HDL Compiler™ for VHDL User Guide

Version W-2024.09-SP3, January 2025

SYNOPSYS®

Feedback

Contents

Feedback

About This Manual

The HDL Compiler tool translates a VHDL hardware language description into a 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 HDL Compiler
- Using HDL Compiler Synthesis directives, attributes, and variables

Audience

The *HDL Compiler for VHDL User Guide* is written for logic designers and electronic engineers who are familiar with the Design Compiler® and Fusion Compiler™ tools. Knowledge of the VHDL 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 Compiler®
- Fusion Compiler™
- DesignWare® components
- Library Compiler™

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
`Courier`	Indicates syntax, such as `write_file`
`Courier italic`	Indicates a user-defined value in syntax, such as `write_file` *`design_list`*
`Courier bold`	Indicates user input—text you type verbatim—in examples, such as `prompt>` **`write_file top`**
`Purple`	• Within an example, indicates information of special interest. • Within a command-syntax section, indicates a default, such as `include_enclosing =` **`true`** `| false`
[ ]	Denotes optional arguments in syntax, such as `write_file [-format` *`fmt`*`]`
...	Indicates that arguments can be repeated as many times as needed, such as *`pin1 pin2 ... pinN`*.
|	Indicates a choice among alternatives, such as `low | medium | high`
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Bold	Indicates a graphical user interface (GUI) element that has an action associated with it.
Edit > Copy	Indicates a path to a menu command, such as opening the **Edit** menu and choosing **Copy**.

Convention	Description
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.

Feedback

1

VHDL for Synthesis

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

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 VHDL Designs

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

- `read_vhdl` or `read_file -format vhdl`

  For example,

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

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

  For example,

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

  This method is recommended because of the following reasons:

  - Recursive elaboration is performed on the entire design, so you do not need an explicit `link` command. The `elaborate` command includes the functions of the `link` command.
  - For designs containing interfaces or parameterized designs, you do not need to set the `hdlin_auto_save_templates` variable to `true`.
  - You can build parameterized designs with nondefault parameter values.
  - The `read_file` command ignores standalone configurations.

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

If you want to read a VHDL netlist, use the specialized VHDL netlist reader instead of HDL Compiler. The VHDL netlist reader reads netlists faster and uses less memory.

Note:

To enable the VHDL netlist reader, you must set `enable_vhdl_netlist_reader` to `true` (the default is `false)` and use the netlist reading commands shown in Summary of Reading Methods.

If the file is not a VHDL netlist or if `enable_vhdl_netlist_reader` is set to `false`, HDL Compiler reads the design.

Specifying the VHDL Version

To specify which VHDL language version to use during the read process, set the `hdlin_vhdl_std` variable . The valid values are `1987`, `1993`, and `2008`, corresponding to the 1987, 1993, and 2008 VHDL LRM releases respectively. The default is `2008`.

Summary of Reading Methods

The recommended and alternative reading commands are shown in the following table:

Type of input	**Reading method**

RTL	Recommended reading method
	`analyze -format vhdl { files}`
	`elaborate topdesign`
	Alternative reading method
	`read_vhdl { files }` (Tcl)
	`read_file -rtl vhdl { files }` (Tcl)

Gate-level netlists	Recommended reading method `read_vhdl -netlist { files }` (Tcl) or `read_file -rtl vhdl -netlist { files }` (Tcl) Alternative reading method You can also use any RTL-reading command to read netlists, but it is slower and uses more memory than the specialized gate-level netlist reader.

Using the analyze and elaborate Commands

When you are elaborating a design, the last analyzed architecture is used if you do not specify an architecture.

To understand how to use the `analyze` and `elaborate` commands, consider Example 1, which represents a single design with no user-defined libraries.

Example 1   Design dff_pos

```
library IEEE ;
use IEEE.std_logic_1164.all;

entity dff_pos is
  port (DATA, CLK : in std_logic;
        Q : out std_logic );
end dff_pos;

architecture rtl of dff_pos is
begin
process (CLK) begin
  if (rising_edge (CLK)) then
    Q <= DATA;
  end if;
end process;
end rtl;
```

Example 2 through Example 4 show various reading methods.

Example 2

```
--The analyze and elaborate commands read design dff_pos which is
--contained in a single file, dff_entity_arch.vhd.

dc_shell> analyze -format vhdl dff_entity_arch.vhd
dc_shell> elaborate dff_pos
```

Example 3

```
--Design dff_pos is contained in two files, dff_entity.vhd and
--dff_arch.vhd. Each file is analyzed by a separate analyze command
--and then the dff_pos design is elaborated.

dc_shell> analyze -format vhdl dff_entity.vhd
dc_shell> analyze -format vhdl dff_arch.vhd
dc_shell> elaborate dff_pos
```

Example 4

```
--Design dff_pos is contained in two files, dff_entity.vhd and
--dff_arch.vhd. Both files are analyzed using a single analyze command,
--and then elaborated.

dc_shell> analyze -format vhdl {dff_entity.vhd dff_arch.vhd}
dc_shell> elaborate dff_pos
```

Using the read_file Command

To use any `read_file` command, all your entity and architecture definitions must be contained in a single read. The entity and architecture definitions can be contained in separate files. Table 1 shows various reading examples.

Table 1 read Command Examples

Example	Description
`dc_shell> read_file -format vhdl ALU.vhd`	The `read_file` command reads the single design in the ALU.vhd file.
`dc_shell> read_vhdl ALU.vhd`	The `read_vhdl` command reads the single design in the ALU.vhd file.
`dc_shell> read_vhdl {ALU_subblock.vhd ALU_top.vhd}`	The `read_vhdl` command reads a design consisting of two files: ALU_subblock.vhd and ALU_top.vhd.

`dc_shell> read_vhdl ALU_subblock.vhd`
`dc_shell> read_vhdl ALU_top.vhd`
	Two `read_vhdl` commands read a design consisting of two files: ALU_subblock.vhd and ALU_top.vhd.

When you use the `read` command, you must also use the `current_design` command, to specify the working design, and the `link` command, to resolve design references, before optimizing the design. These operations are automatically done by the `elaborate` command. If you have configurations, you must use `analyze`. The `read` command ignores configurations and has limited supported for generics. See Configuration Support and Parameterized Models (Generics).

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 VHDL files other than the default (.vhd and .vhdl) and sets the file source lists. The `read_file -autoread` command specifies the top-level design and includes only files with the specified VHDL file extensions.

```
dc_shell> set_app_var hdlin_autoread_vhdl_extensions {.vhdx .vhdlang}
dc_shell> set my_sources {entities/src archs/src}
dc_shell> set my_excludes {entities/src/no_hdl_* archs/src/docs/}
dc_shell> analyze $my_sources -recursive -exclude $my_excludes \
   -autoread -format vhdl -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 .sverilog

File Dependencies

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

- *Analyze dependency*

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

- *Link dependency*

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

- *Include dependency*

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

- *Verilog and SystemVerilog compilation-unit dependency*

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

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)
```

Decrypting VHDL 1735 Files

VHDL RTL files that are encrypted using Synopsys public keys, according to the IEEE 1735 encryption standard, can be decrypted and analyzed in the HDL Compiler tool.

Note:

To to get the public encryption keys, contact your Synopsys representative.

For example, the encrypted VHDL file `top.vhdp` is decrypted and read back in the tool as follows:

```
`protect begin_protected
`protect key_keyowner = "Synopsys"
`protect key_keyname = "SNPS-SYN-EC-RSA-1"
`protect key_method = "rsa"
`protect encoding = (enctype = "base64", line_length = 76, bytes = 256)
`protect key_block
DikTHyPyT3q6ap6EB2fR8C2w0tOHOhoikHbAld3QyhJ/5ryypBEon/AW4RvyMqJd …..
```

Following is an example for decrypted VHDL file analysis:

```
analyze -format vhdl top.vhdp
    ….
    Running PRESTO HDLC
    Presto compilation completed successfully.
   1
```

If decryption fails due to an incorrect key or an error message in the RTL design, the HDL Compiler tool issues the following error:

```
-------------------
      : [Redacted]
         ^^^^^^^^
[Error] in encrypted region
```

Note:

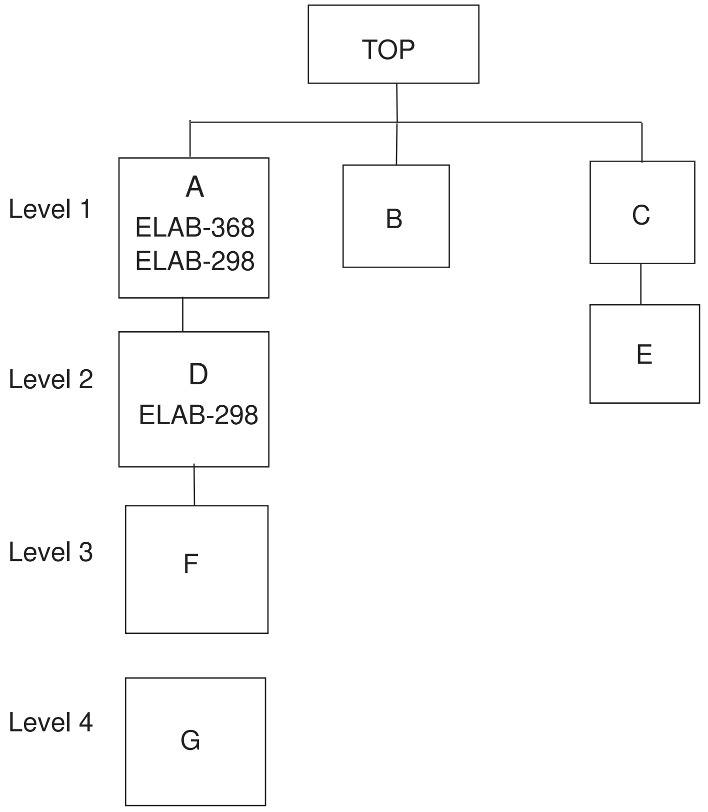
The tool does not include any RTL design details such as the port name in the error message.

Reporting Elaboration Errors

HDL Compiler elaborates designs in a top-down hierarchical order. The elaboration failure of a top-level module prohibits the elaboration of all associated submodules. The `hdlin_elab_errors_deep` variable allows the elaboration of submodules even if the top-level module elaboration fails, enabling HDL Compiler to report more elaboration, link, and VER-37 errors and warnings in a hierarchical design during the first elaboration run.

To understand how the `hdlin_elab_errors_deep` variable works, consider the four-level hierarchical design in Figure 1. This design has elaboration (ELAB) errors as noted in the figure.

Figure 1 Hierarchical Design



Under default conditions, when you elaborate the design, HDL Compiler reports only the errors in the first level (ELAB-368 and ELAB-298 in module A). To find the second-level error (ELAB-298 in submodule D), you need to fix the first-level errors and elaborate again. When you use the `hdlin_elab_errors_deep` variable, you only need to elaborate once to find the errors in module A and submodule D.

The next section describes the `hdlin_elab_errors_deep` variable and provides methodology and examples.

