. The control logic and the SELECT_OP cell are mapped together into an optimal gate structure.

Figure 9 SELECT_OP Cell and Selection Logic Mapped to Target Library

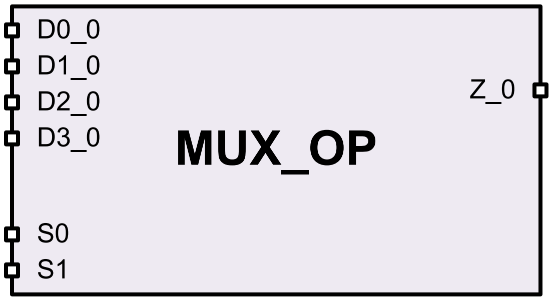


The MUX_OP Cell

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.

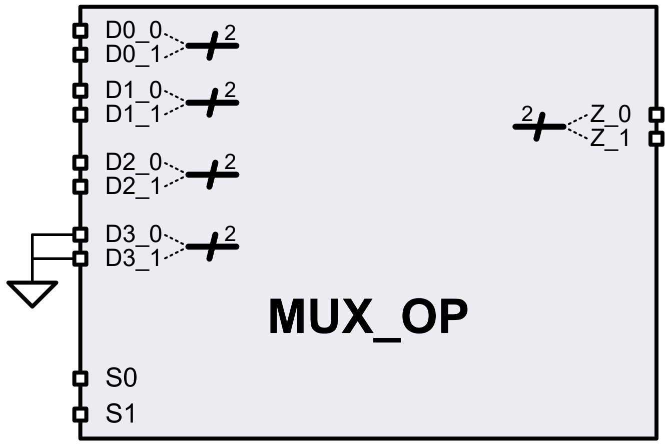
Figure 10 shows a MUX_OP that selects one of four data input bits.

Figure 10 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 11 shows a MUX_OP cell that can select one of three two-bit-wide data inputs.

Figure 11 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 19 shows an example RTL statement that selects from seven data signals using an array read operation.

Example 19 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 12 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 12 Elaborated Inferred MUX_OP Cell

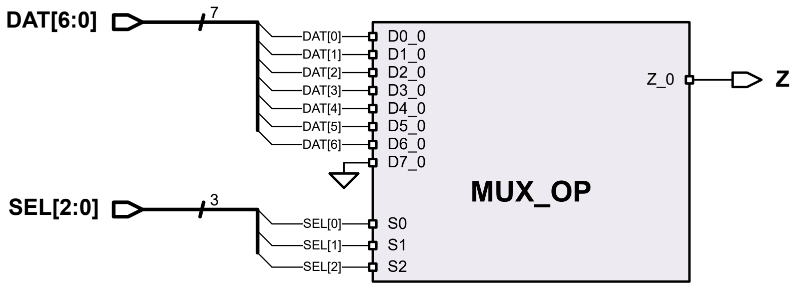
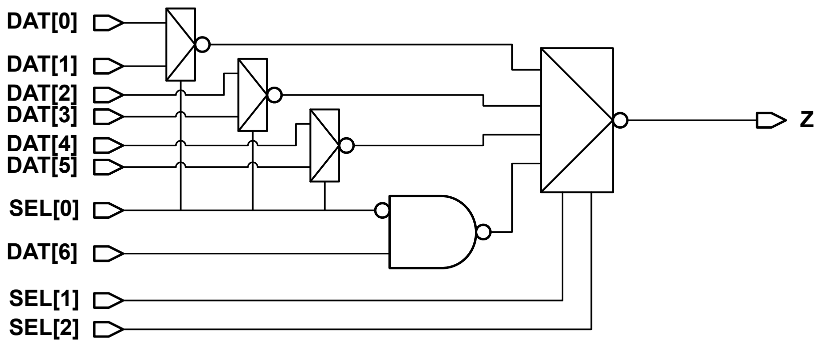


Figure 13 shows the compiled, mapped logic for the previous example. The MUX_OP is implemented using a compact inverting multiplexer tree structure, along with a logic gate that results from optimizing the unused MUX_OP input data bit.

Figure 13 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 6 shows the default inference behavior (when no RTL pragmas are applied).

Table 6 Default SELECT_OP and MUX_OP Inference Behavior

RTL Operator	Default inference behavior
`if` statement	SELECT_OP

Table 6 Default SELECT_OP and MUX_OP Inference Behavior (Continued)

RTL Operator	**Default inference behavior**
`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

The `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 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 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.

The following table shows the valid values and resulting inference behaviors that apply.

Table 8 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 `case` Verilog statement together with the `full_case` and `parallel_case` pragmas, which indicate 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 21) or individually as the case item expressions (Example 20).

Example 20 One-Hot Multiplexer Using One-Hot Case Selection Signals

```
module onehot_1 (in1, in2, in3, sel1, sel2, sel3, out);
input in1, in2, in3;
input sel1, sel2, sel3;
output reg out;

always @*
begin
 case({sel3, sel2, sel1}) //synopsys full_case parallel_case infer_onehot_mux
  3'b001:   out = in1;
```

```
    3'b010:   out = in2;
    3'b100:   out = in3;
    default:  out = 1'bX;
  endcase
 end
 endmodule
```

Example 21   One-Hot Multiplexer Using One-Hot Case Item Signals

```
module onehot_2 (in1, in2, in3, sel1, sel2, sel3, out);
input in1, in2, in3;
input sel1, sel2, sel3;
output reg out;

);
always @*
begin
 case (1'b1) //synopsys full_case parallel_case 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.

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

- The operands have upper bits truncated. For example, if d is 16-bit, d[7:0] truncates the upper eight bits.
- The width of the resulting expression is greater than the width of the truncated operand. In the following example, if e is 9-bit, the width of e is greater than the width of the truncated operand d[7:0]:

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

For lower-bit truncations, the datapath is extracted in all cases. As described in 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];`

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 22 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 22 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 23 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 23 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 24 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 24 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 25 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 25 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
```

Latches in Combinational Logic

Sometimes your Verilog source can imply combinational feedback paths or latches in synthesized logic. This happens when a signal or a variable in a combinational logic block (an always block without a posedge or negedge clock statement) is not fully specified. A variable or signal is fully specified when it is assigned under all possible conditions.

When a variable is not assigned a value for all paths through an always block, the variable is conditionally assigned and a latch is inferred for the variable to store its previous value. To avoid these latches, make sure that the variable is fully assigned in all paths. In Example 26, the variable Q is not assigned if GATE equals 1’b0. Therefore, it is conditionally assigned and Design Compiler creates a latch to hold its previous value.

Example 26   Latch Inference Using an if Statement

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

Example 27 and Example 28 show Q fully assigned—Q is assigned 0 when GATE equals 1’b0. Note that Example 27 and Example 28 are not equivalent to Example 26, in which Q holds its previous value when GATE equals 1’b0.

Example 27   Avoiding Latch Inference—Method 1

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

Example 28   Avoiding Latch Inference—Method 2

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

The code in Example 29 results in a latch because the variable is not fully assigned. To avoid the latch inference, add the following statement before the endcase statement:

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

Example 29   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
```

Latches are also synthesized whenever a for loop statement does not assign a variable for all possible executions of the for loop and when a variable assigned inside the for loop is not assigned a value before entering the enclosing for loop.

4

Sequential Logic

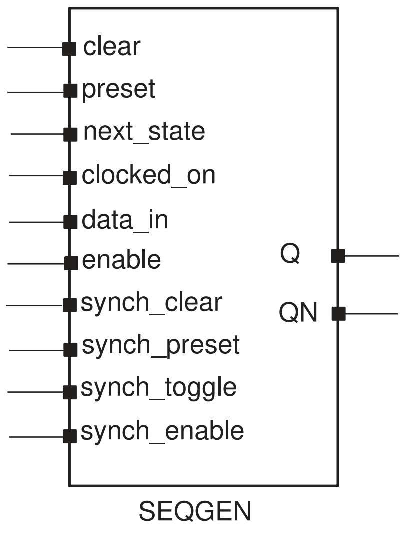
The term register refers to a 1-bit memory device, either a flip-flop or latch. A flip-flop is an edge-triggered memory device, while a latch is a level-sensitive memory device. The following topics describe flip-flop and latch inference:

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

Generic Sequential Cell SEQGEN

When the HDL Compiler tool reads a design, it uses a generic sequential cell SEQGEN shown in Figure 14 to represent an inferred flip-flop or latch.

Figure 14 SEQGEN Cell and Pin Assignments



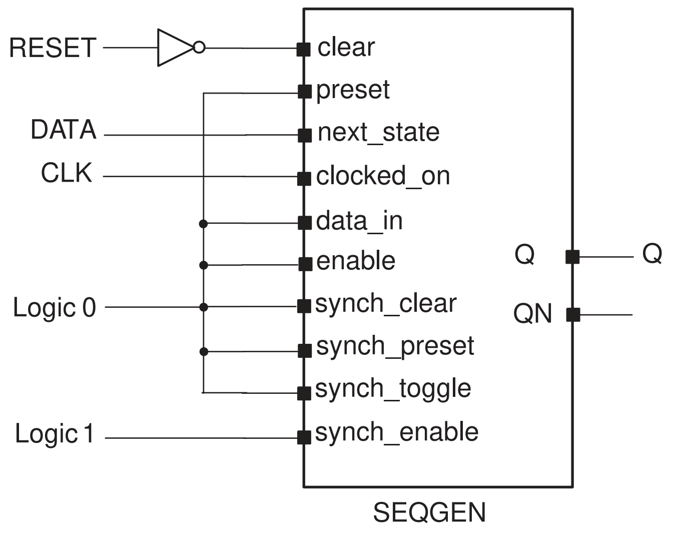
Example 30 shows how to direct the HDL Compiler tool to use a SEQGEN cell to implement a D flip-flop with an asynchronous reset.

Example 30 D Flip-Flop With Asynchronous Reset

```
module dff_async_set (DATA, CLK, RESET, Q);
  input DATA, CLK, RESET;
  output Q;
  reg Q;
  always @(posedge CLK or negedge RESET)
    if (~RESET)
      Q <= 1'b1;
    else
      Q <= DATA;
endmodule
```

Figure 15 shows the SEQGEN implementation.

Figure 15 SEQGEN Implementation



Example 31 shows the `report_cell` output, where the inferred Q_reg flip-flop is mapped to a SEQGEN cell.

Example 31 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 32 shows the GTECH netlist.

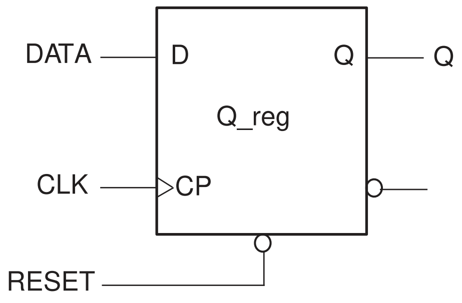
Example 32   GTECH Netlist

```
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 16 shows an example of an implementation after compile.

Figure 16   HDL Compiler 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

HDL Compiler provides inference reports that describe each inferred flip-flop or latch. You can enable or disable the generation of inference reports by using the `hdlin_reporting_level` variable . By default, the level is set to `basic`. When the level is set to `basic` or `comprehensive`, HDL Compiler generates a report similar to Example 33. This basic inference report shows only which type of register was inferred.

Example 33 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 34 shows a verbose inference report.

Example 34 Verbose Inference Report for a D Flip-Flop With Asynchronous Reset

```
===========================================================================
| Register Name |    Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===========================================================================
|     Q_reg     | Flip-flop |   1   |  N  | N  | Y  | N  | N  | N  | N  |
===========================================================================
Sequential Cell (Q_reg)
      Cell Type: Flip-Flop
      Multibit Attribute: N
      Clock: CLK
      Async Clear: RESET
      Async Set: 0
      Async Load: 0
      Sync Clear: 0
      Sync Set: 0
      Sync Toggle: 0
      Sync Load: 1
```

If you do not want the inference report, set the `hdlin_reporting_level` variable to `none`.

See Also

- Reporting Elaboration Errors in the Hierarchy

Register Inference Guidelines

When inferring registers, restrict each always block so that it infers a single type of memory element and check the inference report to verify that HDL Compiler inferred the correct device.

Register inference guidelines are described in the following sections:

- Multiple Events in an always Block
- Minimizing Registers
- Keeping Unloaded Registers
- Preventing Unwanted Latches
- Register Inference Limitations

Multiple Events in an always Block

HDL Compiler supports multiple events in a single `always` block, as shown in Example 35.

Example 35 Multiple Events in a Single always Block

```
module test (
   input [7:0]data,
   input clk,
   output reg [7:0]sum
);
always
begin
   @ (posedge clk)
```

Feedback

```
          sum <= data;
      @ (posedge clk)
          sum <= sum + data;
      @ (posedge clk)
          sum <= sum + data;
end
endmodule
```

Minimizing Registers

An `always` 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 36 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 36   Inefficient Circuit Description With Six Inferred Registers

```
   input clock, reset,
   output reg and_bits, or_bits, xor_bits
);
reg [2:0] count;

always @(posedge clock) begin
   if (reset)
      count <= 0;
   else
      count <= count + 1;
      and_bits <= & count;
      or_bits  <= | count;
      xor_bits <= ^ count;
   end
endmodule
```

Example 37 shows the inference report which contains the six inferred flip-flops.

Example 37   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 38 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 38 creates a more area-efficient design.

Example 38   Circuit With Three Inferred Registers

```
module count (
   input clock, reset,
   output reg and_bits, or_bits, xor_bits
);
reg [2:0] count;

always @(posedge clock)
begin //synchronous block
   if (reset)
      count <= 0;
   else
      count <= count + 1;
end

always @(count)
begin //asynchronous block
   and_bits = & count;
   or_bits  = | count;
   xor_bits = ^ count;
end
endmodule
```

Example 39 shows the inference report, which contains three inferred flip-flops.