Methodology

Use the following methodology to enable HDL Compiler to report elaboration, link, and VER-37 errors across the hierarchy during a single elaboration run.

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

   When you set the `hdlin_elab_errors_deep` variable to `true`, HDL Compiler reports the following:

   ```
   HDLC compilation run in rtl debug mode.
   ```

   Caution:

   HDL Compiler does not create designs when you set `hdlin_elab_errors_deep` to `true`. The tool reports warnings if you try to use commands that require a design. For example, if you run `list_designs`, the tool reports the following:

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

3. Elaborate your design using the `elaborate` command.
4. Fix any elaboration, link, and VER-37 errors. Review the warnings and fix errors as needed.
5. Set the `hdlin_elab_errors_deep` variable to `false`.
6. Elaborate your error-free design.
7. Proceed with your normal synthesis flow.

The next section provides examples showing HDL Compiler reporting all errors across the hierarchy, reducing the need for multiple elaboration runs.

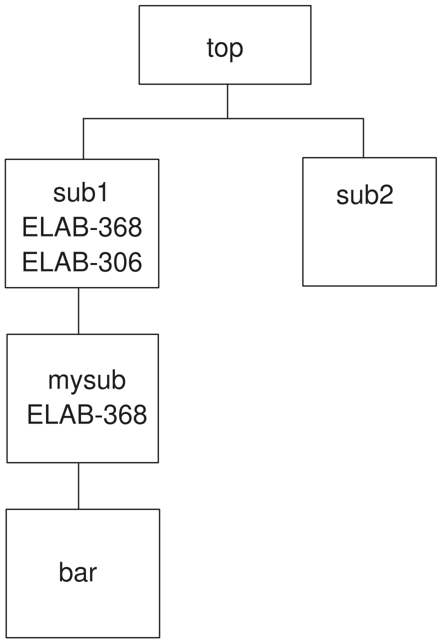
Examples

To enable HDL Compiler to find errors down the hierarchy in one elaboration run, you can set the `hdlin_elab_errors_deep` variable to `true`, changing it from its default of `false`. This variable is designed to speed up the time in finding design elaboration and linking errors.

This section uses the top design in Figure 2 as an example of reporting hierarchical errors. The error messages are shown in the figure. Example 5 lists the RTL code of the top design.

Feedback

Figure 2 Hierarchical Design



Example 5 VHDL RTL for the top Design

```
-- entity top
entity top is
port (clk, a, b : in bit;
      c, o : out bit);
end entity top;
architecture A1 of top is
component sub1 is
port ( a, b     : in bit;
       o        : out bit);
end component sub1;
component sub2 is
port ( a, b     : in bit;
       o        : out bit);
end component sub2;
begin
  sub1_inst: sub1 port map (a, b, c);
  sub2_inst: sub2 port map (a, b, o);
end A1;

-- entity sub1
library IEEE;
use IEEE.std_logic_1164.all;
entity sub1 is
port ( a, b     : in bit;
       o        : out bit);
```

```
end entity sub1;
architecture A1 of sub1 is
component comp1 is
port ( a, b     : in bit;
       o        : out bit);
end component mysub;
  signal r    : bit_vector(1 downto 0);
  signal temp, sig, sig1 : std_logic;
  constant icon : integer := 5;
begin

  temp <= TO_STDULOGIC(a or b);
  temp <= '1';             -- ELAB-368  error
  temp <= sig and 'Z';     -- ELAB-306  error

  mysub_inst: mysub port map (a, b, o);
end A1;

-- entity mysub
library IEEE;
use IEEE.std_logic_1164.all;
entity mysub is
port ( a, b     : in bit;
       o        : out bit);
end entity mysub;
architecture A1 of mysub is
component bar is
port ( a, b     : in bit;
       o        : out bit);
end component bar;
signal temp : bit;
begin
    temp <= a and b;
    temp <= '1';           -- ELAB-368  error
    bar_inst: bar port map(a, b, o);
end A1;

-- entity bar
entity bar is
port ( a, b     : in bit;
       o        : out bit);
end entity bar;
architecture A1 of bar is
begin
    o <= not b;
end A1;

-- entity sub2
entity sub2 is
port ( a, b     : in bit;
       o        : out bit);
end entity sub2;
architecture A1 of sub2 is
```

```
begin
    o <= a or b;
end A1;
```

When you elaborate the top design with the `hdlin_elab_errors_deep` variable set to `false`, HDL Compiler reports the first-level ELAB-368 and ELAB-306 errors in the sub1 module but does not report the ELAB-368 error in the mysub submodule. Example 6 shows the session log.

Example 6   Session Log

```
analyze -f vhdl test.vhd
Running HDLC
Compiling Entity Declaration TOP
Compiling Architecture A1 of TOP
Compiling Entity Declaration SUB1
Compiling Architecture A1 of SUB1
Compiling Entity Declaration MYSUB
Compiling Architecture A1 of MYSUB
Compiling Entity Declaration BAR
Compiling Architecture A1 of BAR
Compiling Entity Declaration SUB2
Compiling Architecture A1 of SUB2
HDLC compilation completed successfully.
Loading db file '/.../lsi_10k.db'
1
elaborate top
Loading db file '/.../gtech.db'
Loading db file '/.../standard.sldb'
  Loading link library 'lsi_10k'
  Loading link library 'gtech'
Running HDLC
HDLC compilation completed successfully.
Elaborated 1 design.
Current design is now 'top'.
Information: Building the design 'sub1'. (HDL-193)
Error:  ./test.vhd:39: Net 'temp', or a directly connected net, is driven
by
more than one source, and at least one source is a constant net.
(ELAB-368)
Error:  ./test.vhd:40: Illegal use of tristate value (HDL-140).
(ELAB-306)
*** HDLC compilation terminated with 2 errors. ***
Information: Building the design 'sub2'. (HDL-193)
HDLC compilation completed successfully.
Warning: Unable to resolve reference 'sub1' in 'top'. (LINK-5)
1
current_design
Current design is 'top'.
{top}
list_designs
```

```
sub2    top (*)
1
```

When you set the `hdlin_elab_errors_deep` variable to `true`, HDL Compiler reports errors down the hierarchy during elaboration. Example 7 shows the session log with all the error messages.

Example 7   Session Log With All the Error Messages

```
set hdlin_elab_errors_deep TRUE
TRUE
analyze -f vhdl test.vhd
Running HDLC
Compiling Entity Declaration TOP
Compiling Architecture A1 of TOP
Compiling Entity Declaration SUB1
Compiling Architecture A1 of SUB1
Compiling Entity Declaration MYSUB
Compiling Architecture A1 of MYSUB
Compiling Entity Declaration BAR
Compiling Architecture A1 of BAR
Compiling Entity Declaration SUB2
Compiling Architecture A1 of SUB2
HDLC compilation completed successfully.
Loading db file '/.../lsi_10k.db'
1
elaborate top
Loading db file '/.../gtech.db'
Loading db file '/.../standard.sldb'
  Loading link library 'lsi_10k'
  Loading link library 'gtech'
Running HDLC
*** HDLC compilation run in rtl debug mode. ***
HDLC compilation completed successfully.
Elaborated 1 design.
Current design is now 'top'.
Information: Building the design 'sub1'. (HDL-193)
*** HDLC compilation run in rtl debug mode. ***
Error:  ./test.vhd:39: Net 'temp', or a directly connected net, is driven
by more than one source, and at least one source is a constant net.
(ELAB-368)
Error:  ./test.vhd:40: Illegal use of tristate value (HDL-140).
(ELAB-306)
*** HDLC compilation terminated with 2 errors. ***
*** HDLC compilation run with backup flow. ***
Information: Building the design 'sub2'. (HDL-193)
*** HDLC compilation run in rtl debug mode. ***
HDLC compilation completed successfully.
Information: Building the design 'mysub'. (HDL-193)
*** HDLC compilation run in rtl debug mode. ***
Error: ./test.vhd:60: Net 'temp', or a directly connected net, is driven
by more than one source, and at least one source is a constant net.
(ELAB-368)
```

```
*** HDLC compilation terminated with 1 errors. ***
*** HDLC compilation run with backup flow. ***
Information: Building the design 'bar'. (HDL-193)
*** HDLC compilation run in rtl debug mode. ***
HDLC compilation completed successfully.
1
current_design
Error: Current design is not defined. (UID-4)
list_designs
Warning: No designs to list. (UID-275)
0
```

By default, only the top-level errors are reported, that is, the ELAB-368 and ELAB-306 errors in the sub1 module. To report the child-level ELAB-368 error in the mysub submodule, you need to fix all the errors in the sub1 module and run the `elaborate` command again. However, when you set the `hdlin_elab_errors_deep` variable to `true`, HDL Compiler reports all errors down the hierarchy in one elaboration run:

- ELAB-368 and ELAB-306 in the sub1 module
- ELAB-368 in the mysub submodule

When the `hdlin_elab_errors_deep` variable to `true`, note the following guidelines:

- No designs are saved because the designs could be erroneous.
- The `compile_ultra` and `list_designs` commands do not work.
- You should use the `analyze` command rather than the `read_file` command to read your design because the `read_file` command has no link functionality and accepts no command-line parameter specifications.
- All syntax errors are reported when you run the `analyze` command, but HDL Compiler is not a linting tool. You should use the `check_design` command in HDL Compiler for linting.
- The runtime during elaboration might increase slightly.

Caution:

HDL Compiler does not create designs when the `hdlin_elab_errors_deep` variable is set to `true`. If you run the `list_designs` command, HDL Compiler reports the following warning message:

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

Customizing Elaboration Reports

By default, the tool displays inferred sequential elements, MUX_OPs, and inferred three-state elements in elaboration reports using the `basic` setting, as shown in Table 2.

You can customize the report by setting the `hdlin_reporting_level` variable to `none`, `comprehensive`, or `verbose`. A true, false, or verbose setting indicates that the corresponding information is included, excluded, or detailed respectively in the report.

Table 2 Basic Reporting Level Variable Settings

Information Displayed (Information Keyword)	**Basic (Default)**	**None**	**Comprehensive**	**Verbose**
Floating net to ground connections (`floating_net_to_ground`)	false	false	true	true
Inferred state variables (`fsm`)	false	false	true	true
Inferred sequential elements (`inferred_modules`)	true	false	true	true
MUX_OPs (`mux_op`)	true	false	true	true
Synthetic modules (`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 modules, 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}
```

Parameterized Models (Generics)

HDL Compiler fully supports generic declarations. Generics enable you to assign unique parameter values to each model instance when you elaborate your design.

The model in Example 8 uses a generic declaration to determine the bit-width of a register input; the default width is declared as 2.

Example 8 Generic Register Model

```
LIBRARY IEEE, SYNOPSYS;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_MISC.ALL;
USE SYNOPSYS.ATTRIBUTES.ALL;

entity DFF is
 generic(N : INTEGER := 2);  --flip flop is N bits wide
 port(input  :   in STD_LOGIC_VECTOR (N - 1 downto 0);
      clock  :   in STD_LOGIC;
      output :   out STD_LOGIC_VECTOR (N - 1 downto 0) );
end DFF;

architecture RTL of DFF is
begin

entry : process (clock)
   variable tmp: STD_LOGIC_VECTOR (N - 1 downto 0);
begin
 if (clock = '0') then
   tmp := input;
 else
   if (clock = '1') then
     output <= tmp;
   end if;
 end if;
end process;
end RTL;
```

Assuming that the file n-register.vhd contains the model in Example 8, you can analyze the model and store the results in the user-specified design library, mylib, with the following command:

```
dc_shell> analyze -format vhdl n-register.vhd -library  mylib
```

To specify that an instance of the register model should have a bit-width of 3, use the `elaborate` command as follows:

```
dc_shell> elaborate DFF -parameters N=3
```

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

```
dc_shell> list_designs
 Design
--------
* DFF_N3
```

Using the `read_file` command with generics is not recommended, because you can build only the default of the generic. If you do not specify a default generic value, HDL Compiler reports the following:

```
Warning:  filename:line: Generic N does not have default
value. (ELAB-943).
```

In addition, you need to either set the `hdlin_auto_save_templates` variable to `true` or insert the `--synopsys template` directive in the entity declaration, as follows:

```
entity DFF is
 generic(N : INTEGER := 2);  --flip flop is N bits wide
 port(input  :   in STD_LOGIC_VECTOR (N - 1 downto 0);
      clock  :   in STD_LOGIC;
      output :   out STD_LOGIC_VECTOR (N - 1 downto 0) );
  -- synopsys template
end DFF;
```

The variables described in Table 3 control the naming of designs based on generic models. To list their current values, enter the following command:

```
dc_shell> report_app_var template_*
```

Table 3 Template Naming Variables

Variable	Description
`template_naming_stylehdlin.naming.template_naming_style`	Controls how templates (VHDL generics) are named. The default value is %s_%p, where %s is the name of the source design and %p is the parameter list. By default, the design name and the parameter list are separated by an underscore (_).
`template_separator_stylehdlin.naming.template_separator_style`	Provides a separator character for multiple parameters in a template name. The default value is an underscore (_).
`template_parameter_stylehdlin.naming.template_parameter_style`	Controls how template parameters are named. The default value is %s%d, where %s is the name of a parameter and %d is the value of that parameter. By default, the parameter name and value are not separated.

Configuration Support

To enable configuration support, set the `hdlin_enable_configurations` variable to `true`. The default is `false`. When this is set to `true`, you cannot use the `-autoread` option of the `analyze` or `read_file` command. For more information about the `-autoread` option, see Automated Process of Reading Designs With Dependencies.

Configurations bind entity design units to architecture design units. To specify a configuration, you must use the `analyze` command. For example, if file.vhdl contains the configuration my_configuration, read the design as follows:

```
analyze -format vhdl file.vhdl
elaborate my_configuration
```

VHDL allows different entities to have different architecture definitions of the same name. For example,

```
arch RTL1 of entity1 is
....
arch RTL1 of entity2 is
...
```

However, the same does not hold for configurations. For example, the following is not supported.

```
configuration CNFG1 of entity1 is
....
configuration CNFG1 of entity2 is
...
```

HDL Compiler binds the last read definition of CNFG1 to both entities. Therefore, configuration names for different entities must be unique. There is no configuration/entity pair concept. If configurations for different entities have the same name and they are analyzed sequentially, only the last one remains, as shown:

```
entity conf_0_vhdl is
   port(x: in BIT; y: out BIT);
end conf_0_vhdl;

architecture design_0_vhdl of conf_0_vhdl is
  begin
    y <= x;
end design_0_vhdl;

configuration trivial_config of work.conf_0_vhdl is

  for design_0_vhdl
  end for;

end trivial_config;
```

Example 9 shows the tool log output.

Example 9 Tool Log Output

```
dc_shell> elaborate trivial_config
Running HDLC
Loading db file '/SYNOPSYS_ROOT_DIRECTORY/libraries/syn/standard.sldb'
Loading db file '/SYNOPSYS_ROOT_DIRECTORY/libraries/syn/gtech.db'
Loading db file '/SYNOPSYS_ROOT_DIRECTORY/libraries/syn/lsi_10k.db'
```

```
format: vhdl
HDLC compilation completed successfully.
Current design is now 'conf_0_vhdl'
```

This following subsections describe HDL Compiler support for configurations:

- Bind Entity to Architecture
- Architectures From the Same Library
- Architectures From Different Libraries
- Component Inside a Concurrent Block
- Generic in a Configuration
- Port Map in a Configuration
- Nested Configurations
- Indirectly Nested Configurations
- Embedded Configurations
- Multiple Architectures in Embedded Configurations
- Combinations of Embedded, Nested, and Standalone Configurations
- Mixed Language Support

Bind Entity to Architecture

Example 10 uses configurations to bind components C1 and C2 to specific entity and architecture combinations.

Example 10 Binding Entities to Architectures

```
entity a_bar_b is
  port(a, b: in bit; c: out bit);
end a_bar_b;

architecture struct of a_bar_b is
  begin
     c <= not(a) and b;
end struct;

entity a_b_bar is
   port(a, b: in bit; c: out bit);
end a_b_bar;

architecture struct of a_b_bar is
   begin
      c <= a and not(b);
end struct;
entity conf_1_vhdl is
```

```
    port(a, b: in bit; c: out bit);
end conf_1_vhdl;

architecture struct of conf_1_vhdl is
     component a_bar_b port(a, b: in bit; c :out bit); end component;
     component a_b_bar port(a, b: in bit; c :out bit); end component;

     signal a_not_b, not_a_b: bit;
        begin
            C1: a_bar_b port map(a, b, not_a_b);
            C2: a_b_bar port map(a, b, a_not_b);
            c <= not_a_b or a_not_b;
end struct;

configuration config_example1 of conf_1_vhdl is
    for struct  -- of conf_1_vhdl
      for C1: a_bar_b
          use entity work.a_bar_b(struct);
      end for;
      for C2: a_b_bar
          use entity work.a_b_bar(struct);
      end for;
   end for;
end config_example1;
```

Example 11 shows the tool log output.

Example 11   Tool Log Output

```
dc_shell> elaborate config_example1
Running HDLC
Loading db file '/SYNOPSYS_ROOT_DIRECTORY/libraries/syn/standard.sldb'
Loading db file '/SYNOPSYS_ROOT_DIRECTORY/libraries/syn/gtech.db'
Loading db file '/SYNOPSYS_ROOT_DIRECTORY/libraries/syn/lsi_10k.db'
Component WORK.STRUCT/CONF_1_VHDL.STRUCT.C1 has been configured to use
the following implementation:
      Work Library: WORK
      Design Name:  A_BAR_B
      Architecture Name: STRUCT
Component WORK.STRUCT/CONF_1_VHDL.STRUCT.C2 has been configured to use
the following implementation:
      Work Library: WORK
      Design Name:  A_B_BAR
      Architecture Name: STRUCT
HDLC compilation completed successfully.
Information: Building the design 'A_BAR_B'. (HDL-193)
Running HDLC
HDLC compilation completed successfully.
Information: Building the design 'A_B_BAR'. (HDL-193)
Running HDLC
HDLC compilation completed successfully.
Current design is now 'conf_1_vhdl'
```

Architectures From the Same Library

Example 12 uses configurations to bind two instances of the same component to different architectures of the same entity. The component C1 is implemented using architecture struct1, while C2 is implemented using architecture struct2.

Example 12   Using Architectures From the Same Library

```
entity a_bar_b is
    port(a, b: in bit; c: out bit);
end a_bar_b;

architecture struct1 of a_bar_b is
begin
    c <= not(a) and b;
end struct1;

architecture struct2 of a_bar_b is
begin
    c <= a and not(b);
end struct2;

entity conf_2_vhdl is
    port(a, b: in bit; c: out bit);
end conf_2_vhdl;

architecture struct of conf_2_vhdl is
    component a_bar_b port(a, b: in bit; c :out bit); end component;

    signal a_not_b, not_a_b: bit;
begin
    C1: a_bar_b port map(a, b, not_a_b);
    C2: a_bar_b port map(a, b, a_not_b);
    c <= not_a_b or a_not_b;
end struct;

configuration config_example2 of conf_2_vhdl is
    for struct  -- of conf_2_vhdl
        for C1: a_bar_b
            use entity work.a_bar_b(struct1);
        end for;
        for C2: a_bar_b
            use entity work.a_bar_b(struct2);
        end for;
    end for;
end config_example2;
```

Example 13 shows the tool log output.

Example 13   Tool Log Output

```
dc_shell> elaborate config_example2
Running HDLC
Loading db file '/SYNOPSYS_ROOT_DIRECTORY/libraries/syn/standard.sldb'
Loading db file '/SYNOPSYS_ROOT_DIRECTORY/libraries/syn/gtech.db'
Loading db file '/SYNOPSYS_ROOT_DIRECTORY/libraries/syn/lsi_10k.db'
Component WORK.STRUCT/CONF_2_VHDL.STRUCT.C1 has been configured to use
the following implementation:
      Work Library: WORK
      Design Name:  A_BAR_B
      Architecture Name: STRUCT1
Component WORK.STRUCT/CONF_2_VHDL.STRUCT.C2 has been configured to use
the following implementation:
      Work Library: WORK
      Design Name:  A_BAR_B
      Architecture Name: STRUCT2
HDLC compilation completed successfully.
Information: Building the design 'A_BAR_B'. (HDL-193)
Running HDLC
HDLC compilation completed successfully.
Information: Building the design 'A_BAR_B'. (HDL-193)
Running HDLC
HDLC compilation completed successfully.
Current design is now 'conf_2_vhdl'
```

Architectures From Different Libraries

Example 14 uses configurations to bind two instances of the same component to different architectures in different libraries as follows:

- Bind architecture struct1 in library lib1 to an instantiation C1 of component a_bar_b
- Bind architecture struct2 in library lib2 to an instantiation C2 of component a_bar_b

Example 14   Using Architectures From Different Libraries

```
--File config3.1.vhdl

 entity a_bar_b is
    port(a, b: in bit; c: out bit);
end a_bar_b;

architecture struct1 of a_bar_b is
begin
    c <= not(a) and b;
end struct1;

--File config3.2.vhdl

entity a_bar_b is
    port(a, b: in bit; c: out bit);
```

```
end a_bar_b;

architecture struct2 of a_bar_b is
begin
    c <= a and not(b);
end struct2;

--File config3.3.vhdl

library lib1, lib2;
use lib1.all;
use lib2.all;

entity conf_3_vhdl is
    port(a, b: in bit; c: out bit);
end conf_3_vhdl;

architecture struct of conf_3_vhdl is
    component a_bar_b port(a, b: in bit; c :out bit); end component;

    signal a_not_b, not_a_b: bit;
begin
    C1: a_bar_b port map(a, b, not_a_b);
    C2: a_bar_b port map(a, b, a_not_b);

    c <= not_a_b or a_not_b;

end struct;

configuration config_example3 of conf_3_vhdl is
    for struct  -- of conf_3_vhdl
        for C1: a_bar_b
            use entity lib1.a_bar_b(struct1);
        end for;
        for C2: a_bar_b
            use entity lib2.a_bar_b(struct2);
        end for;
    end for;
end config_example3;
```

Example 15 shows the script file.

Example 15   Script File