Example 39   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 40 and Example 41.

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 40 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 41.

Example 40 Retains an Unloaded Cell (sum2) and Two Adders

```
module mydesign (in1, in2, in3, out, clk);
   input clk,
   input [0:1] in1, in2, in3,
   output [0:3] out
);
reg sum1, sum2 /*  synopsys preserve_sequential */;
wire [0:4] save;
always @ (posedge clk)
begin
   sum1 <= in1 + in2;
   sum2 <= in1 + in2 + in3; // this combinational logic is saved
end
assign out = ~sum1;
assign save = sum1 + sum2; // this combinational logic is not saved
                           // because it is after the saved reg, sum2
endmodule
```

Example 41 preserves all combinational logic before reg save.

Example 41   Retains an Unloaded Cell and Three Adders

```
module mydesign (
   input clk,
   input [0:1] in1, in2, in3,
   output [0:3] out
);
reg sum1, sum2, save /* synopsys preserve_sequential */;
always @ (posedge clk)
begin
   sum1 <= in1 + in2;
   sum2 <= in1 + in2 + in3; // this combinational logic is saved
end
assign out = ~sum1;
always @ (posedge clk)
begin
   save <= sum1 + sum2; // this combinational logic is saved
end
endmodule
```

The `preserve_sequential` directive and the `hdlin_preserve_sequential` variable enable you to preserve cells that are inferred but optimized away by HDL Compiler. If a cell is never inferred, the `preserve_sequential` directive and the `hdlin_preserve_sequential` variable have no effect because there is no inferred cell to act on. In Example 42, sum2 is not inferred, so `preserve_sequential` does not save sum2.

Example 42   preserve_sequential Has No Effect on Cells Not Inferred

```
module mydesign (
   input clk,
   input [0:1] in1, in2,
   output [0:3] out
);
reg sum1, sum2 /*  synopsys preserve_sequential */;
wire [0:4] save;
always @ (posedge clk)
begin
   sum1 <= in1 + in2;
end
assign out = ~sum1;
assign save = sum2; // Although the preserve_sequential directive is on
                    // sum2, it is not saved due to sum2 is not inferred
endmodule
```

Note:

By default, the `hdlin_preserve_sequential` variable does not preserve variables used in for loops as unloaded registers. To preserve such variables, you must set `hdlin_preserve_sequential` to `ff+loop_variables`.

In addition to preserving sequential cells with the `hdlin_preserve_sequential` variable and the `preserve_sequential` directive, you can also use the `hdlin_keep_signal_name` variable and the `keep_signal_name` directive. For more information, see Keeping Signal Names.

Note:

The tool does not distinguish between unloaded cells (those not connected to any output ports) and feedthroughs. See Example 43 for a feedthrough.

Example 43   Feedthrough Example

```
module test (
   input clk,
   input in,
   output reg out
);
reg tmp1;
always@(posedge clk)
begin : storage
   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 44   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). If you do not want to infer latches, set the `hdlin_check_no_latch` variable to `true`, which causes the tool to issue ELAB-395 warning messages for latch inference.

As shown in Example 45, one branch of the `case` statement is commented out, so output DOUT is not fully specified and the tool infers a latch.

Example 45

```
module selector (SEL, DIN, DOUT);
input [1:0] SEL;
input [3:0] DIN;
output reg DOUT;

always @*
case (SEL)
  2'b00: DOUT = DIN[0];
  2'b01: DOUT = DIN[1];
  2'b10: DOUT = DIN[2];
//  2'b11: DOUT = DIN[3];
endcase
endmodule
```

Register Inference Limitations

Note the following limitations when inferring registers:

- The tool does not support more than one independent if-block when asynchronous behavior is modeled within an always block. If the always block is purely synchronous, multiple independent if-blocks are supported by the tool.
- The HDL Compiler tool cannot infer flip-flops and latches with three-state outputs. You must instantiate these components in your Verilog description.
- The HDL Compiler tool cannot infer flip-flops with bidirectional pins. You must instantiate these components in the RTL.
- The HDL Compiler tool cannot infer flip-flops with multiple clock inputs. You must instantiate these components in the RTL.
- The HDL Compiler tool cannot infer multiport latches. You must instantiate these components in the RTL.
- The HDL Compiler tool cannot infer register banks (register files). You must instantiate these components in the RTL.
- Although you can instantiate flip-flops with bidirectional pins, the HDL Compiler tool interprets these cells as black boxes.

- If you use an `if` statement to infer D flip-flops, the `if` statement must occur at the top level of the `always` block.

  The following example is invalid because the `if` statement does not occur at the top level:

  ```
  always @(posedge clk or posedge reset) begin
   temp = reset;
   if (reset)
   ...
   end
  ```

  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

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 46 shows that a D latch is inferred for the `always@` construct.

Example 46   D Latch Code

```
module d_latch (
   input GATE, DATA,
   output reg Q
);
always @(GATE or DATA)
if (GATE)
   Q <= DATA;
endmodule
```

The HDL Compiler tool generates the inference report shown in Example 47.

Example 47   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 48 shows the recommended coding style for an asynchronously set latch using the `async_set_reset` directive.

Example 48   D Latch With Asynchronous Set: Uses async_set_reset

```
module d_latch_async_set (
   input GATE, DATA, SET,
   output reg Q
);

// synopsys async_set_reset "SET"
always @(GATE or DATA or SET)
if (~SET)
   Q = 1'b1;
else if (GATE)
   Q = DATA;
endmodule
```

The tool generates the inference report shown in Example 49.

Example 49   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 50 shows the recommended coding style for an asynchronously reset latch using the `async_set_reset` directive.

Example 50 D Latch With Asynchronous Reset: Uses async_set_reset

```
module d_latch_async_reset (
   input RESET, GATE, DATA,
   output reg Q
);
//synopsys async_set_reset "RESET"
always @ (RESET or GATE or DATA)
   if (~RESET) Q <= 1'b0;
   else if (GATE) Q <= DATA;
endmodule
```

The tool generates the inference report shown in Example 51.

Example 51 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 52.

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 52 D Latch With Asynchronous Set and Reset: Uses hdlin_latch_always_async_set_reset

```
// Set hdlin_latch_always_async_set_reset to true.
module d_latch_async (
   input GATE, DATA, RESET, SET,
   output reg Q
);
// synopsys one_cold "RESET, SET"
always @ (GATE or DATA or RESET or SET)
begin : infer
   if (!SET) Q <= 1'b1;
   else if (!RESET) Q <= 1'b0;
   else if (GATE) Q <= DATA;
end
endmodule
```

Example 53 shows the inference report.

Example 53 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  | -  | -  | -  |
===========================================================================
```

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 @ (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 17, or you can synthesize a flip-flop with built-in reset functionality, as shown in Figure 18.

Figure 17 Flip-Flop With External Combinational Logic to Represent Reset

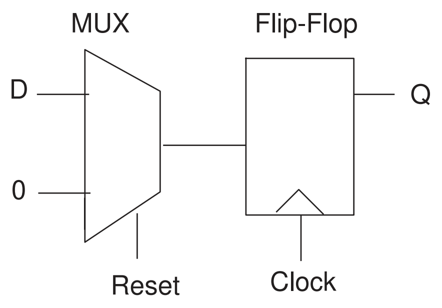
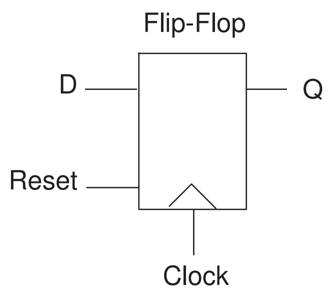


Figure 18 Flip-Flop With Built-In Reset Functionality



The intended implementation is not apparent from the RTL code. You should specify HDL Compiler synthesis directives or HDL Compiler variables to guide the tool to create the proper synchronous set and reset signals.

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

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 54 infers a basic D flip-flop.

Example 54 Basic D Flip-Flop

```
module dff_pos (DATA, CLK, Q);
  input DATA, CLK;
  output Q;
```

```
    reg Q;
    always @(posedge CLK)
      Q <= DATA;
  endmodule
```

HDL Compiler generates the inference report shown in Example 55.

Example 55 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 56 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 56 D Flip-Flop With Asynchronous Reset Using ?: Construct

```
module test(input clk, rst, din, output reg dout);
  always@(posedge clk or negedge rst)
  dout <= (!rst) ? 1'b0 : din;
endmodule
```

HDL Compiler generates the inference report shown in Example 57.

Example 57 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 58 infers a D flip-flop with an asynchronous reset.

Example 58 D Flip-Flop With Asynchronous Reset

```
module dff_async_reset (DATA, CLK, RESET, Q);
 input DATA, CLK, RESET;
  output Q;
  reg Q;
  always @(posedge CLK or posedge RESET)
  if (RESET)
    Q <= 1'b0;
  else
```

```
        Q <= DATA;
endmodule
```

HDL Compiler generates the inference report shown in Example 59.

Example 59   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 60 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 Macros).

Example 60   D Flip-Flop With Asynchronous Set and Reset

```
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
```

Example 61 shows the inference report.

Example 61   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 @(posedge CLK)
if (SET) Q <= 1'b1;
else     <= DATA;
endmodule
```

Inference Report

```
module dff_sync_set (
   input DATA, CLK, SET;
   output reg Q
);
//synopsys sync_set_reset "SET"
always @(posedge CLK)
   if (SET) Q <= 1'b1;
   else Q <= DATA;
endmodule
```

D Flip-Flop With Synchronous Reset: Use sync_set_reset

Example 62 infers a D flip-flop with synchronous reset. The `sync_set_reset` directive is applied to the RESET signal.

Example 62   D Flip-Flop With Synchronous Reset: Use sync_set_reset

```
module dff_sync_reset (
  input DATA, CLK, RESET,
  output reg Q
);
  //synopsys sync_set_reset "RESET"
  always @(posedge CLK)
    if (~RESET)
```

```
        Q <= 1'b0;
      else
        Q <= DATA;
endmodule
```

HDL Compiler generates the inference report shown in Example 63.

Example 63   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 64 to infer a D flip-flop with both synchronous and asynchronous load signals.

Example 64   Synchronous and Asynchronous Loads

```
module dff_a_s_load (ALOAD, SLOAD, ADATA, SDATA, CLK, Q);
  input ALOAD, ADATA, SLOAD, SDATA, CLK;
  output Q;
  reg Q;
  wire asyn_rst, asyn_set;

  assign asyn_rst = ALOAD && !ADATA;
  assign asyn_set = ALOAD && ADATA;

//synopsys one_cold "ALOAD, ADATA"

  always @ (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
```

HDL Compiler generates the inference report shown in Example 65.

Example 65   D Flip-Flop With Synchronous and Asynchronous Load Inference Report

```
==============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST
 |
==============================================================================
|       Q_reg         | Flip-flop |   1   |  N  | N  | Y  | Y  | N  | N  | N
 |
==============================================================================
Sequential Cell (Q_reg)
```

```
Cell Type: Flip-Flop
Multibit Attribute: N
Clock: CLK
Async Clear: ADATA' ALOAD
Async Set: ADATA ALOAD
Async Load: 0
Sync Clear: 0
Sync Set: 0
Sync Toggle: 0
Sync Load: SLOAD
```

D Flip-Flops With Complex Set and Reset Signals

While many set and reset signals are simple signals, some include complex logic. To enable HDL Compiler to generate a clean set/reset (that is, a set/reset signal attached only to the appropriate set/reset pins), use the following coding guidelines:

- Apply the appropriate set/reset compiler directive (`//synopsys sync_set_reset` or `//synopsys async_set_reset`) to the set/reset signal.
- Use no more than two operands in the set/reset logic expression conditional.
- Use the set/reset signal as the first operand in the set/reset logic expression conditional.

This coding style supports usage of the negation operator on the set/reset signal and the logic expression. The logic expression can be a simple expression or any expression contained inside parentheses. However, any deviation from these coding guidelines is not supported. For example, using a more complex expression other than the OR of two expressions, or using a rst (or ~rst) that does not appear as the first argument in the expression is not supported.

Examples

```
//synopsys sync_set_reset "rst"
always @(posedge clk)
if (rst | logic_expression)
   q <= 0;
else ...
else ...
...

//synopsys sync_set_reset "rst"
assign a = rst |  ~( a | b & c);
always @(posedge clk)
if (a)
   q <= 0;
else ...;
else ...;
...

//synopsys sync_set_reset "rst"
always @(posedge clk)
```

```
if ( ~ rst |  ~ (a | b | c))
   q <= 0;
else ...
else ...
...