```
sh mkdir ./lib1
sh mkdir ./lib2
define_design_lib lib1 -path ./lib1
define_design_lib lib2 -path ./lib2



analyze -format vhdl config3.1.vhdl -library  lib1
analyze -format vhdl config3.2.vhdl -library  lib2
analyze -format vhdl config3.3.vhdl
elaborate config_example3
```

Example 16 shows the tool log output.

Example 16 Tool Log Output

```
dc_shell> analyze -format vhdl config3.1.vhdl -library  lib1
Running HDLC
Input files:
/TEST_DIRECTORY/config3.1.vhdl
Compiling Entity Declaration A_BAR_B
Compiling Architecture STRUCT1 of A_BAR_B
HDLC compilation completed successfully.
1
dc_shell> analyze -format vhdl config3.2.vhdl -library  lib2
Running HDLC
Input files:
config3.2.vhdl
Compiling Entity Declaration A_BAR_B
Compiling Architecture STRUCT2 of A_BAR_B
HDLC compilation completed successfully.
1
dc_shell> analyze -format vhdl config3.3.vhdl
Running HDLC
Input files:
config3.3.vhdl
Compiling Entity Declaration CONF_3_VHDL
Compiling Architecture STRUCT of CONF_3_VHDL
Compiling Configuration CONFIG_EXAMPLE3 of CONF_3_VHDL
HDLC compilation completed successfully.
1
dc_shell> elaborate config_example3
Running HDLC
Loading db file '/SYNOPSYS_ROOT_DIRECTORY/libraries/syn/standard.sldb'
Loading db file '/SYNOPSYS_ROOT_DIRECTORY/libraries/syn/gtech.db'
Loading db file '/SYNOPSYS_ROOT_DIRECTORY/libraries/syn/lsi_10k.db'
Component WORK.STRUCT/CONF_3_VHDL.STRUCT.C1 has been configured to use
the following implementation:
      Work Library: LIB1
      Design Name:  A_BAR_B
      Architecture Name: STRUCT1
Component WORK.STRUCT/CONF_3_VHDL.STRUCT.C2 has been configured to use
the following implementation:
      Work Library: LIB2
      Design Name:  A_BAR_B
      Architecture Name: STRUCT2
HDLC compilation completed successfully.
Information: Building the design 'A_BAR_B'. (HDL-193)
Running HDLC
HDLC compilation completed successfully.
Information: Building the design 'A_BAR_B'. (HDL-193)
Running HDLC
```

```
HDLC compilation completed successfully.
Current design is now 'conf_3_vhdl'
```

Component Inside a Concurrent Block

Example 17 shows how to configure a component inside a concurrent block.

Example 17   Configuration With a Component Inside a Concurrent Block

```
entity my_or is
port (a, b: bit; z:out bit);
end entity my_or;

architecture beh of my_or is
begin
  z <= a or b;
end;

entity conf_4_vhdl is
  port(A, B: BIT; Z: out BIT);
end;

architecture BEH of conf_4_vhdl is
  component MY_AND
    port(A, B: BIT;
          Z: out BIT);
  end component;

begin
  Z <= A;
  L1: for I in 3 downto 0 generate
    L2: for J in I downto 0 generate
      L3: if J < I generate
        U1: MY_AND port map ( A, B, Z);
      end generate;
    end generate;
  end generate;
end;
configuration config_example4 of conf_4_vhdl is
    for beh
        for L1
          for L2
            for L3
              for U1: MY_AND
                use entity work.my_or (beh);
              end for;
            end for;
          end for;
        end for;
    end for;
end config_example4;
```

Example 18 shows the tool log output.

Example 18   Tool Log Output

```
dc_shell> elaborate config_example4
Running HDLC
Loading db file '/SYNOPSYS_ROOT_DIRECTORY/libraries/syn/standard.sldb'
Loading db file '/SYNOPSYS_ROOT_DIRECTORY/libraries/syn/gtech.db'
Loading db file '/SYNOPSYS_ROOT_DIRECTORY/libraries/syn/lsi_10k.db'
Component WORK.BEH/CONF_4_VHDL.BEH.L1.L2.L3.U1 has been configured to use
the following implementation:
      Work Library: WORK
      Design Name:  MY_OR
      Architecture Name: BEH
HDLC compilation completed successfully.
Information: Building the design 'MY_OR'. (HDL-193)
Running HDLC
HDLC compilation completed successfully.
Current design is now 'conf_4_vhdl'
```

Generic in a Configuration

Example 19 shows a component configuration used with a generic. Notice that the MY_AND component is configured to use the MY_OR (width = 8) implementation.

Example 19   Generic in a Configuration

```
entity my_or is
generic (width : integer);
port (a, b: in bit_vector (width - 1 downto 0);
         z: out bit_vector (width - 1 downto 0));
end entity my_or;

architecture beh of my_or is
begin
  z <= a or b;
end;
entity conf_5_vhdl is
  port(A, B: in BIT_VECTOR (7 downto 0);
        Z:   out BIT_VECTOR (7 downto 0));
end;

architecture BEH of conf_5_vhdl is
  component MY_AND
    generic (width: integer := 5);
    port(A, B: in BIT_VECTOR (width - 1 downto 0);
         Z: out BIT_VECTOR (width - 1 downto 0));
  end component;

begin
  Z <= A;
  L1: for I in 3 downto 0 generate
```

```
        U1: MY_AND port map ( A, B, Z);
  end generate;
end;

configuration config_example5 of conf_5_vhdl is
    for beh
        for L1
            for U1: MY_AND
              use entity work.my_or (beh) generic map (width => 8);
            end for;
        end for;
    end for;
end config_example5;
```

Example 20 shows the tool log output.

Example 20   Tool Log Output

```
dc_shell> elaborate config_example5
 Running HDLC
 Loading db file '/SYNOPSYS_ROOT_DIRECTORY/libraries/syn/
standard.sldb'
 Loading db file '/SYNOPSYS_ROOT_DIRECTORY/libraries/syn/gtech.db'
 Loading db file '/SYNOPSYS_ROOT_DIRECTORY/libraries/syn/lsi_10k.db'
 Component WORK.BEH/CONF_5_VHDL.BEH.L1.U1 has been configured to use
the following implementation:
       Work Library: WORK
       Design Name:  MY_OR
       Architecture Name: BEH
 HDLC compilation completed successfully.
 Information: Building the design 'MY_OR' instantiated from design
'conf_5_vhdl' with the parameters "width=8". (HDL-193)
 Running HDLC
 HDLC compilation completed successfully.
 Current design is now 'conf_5_vhdl'
```

Port Map in a Configuration

Example 21 uses a port map in the configuration.

Example 21   Port Map in a Configuration

```
library ieee;
use ieee.std_logic_1164.all;
use std.standard.time;

entity INVERTER is
port (IN1 : in BIT; OUT1 : out BIT);
end INVERTER;

architecture STRUCT_I of INVERTER is
  begin
```

```
        out1 <= not in1;
    end STRUCT_I;
entity CONFIG_TEST1_VHDL is end CONFIG_TEST1_VHDL;

architecture STRUCT_T of CONFIG_TEST1_VHDL is
    signal S1, S2 : BIT := '1';

component INV_COMP is
    port (IN_A : in BIT; OUT_A : out BIT);
end component;

    begin
       lh : inv_comp port map (S1, S2);
end STRUCT_T;

configuration CONFIG_INV of CONFIG_TEST1_VHDL is
for STRUCT_T
    for LH : INV_COMP
      use entity WORK.INVERTER (STRUCT_I)
      generic map (PropTime => TimeH)
      port map (IN1 => IN_A, OUT1 => OUT_A);
    end for;
end for;
end CONFIG_INV;
```

Nested Configurations

Example 22 uses a configuration inside a configuration.

Example 22   Nested Configurations

```
entity MY_AND is
port ( A, B      : in bit;
       O         : out bit);
end entity MY_AND;

architecture STRUCT1 of MY_AND is
begin
O <= A and B;
end STRUCT1;

entity MY_XOR is
port ( A, B      : in bit;
       O         : out bit);
end entity MY_XOR;

architecture STRUCT1 of MY_XOR is
component MY_AND is
port (A, B : in bit;
      O: out bit);
end component;
begin
```

```
  U1: MY_AND port map (A, B, O);
end STRUCT1;
architecture STRUCT2 of MY_XOR is
signal S1, S2, S3, S4 : bit;
begin
S1 <= A and (not B);
S2 <= (not A) and B;
O <= S1 or S2;
end STRUCT2;

entity CONFIG_FLOW_VHDL is
port (A1, A2, A3, A4, A5, B1, B2, B3, B4, B5     : in bit;
      O1, O2, O3, O4, O5                         : out bit);
end CONFIG_FLOW_VHDL;

architecture A1 of CONFIG_FLOW_VHDL is
component MY_XOR_COM is
port ( A, B      : in bit;
       O         : out bit);
end component;
component MY_AND is
port ( A, B      : in bit;
       O         : out bit);
end component ;
begin
U1 : MY_XOR_COM port map (A1, B1, O1);
U2 : MY_XOR_COM port map (A2, B2, O2);
U3 : MY_XOR_COM port map (A3, B3, O3);
U4 : MY_XOR_COM port map (A4, B4, O4);
V1 : MY_AND port map (A5, B5, O5);
end A1;

configuration TEST_CONFIG of MY_XOR is
  for STRUCT1
    for U1: MY_AND
      use entity WORK.MY_AND;
    end for;
  end for;
end TEST_CONFIG;

configuration MY_CONFIG of CONFIG_FLOW_VHDL is
use WORK.all;
for A1
  for U1, U2 : MY_XOR_COM
     use entity WORK.MY_XOR (STRUCT1);
  end for;
  for U3 : MY_XOR_COM
    use entity WORK.MY_XOR (STRUCT2);
  end for;
  for U4 : MY_XOR_COM
     use configuration WORK.TEST_CONFIG;
  end for;
  for V1 : MY_AND
```

```
      -- Use default
    end for;
end for;
end MY_CONFIG;
```

Indirectly Nested Configurations

A directly nested configuration is a nested configuration that configures its subdesign by using a “use configuration subconfigure” clause, as shown in Example 23; an indirectly nested configuration is a nested configuration that configures its subdesign by using a “for” clause, as shown in Example 24.

Example 23   Directly Nested Configuration

```
configuration conf_in_conf_configuration of conf_in_conf is
   for test
      for all : conf_test
         use configuration WORK.TEST_CONFIG; -- nested configuration
      end for;
   end for;
end conf_in_conf_configuration;
```

Example 24   Indirectly Nested Configuration

```
configuration conf_in_conf_configuration of conf_in_conf is
   for test
      for all : conf_test
         use entity work.conf_test(test);
         for test    -- here is the nested configuration
            for all : multi_and   -- component name multi_and
               use entity lib1.multiand(rtl); -- configured entity name
multiand
            end for;
         end for;
      end for;
   end for;
end conf_in_conf_configuration;
```

Embedded Configurations

The HDL Compiler tool supports embedded configurations. To enable this feature, set the `hdlin_enable_configurations` variable to `true`. Example 25 shows an embedded configuration.

Example 25   Embedded Configuration

```
entity Buf is
   port (Input_pin: in Bit;Output_pin: out Bit);
end Buf;
architecture DataFlow of Buf is begin
```

```
   Output_pin <= Input_pin;
end DataFlow;

entity Test_Bench is
end Test_Bench;
  architecture Structure of Test_Bench is
  component Buf is
  port (Comp_I: in Bit; Comp_O: out Bit);
  end component;
  -- A binding indication; generic and port map aspects within a
  -- binding indication associate actuals (Comp_I, etc.) with
  -- formals of the entity declaration (Input_pin, etc.):
  for UUT: Buf
  use entity Work.Buf(DataFlow)
  port map (Input_pin => Comp_I,
  Output_pin=> Comp_O);
  signal S1,S2: Bit;
  begin
  -- A component instantiation statement; generic and portmap aspects
  -- within a component instantiation statement associate actuals
  -- (S1, etc.) with the formals of a component (Comp_I, etc.):
  UUT: Buf
  port map (Comp_I => S1, Comp_O => S2);
  -- A block statement; generic and port map aspects within the
  -- block header of a block statement associates actuals (in this
  -- case, 4), with the formals defined in the block header:
   B: block
   begin
   end block;
end Structure;
```

You can use an embedded configuration in only one `for` construct in one component, as shown in Example 26. However, as a standalone configuration, you can use embedded configurations in multi-nested configurations and in several `for` keywords, as shown in Example 27.

Example 26   Embedded Configuration

```
for u1: embed_shift use entity work.embed_shift;
```

Example 27   Standalone Configuration

```
configuration embed_top_config of embed_top is
 for arch1
  for swap_exp1 : work.components.swap_exp
   use entity work.swap_exp(comb_seq)
    generic map (width1 => width + 1);
 for comb_seq
  for seq_gen
   for all: work.components.sync_async
    use entity work.sync_async(sync_logic)
    generic map (width2 => width1 + 1);
    …
```

To read designs with embedded configurations, read or elaborate the design using the entity name. To help understand how the tool processes embedded configurations, consider the design in Example 28. When you read the RTL using `read_vhdl` and `link`, the tool creates the log shown in Example 29 ; if you use `analyze` and `elaborate`, the tool creates the log shown in Example 30.

Note:

If you have multiple embedded architectures, you need to follow the usage guidelines described in Multiple Architectures in Embedded Configurations.

Example 28   Module config_simple_embed.vhd

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

entity embed_shift is
 generic (width: integer := 4);
 port (
  a : in bit_vector(0 to width - 1);
  b : in integer;
        o : out bit_vector(0 to width - 1)
 );
end entity embed_shift;

architecture tmp1 of embed_shift is
 begin
    o <= a sll  b;
end architecture tmp1;

entity mix_embed_config is
 port (
  a : in bit_vector(0 to 7);
  b : in integer;
        o : out bit_vector(0 to 7)
 );
end entity mix_embed_config;
architecture tmp of mix_embed_config is
 component my_shift is
 generic (width: integer := 8);
 port (
  a : in bit_vector(0 to width - 1);
  b : in integer;
  o : out bit_vector(0 to width - 1)
 );
 end component my_shift;
 for u1: my_shift use entity work.embed_shift;
-- use an embedded configuration because the component name is different
-- than the library name
 begin
 u1: my_shift
```

```
      port map (a, b, o);
end architecture tmp;
```

Example 29   HDL Compiler Report Log for read_vhdl and link

```
dc_shell> read_vhdl config.support_embedded.config_2.vhd
Loading db file '.../libraries/syn/lsi_10k.db'
Loading db file '.../libraries/syn/gtech.db'
Loading db file '.../libraries/syn/standard.sldb'
  Loading link library 'lsi_10k'
  Loading link library 'gtech'
Loading vhdl file './config_simple_embed.vhd'
Running HDLC
Compiling Entity Declaration EMBED_SHIFT
Compiling Architecture TMP1 of EMBED_SHIFT
Compiling Entity Declaration MIX_EMBED_CONFIG
Compiling Architecture TMP of MIX_EMBED_CONFIG
Component WORK.TMP/MIX_EMBED_CONFIG.TMP.U1 has been configured to use the
following implementation:
      Work Library: WORK
      Design Name:  EMBED_SHIFT
HDLC compilation completed successfully.
Current design is now './embed_shift.db:embed_shift'
Loaded 2 designs.
Current design is 'embed_shift'.
embed_shift mix_embed_config
dc_shell> link
  Linking design 'embed_shift'
  Using the following designs and libraries:
-------------------------------------------------------------------------

  embed_shift                  ./embed_shift.db
  lsi_10k (library)            .../libraries/syn/lsi_10k.db
  mix_embed_config             ./mix_embed_config.db
1
```

Example 30   HDL Compiler Report Log for Analyze and Elaborate With Entity Name

```
analyze -format vhdl config_simple_embed.vhd
Running HDLC
Compiling Entity Declaration EMBED_SHIFT
Compiling Architecture TMP1 of EMBED_SHIFT
Compiling Entity Declaration MIX_EMBED_CONFIG
Compiling Architecture TMP of MIX_EMBED_CONFIG
HDLC compilation completed successfully.
1
elaborate mix_embed_config
Loading db file '.../libraries/syn/gtech.db'
Loading db file '.../libraries/syn/standard.sldb'
  Loading link library 'gtech'
Running HDLC
Component WORK.TMP/MIX_EMBED_CONFIG.TMP.U1 has been configured to use the
following implementation:
      Work Library: WORK
      Design Name:  EMBED_SHIFT
HDLC compilation completed successfully.
Elaborated 1 design.
Current design is now 'mix_embed_config'.
Information: Building the design 'EMBED_SHIFT' instantiated from design
'mix_embed_config' with the parameters "width=8". (HDL-193)
HDLC compilation completed successfully.
1
```

Multiple Architectures in Embedded Configurations

If you define multiple architectures for an entity and you instantiate that entity without a specific binding to a specific architecture, the tool chooses the last architecture that is read as the architecture for that entity. Consider Example 31 in which the MY_AND entity is instantiated by the U2 component. The tool by default associates the last architecture read, STRUCT4, with U2 and generates a warning shown in Example 32.

To avoid this warning, you need to tell the tool which architecture to choose by specifying the binding in the embedded configuration. If you have several nested designs, you can create a standalone configuration to connect the correct working libraries to the appropriate components and configurations. In HDL Compiler, you can select a configuration identifier and use it to elaborate. To ensure your design intent is correctly read, elaborate with the configuration identifier, as described in the next section.

Example 31   Default Architecture for U2 Component is STRUCT4

```
entity MY_AND is
  port (O : out bit);
end entity MY_AND;

architecture STRUCT3 of MY_AND is
begin
  O <= '1';
end STRUCT3;

architecture STRUCT4 of MY_AND is
begin
  O <= '0';
end STRUCT4;


entity E1 is
  port (O : out bit);
end E1;

architecture A1 of E1 is
  component MY_AND is       -- same name as a previously found entity
    port(O : out bit);
  end component;
```

```
begin
  U2 : MY_AND port map (O);
end A1;
```

Example 32   Tool Warns When Multiple Architectures Are Associated With an Entity

```
dc_shell> analyze -format vhdl test.vhd
Loading db file '.../libraries/syn/gtech.db'
Loading db file '.../libraries/syn/standard.sldb'
  Loading link library 'gtech'
Loading vhdl file '.../test.vhd'
Running HDLC
Compiling Entity Declaration MY_AND
Compiling Architecture STRUCT3 of MY_AND
Compiling Architecture STRUCT4 of MY_AND
Compiling Entity Declaration E1
Compiling Architecture A1 of E1
Warning:  The entity 'MY_AND' has multiple architectures defined. The
last defined architecture 'STRUCT4' will be used to build the design
by default. (VHD-6)
```

Combinations of Embedded, Nested, and Standalone Configurations

In order for the tool to correctly read your design when it contains a combination of embedded, nested, and standalone configurations, you must elaborate with the configuration identifier instead of with the entity name. To help understand this requirement, consider the design in Example 34 through Example 38. In this design, if you want the tool to select the “swap_seq of swap_exp” architecture, you must elaborate the design using the configuration identifier, as shown in Example 33.

Example 33   Correct Way to Read the Design With Embedded, Nested, and Standalone Configurations - Use the Configuration Identifier

```
analyze -format vhdl config_nested_sync_async.vhd
analyze -format vhdl config_nested_swap_exp.vhd
analyze -format vhdl embed_nested_standalone_top.vhd
elaborate embed_top_config
```

If you elaborate with the entity name (or use `read_vhdl`), the tool chooses the last defined architecture it sees, which is “comb_seq of swap_exp” for this design. The tool does not see the configuration defined in embed_nest_standalone_top.vhd. When you use the configuration identifier (embed_top_config) to elaborate, the tool reads both the architecture and configuration code and uses the embedded and standalone configuration's declarations for its library and it chooses the correct components.

Example 34 Top Module in embed_nest_standalone_top.vhd

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

entity embed_top is
 -- generic (width: integer:= 3);
 generic (width: integer:= 2);
 port (
  clk : in std_logic;
  rst : in std_logic;
  din : in std_logic_vector(0 to width);
  d2 : in std_logic_vector(0 to width);

       dout : out std_logic_vector(0 to width)
 );
end entity embed_top;

architecture arch1 of embed_top is
 signal tmp_data1, tmp_data2 : std_logic_vector(0 to width);
 for swap_exp2 : work.components.swap_exp use entity
work.swap_exp(swap_seq)
generic map (width1 => 3);

begin
 swap_exp1 : component work.components.swap_exp
    port map (clk => clk, rst => rst, din1 => din, din2 => d2, dout =>
tmp_data1);
 swap_exp2 : component work.components.swap_exp
    port map (clk => clk, rst => rst, din1 => tmp_data1, din2 => d2, dout =>
tmp_data2);
 swap_exp3 : component work.components.swap_exp
    port map (clk => clk, rst => rst, din1 => tmp_data2, din2 => d2, dout =>
dout);
end architecture arch1;

configuration embed_top_config of embed_top is
   for arch1
       for swap_exp1 : work.components.swap_exp
    use entity work.swap_exp(comb_seq) generic map (width1 => width + 1);
    for comb_seq
        for seq_gen
            for all: work.components.sync_async
     use entity work.sync_async(sync_logic) generic map (width2 => width1
+ 1);
     for sync_logic
   for next_level : work.components.and_or
       use entity work.and_or(and_of_logic);
          end for;
     end for;
     end for;
        end for;
    end for;
       end for;