//synopsys sync_set_reset "rst"
assign a =  ~ rst |  ~ logic_expression;
always @(posedge clk)
if (a)
   q <= 0;
else ...;
else ...;
...
```

Multiple Flip-Flops With Asynchronous and Synchronous Controls

In Example 66, 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 66   Multiple Flip-Flops With Asynchronous and Synchronous Controls

```
module multi_attr (DATA1, DATA2, CLK, RESET, SLOAD, Q1, Q2);
  input DATA1, DATA2, CLK, RESET, SLOAD;
  output Q1, Q2;
  reg Q1, Q2;

  //synopsys sync_set_reset "RESET"
  always @(posedge CLK)
  begin : infer_sync
    if (~RESET)
      Q1 <= 1'b0;
    else if (SLOAD)
      Q1 <= DATA1;    // note: else hold Q1
  end
  always @(posedge CLK or negedge RESET)
  begin: infer_async
    if (~RESET)
      Q2 <= 1'b0;
    else if (SLOAD)
      Q2 <= DATA2;
  end
endmodule
```

Example 67 shows the inference report.

Example 67   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
 |
=============================================================================
```

Feedback

5

Modeling Three-State Buffers

HDL Compiler infers a three-state driver when you assign the value z (high impedance) to a variable. HDL Compiler infers 1 three-state driver per variable per always block. You can assign high-impedance values to single-bit or bused variables. A three-state driver is represented as a TSGEN cell in the generic netlist. Three-state driver inference and instantiation are described in the following sections:

- Using z Values
- Three-State Driver Inference Report
- Assigning a Single Three-State Driver to a Single Variable
- Assigning Multiple Three-State Drivers to a Single Variable
- Registering Three-State Driver Data
- Instantiating Three-State Drivers
- Errors and Warnings

Using z Values

You can use the z value in the following ways:

- Variable assignment
- Function call argument
- Return value

You can use the z value only in a comparison expression, such as in

```
if (IN_VAL == 1'bz) y=0;
```

This statement is permissible because `IN_VAL == 1'bz` is a comparison. However, it always evaluates to false, so it is also a simulation/synthesis mismatch. See Unknowns and High Impedance in Comparison.

This code,

```
OUT_VAL = (1'bz && IN_VAL);
```

is not a comparison expression. HDL Compiler generates an error for this expression.

Three-State Driver Inference Report

The `hdlin_reporting_level` variable determines whether HDL Compiler generates a three-state inference report. If you do not want inference reports, set the level to `none`. The default is `basic`, which indicates to generate a report. Example 68 shows a three-state inference report:

Example 68   Three-State Inference Report

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

The first column of the report indicates the name of the inferred three-state device. The second column indicates the type of inferred device. The third column indicates the width of the inferred device. HDL Compiler generates the same report for the default and verbose reports for three-state inference. For more information about the `hdlin_reporting_level` variable to `basic+fsm`, see Customizing Elaboration Reports.

Assigning a Single Three-State Driver to a Single Variable

Example 69 infers a single three-state driver and shows the associated inference report.

Example 69   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 70   Inference Report

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

Example 71 infers a single three-state driver with MUXed inputs and shows the associated inference report.

Example 71 Single Three-State Driver With MUXed Inputs

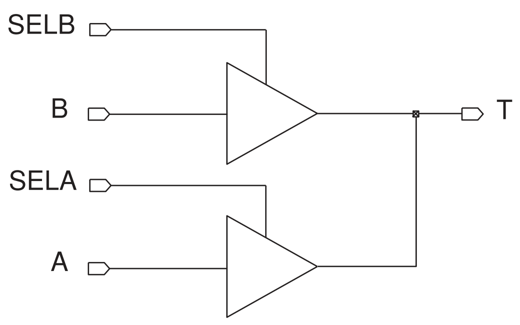
```
module three_state (A, B, SELA, SELB, T);
  input  A, B, SELA, SELB;
  output T;
  reg T;
  always @(SELA or SELB or A or B) begin
    T = 1'bz;
    if (SELA)
      T = A;
      if (SELB)
      T = B;
    end
endmodule


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 19, always use assign statements, as shown in Example 72.

Figure 19 Two Three-State Drivers Assigned to a Single Variable



Example 72 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 73). 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 73 Incorrect Method

```
module three_state (A, B, SELA, SELB, T);
  input  A, B, SELA, SELB;
  output T;
  reg T;
  always @(SELA or A)
    if (SELA)
      T = A;
    else
      T = 1'bz;
  always @(SELB or B)
    if (SELB)
      T = B;
    else
      T = 1'bz;
endmodule
```

Registering Three-State Driver Data

When a variable is registered in the same block in which it is defined as a three-state driver, HDL Compiler also registers the driver's enable signal, as shown in Example 74. Figure 20 shows the compiled gates and the associated inference report.

Example 74 Three-State Driver With Enable and Data Registered

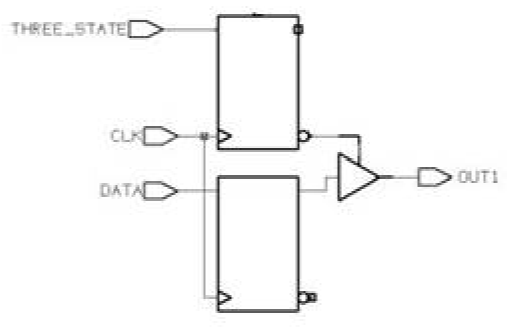
```
module ff_3state (DATA, CLK, THREE_STATE, OUT1);
  input DATA, CLK, THREE_STATE;
  output OUT1;
  reg OUT1;
always @ (posedge CLK) begin
  if (THREE_STATE)
    OUT1 <= 1'bz;
  else
    OUT1 <= DATA;
end
endmodule
```

Example 75 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 20   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 76 shows a three-state gate instantiation with an active-high enable and no inverted output.

Example 76   Three-State Gate Instantiation

```
module three_state (in1,out1,cntrl1);
 input in1,cntrl1;
 output out1;

 bufif1 (out1,in1,cntrl1);
endmodule
```

Errors and Warnings

When you use the coding styles recommended in this chapter, you do not need to declare variables that drive multiply driven nets as tri data objects. But if you don't use these coding styles, or you don't declare the variable as a tri data object, HDL Compiler issues an ELAB-366 error message and terminates. To force HDL Compiler to warn for this condition (ELAB-365) but continue to create a netlist, set the `hdlin_prohibit_nontri_multiple_drivers` variable to false (the default is true). With this variable false, HDL Compiler builds the generic netlist for all legal designs. If a design is illegal, such as when one of the drivers is a constant, HDL Compiler issues an error message.

The following code generates an ELAB-366 error message (OUT1 is a reg being driven by two always@ blocks):

```
 module three_state (ENABLE, IN1, RESET, OUT1);

 input IN1, ENABLE, RESET;
 output OUT1;
 reg OUT1;

always @(IN1 or ENABLE)
     if (ENABLE)
     OUT1 = IN1;

always@ (RESET)
    if (RESET)
    OUT1 = 1'b0;
endmodule
```

The ELAB-366 error message is

```
Error:  Net '/...v:14: OUT1' or a directly connected net is
driven by more than one source, and not all drivers are
three-state. (ELAB-366)
```

Feedback

6 HDL Compiler Synthesis Directives

HDL Compiler synthesis directives are special comments that affect how synthesis processes the RTL. 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 124 for details.

The following sections describe the HDL Compiler synthesis directives:

- 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
- translate_off and translate_on (Deprecated)
- Directive Support by Pragma Prefix

async_set_reset

When you set the `async_set_reset` directive on a single-bit signal, HDL Compiler searches for a branch that uses the signal as a condition and then checks whether the branch contains an assignment to a constant value. If the branch does, the signal becomes an asynchronous reset or set. Use this directive on single-bit signals.

The syntax is

```
// synopsys async_set_reset "signal_name_list"
```

See Also

- Inferring Latches

async_set_reset_local

When you set the `async_set_reset_local` directive, HDL Compiler treats listed signals in the specified block as if they have the `async_set_reset` directive set. Attach the `async_set_reset_local` directive to a block label using the following syntax:

```
// synopsys async_set_reset_local block_label "signal_name_list"
```

async_set_reset_local_all

When you set the `async_set_reset_local_all` directive, HDL Compiler treats all listed signals in the specified blocks as if they have the `async_set_reset` directive set. Attach the `async_set_reset_local_all` directive to a block label using the following syntax:

```
// synopsys async_set_reset_local_all "block_label_list"
```

To enable the `async_set_reset_local_all` behavior, you must set `hdlin_ff_always_async_set_reset` to false and use the coding style shown in Example 77.

Example 77 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 78 and Example 79.

Example 78 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 79 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 80 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 80   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 81 shows an invalid `enum` declaration.

Example 81   Invalid enum Declaration

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

Example 82 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 82   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 83 relates to Example 82 and shows an invalid encoding for colors.

Example 83   Invalid Bit Value Encoding for Colors

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

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

You can use enumeration literals just like constants, as shown in Example 84.

Example 84 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 85 shows the declaration of the enumeration.

Example 85 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 86 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 86 Incorrect Enumerated Type Declaration for a Port

```
module bad_example (a,b);
  parameter [1:0] /* synopsys enum colors */
 green = 2'b00, white = 2'b11;
  input a;
  output [1:0] b;
  reg [1:0] /* synopsys enum colors */ b;
...
endmodule
```

full_case

This directive prevents HDL Compiler from generating logic to test for any value that is not covered by the case branches and creating an implicit default branch. Set the `full_case` directive on a case statement when you know that all possible branches of the case statement are listed within the case statement. When a variable is assigned in a case statement that is not full, the variable is conditionally assigned and requires a latch.

Caution:

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

The syntax for the `full_case` directive is

Feedback

```
// synopsys full_case
```

In Example 87, `full_case` is set on the first case statement and `parallel_case` and `full_case` directives are set on the second case statement.

Example 87 // 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 occurs 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, HDL Compiler generates logic to test for any value that is not covered by the case branches and creates an implicit “default” branch that contains no actions. When a variable is assigned in a case statement that is not full, the variable is conditionally assigned and requires a latch.

infer_multibit and dont_infer_multibit

The HDL Compiler tool can infer registers that have identical structures as multibit components.

The following sections describe how to use the multibit inference directives:

- Using the infer_multibit Directive
- Using the dont_infer_multibit Directive
- Reporting Multibit Components

Multibit sequential mapping does not pull in as many levels of logic as single-bit sequential mapping. Therefore, HDL Compiler 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 the 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 88 shows usage.

Example 88   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 89 shows the inference report.

Example 89   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 90 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 91 shows the multibit inference report.

Example 91   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
 |
```

Feedback

```
============================================================================
===
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 92 shows a multibit component report.

Example 92 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 on page 66.

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
```

Feedback

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 71.

keep_signal_name

Use the `keep_signal_name` directive to provide HDL Compiler with guidelines for preserving signal names.

The syntax is

```
// synopsys keep_signal_name "signal_name_list"
```

Set the `keep_signal_name` directive on a signal before any reference is made to that signal; for example, one methodology is to put the directive immediately after the declaration of the signal.

See Also

- Keeping Signal Names

one_cold

A one-cold implementation indicates that all signals in a group are active-low and that only one signal can be active at a given time. Synthesis implements the `one_cold` directive by omitting a priority circuit in front of the flip-flop. Simulation ignores the directive. The `one_cold` directive prevents 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 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 D Flip-Flop With Asynchronous Set and 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 HDL Compiler from building additional logic to ensure the first occurrence of a true branch is executed if more than one branch were true at one time.

Caution:

> Marking a case statement as parallel when it actually is not parallel can cause the simulation to behave differently from the logic HDL Compiler synthesizes because HDL Compiler does not generate priority encoding logic to make sure that the branch listed first in the case statement takes effect.

The syntax for the `parallel_case` directive is

```
// synopsys parallel_case
```

Use the `parallel_case` directive immediately after the case expression. In Example 93, 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 93 parallel_case Directives

```
reg [3:0] current_state, next_state;
parameter state1 = 4'b0001, state2 = 4'b0010,
 state3 = 4'b0100, state4 = 4'b1000;
case (1) //synopsys parallel_case
     current_state[0] : next_state = state2;
     current_state[1] : next_state = state3;
     current_state[2] : next_state = state4;
     current_state[3] : next_state = state1;
endcase
```

When a case statement is not parallel (more than one branch evaluates to true), priority encoding is needed to ensure that the branch listed first in the case statement takes effect.

The following table summarizes the types of case statements.

Case statement description	Additional logic
Full and parallel	No additional logic is generated.
Full but not parallel	Priority-encoded logic: HDL Compiler generates logic to ensure that the branch listed first in the case statement takes effect.
Parallel but not full	Latches created: HDL Compiler generates logic to test for any value that is not covered by the case branches and creates an implicit “default” branch that requires a latch.
Not parallel and not full	Priority-encoded logic: HDL Compiler generates logic to make sure that the branch listed first in the case statement takes effect.Latches created: HDL Compiler generates logic to test for any value that is not covered by the case branches and creates an implicit “default” branch that requires a latch.

preserve_sequential

The `preserve_sequential` directive allows you to preserve specific cells that would otherwise be optimized away by HDL Compiler. See Keeping Unloaded Registers on page 84.

sync_set_reset

Use the `sync_set_reset` directive to infer a D flip-flop with a synchronous set/reset. When you compile your design, the SEQGEN inferred by HDL Compiler is mapped to a flip-flop in the logic library with a synchronous set/reset pin, or HDL Compiler 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 HDL Compiler that the signal should be kept as close to the register as possible during mapping, preventing a simulation/synthesis mismatch which can occur if the set/reset signal is masked by the X during initialization in simulation. When a single-bit signal has this directive set to true, HDL Compiler checks the signal to determine whether it synchronously sets or resets a register in the design. Attach this directive to single-bit signals. Use the following syntax:

```
//synopsys sync_set_reset "signal_name_list"
```

For an example of a D flip-flop with a synchronous set signal that uses the `sync_set_reset` directive, see D Flip-Flop With Synchronous Set: Use sync_set_reset on page 95.

For an example of a D flip-flop with a synchronous reset signal that uses the `sync_set_reset` directive, see D Flip-Flop With Synchronous Reset: Use sync_set_reset.

For an example of multiple flip-flops with asynchronous and synchronous controls, see Multiple Flip-Flops With Asynchronous and Synchronous Controls.

sync_set_reset_local

The `sync_set_reset_local` directive instructs HDL Compiler to treat signals listed in a specified block as if they have the `sync_set_reset` directive set to true. Attach this directive to a block label, using the following syntax:

```
 //synopsys sync_set_reset_local block_label "signal_name_list"
```

Example 94 shows the usage.