-- for swap_exp2 : is simple so it embeds on architecture

       -- for swap_exp3 : work.components.sync_async
  --   use entity work.sync_async(comb_seq);
       for swap_exp3 : work.components.swap_exp
    use entity work.swap_exp(comb_seq);

    for comb_seq
        for seq_gen
            for all : work.components.sync_async
         use entity work.sync_async(async_logic);
            end for;
        end for;
    end for;
       end for;
   end for;
end configuration embed_top_config;
```

Example 35   Submodule in config_nested_swap_exp.vhd

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

     entity swap_exp is
      -- generic (width1 : integer := 4);
      generic (width1 : integer := 3);
      port (clk : in std_logic;
            rst : in std_logic;
            din1 : in std_logic_vector(0 to width1 - 1);
            din2 : in std_logic_vector(0 to width1 - 1);

            dout : out std_logic_vector(0 to width1 - 1));
     end entity swap_exp;

     architecture swap_seq of swap_exp is
     signal int_data : std_logic_vector (0 to width1 - 1);
     begin
      swap : process (din1) is
      begin
         int_data(2) <= din1(0);
         int_data(1) <= din1(1);
         int_data(0) <= din1(2);
      end process swap;

      seq : process (clk, rst) is
      begin
         if clk'event and clk = '1' then
       if (rst ='0') then
           -- dout <= 0;
           dout <= (others => '0');
       else
           dout <= int_data;
       end if;
         end if;
      end process seq;
     end architecture swap_seq;
```

```
architecture comb_seq of swap_exp is
signal int_data : std_logic_vector (0 to 3);
begin
 comb : process (din1, din2) is
 begin
    int_data(2) <= not(din1(0) or din2(0));
    int_data(1) <= din1(2) xor din2(1);
    int_data(0) <= din1(1) and din2(2);
 end process comb;

 -- seq_gen: for i in 0 to 3 generate
 seq_gen: for i in 0 to 2 generate
 begin
       reg : component work.components.sync_async
       port map (clk => clk, rst => rst, d1 => int_data(i), d2 =>
int_data(i),
q => dout(i));
 end generate seq_gen;

end architecture comb_seq;
```

Example 36   Submodule in config_nested_sync_async.vhd

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


entity sync_async is
 generic (width2 : integer := 8);
 port (clk : in std_logic;
       rst : in std_logic;
       d1 : in std_logic;
       d2 : in std_logic;

       q : out std_logic);
end entity sync_async;

architecture sync_logic of sync_async is
    signal tmp_data1, tmp_data2 : std_logic_vector(0 to width2-1);
    begin
 next_level : component work.components.and_or
     port map (a => d1, b => d2,  o => tmp_data1(0));

 reg: process (clk, rst) is
 begin
    if (clk'event and clk ='1') then
  if (rst = '0') then
     q <= '0';
  else
     q <= tmp_data1(0);
  end if;
    end if;
 report ("in SYNC");
 end process reg;
end architecture sync_logic;

architecture async_logic of sync_async is
    signal tmp_data1, tmp_data2 : std_logic_vector(0 to width2-1);
    begin
 process (clk, rst) is
 begin
    if rst = '0' then
       q <= '0';
    elseif clk'event and clk = '1' then
       q <=  d1;
    end if;
 report ("in ASYNC");
 end process;
end architecture async_logic;
```

Example 37   Submodule in config_nested_and_or.vhd

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


entity and_or is
 port (a : in std_logic;
       b : in std_logic;
       o : out std_logic);
end entity and_or;

architecture and_of_logic of and_or is
 begin
    o <= a and b;
end architecture and_of_logic;

architecture or_of_logic of and_or is
 begin
    o <= a or b;
end architecture or_of_logic;
```

Example 38   Submodule in config_nested_pkg.vhd

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

package components is

   component and_or is
 port (a : in std_logic;
       b : in std_logic;
       o : out std_logic);
```

```
    end component and_or;

    component sync_async is
  port (clk : in std_logic;
        rst : in std_logic;
         d1 : in std_logic;
         d2 : in std_logic;
          q : out std_logic);
    end component sync_async;

    component swap_exp is
  port (clk : in std_logic;
        rst : in std_logic;
        din1 : in std_logic_vector(0 to 2);
        din2 : in std_logic_vector(0 to 2);
        dout : out std_logic_vector(0 to 2));
    end component swap_exp;

end package components;
```

Example 39   HDL Compiler Log Report

```
analyze -f vhdl config_nested_and_or.vhd
Running HDLC
Compiling Entity Declaration AND_OR
Compiling Architecture AND_OF_LOGIC of AND_OR
Compiling Architecture OR_OF_LOGIC of AND_OR
Warning:  The entity 'and_or' has multiple architectures defined. The
last defined
architecture 'or_of_logic' will be used to build the design by default.
(VHD-6)
HDLC compilation completed successfully.
1
analyze -f vhdl config_nested_pkg.vhd
Running HDLC
Compiling Package Declaration COMPONENTS
HDLC compilation completed successfully.
1
analyze -f vhdl config_nested_sync_async.vhd
Running HDLC
Compiling Entity Declaration SYNC_ASYNC
Compiling Architecture SYNC_LOGIC of SYNC_ASYNC
Compiling Architecture ASYNC_LOGIC of SYNC_ASYNC
Warning:  The entity 'sync_async' has multiple architectures defined. The
last defined
architecture 'async_logic' will be used to build the design by default.
(VHD-6)
HDLC compilation completed successfully.
1
analyze -f vhdl config_nested_swap_exp.vhd
Running HDLC
Compiling Entity Declaration SWAP_EXP
Compiling Architecture SWAP_SEQ of SWAP_EXP
Compiling Architecture COMB_SEQ of SWAP_EXP
Warning:  The entity 'swap_exp' has multiple architectures defined. The
last defined
architecture 'comb_seq' will be used to build the design by default.
(VHD-6)
```

```
HDLC compilation completed successfully.
1
analyze -f vhdl embed_nest_standalone_top.vhd
Running HDLC
Compiling Entity Declaration EMBED_TOP
Compiling Architecture ARCH1 of EMBED_TOP
Compiling Configuration EMBED_TOP_CONFIG of EMBED_TOP
HDLC compilation completed successfully.
1
elaborate embed_top_config
Loading db file '.../libraries/syn/gtech.db'
Loading db file '.../libraries/syn/standard.sldb'
  Loading link library 'gtech'
Running HDLC
Component WORK.ARCH1/EMBED_TOP.ARCH1.SWAP_EXP1 has been configured to use
the
following
implementation:
      Work Library: WORK
      Design Name:  SWAP_EXP
      Architecture Name: COMB_SEQ
Component WORK.ARCH1/EMBED_TOP.ARCH1.SWAP_EXP2 has been configured to use
the following implementation:
      Work Library: WORK
      Design Name:  SWAP_EXP
      Architecture Name: SWAP_SEQ
Component WORK.ARCH1/EMBED_TOP.ARCH1.SWAP_EXP3 has been configured to use
the following implementation:
      Work Library: WORK
      Design Name:  SWAP_EXP
      Architecture Name: COMB_SEQ
HDLC compilation completed successfully.
Elaborated 1 design.
Current design is now 'embed_top'.
Information: Building the design 'SWAP_EXP' instantiated from design
'embed_top' with
the parameters "width1=3". (HDL-193)
Component WORK.COMB_SEQ/SWAP_EXP.COMB_SEQ.SEQ_GEN.REG has been configured
to use the following implementation:

      Work Library: WORK
      Design Name:  SYNC_ASYNC
      Architecture Name: SYNC_LOGIC
      Configuration Name :
WORK.EMBED_TOP_CONFIG.ARCH1.SWAP_EXP1.COMB_SEQ.SEQ_GEN.REG
HDLC compilation completed successfully.
Information: Building the design 'SWAP_EXP' instantiated from design
'embed_top' with
the parameters "width1=3". (HDL-193)

Inferred memory devices in process
 in routine swap_exp_width13 line 28 in file
  '../vhdl_rtl/config_nested_swap_exp.vhd'.
============================================================================
===
|    Register Name     |    Type    | Width | Bus | MB | AR | AS | SR | SS | ST
 |
============================================================================
===
|      dout_reg        | Flip-flop |   3   |  Y  | N  | N  | N  | N  | N  | N
 |
============================================================================
===
HDLC compilation completed successfully.
Warning: Design 'swap_exp_width13' was renamed to 'swap_exp_width13_1' to
avoid
a conflict with another design that has the same name but different
parameters.
(LINK-17)
Information: Building the design 'SWAP_EXP' instantiated from design
'embed_top' with
the parameters "width1=3". (HDL-193)
Component WORK.COMB_SEQ/SWAP_EXP.COMB_SEQ.SEQ_GEN.REG has been configured
to use the
following implementation:
      Work Library: WORK
      Design Name:  SYNC_ASYNC
      Architecture Name: ASYNC_LOGIC
      Configuration Name :
HDLC compilation completed successfully.
Warning: Design 'swap_exp_width13' was renamed to 'swap_exp_width13_2' to
avoid a
conflict
with another design that has the same name but
different parameters. (LINK-17)
Information: Building the design 'SYNC_ASYNC' instantiated from design
'swap_exp_width13'
with the parameters "width2=4". (HDL-193)
Component WORK.SYNC_LOGIC/SYNC_ASYNC.SYNC_LOGIC.NEXT_LEVEL has been
configured to use
the following implementation:
      Work Library: WORK
      Design Name:  AND_OR
      Architecture Name: AND_OF_LOGIC
      Configuration Name :
'report' output: in SYNC

Inferred memory devices in process
 in routine sync_async_width24 line 22 in file
  '../vhdl_rtl/config_nested_sync_async.vhd'.
============================================================================
===
|    Register Name     |    Type    | Width | Bus | MB | AR | AS | SR | SS | ST
 |
============================================================================
===
|        q_reg         | Flip-flop |   1   |  N  | N  | N  | N  | N  | N  | N
 |
============================================================================
===
HDLC compilation completed successfully.
Information: Building the design 'SYNC_ASYNC' instantiated from design
'swap_exp_width13_2' with the parameters "width2=8". (HDL-193)
'report' output: in ASYNC

Inferred memory devices in process
 in routine sync_async_width28 line 38 in file
  '../vhdl_rtl/config_nested_sync_async.vhd'.
============================================================================
===
|    Register Name     |    Type    | Width | Bus | MB | AR | AS | SR | SS | ST
 |
=============================================================================
===
|        q_reg        | Flip-flop |   1   |  N   | N   | Y   | N   | N   | N   | N
 |
=============================================================================
===
HDLC compilation completed successfully.
Information: Building the design 'AND_OR'. (HDL-193)
HDLC compilation completed successfully.
1
```

Tool Behavior When Using Elaborate With the Entity Name

As noted in the previous sections, under certain conditions you need to elaborate using the configuration identifier. This section describes the problems that arise when you elaborate with the entity name.

Example 40 shows an incorrect way to read the design described in Example 34 through Example 38. Recall that for this design you want the tool to select the “swap_seq of swap_exp” architecture. An explanation of the tool behavior follows the examples.

Example 40   Incorrect Way to Read When Using the Standalone Configuration: Elaborate With the Entity Name