Example 94   sync_set_reset_local Usage

```
module m1 (input d1,d2,clk, set1, set2, rst1, rst2, output reg q1,q2);

// synopsys sync_set_reset_local sync_rst "rst1"
//always@(posedge clk or negedge rst1)
  always@(posedge clk )
    begin: sync_rst
      if(~rst1)
        q1 <= 1'b0;
      else if (set1)
        q1 <= 1'b1;
      else
      q1 <= d1;
    end

  always@(posedge clk)
    begin: default_rst
      if(~rst2)
        q2 <= 1'b0;
      else if (set2)
        q2 <= 1'b1;
      else
       q2 <= d2;
    end

endmodule
```

sync_set_reset_local_all

The `sync_set_reset_local_all` directive instructs HDL Compiler to treat all signals listed in the specified blocks as if they have the `sync_set_reset` directive set to true. Attach this directive to a block label, using the following syntax:

```
// synopsys sync_set_reset_local_all "block_label_list"
```

Example 95 shows usage.

Example 95   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 96 shows usage.

Example 96   template Directive

```
module template (a, b, c);
  input a, b, c;
  // synopsys template
  parameter width = 8;
```

```
.
.
.
endmodule
```

For more information, see Parameterized Designs on page 31.

translate_off and translate_on (Deprecated)

The `translate_off` and `translate_on` directives are deprecated. To suspend translation of the source code for synthesis, use the `SYNTHESIS` macro and the appropriate conditional directives (`` `ifdef ``, `` `ifndef ``, `` `else ``, `` `endif ``) rather than `translate_off` and `translate_on`.

The `SYNTHESIS` macro replaces the `DC` macro (`DC` is still supported for backward compatibility). See Predefined Macros on page 35.

Directive Support by Pragma Prefix

Not all pragma prefixes support all directives:

- The `synopsys` prefix is intended for directives specific to HDL Compiler. The tool issues an error message if an unknown directive is encountered.
- The `pragma` and `synthesis` prefixes are intended for industry-standard directives. The tool ignores any unsupported directives to allow for directives intended for other tools. Directives specific to HDL Compiler are not supported.

Table 9 shows how each directive is handled by each pragma prefix.

Table 9 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

Table 9 Directive Support by Pragma Prefix (Continued)

Directive	**// synopsys, // $s**	**// pragma**	**// synthesis**
`full_case` `parallel_case`	Used	Ignored	Ignored
`infer_multibit` `dont_infer_multibit`	Used	Ignored	Ignored
`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

A

Verilog Design Examples

These Verilog examples describe the coding techniques for late-arriving signals and master-slave latch inferences.

- 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 defines the Design Compiler installation location.

Coding for Late-Arriving Signals

The following topics describe coding techniques for late-arriving signals:

- Duplicating Datapaths
- Moving Late-Arriving Signals Close to Output

Note:

These techniques apply to the HDL Compiler output. When this output is constrained and optimized by the HDL Compiler tool, 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.

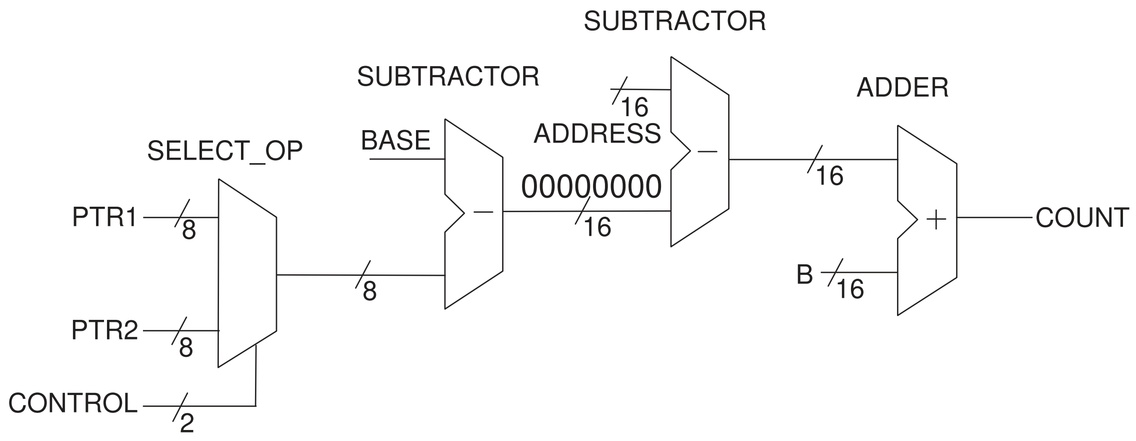
Original RTL

In Example 97, 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 21, 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 98.

Example 97 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 21 Schematic of the Original RTL



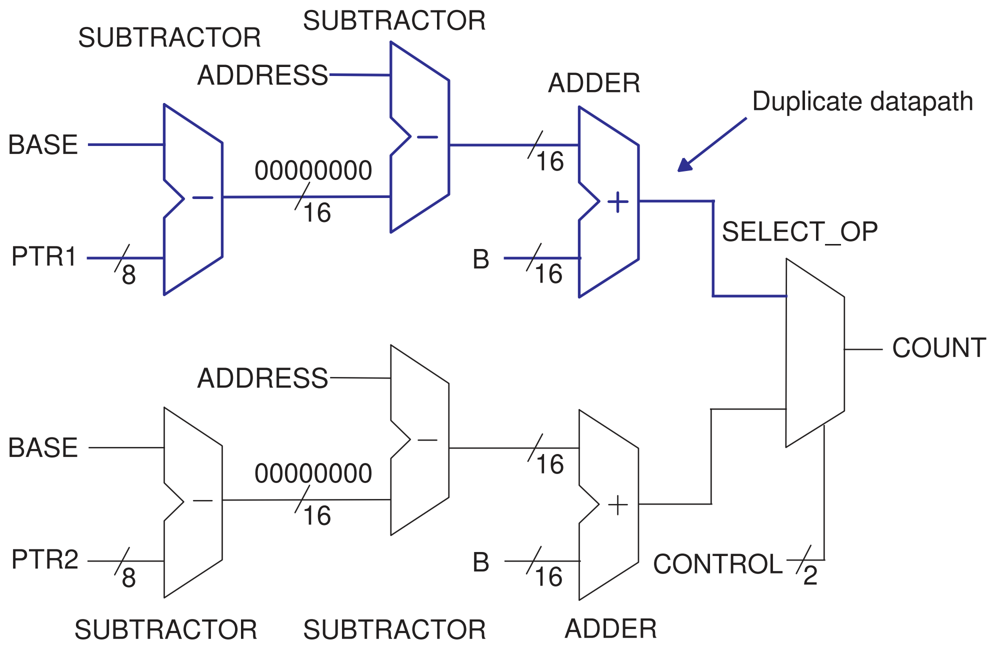
Modified RTL With the Duplicate Datapath

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 98 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 22 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.

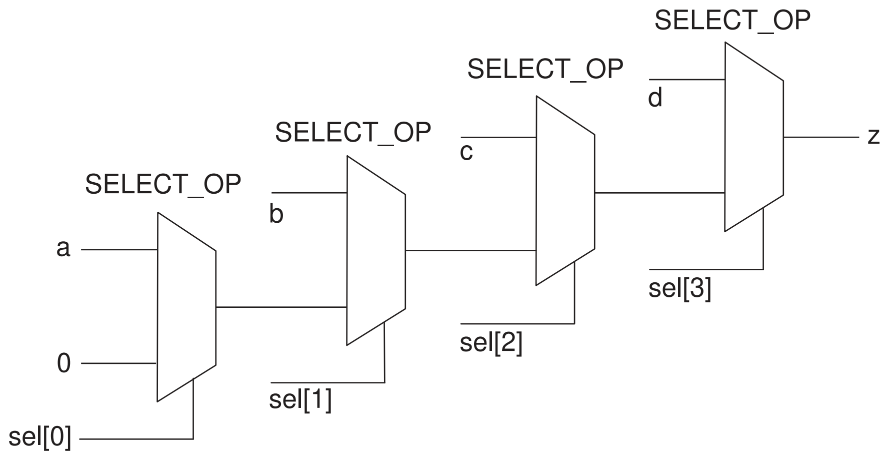
RTL With Sequential if Statements

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 99 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 23 Schematic of the RTL



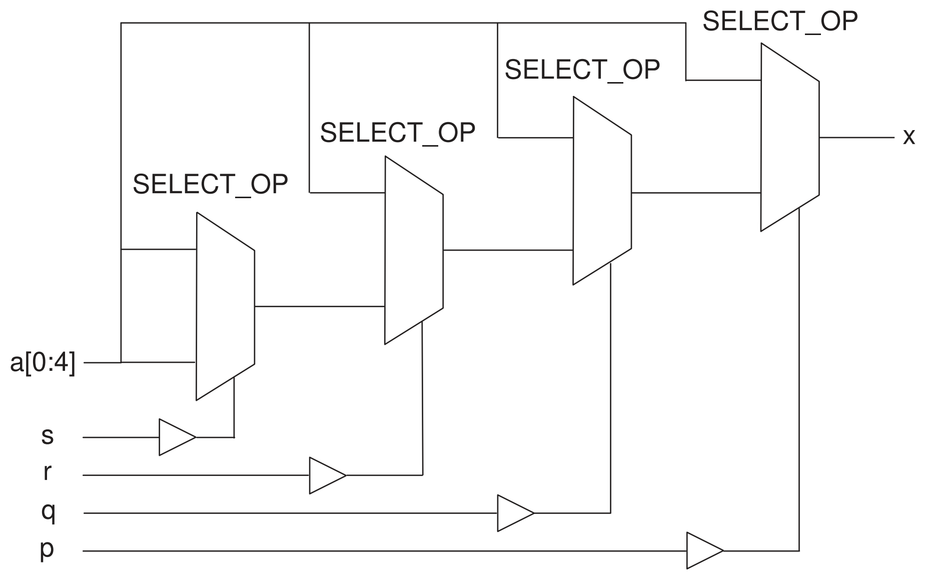
Modified RTL With Named begin-end Blocks

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 100 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 24 Schematic of the Modified RTL



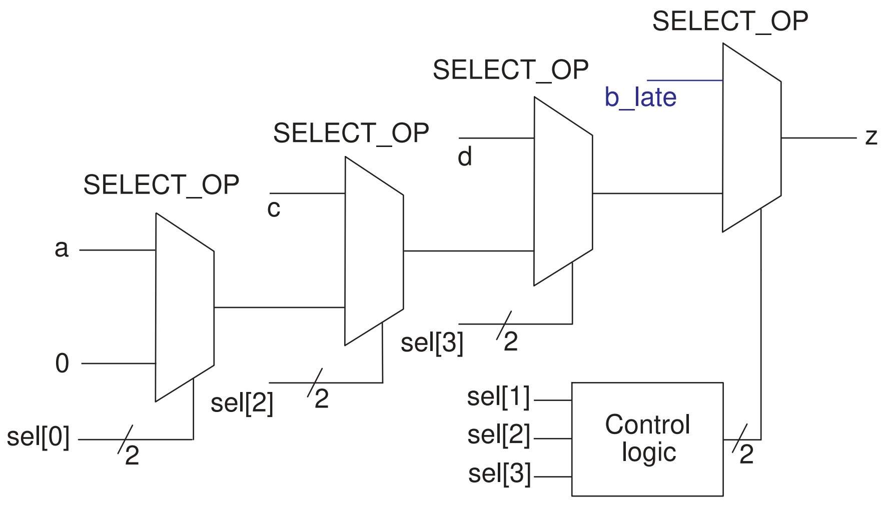
Late-Arriving Data Signal Example 1

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

Example 101 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 25 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.

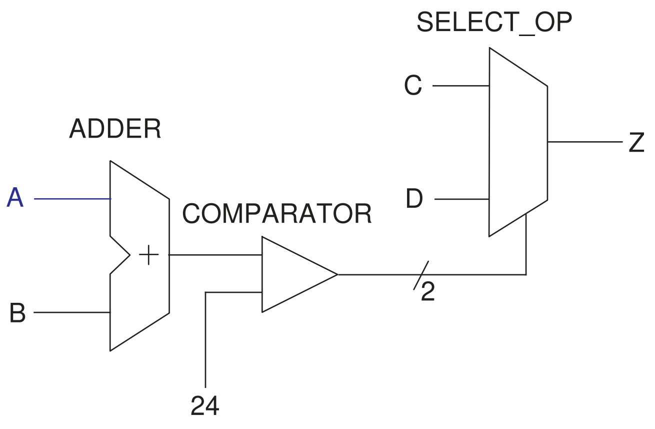
Original RTL Containing the Late-Arriving Input A

The original RTL contains input A that is late arriving.

Example 102 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 26 Schematic of the Original RTL



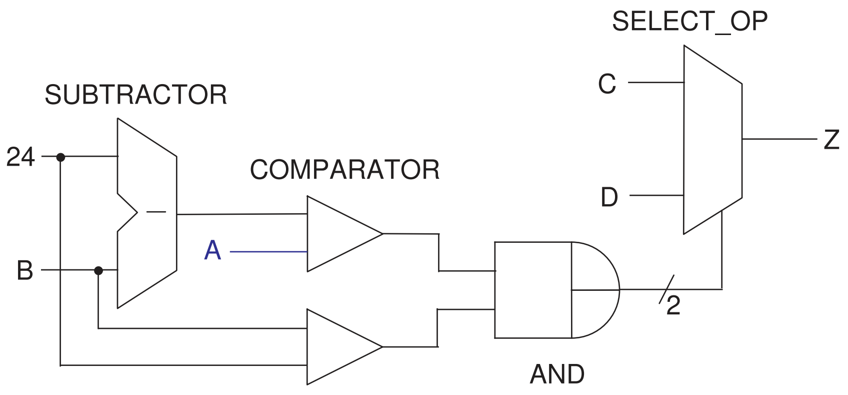
Modified RTL

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

Example 103 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 27 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.

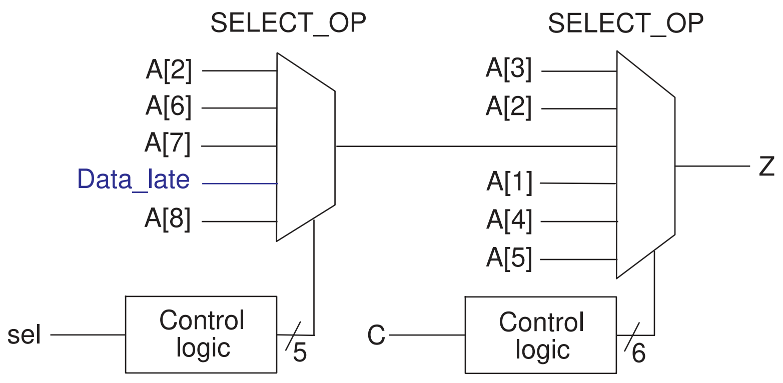
Original RTL Containing a Late-Arriving Input Data_late

The original RTL contains input Data_late that is late arriving.

Example 104 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 28 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 105 Modified RTL

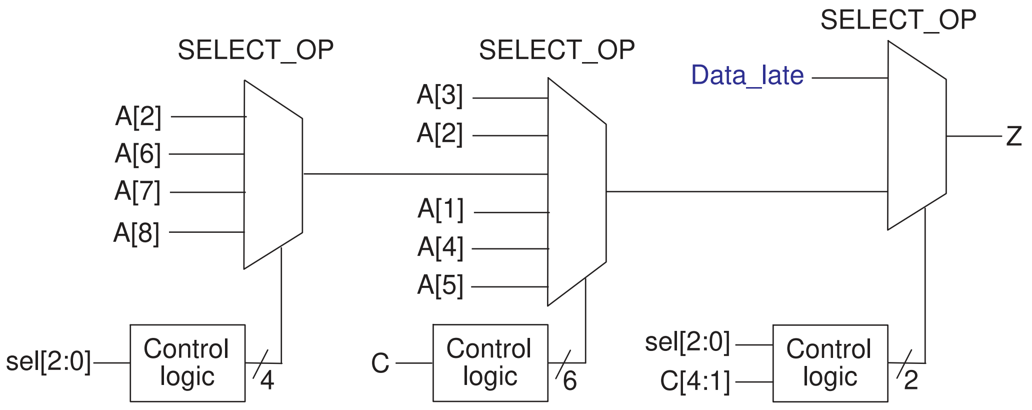
```
module case_in_if_01_improved (
   input [8:1] A,
   input Data_late,
   input [2:0] sel,
   input [5:1] C,
   output logic Z
);
logic Z1, FIRST_IF;

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 29   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.

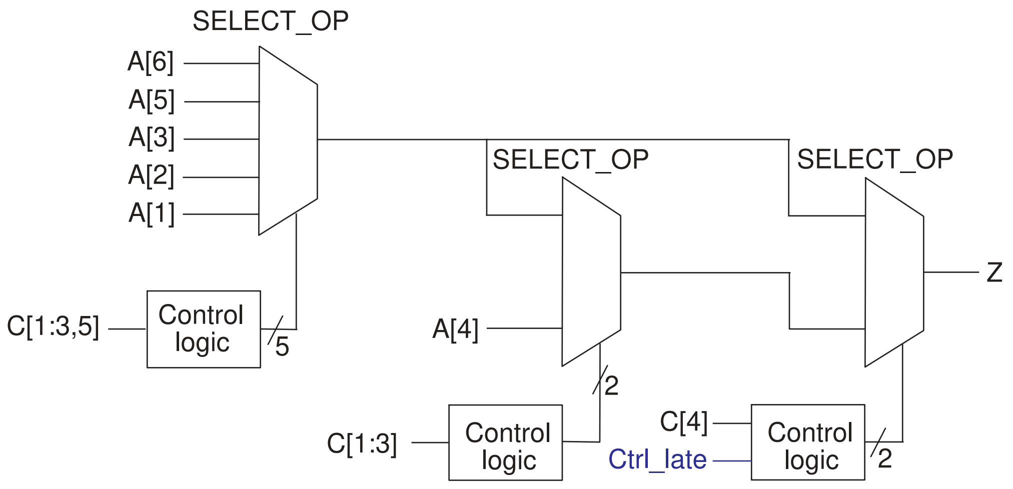
Example 106 RTL With a Late-Arriving Control Signal

```
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 30 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 107 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 108 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

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 109 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 @ (posedge SCK) Q <= DATA;
endmodule
```

Example 110 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 111 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 @ (posedge SCK1) Q1 <= D1;
always @ (posedge SCK2) Q2 <= D2;
endmodule
```

Example 112 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 113 RTL for Two-Phase Clocks

```
module latch_verilog (
   input DATA, MCK, SCK,
   output reg Q
);
reg TEMP;

always @(DATA or MCK)
   if (MCK) TEMP <= DATA;

always @(TEMP or SCK)
   if (SCK) Q <= TEMP;

endmodule
```

Example 114 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  | -  | -  | -  |
========================================================================
```

B

Verilog Language Support

The following sections describe the Verilog language as supported by HDL Compiler:

- Syntax
- Verilog Keywords
- Unsupported Verilog Language Constructs
- Construct Restrictions and Comments
- Verilog 2001 and 2005 Supported Constructs
- Ignored Constructs
- Verilog 2001 Feature Examples
- Verilog 2005 Feature Example

Syntax

Synopsys supports the Verilog syntax as described in the *IEEE Std 1364-2005*.

The lexical conventions HDL Compiler uses are described in the following sections:

- Comments
- Numbers

Comments

You can enter comments anywhere in a Verilog description, in two forms:

- Beginning with two slashes //

  HDL Compiler ignores all text between these characters and the end of the current line.
- Beginning with the two characters /* and ending with */

HDL Compiler ignores all text between these characters, so you can continue comments over more than one line.

Note:

You cannot nest comments.

Numbers

You can declare numbers in several different radices and bit-widths. A radix is the base number on which a numbering system is built. For example, the binary numbering system has a radix of 2, octal has a radix of 8, and decimal has a radix of 10.

You can use these three number formats:

- A simple decimal number that is a sequence of digits in the range of 0 to 9. All constants declared this way are assumed to be 32-bit numbers.
- A number that specifies the bit-width as well as the radix. These numbers are the same as those in the previous format, except that they are preceded by a decimal number that specifies the bit-width.
- A number followed by a two-character sequence prefix that specifies the number's size and radix. The radix determines which symbols you can include in the number. Constants declared this way are assumed to be 32-bit numbers. Any of these numbers can include underscores (_), which improve readability and do not affect the value of the number. Table 10 summarizes the available radices and valid characters for the number.

Table 10 Verilog Radices

Name	Character prefix	Valid characters
Binary	’b	0 1 x X z Z _ ?
Octal	’o	0–7 x X z Z _ ?
Decimal	’d	0–9 _
Hexadecimal	’h	0–9 a–f A–F x X z Z _ ?

Example 115 shows some valid number declarations.

Example 115 Valid Verilog Number Declarations

```
391                 //  32-bit decimal number
'h3a13              //  32-bit hexadecimal number
10'o1567            //  10-bit octal number
```

```
3'b010           //  3-bit binary number
4'd9             //  4-bit decimal number
40'hFF_FFFF_FFFF //  40-bit hexadecimal number
2'bxx            //  2-bits don't care
3'bzzz           //  3-bits high-impedance
```

Verilog Keywords

Table 11 lists the Verilog keywords. You cannot use these words as user variable names unless you use an escape identifier.

Caution:

Configuration-related keywords are not treated as keywords outside of configurations. HDL Compiler does not support configurations at this time.

Table 11 Verilog Keywords

always	and	assign	automatic	begin	buf
bufif0	bufif1	case	casex	casez	cell
cmos	config	deassign	default	defparam	design
disable	edge	else	end	endcase	endconfig
endfunction	endgenerate	endmodule	endprimitive	endspecify	endtable
endtask	event	for	force	forever	fork
function	generate	genvar	highz0	highz1	if
ifnone	incdir	include	initial	inout	input
instance	integer	join	large	liblist	library
localparam	macromodule	medium	module	nand	negedge
nmos	nor	noshowcancelled	not	notif0	notif1
or	output	parameter	pmos	posedge	primitive
pull0	pull1	pulldown	pullup	pulsestyle_onevent	pulsestyle_ondetect
rcmos	real	realtime	reg	release	repeat
rnmos	rpmos	rtran	rtranif0	rtranif1	scalared
showcancelled	signed	small	specify	specparam	strong0

Table 11   Verilog Keywords (Continued)

strong1	supply0	supply1	table	task	time
tran	tranif0	tranif1	tri	tri0	tri1
triand	trior	trireg	unsigned	use	vectored
wait	wand	weak0	weak1	while	wire
wor	xnor	xor			

Unsupported Verilog Language Constructs

HDL Compiler does not support the following constructs:

- Configurations
- Unsupported definitions and declarations
    - primitive definition
    - time declaration
    - event declaration
    - triand, trior, tri1, tri0, and trireg net types
    - Ranges for integers
- Unsupported statements
    - initial statement
    - repeat statement
    - delay control
    - event control
    - forever statement (The forever loop is only supported if it has an associated disable condition, making the exit condition deterministic.)
    - fork statement
    - deassign statement

    - force statement
    - release statement
- Unsupported operators
    - Case equality and inequality operators (=== and !==)
- Unsupported gate-level constructs
    - nmos, pmos, cmos, rnmos, rpmos, rcmos
    - pullup, pulldown, tranif0, tranif1, rtran, rtrainf0, and rtrainf1 gate types
- Unsupported miscellaneous constructs
    - hierarchical names within a module

If you use an unsupported construct, HDL Compiler issues a syntax error such as

```
event is not supported
```

Construct Restrictions and Comments

Construct restrictions and guidelines are described in the following sections:

- always Blocks
- generate Statements
- Real Math Functions
- Conditional Expressions (?:) Resource Sharing
- Case
- defparam
- disable
- Blocking and Nonblocking Assignments
- Macromodule
- inout Port Declaration
- tri Data Type
- HDL Compiler Directives
- reg Types

- Types in Busing
- Combinational while Loops

always Blocks

The tool does not support more than one independent `if` block when asynchronous behavior is modeled within an `always` block. If the `always` block is purely synchronous, the tool supports multiple independent `if` blocks. In addition, the tool does not support more than one conditional operator (?:) inside an `always` block.

Note:

If an `always` block is very small, the tool might move the logic inside the block during synthesis.

generate Statements

Synopsys support of the `generate` statement is described in the following sections:

- Generate Overview
- Types of generate Blocks
- Anonymous generate Blocks
- Loop Generate Blocks and Conditional Generate Blocks
- Restrictions

Generate Overview

HDL Compiler supports both the 2001 and the 2005 standards for the `generate` statement. The default is the 2005 standard; to enable the 2001 standard, set the `hdlin_vrlg_std` variable to `2001`. The following subsections describe the naming-style differences between these two standards.

Types of generate Blocks

Standalone generate Blocks

Standalone generate blocks are blocks using the `begin` statement that are not associated with a *conditional generate* or *loop generate* block. These are legal under the 2001 standard, but are illegal according to the Verilog 2005 LRM, as illustrated in the following example.

Example 116 Standalone generate Block

```
module top ( input in1, output out1 );
  generate
begin : b1
  mod1 U1(in1, out1);
end
endgenerate
endmodule

module mod1( input in1, output out1 );
endmodule
```

When you use the 2001 standard, HDL Compiler creates the name b1.U1 for mod 1:

```
Cell           Reference       Library             Area  Attributes
-------------------------------------------------------------------
b1.U1          mod1                            0.000000  b
-------------------------------------------------------------------
Total 1 cells                                  0.000000
```

When you use the 2005 standard, HDL Compiler issues a VER-946 error message:

```
Compiling source file RTL/t1.v
Error:  RTL/t1.v:3: Syntax error on an obsolete Verilog 2001 construct
 standalone generate block 'b1'. (VER-946)
*** Presto compilation terminated with 1 errors. ***
```

Anonymous generate Blocks

Anonymous generate blocks are `generate` blocks that do not have a user-defined label. They are also referred to as unnamed blocks.

According to the 2001 Verilog LRM, anonymous blocks do not create their own scope, but the 2005 standard has an implicit naming convention that allows scope creation. The Verilog 2005 standard assigns a number to every `generate` construct in a given scope. The number is 1 for the first construct and is incremented by 1 for each subsequent `generate` construct in the scope. All unnamed `generate` blocks are given the name genblk`n`, where `n` is the number assigned to the enclosing `generate` construct. If the name conflicts with an explicitly declared name, leading zeros are added in front of the number until the conflict is resolved.

The following example shows the difference between the two standards.

Example 117 Anonymous generate Block

```
module top( input [0:3] in1, output [0:3] out1 );
genvar I;
generate
for( I = 0; I < 3; I = I+1 ) begin: b1
  if( 1 ) begin : b2
    if( 1 )
      if( 1 )
```