```
...
analyze -format vhdl config_nested_sync_async.vhd
analyze -format vhdl config_nested_swap_exp.vhd
analyze -format vhdl embed_nested_standalone_top.vhd
elaborate embed_top
...
```

Example 41   Incorrect Way to Read When Using Standalone Configurations: read_vhdl and link

```
...
read_vhdl config_nested_sync_async.vhd
read_vhdl config_nested_swap_exp.vhd
read_vhdl embed_nested_standalone_top.vhd
link
...
```

If you use the reading style in Example 40 or Example 41, HDL Compiler only checks the contents from the architecture_body in the embed_nest_standalone_top.vhd file. It ignores the rest of the RTL code in that module. In this case, it ignores the entire configuration portion of the top module even if you declare the configuration in the RTL.

The design architecture contains three components: swap_exp1, swap_exp2, and swap_exp3.

For the first component, swap_exp1, HDL Compiler ignores whatever you code in the configuration declarative section. Instead, the tool picks up the last defined architecture (comb_seq of swap_exp) by default and does not select the “swap_seq of swap_exp” architecture in the config_nested_swap_exp.vhd module, which is declared in the architecture statements part in the top module. Next, since the architecture “comb_seq of swap_exp” has its own library declared, the tool searches for the last defined architecture in the module of config_nested_sync_async.vhd, which is the architecture “async_logic of sync_async”, and uses it by default instead of other architectures in that module.

For the second component, swap_exp2, the tool is referred by the embedded configuration and links to the architecture “swap_seq of swap_exp” in the config_nested_swap_exp.vhd module, where it clearly states “use entity work.swap_exp(swap_seq)” as the embedded configuration in the architecture body.

For the swap_exp3 component, HDL Compiler chooses the last defined architecture “async_logic of sync_async” by default for the same reasons described previously for the swap_exp1 component.

Mixed Language Support

HDL Compiler supports Verilog and SystemVerilog subblock instantiations by using configured or direct methods. To enable this mixed language capability, set the `hdlin_vhdl_mixed_language_instantiation` and `hdlin_enable_configurations` variables to `true`. The default for both is `false`.

To illustrate mixed language support, consider a VHDL design A that contains two Verilog B designs from two different libraries. Example 42 shows the B instantiations. Example 43 shows a script that analyzes and elaborates the two Verilog subblocks into the VHDL top design.

Example 42   VHDL Subblocks Containing Verilog RTL

```
I1 : entity B_one.test
 port map(x => x1);
I2 : entity B_two.test
 port map(x => x2);
```

Example 43   Analyze and Elaborate Verilog Subblocks

```
define_design_lib work -path ./work
define_design_lib B_one -path ./d_one
define_design_lib B_two -path ./d_two
set hdlin_vhdl_mixed_language_instantiation true
analyze -format verilog -library B_one B_1.v
analyze -format verilog -library B_two B_2.v
analyze -format vhdl A.vhd
elaborate A
```

When configuring Verilog blocks in VHDL, note the following guidelines:

- The HDL Compiler tool checks ports for bit-width, not data type.
- All generics must be fully mapped.

- When a conflict occurs between a VHDL block and a Verilog block, the HDL Compiler tool chooses the VHDL block.
- The default parameter values cannot be used.

Design Libraries

This section contains the following subsections:

- Predefined Design Libraries
- Creating User-Defined Design Libraries
- User-Defined Design Library Example
- Using Design Units From Design Libraries
- Design Library Reports

Predefined Design Libraries

The following packages are analyzed for you:

- std_logic_1164
- std_logic_arith
- numeric_std
- std_logic_misc
- Standard package
- Synopsys ATTRIBUTES package

These packages are contained in the logical libraries IEEE, WORK, DEFAULT, and SYNOPSYS, which are defined during installation. Their default physical locations are defined in the `.synopsys_vss.setup` file, located in the Synopsys synthesis root installation subdirectory. These packages are described in Predefined Libraries.

Packages defined in these libraries can be used by your VHDL source code and are found automatically. Example 44 shows how to use the predefined std_logic_1164 package from the IEEE library.

Example 44   Using Predefined Libraries

```
library IEEE;
use IEEE.std_logic_1164.all;
...
```

Unlike all the other predefined packages, the Standard package does not require a use clause to enable your design to use the functions with the package.

Note:

These predefined packages are compatible only with the tool version they are released with; they are not compatible with other releases of the tool. A version number is stored in the intermediate format file, and the file can be read in only by the version in which it was created.

Creating User-Defined Design Libraries

When designs or packages are analyzed, the analyzed results are stored in the WORK library by default. If you want to store the results in a user-defined library, for example, MYLIB, with a physical location of MYLIB_LOC, you can use one of two methods. In the first method, use the `define_design_lib` command; in the second, you use the `.synopsys_vss.setup` file .

Table 4 Methods for Creating User-Defined Libraries

Method	Description

define_design_lib	Use the `define_design_lib` command to specify the library name and location.
	`define_design_lib MYLIB -path /MYLIB_LOC`
	Then analyze your design to MYLIB.
	`analyze -library MYLIB -format vhdl {{....design}}`

.synopsys_vss.setup	Add the user-specified library name and mapping to your `.synopsys_vss.setup` file.
	`MYLIB: ./ MYLIB_LOC`
	Then, analyze your design to MYLIB.
	`analyze -library MYLIB -format vhdl <{....design}>`

User-Defined Design Library Example

The following steps show you how to store the analysis of two packages in a user-defined library named COMMON-TLS and use these packages in the ALU design.

1. Define a logical library called “COMMON-TLS” and map it to a physical location, using the `define_design_lib` command.

   ```
   dc_shell> define_design_lib COMMON-TLS -path ./COMMON
   ```

2. Store the analysis of the package files in the user-defined library COMMON-TLS (The packages reside in the files types.vhd and functions.vhd).

```
dc_shell> analyze -format vhdl -library  COMMON-TLS
{types.vhd functions.vhd}
```

The `-library` option indicates the library name where the analyzed file should be stored.

Declare the COMMON-TLS library in the ALU code:

```
library COMMON-TLS;
use COMMON-TLS.types.all;
use COMMON-TLS.functions.all;
```

Example 45 shows the complete flow using the `analyze` and `elaborate` commands. (Design ALU is defined in the files ALU_entity.vhd and ALU_arch.vhd.)

Example 45   Flow for User-Defined Library Using analyze

```
dc_shell> define_design_lib WORK -path ./work
dc_shell> define_design_lib COMMON-TLS -path ./COMMON


dc_shell> analyze -format vhdl \
   -library  COMMON-TLS {types.vhdfunctions.vhd}
dc_shell> analyze -format vhdl {ALU_entity.vhd ALU_arch.vhd}
dc_shell> elaborate ALU_top
```

Example 46 shows the flow using the `read` command.

Example 46   Flow for User-Defined Library Using read

```
dc_shell> define_design_lib WORK -path ./work
dc_shell> define_design_lib COMMON-TLS -path ./COMMON
dc_shell> read_file -format vhdl \
   -library COMMON-TLS {types.vhd functions.vhd}
dc_shell> read_file -format vhdl {ALU_subblock.vhd ALU_top.vhd}
dc_shell> current_design ALU_top
dc_shell> link
```

Using Design Units From Design Libraries

Design libraries contain analyzed designs and packages used when you

- Elaborate designs

  During elaboration, subdesigns are first linked by a search through designs in memory. HDL Compiler then searches the current design library for preexisting analyzed files of the subdesigns. The .db files in the search path are also automatically linked during elaboration of a top-level design. See the `elaborate` man page for syntax details.

- Instantiate design units

  Design units from design libraries can be instantiated into other designs. For example, you can instantiate the design adder by using

  ```
  U1: entity adder (adder_arch)
      generic map (N => 16)
      port map (A,B,Z);
  ```

- Call a package in a library with the use clause

  The use clause allows an entity to use a package from a library. Reference these packages in the declaration section of the entity description.

Design Library Reports

To get a complete list of design libraries, use the `report_design_lib -libraries` command. To view the contents of an individual library, such as the IEEE library, use the `report_design_lib IEEE` command. To find out a library's physical location, use the `get_design_lib_path` command.

For more information about these commands, see the man pages.

Package Support

HDL Compiler supports the following packages:

- IEEE Package—std_logic_1164
- IEEE Package—std_logic_arith
- IEEE Package—numeric_std (IEEE Standard 1076.3)
- IEEE Package—std_logic_misc
- Standard Package
- Synopsys Package—ATTRIBUTES

For more information on these packages, see Predefined Libraries.

Array Naming Variables

The three variables described in Table 5 affect how array elements are named. To list their current values, run the following command:

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

Table 5 Array Naming Variables

Variable	Description
`bus_naming_style`	Describes how to name the bits in port, cell, and net arrays. When a multiple-bit array is read in, it is converted to a set of individual single-bit names. The value is a string containing the characters %s and %d, which are replaced by the array name and the bit (element) index, respectively. The default is "%s[%d]".Example: Array X_ARRAY is indexed from 0 to 7 and has bus_naming_style = "%s.%d." HDL Compiler names the third element of X_ARRAY as X_ARRAY.2. Note: It is recommended that you do not change this default if the netlist is written out in Verilog format.
`bus_dimension_separator_style`	Specifies the style to use in separating multidimensional array indexes. The default is "][".
`bus_minus_style`	Describes how to represent negative indexes in port, cell, and net names. The value is a string containing the characters %d (replaced by the absolute value of a negative index). The default is "–%d".Example: If bus_minus_style = "M%d", the index value of negative 3 is represented as "M3."

Supporting Waived Messages

VC SpyGlass Lint provides a set of rules to report synthesis, simulation, connectivity, and structural checks. The HDL Compiler tool, also provides similar checks. This might cause you to analyze the same message again. The VC SpyGlass Waiver Reuse feature enables the waivers generated in VC SpyGlass to be used across the HDL Compiler tool to prevent from reanalyzing the same message.

This section provides the details for reporting waived messages using the WDF file:

- Using the WDF File
- Reporting Waived Messages Using WDF Files
- Use Model for HDL Compiler tool Run
- Example Output From the HDL Compiler tool
- Comparing Reports

Using the WDF File

The `set_waiver` command manages the waiver suppression flow. The `wdf file list`, `reset`, and `print` options are mutually exclusive. After generating the WDF file, you can specify the file as an input for the HDL Compiler tool.

Use this command to provide the configuration for waiver data file generation.

Syntax

```
set_waiver
         {[<wdf file list>]}
         [-report <filename>]
         [-reset]
         [-print <summary|detailed>]
```

Arguments

Argument	Description
`wdf file list`	Specifies the list of waiver data files to process. Enables the waiver mode when a valid WDF file list is given as input.
`-report` *`filename`*	Records the waived messages in the specified file name.
`-reset`	Resets the waiver mode and all the waiver data. The `-reset` option writes out the waived messages to the specified default waived messages file
`-print summary|detailed`	Generates the information for current waiver configuration and WDF data. Since the object level data can be large, specify the required value as summary or detailed.

Examples

The following command specifies the WDF file `wdf.json.en.gz` as input and maps the waived messages to the `report_wdf.log` file.