```
        if( 1 )
          mod1 U1(in1[I], out1[I]);
   end
end
endgenerate
endmodule

module mod1( input in1, output out1 );
endmodule
```

When you use the Verilog 2001 standard, HDL Compiler creates the names b1[0].b2.U1, b1[1].b2.U1, and b1[2].b2.U1 for the instantiated subblocks:

```
Cell                        Reference       Library       Area  Attributes
--------------------------------------------------------------------------
b1[0].b2.U1                 mod1                      0.000000  b
b1[1].b2.U1                 mod1                      0.000000  b
b1[2].b2.U1                 mod1                      0.000000  b
--------------------------------------------------------------------------
Total 3 cells                                         0.000000
```

When you use the Verilog 2005 standard, HDL Compiler creates the names b1[0].b2.genblk1.U1, b1[1].b2.genblk1.U1, and b1[2].b2.genblk1.U1. Note that there are no multiple genblk1's for the nested anonymous `if` blocks:

```
Cell                        Reference       Library       Area  Attributes
--------------------------------------------------------------------------
b1[0].b2.genblk1.U1          mod1                      0.000000  b
b1[1].b2.genblk1.U1          mod1                      0.000000  b
b1[2].b2.genblk1.U1          mod1                      0.000000  b
--------------------------------------------------------------------------
Total 3 cells                                         0.000000
```

Another type of anonymous `generate` block is created when the block does not have a label, but each block has a `begin` ...`end` statement:

Example 118 Anonymous generate Block With begin...end

```
module top( input [0:3] in1, output [0:3] out1 );
genvar I;
generate
for( I = 0; I < 3; I = I+1 ) begin: b1
  if( 1 ) begin : b2
    if( 1 ) begin
      if( 1 ) begin
        if( 1 ) begin
          mod1 U1(in1[I], out1[I]);
        end
      end
    end
  end
end
endgenerate
```

```
endmodule

module mod1( input in1, output out1 );
endmodule
```

When you use the 2001 standard, HDL Compiler creates the names b1[0].b2.U1, b1[1].b2.U1, and b1[2].b2.U1 for the instantiated subblocks:

```
Cell                      Reference       Library       Area  Attributes
-------------------------------------------------------------------------
b1[0].b2.U1               mod1                      0.000000  b
b1[1].b2.U1               mod1                      0.000000  b
b1[2].b2.U1               mod1                      0.000000  b
-------------------------------------------------------------------------
Total 3 cells                                       0.000000
```

When you use the 2005 standard, the tool creates the names b1[0].b2.genblk1.genblk1.genblk1.U1, b1[1].b2.genblk1.genblk1.genblk1.U1, b1[2].b2.genblk1.genblk1.U1:

```
Cell                    Reference       Library         Area  Attributes
-------------------------------------------------------------------------
b1[0].b2.genblk1.genblk1.genblk1.U1
                          mod1                      0.000000  b
b1[1].b2.genblk1.genblk1.genblk1.U1
                          mod1                      0.000000  b
b1[2].b2.genblk1.genblk1.genblk1.U1
                          mod1                      0.000000  b
-------------------------------------------------------------------------
Total 3 cells                                       0.000000
```

Note that there is a genblk1 for each of the nested `begin...end if` blocks that creates a new scope.

The following example illustrates how scope creation can produce an error under the Verilog 2005 standard from code that compiles cleanly under the Verilog 2001 standard:

Example 119 Scope Creation

```
module top(input in, output out);
generate if(1) begin
  wire w = in;
end endgenerate
assign out = w;
endmodule
```

Under the Verilog 2001 standard, `w` is visible in the `assign` statement, but under the Verilog 2005 standard, scope creation makes `w` invisible outside the `generate` block, and HDL Compiler issues an error message:

```
Error:  RTL/t5.v:5: The symbol 'w' is not defined. (VER-956)
```

Loop Generate Blocks and Conditional Generate Blocks

Loop generate blocks are `generate` blocks that contain a `for` loop. *Conditional generate blocks* are `generate` blocks that contain an `if` statement. Loop generate blocks and conditional generate blocks can be nested, as shown in the following example.

Example 120 Loop and Conditional generates

```
module top( input D1, input clk, output Q1 );
genvar i, j;
parameter param1 = 0;
parameter param2 = 1;

generate
for (i=0; i < 3; i=i+1) begin : loop1
  for (j=0; j < 2; j=j+1) begin : loop2
    if (j == param1) begin : if1_label
    memory U_00 (D1,clk,Q1);
    end
    if (j == param2) begin : if2_label
    memory U_00 (D1,clk,Q1);
    end
  end //loop2
end //loop1
endgenerate
endmodule

module memory( input D1, input clk, output Q1 );
endmodule
```

In this case, the instance name is the same under both standards:

```
Cell                        Reference       Library        Area  Attributes
-------------------------------------------------------------------------
loop1[0].loop2[0].if1_label.U_00
                            memory                     0.000000  b
loop1[0].loop2[1].if2_label.U_00
                            memory                     0.000000  b
loop1[1].loop2[0].if1_label.U_00
                            memory                     0.000000  b
loop1[1].loop2[1].if2_label.U_00
                            memory                     0.000000  b
loop1[2].loop2[0].if1_label.U_00
                            memory                     0.000000  b
loop1[2].loop2[1].if2_label.U_00
                            memory                     0.000000  b
-------------------------------------------------------------------------
Total 6 cells
```

Restrictions

- Hierarchical Names (Cross Module Reference)

  HDL Compiler supports hierarchical names or cross-module references, if the hierarchical name remains inside the module that contains the name and each item on the hierarchical path is part of the module containing the reference.

  In the following code, the item is not part of the module and is not supported.

```
  module top ();
    wire x;
    down d ();
  endmodule

  module down ();
    wire y, z;
    assign t = top.d.z;
// not supported:
// hier. ref. starts outside current module
  endmodule
```

- Parameter Override (defparam)

  The use of defparam is highly discouraged in synthesis because of ambiguity problems. Because of these problems, defparam is not supported inside generate blocks. For details, see the Verilog 1800 LRM.

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 12.

Table 12 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

Table 12 Unary System Functions to C Language Real Math Functions Cross-Listing (Continued)

Unary System Function	Equivalent C Language Function	Description
$sin (x)	sin (x)	Sine
$cos (x)	cos (x)	Cosine
$tan (x)	tan (x)	Tangent
$asin (x)	asin (x)	Arc-sine
$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

Conditional Expressions (?:) Resource Sharing

HDL Compiler supports resource sharing in conditional expressions such as

```
dout = sel ? (a + b) : (a + c);
```

In such cases, HDL Compiler marks the adders as sharable; HDL Compiler determines the final implementation during timing-drive resource sharing.

The tool does not support more than one ?: operator inside an always block. For more information, see always Blocks on page 147.

Case

The case construct is discussed in the following sections:

- casez and casex
- Full Case and Parallel Case

casez and casex

HDL Compiler allows ? and z bits in casez items but not in expressions; that is, the z bits are allowed in the branches of the case statement but not in the expression immediately following the casez keyword.

```
casez (y)   // y is referred to as the case expression

2'b1z:    //2'b1z is referred to as the item
```

Example 121 shows an invalid expression in a casez statement.

Example 121 Invalid casez Expression

```
casez (1'bz)  //illegal testing of an expression
    ...
endcase
```

The same holds true for casex statements using x, ?, and z. The code

```
casex (a)
2'b1x : // matches 2'b10 and 2'b11
endcase
```

does not equal the following code:

```
b = 2'b1x;
casex (a)
b:    // in this case, 2'b1x only matches 2'b10
endcase
```

When x is assigned to a variable and the variable is used in a casex item, the x does not match both 0 and 1 as it would for a literal x listed in the case item.

Full Case and Parallel Case

Case statements can be full or parallel. HDL Compiler can usually determine automatically whether a case statement is full or parallel. Example 122 shows a case statement that is both full and parallel.

Example 122 A case Statement That Is Both Full and Parallel

```
input [1:0] a;
always @(a or w or x or y or z) begin
 case (a)
  2'b11:
      b = w ;
  2'b10:
      b = x ;
  2'b01:
      b = y ;
  2'b00:
      b = z ;
 endcase
end
```

In Example 123, the case statement is not parallel or full, because the values of inputs w and x cannot be determined.

Example 123 A case Statement That Is Not Full and Not Parallel

```
always @(w or x) begin
 case (2'b11)
  w:
      b = 10 ;
  x:
      b = 01 ;
 endcase
end
```

However, if you know that only one of the inputs equals 2’b11 at a given time, you can use the `parallel_case` directive to avoid synthesizing an unnecessary priority encoder.

If you know that either w or x always equals 2’b11 (a situation known as a one-branch tree), you can use the `full_case` directive to avoid synthesizing an unnecessary latch. A latch is necessary whenever a variable is conditionally assigned. Marking a case as full tells the compiler that some branch is taken, so there is no need for an implicit default branch. If a variable is assigned in all branches of the case, HDL Compiler then knows that the variable is not conditionally assigned in that case, and, therefore, that particular case statement does not result in a latch for that variable.

However, if the variable is assigned in only some branches of the case statement, a latch is still required as shown in Example 124. In addition, other case statements might cause a latch to be inferred for the same variable.

Example 124 Latch Result When Variable Is Not Fully Assigned

```
reg a, b;
reg [1:0] c;
case (c)     // synopsys full_case
    0: begin a = 1; b = 0; end
    1: begin a = 0; b = 0; end
    2: begin a = 1; b = 1; end
    3: b = 1;  // a is not assigned here
endcase
```

For more information, see parallel_case and full_case.

defparam

Use of defparam is highly discouraged in synthesis because of ambiguity problems. Because of these problems, defparam is not supported inside generate blocks. For details, see the Verilog LRM.

disable

HDL Compiler supports the disable statement when you use it in named blocks and when it is used to disable an enclosing block. When a disable statement is executed, it causes the named block to terminate. You cannot disable a block that is not in the same always block or task as the disable statement. A comparator description that uses disable is shown in Example 125.

Example 125 Comparator Using disable

```
begin : compare
 for (i = 7; i >= 0; i = i - 1) begin
     if (a[i] != b[i]) begin
         greater_than = a[i];
         less_than = ~a[i];
         equal_to = 0;
         //comparison is done so stop looping
         disable compare;
      end
 end

// If you get here a == b
// If the disable statement is executed, the next three
// lines will not be executed
   greater_than = 0;
   less_than = 0;
   equal_to = 1;
end
```

You can also use a disable statement to implement a synchronous reset, as shown in Example 126.

Example 126 Synchronous Reset of State Register Using disable in a forever Loop

```
always
 begin: test
  @ (posedge clk)
```

```
     if (Reset)
      begin
       z <= 1'b0;
       disable test;
      end
       z <= a;
   end
```

The disable statement in Example 126 causes the test block to terminate immediately and return to the beginning of the block.

Blocking and Nonblocking Assignments

HDL Compiler does not allow both blocking and nonblocking assignments to the same variable within an always block.

The following code applies both blocking and nonblocking assignments to the same variable in one always block.

```
always @(posedge clk or negedge reset) begin
  if (~ reset)
    q = 1'b0;
  else
    q <= d;
end
```

HDL Compiler does not permit this and generates an error message.

During simulation, race conditions can result from blocking assignments, as shown in Example 127. In this example, the value of x is indeterminate, because multiple procedural blocks run concurrently, causing y to be loaded into x at the same time z is loading into y. The value of x after the first @ (posedge clk) is indeterminate. Use of nonblocking assignments solves this race condition, as shown in Example 128.

In Example 127 and Example 128, HDL Compiler creates the gates shown in Figure 31.

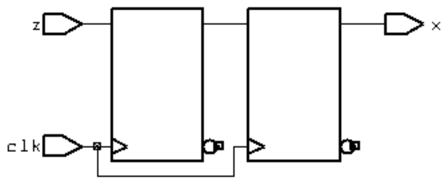
Example 127 Race Condition Using Blocking Assignments

```
always @(posedge clk)
  x = y;
always @(posedge clk)
  y = z;
```

Example 128 Race Solved With Nonblocking Assignments

```
always @(posedge clk)
  x <= y;
always @(posedge clk)
  y <= x;
```

Figure 31 Simulator Race Condition—Synthesis Gates



If you want to switch register values, use nonblocking assignments, because blocking assignments do not accomplish the switch. For example, in Example 129, the required outcome is a swap of the x and y register values. However, after the positive clock edge, y does not end up with the value of x; y ends up with the original value of y. This happens because blocking statements are order dependent and each statement within the procedural block is executed before the next statement is evaluated and executed. In Example 130, the swap is accomplished with nonblocking assignments.

Example 129 Swap Problem Using Blocking Assignments

```
always @(posedge clk)
begin
  x = y;
  y = x;
end
```

Example 130 Swap Accomplished With Nonblocking Assignments

```
always @(posedge clk)
  x <= y;
  y <= z;
```

Macromodule

HDL Compiler treats the macromodule construct as a module construct. Whether you use module or macromodule, the synthesis results are the same.

inout Port Declaration

HDL Compiler allows you to connect inout ports only to module or gate instantiations. You must declare an inout before you use it.

tri Data Type

The tri data type allows multiple three-state devices to drive a wire. When inferring three-state devices, you need to ensure that all the drivers are inferred as three-state devices and that all inputs to a device are z, except the one variable driving the three-state device which have a 1.

HDL Compiler Directives

HDL Compiler directives are discussed in the following sections:

- `define
- `include
- `ifdef, `else, `endif, `ifndef, and `elsif
- `undef

`define

The `` `define `` directive can specify macros that take arguments. For example,

```
`define BYTE_TO_BITS(arg)  ((arg) << 3)
```

The `` `define `` directive can do more than simple text substitution. It can also take arguments and substitute their values in its replacement text.

Macro substitution assigns a string of text to a macro variable. The string of text is inserted into the code where the macro is encountered. The definition begins with the back quotation mark (`), followed by the keyword `define`, followed by the name of the macro variable. All text from the macro variable until the end of the line is assigned to the macro variable.

You can declare and use macro variables anywhere in the description. The definitions can carry across several files that are read into HDL Compiler at the same time. To make a macro substitution, type a back quotation mark (`) followed by the macro variable name.

Some example macro variable declarations are shown in Example 131.

Example 131 Macro Variable Declarations

```
`define highbits      31:29
`define bitlist       {first, second, third}
wire [31:0] bus;
`bitlist = bus['highbits];
```

The `analyze -define` command allows macro definition on the command line. Only one `-define` per `analyze` command is allowed but the argument can be a list of macros, as shown in Example 132.

Note:

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

Curly brackets are not required to enclose one macro, as shown in Example 133. However, if the argument is a list of macros, curly brackets are required.

Example 132 analyze Command With List of Defines

```
analyze -format verilog -define { RIPPLE, SIMPLE } mydesign.v
```

Example 133 analyze Command With One Define

```
analyze -format verilog -define ONLY_ONE mydesign.v
```

See Also

- Predefined Macros

`include

The `` `include `` construct in Verilog is similar to the `#include` directive in the C language. You can use this construct to include Verilog code, such as type declarations and functions, from one module in another module. Example 134 shows an application of the `` `include `` construct.

Example 134 Including a File Within a File

```
Contents of file1.v
`define WORDSIZE 8

function [`WORDSIZE-1:0] fastadder;
  input [`WORDSIZE-1:0] fin1, fin2;
  fastadder = fin1 + fin2;
endfunction

Contents of file2.v
module secondfile (clk, in1, in2, out);

`include "file1.v"
. . .
wire [`WORDSIZE-1:0] temp;
assign temp = fastadder (in1,in2);
. . .
endmodule
```

Included files can include other files, with up to 24 levels of nesting. You cannot use the `` `include `` construct recursively.

When your design contains multiple files for multiple subblocks and include files for subblocks, in their respective sub directories, you can elaborate the top-level design without making any changes to the search path. The tool automatically finds the include files. For example, if your structure is as follows:

```
Rtl/top.v
Rtl/sub_module1/sub_module1.v
Rtl/sub_module2/sub_module2.v
Rtl/sub_module1/sub_module1_inc.v
Rtl/sub_module2/sub_module2_inc.v
```

You do not need to add Rtl/sub_module1/ and Rtl/sub_module2/ to your search path to enable the tool to find the include files sub_module1_inc.v and sub_module2_inc.v when you elaborate top.v.

`ifdef, `else, `endif, `ifndef, and `elsif

These directives allow the conditional inclusion of code.

- The `` `ifdef `` directive executes the statements following it if the indicated macro is defined; if the macro is not defined, the statements after `` `else `` are executed.
- The `` `ifndef `` directive executes the statements following it if the indicated macro is not defined; if the macro is defined, the statements after `` `else `` are executed.
- The `` `elsif `` directive allows one level of nesting and is equivalent to the `` `else `ifdef ... `endif `` directive sequence.

Example 135 illustrates usage. Use the `` `define `` directive to define the macros that are arguments to the `` `ifdef `` directive; see `define.

Example 135 Design Using `ifdef...`else...`endif Directives

```
`ifdef SELECT_XOR_DESIGN
module selective_design(a,b,c);
  input a, b;
  output c;
    assign c = a ^ b;
endmodule

`else

module selective_design(a,b,c);
  input a, b;
  output c;
    assign c = a | b;
endmodule
`endif
```

`undef

The `` `undef `` directive resets the macro immediately following it.

reg Types

The Verilog language requires that any value assigned inside an always statement must be declared as a reg type. HDL Compiler returns an error if any value assigned inside an always block is not declared as a reg type.

Types in Busing

HDL Compiler maintains types throughout a design, including types for buses (vectors). Example 136 shows a Verilog design read into HDL Compiler containing a bit vector that is NOTed into another bit vector.

Example 136 Bit Vector in Verilog

```
module test_busing_1 ( a, b );
  input  [3:0] a;
  output [3:0] b;

  assign b = ~a;
endmodule
```

Example 137 shows the same description written out by HDL Compiler. The description contains the original Verilog types of ports. Internal nets do not maintain their original bus types. Also, the NOT operation is instantiated as single bits.

Example 137 Bit Blasting

```
module test_busing_2 ( a, b );
  input  [3:0] a;
  output [3:0] b;
    assign b[0] = ~a[0];
    assign b[1] = ~a[1];
    assign b[2] = ~a[2];
    assign b[3] = ~a[3];
endmodule
```

Combinational while Loops

To create a combinational while loop, write the code so that an upper bound on the number of loop iterations can be determined. The loop iterative bound must be statically determinable; otherwise an error is reported.

HDL Compiler needs to be able to determine an upper bound on the number of trips through the loop at compile time. In HDL Compiler, there are no syntax restrictions on the loops; while loops that have no events within them, such as in the following example, are supported.

```
input [9:0] a;
// ....
i = 0;
while ( i < 10 && !a[i] ) begin
  i = i + 1;
  // loop body
end
```

To support this loop, HDL Compiler interprets it like a simulator. The tool stops when the loop termination condition is known to be false. Because HDL Compiler can’t determine when a loop is infinite, it stops and reports an error after an arbitrary (but user-defined) number of iterations (the default is 1024).

To exit the loop, HDL Compiler allows additional conditions in the loop condition that permit more concise descriptions.

```
for (i = 0; i < 10 && a[i]; i = i+1) begin
  // loop body
end
```

A loop must unconditionally make progress toward termination in each trip through the loop, or it cannot be compiled. The following example makes progress (that is, increments i) only when !done is true and does not terminate.

```
while ( i < 10 ) begin
  if ( ! done )
    done = a[i];
    // loop body
    i = i + 1;
  end
end
```

The following modified version, which unconditionally increments i, terminates. This code creates the required logic.

```
while ( i < 10 ) begin
  if ( ! done ) begin
    done = a[i];
    end// loop body
    i = i + 1;
end
```

In the next example, loop termination depends on reading values stored in x. If the value is unknown (as in the first and third iterations), HDL Compiler assumes it might be true and generates logic to test it.

```
x[0] = v;       // Value unknown: implies "if(v)"
x[1] = 1;       // Known TRUE: no guard on 2nd trip
x[2] = w;       // Not known: implies "if(w)"
x[3] = 0;       // Known FALSE: stop the loop

i = 0;
```

```
while( x[i] ) begin
  // loop body
  i = i + 1;
end
```

This code terminates after three iterations when the loop tests x[3], which contains 0.

In Example 138, a supported combinational while loop, the code produces gates, and an event control signal is not necessary.

Example 138 Supported while Loop Code

```
module modified_s2 (a, b, z);
parameter N = 3;
input [N:0] a, b;
output [N:1] z;
reg [N:1] z;
integer i;
always @(a or b or z)
    begin
        i = N;
        while (i)
            begin
                z[i] = b[i] + a[i-1];
                i = i - 1;
            end
      end
endmodule
```

In Example 139, a supported combinational while loop, no matter what *x* is, the loop runs for 16 iterations at most because HDL Compiler can keep track of which bits of *x* are constant. Even though it doesn't know the initial value of *x*, it does know that *x* >> 1 has a zero in the most significant bit (MSB). The next time *x* is shifted right, it knows that x has two zeros in the MSB, and so on. HDL Compiler can determine when *x* becomes all zeros.

Example 139 Supported Combinational while Loop

```
module while_loop_comb1(x, count);
  input [7:0] x;
  output [2:0] count;
  reg [7:0] temp;
  reg [2:0] count;
  always @ (x)
  begin
     temp = x;
    count = 0;
    while (temp != 0)
    begin
    count = count + 1;
    temp = temp >> 1;
  end
```

```
    end
  endmodule
```

In Example 140, a supported combinational while loop, HDL Compiler knows the initial value of *x* and can determine *x+1* and all subsequent values of *x*.

Example 140 Supported Combinational while Loop

```
module while_loop_comb2(y, count1, z);
  input [3:0] y, count1;output [3:0] z;
  reg [3:0] x, z, count;
  always @ (y, count1)
  begin
    x = 2;
    count = count1;
    while (x < 15)
      begin
        count = count + 1;
        x = x + 1;
      end
    z = count;
  end
endmodule
```

In Example 141, HDL Compiler cannot detect the initial value of i and so cannot support this while loop. Example 142 is supported because i is determinable.

Example 141 Unsupported Combinational while Loop

```
module my_loop1 #(parameter N=4) (input [N:0] in, output reg [2*N:0] out);
  reg [N:0] i;
  always @* begin
  i = in;
  out = 0 ;
  while (i>0) begin
    out = out + i;
    i = i - 1;
  end
end
endmodule
```

Example 142 Supported Combinational while Loop

```
module my_loop2 #(parameter N=4) (input [N:0] in, output reg [2*N:0] out);
  reg [N:0] i;
  reg [N+1:0] j;
  always @*
  for (j = 0 ; j < (2<<N) ; j = j+1 )
    if (j==in) begin
      i = j;
      out = 0 ;
      while (i>0) begin
        out = out + i;
        i = i - 1;
      end
    end
endmodule
```

Verilog 2001 and 2005 Supported Constructs

Table 13 lists the Verilog 2001 and 2005 features implemented by HDL Compiler. For additional information about these features, see the *IEEE Std 1364-2001*.

Table 13 Supported Verilog 2001 and 2005 Constructs

Feature	Description
Automatic tasks and functions	Fully supported
Constant functions	Fully supported
Local parameter	Fully supported
generate statement	See generate Statements.
Real math functions	See Real Math Functions.
SYNTHESIS macro	Fully supported
Implicit net declarations for continuous assignments	Fully supported
`line directive	Fully supported
ANSI-C-style port declarations	Fully supported
Casting operators	Fully supported
Parameter passing by name (IEEE 12.2.2.2)	Fully supported
Implicit event expression list (IEEE 9.7.5)	Fully supported
ANSI-C-style port declaration (IEEE 12.3.3)	Fully supported
Signed/unsigned parameters (IEEE 3.11)	Fully supported
Signed/unsigned nets and registers (IEEE 3.2, 4.3)	Fully supported
Signed/unsigned sized and based constants (IEEE 3.2)	Fully supported
Multidimensional arrays and arrays of nets (IEEE 3.10)	Fully supported
Part select addressing ([+:] and [-:] operators) (IEEE 4.2.1)	Fully supported

Table 13 Supported Verilog 2001 and 2005 Constructs (Continued)

Feature	Description
Power operator (**) (IEEE 4.1.5)	Fully supported
Arithmetic shift operators (<<< and >>>) (IEEE 4.1.12)	Fully supported
Sized parameters (IEEE 3.11.1)	Fully supported
`ifndef, `elsif, `undef (IEEE 19.4,19.3.2)	Fully supported
`ifdef VERILOG_2001 and `ifdef VERILOG_1995	Fully supported
Comma-separated sensitivity lists (IEEE 4.1.15 and 9.7.4)	Fully supported

Ignored Constructs

The following sections include directives that HDL Compiler accepts but ignores.

Simulation Directives

The following directives are special commands that affect the operation of the Verilog HDL simulator:

```
'accelerate
'celldefine
'default_nettype
'endcelldefine
'endprotect
'expand_vectornets
'noaccelerate
'noexpand_vectornets
'noremove_netnames
'nounconnected_drive
'protect
'remove_netnames
'resetall
'timescale
'unconnected_drive
```

You can include these directives in your design description; HDL Compiler accepts but ignores them.

Verilog System Functions

Verilog system functions are special functions that Verilog HDL simulators implement. Their names start with a dollar sign ($). All of these functions are accepted but ignored by HDL Compiler with the exception of $display, which can be useful during synthesis elaboration. See Use of $display During RTL Elaboration.

Verilog 2001 Feature Examples

This section provides examples for Verilog 2001 features in the following sections:

- Multidimensional Arrays and Arrays of Nets
- Signed Quantities
- Comparisons With Signed Types
- Controlling Signs With Casting Operators
- Part-Select Addressing Operators ([+:] and [-:])
- Power Operator (**)
- Arithmetic Shift Operators (<<< and >>>)

Multidimensional Arrays and Arrays of Nets

HDL Compiler supports multidimensional arrays of any variable or net data type. This added functionality is shown in the following examples.

Example 143 Multidimensional Arrays

```
module m (a, z);
  input [7:0] a;
  output z;
  reg t [0:3][0:7];
  integer i, j;
  integer k;
  always @(a)
    begin
     for (j = 0; j < 8; j = j + 1)
      begin
        t[0][j] = a[j];
      end
     for (i = 1; i < 4; i = i + 1)
      begin
        k = 1 << (3-i);
       for (j = 0; j < k; j = j + 1)
         begin
```

```
              t[i][j] = t[i-1][2*j] ^ t[i-1][2*j+1];
            end
         end
     end
     assign z = t[3][0];
   endmodule
```

Example 144 Arrays of Nets

```
   module m (a, z);
     input [0:3] a;
     output z;
     wire x [0:2] ;
     assign x[0] = a[0] ^ a[1];
     assign x[1] = a[2] ^ a[3];
     assign x[2] = x[0] ^ x[1];
     assign z = x[2];
   endmodule
```

Example 145 Multidimensional Array Variable Subscripting

```
   reg [7:0] X [0:7][0:7][0:7];

   assign out = X[a][b][c][d+:4];
```

Verilog 2001 allows more than one level of subscripting on a variable, without use of a temporary variable.

Example 146 Multidimensional Array

```
module test(in, en, out, addr_in, addr_out_reg, addr_out_bit, clk);

  input [7:0] in;
  input en, clk;
  input [2:0] addr_in, addr_out_reg, addr_out_bit;
  reg [7:0] MEM [0:7];
  output out;

  assign out = MEM[addr_out_reg][addr_out_bit];

  always @(posedge clk) if (en) MEM[addr_in] = in;
endmodule
```

Signed Quantities

HDL Compiler supports signed arithmetic extensions. Function returns and reg and net data types can be declared as signed. This added functionality is shown in the following examples.

Example 147 results in a sign extension, that is, z[0] connects to a[0].

Example 147 Signed I/O Ports

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

In Example 148, because 3’sb111 is signed, the tool infers a signed adder. In the generic netlist, the ADD_TC_OP cell denotes a 2’s complement adder and z[0] is not logic 0.

Example 148 Signed Constants: Code and GTECH Gates

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

In Example 149, because 4’sd5 is signed, a signed comparator (LT_TC_OP) is inferred.

Example 149 Signed Registers: Code and GTECH Gates

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

In Example 150, because in1, in2, and out are signed, a signed multiplier (MULT_TC_OP_8_8_8) is inferred.

Example 150 Signed Types: Code and Gates

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

The code in Example 151 results in a signed subtractor (SUB_TC_OP).

Example 151 Signed Nets: Code and Gates

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

In Example 152, because 4’sd5 is signed, a signed comparator (LT_TC_OP) is inferred.

Example 152 Signed Values

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

Verilog 2001 adds the signed keyword in declarations: `reg signed [7:0] x;`

It also adds support for signed, sized constants. For example, 8'sb11111111 is an 8-bit signed quantity representing -1. If you are assigning it to a variable that is 8 bits or less, 8'sb11111111 is the same as the unsigned 8'b11111111. A behavior difference arises when the variable being assigned to is larger than the constant. This difference occurs because signed quantities are extended with the high-order bit of the constant, whereas unsigned quantities are extended with 0s. When used in expressions, the sign of the constant helps determine whether the operation is performed as signed or unsigned.

HDL Compiler enables signed types by default.

Note:

If you use the `signed` keyword, any signed constant in your code, or explicit type casting between signed and unsigned types, HDL Compiler issues a warning.

Comparisons With Signed Types

Verilog sign rules are tricky. All inputs to an expression must be signed to obtain a signed operator. If one is signed and one unsigned, both are treated as unsigned. Any unsigned quantity in an expression makes the whole expression unsigned; the result doesn’t depend on the sign of the left side. Some expressions always produce an unsigned result; these include bit and part-select and concatenation. See IEEE P1364/P5 Section 4.5.1.You need to control the sign of the inputs yourself if you want to compare a signed quantity against an unsigned one. The same is true for other kinds of expressions. See Example 153 and Example 154.

Example 153 Unsigned Comparison Results When Signs Are Mismatched

```
module m8 (in1, in2, lt);
// in1 is signed but in2 is unsigned
   input signed [7:0] in1;
   input        [7:0] in2;
   output lt;
   wire uns_lt, uns_in1_lt_64;
/* comparison is unsigned because of the sign mismatch, in1
is signed but in2 is unsigned */
   assign uns_lt = in1 < in2;
/* Unsigned constant causes unsigned comparison; so negative
values of in1 would compare as larger than 8'd64 */
   assign uns_in1_lt_64 = in1 < 8'd64;
   assign lt = uns_lt + uns_in1_lt_64;
endmodule
```

Example 154 Signed Values

```
module m7 (in1, in2, lt, in1_lt_64);
   input  signed [7:0] in1, in2;  // two signed inputs
   output lt, in1_lt_64;
   assign lt  =  in1 < in2;     // comparison is signed
   // using a signed constant results in a signed comparison
   assign in1_lt_64 = in1 < 8'sd64;
endmodule
```

Controlling Signs With Casting Operators

Use the Verilog 2001 casting operators, $signed() and $unsigned(), to convert an unsigned expression to a signed expression. In Example 155, the casting operator is used to obtain a signed comparator. Note that simply marking an expression as signed might give undesirable results because the unsigned value might be interpreted as a negative number. To avoid this problem, zero-extend unsigned quantities, as shown in Example 155.

Example 155 Casting Operators

```
module m9 (in1, in2, lt);
   input signed [7:0] in1;
   input        [7:0] in2;
   output lt;
   assign lt = in1 < $signed ({1'b0, in2});
  //Cast to get signed comparator.
  //Zero-extend to preserve interpretation of unsigned value as positive
number.
```

Part-Select Addressing Operators ([+:] and [-:])

Verilog 2001 introduced variable part-select operators. These operators allow you to use variables to select a group of bits from a vector. In some designs, coding with part-select operators improves elaboration time and memory usage.

Variable part-select operators are discussed in the following sections:

Variable Part-Select Overview

A Verilog 1995 part-select operator requires that both upper and lower indexes be constant: a[2:3] or a[value1:value2].

The variable part-select operator permits selection of a fixed-width group of bits at a variable base address and takes the following form:

- [base_expr +: width_expr] for a positive offset
- [base_expr -: width_expr] for a negative offset

The syntax specifies a variable base address and a known constant number of bits to be extracted. The base address is always written on the left, regardless of the declared direction of the array. The language allows variable part-select on the left side and the right side of an expression. All of the following expressions are allowed:

- data_out = array_expn[index_var +: 3]

  (part select is on the right side)
- data_out = array_expn[index_var -: 3]

  (part select is on the right side)
- array_expn[index_var +: 3] = data_in

  (part select is on the left side)
- array_expn[index_var -: 3] = data_in

  (part select is on the left side)

This table shows examples of Verilog 2001 syntax and the equivalent Verilog 1995 syntax.

Verilog 2001 syntax	Equivalent Verilog 1995 syntax	
a[x +: 3] for a descending array	{ a[x+2], a[x+1], a[x] }	a[x+2 : x]
a[x -: 3] for a descending array	{ a[x], a[x-1], a[x-2] }	a[x : x-2]
a[x +: 3] for an ascending array	{ a[x], a[x+1], a[x+2] }	a[x : x+2]
a[x -: 3] for an ascending array	{ a[x-2], a[x-1], a[x] }	a[x-2 : x]

The original HDL Compiler tool allows nonconstant part-selects if the width is constant; HDL Compiler permits only the new syntax.

Example—Ascending Array and -:

The following Verilog code uses the -: operator to select bits from Ascending_Array.

```
reg [0:7] Ascending_Array;
...
   Data_Out = Ascending_Array[Index_Var -: 3];
```

The value of Index_Var determines the starting point for the bits selected. In the following table, the bits selected are shown as a function of Index_Var.

Ascending_Array	**[**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7]**
Index_Var = 0	not valid, synthesis/simulation mismatch								
Index_Var = 1	not valid, synthesis/simulation mismatch								
Index_Var = 2		•	•	•	•	•	•	•	•
Index_Var = 3		•	•	•	•	•	•	•	•
Index_Var = 4		•	•	•	•	•	•	•	•
Index_Var = 5		•	•	•	•	•	•	•	•
Index_Var = 6		•	•	•	•	•	•	•	•
Index_Var = 7		•	•	•	•	•	•	•	•

Ascending_Array[Index_Var -: 3] is functionally equivalent to the following part-select that is not computable:Ascending_Array[Index_Var - 2 : Index_Var]

Example—Ascending Array and +:

The following Verilog code uses the +: operator to select bits from Ascending_Array.

```
reg [0:7] Ascending_Array;
...
   Data_Out = Ascending_Array[Index_Var +: 3];
```

The value of Index_Var determines the starting point for the bits selected. In the following table, the bits selected are shown as a function of Index_Var.

Ascending_Array [	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7]**
Index_Var = 0	•	•	•	•	•	•	•	•
Index_Var = 1	•	•	•	•	•	•	•	•
Index_Var = 2	•	•	•	•	•	•	•	•
Index_Var = 3	•	•	•	•	•	•	•	•
Index_Var = 4	•	•	•	•	•	•	•	•
Index_Var = 5	•	•	•	•	•	•	•	•
Index_Var = 6	not valid, synthesis/simulation mismatch; see the following note.							
Index_Var = 7	not valid, synthesis/simulation mismatch; see the following note.							

Note:

- Ascending_Array[Index_Var +: 3] is functionally equivalent to the following part-select that is not computable: Ascending_Array[Index_Var : Index_Var + 2]
- Noncomputable part-selects are not supported by the Verilog language. Ascending_Array[7 +:3] corresponds to elements Ascending_Array[7 : 9] but elements Ascending_Array[8] and Ascending_Array[9] do not exist. A variable part-select must always compute to a valid index; otherwise, a synthesis elaborate error and a runtime simulation error results.

Example—Descending Array and the -: Operator

The following code uses the -: operator to select bits from Descending_Array.

```
reg [7:0] Descending_Array;
...
   Data_Out = Descending_Array[Index_Var -: 3];
```

The value of Index_Var determines the starting point for the bits selected. In the following table, the bits selected are shown as a function of Index_Var.

Descending_Array	[7	6	5	4	3	2	1	0]
Index_Var = 0	not valid, synthesis/simulation mismatch							
Index_Var = 1	not valid, synthesis/simulation mismatch							
Index_Var = 2	•	•	•	•	•	•	•	•
Index_Var = 3	•	•	•	•	•	•	•	•
Index_Var = 4	•	•	•	•	•	•	•	•
Index_Var = 5	•	•	•	•	•	•	•	•
Index_Var = 6	•	•	•	•	•	•	•	•
Index_Var = 7	•	•	•	•	•	•	•	•

Descending_Array[Index_Var -: 3] is functionally equivalent to the following noncomputable part-select: Descending_Array[Index_Var : Index_Var - 2]

Example—Descending Array and the +: Operator

The following Verilog code uses the +: operator to select bits from Descending_Array.

```
reg [7:0] Descending_Array;
...
   Data_Out = Descending_Array[Index_Var +: 3];
```

The value of Index_Var determines the starting point for the bits selected. In the following table, the bits selected are shown as a function of Index_Var.

Descending_Array	[7	6	5	4	3	2	1	0]
Index_Var = 0	•	•	•	•	•	•	•	•
Index_Var = 1	•	•	•	•	•	•	•	•
Index_Var = 2	•	•	•	•	•	•	•	•
Index_Var = 3	•	•	•	•	•	•	•	•
Index_Var = 4	•	•	•	•	•	•	•	•
Index_Var = 5	•	•	•	•	•	•	•	•

Descending_Array	**[7**	**6**	**5**	**4**	**3**	**2**	**1**	**0]**
Index_Var = 6	not valid, synthesis/simulation mismatch							
Index_Var = 7	not valid, synthesis/simulation mismatch							

Descending_Array[Index_Var +: 3] is functionally equivalent to the following noncomputable part-select: Descending_Array[Index_Var + 2 : Index_Var]

Noncomputable part-selects are not supported by the Verilog language. Descending_Array[7 +:3] corresponds to elements Descending_Array[9 : 7] but elements Descending_Array[9] and Descending_Array[8] do not exist. A variable part-select must always compute to a valid index; otherwise, a synthesis elaborate error and a runtime simulation error results.

Power Operator (**)

This operator performs y^x, as shown in Example 156.

Example 156 Power Operators

```
module m #(parameter b=2, c=4) (a, x, y, z);
  input [3:0] a;
  output [7:0] x, y, z;

  assign z = 2 ** a;
  assign x = a ** 2;
  assign y = b ** c; // where b and c are constants

endmodule
```

Arithmetic Shift Operators (<<< and >>>)

The arithmetic shift operators allow you to shift an expression and still maintain the sign of a value, as shown in Example 157. When the type of the result is signed, the arithmetic shift operator (>>>) shifts in the sign bit; otherwise it shifts in zeros.

Example 157 Shift Operator Code and Gates

```
module s1 (A, S, Q);
  input signed [3:0] A;
  input [1:0] S;
  output [3:0] Q;
  reg [3:0] Q;
  always @(A or S)
  begin

// arithmetic shift right,
// shifts in sign-bit from left

    Q = A >>> S;
  end
endmodule
```

Verilog 2005 Feature Example

Zero Replication

According to the Verilog 2005 LRM, a replication operation with a zero replication constant is considered to have a size of zero and is ignored. Such an operation can appear only within a concatenation in which at least one of the operands of the concatenation has a positive size.

Zero replication can be useful for parameterized designs. In the following example, the valid values for parameter P are 1 to 32.

```
module top  #(parameter P = 32)  ( input [32-1:0]a, output [32-1:0] b);
assign b = {{32-P{1'b1}}, a[P-1:0]};
endmodule
```

When the `hdlin_vrlg_std` variable is set to `2005`, and you analyze replication operations whose elaboration-time constant is zero or negative, the repeated expressions elaborate once (for their side-effects). But they do not contribute result values to a surrounding concatenation or assignment pattern. The Verilog 2005 standard permits such empty replication results only within an otherwise nonempty concatenation

Note:

Nonstandard replication operations that are analyzed when the Verilog version is set to `1995` or `2001` return 1'b0. This is compatible with an extension made by Synopsys Verilog products of that era.

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.

HDL Compiler
The Synopsys Verilog synthesis product.

identifier
A sequence of letters, underscores, and numbers. An identifier cannot be a Verilog reserved word, such as *type* or *loop*. An identifier must begin with a letter or an underscore.

latch
A level-sensitive memory device.

netlist
A network of connected components that together define a design.

optimization
The modification of a design in an attempt to improve some performance aspect. 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.

Feedback

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.