```
set_waiver {wdf/wdf.json.en.gz} -report report_wdf.log
```

Reporting Waived Messages Using WDF Files

The `report_waived_messages` prints out the suppressed message list. The `report_waived_messages` command can be invoked multiple times and can also be issued between different commands.

Use this command to specify the waived messages.

Syntax

```
report_waived_messages
          [-log <filename>]
          [-status <waived|unwaived|both>]
```

Arguments

Argument	Description
`-log` *`filenam`*	Prints out the suppressed message list in the specified file name.
`-status` *`waived|unwaived|both`*	Specifies the category of messages to be displayed. The status of the suppressed message list can be specified as `waived`, `unwaived`, or `both`.

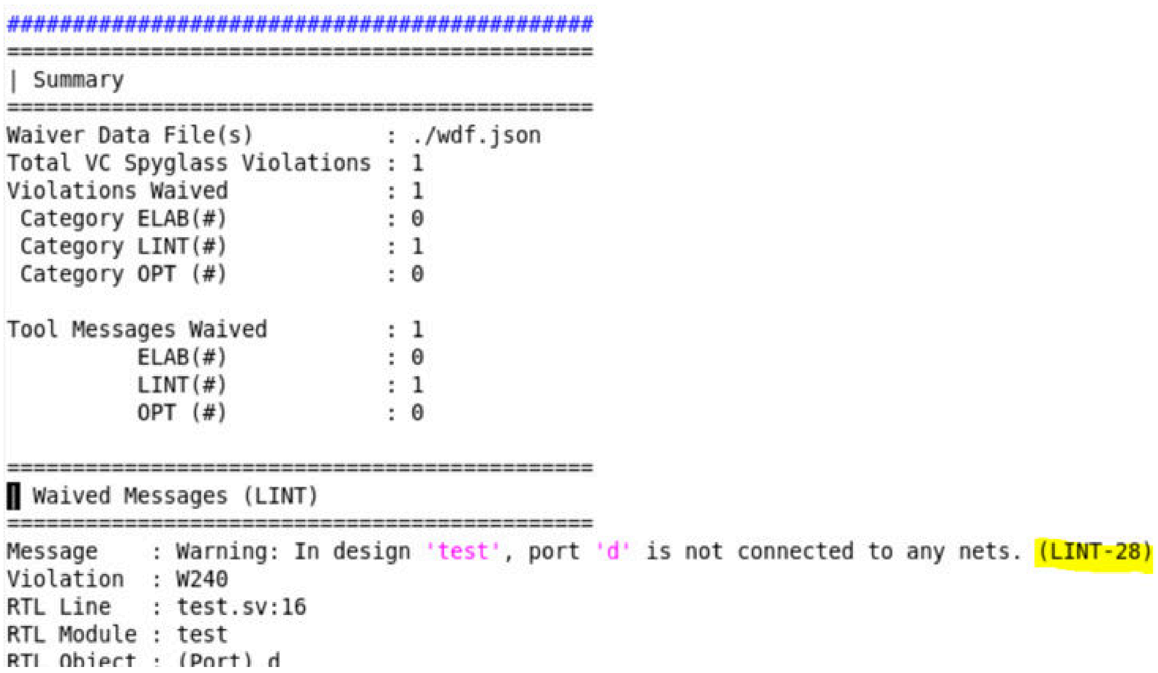
Use Model for HDL Compiler tool Run

Use the following Tcl file for generating the WDF file in the HDL Compiler tool:

```
#### Specify the list of WDF files generated in VC SpyGlass
set_waiver {wdf/wdf.json.en.gz} -report report_wdf.log

#### Design Read
analyze -f sverilog { top.v }
elaborate top

#### Check Design, Compile Ultra
check_design
compile_ultra
check_design

#### Report the waived messages (optional)
report_waived_messages -log report_waiver_message.log -status both
```

Example Output From the HDL Compiler tool

The following figure displays the output generated from the HDL Compiler tool after applying waivers to the mapped messages:

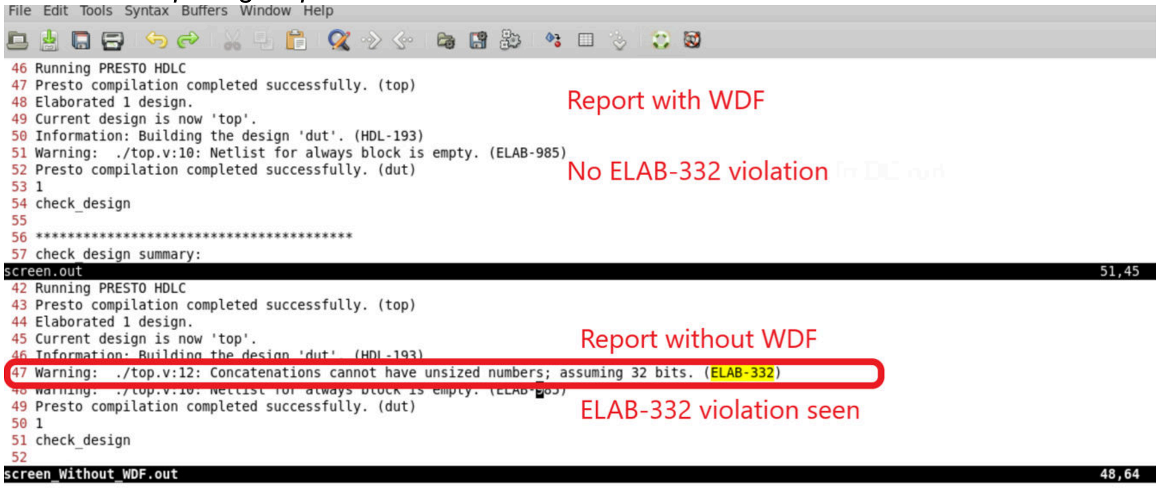


```
################################################
================================================
| Summary
================================================
Waiver Data File(s)          : ./wdf.json
Total VC Spyglass Violations : 1
Violations Waived            : 1
 Category ELAB(#)            : 0
 Category LINT(#)            : 1
 Category OPT (#)            : 0

Tool Messages Waived         : 1
         ELAB(#)             : 0
         LINT(#)             : 1
         OPT (#)             : 0

================================================
| Waived Messages (LINT)
================================================
Message     : Warning: In design 'test', port 'd' is not connected to any nets. (LINT-28)
Violation   : W240
RTL Line    : test.sv:16
RTL Module  : test
RTL Object  : (Port) d
```

Comparing Reports

The following figure illustrates the difference in the reports after the waiver data is shared with the HDL Compiler tool using a WDF file:

Figure 3 Comparing Reports



```
Running PRESTO HDLC
Presto compilation completed successfully. (top)
Elaborated 1 design.
Current design is now 'top'.
Information: Building the design 'dut'. (HDL-193)
Warning:  ./top.v:10: Netlist for always block is empty. (ELAB-985)
Presto compilation completed successfully. (dut)
1
check_design

****************************************
check_design summary:
screen.out                                                          51,45
Running PRESTO HDLC
Presto compilation completed successfully. (top)
Elaborated 1 design.
Current design is now 'top'.
Information: Building the design 'dut'. (HDL-193)
Warning:  ./top.v:12: Concatenations cannot have unsized numbers; assuming 32 bits. (ELAB-332)
Warning:  ./top.v:10: Netlist for always block is empty. (ELAB-985)
Presto compilation completed successfully. (dut)
1
check_design

screen_Without_WDF.out                                              48,64
```

Licenses

The 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	Not applicable	Not applicable
VHDL netlist reader	Not applicable	No	No	No

2

General Coding Considerations

This chapter discusses coding issues specific to HDL Compiler, in the following sections:

Declarative Region in generate Statements

HDL Compiler allows declarations within generate statements. Each iteration of the generate statement declares new copies of the objects in the declarative region. Consider Example 47, which describes four AND gates and four inverters. This code produces four independent signals named “S”. HDL Compiler distinguishes these signals using a user-specified naming convention. This convention is determined by the `hdlin_generate_naming_style` and `hdlin_generate_separator_style` variables. See the man pages for information about these variables.

Example 47   Signal Declarations Within generate Statements

```
G: for I in 0 to 3 generate
  signal S: BIT;
 begin
  S <= A(I) and B(I);
  Z(I) <= not S;
 end generate;
```

Example 48 shows that you can declare objects other than signals in a generate statement.

Example 48   Function Declarations Within Generate Statements

```
G: for I in 0 to 3 generate
  function F (X: in BIT_VECTOR(0 to I)) return BIT is
   variable R: BIT := '1';
  begin
   for J in X'RANGE loop
    R := R and x(j);
   end loop;
   return R;
  end function;
 begin
  z(i) <= f (a(0 to i));
 end generate;
```

Here, four versions of the function “f” are created, one for each iteration of the generate loop. Because the function is declared in the generate declarative region, it can only be called from the generate body.

Design Units

Design unit requirements specific to HDL Compiler are discussed in the following subsections:

- Direct Instantiation of Components
- Default Component Port Assignments
- Component Name Restrictions
- Component Sources
- Component Port Consistency
- Instantiating Technology-Independent Components
- Component Architecture
- Package Names
- Procedures and Functions as Design Components

Direct Instantiation of Components

HDL Compiler allows components to be directly instantiated in the design without a component declaration. This is a VHDL-93 feature that provides a more concise method of instantiating subdesigns. The following notation is used:

```
instance_label: entity entity_name (architecture_name)
                                   generic map (...) port map (...);
```

HDL Compiler always picks the last architecture analyzed for synthesis. In the following examples, design1 in Example 49 is analyzed and saved in a library called DESIGN1_LIB and design2 instantiates design1, as shown in Example 50.

Example 49   Design 1 Can Be Instantiated by Design 2

```
-- design1 is contained in the design1.vhd file
-- ============================================
entity design1 is
  port (a: in bit;
        z: out bit);
end;

architecture rtl of design1 is
begin
  z <= a;
end;
```

HDL Compiler supports the direct instantiation of design1 without a component declaration as shown in Example 50. Notice that design2 now requires fewer lines of code.

Example 50   Design 2 Instantiates Design 1

```
-- design2 is contained in the design2.vhd file
-- ====================================================
library DESIGN1_LIB;
use DESIGN1_LIB.all;

entity design2 is
  port (b: in bit;
        y: out bit);
end;

architecture rtl of design2 is
begin
  u1: entity DESIGN1_LIB.design1(rtl)
      port map (a => b, z => y);
end;
```

Default Component Port Assignments

HDL Compiler supports the use of default assignments for component port declarations as shown in Example 51. This simplifies coding by allowing ports with default assignments to be omitted during component instantiation.

Note:

Default assignments for entity port declarations is parsed but ignored by HDL Compiler.

Example 51   Default Port Assignments

```
component AND3 is
  port (A: in bit;
        B: in bit;
        C: in bit:= '1'; -- default assignment to component port
        Z : out bit
       );
end component;
.
.
.
  U1 : AND3
     port map (A => A1,
               B => B1,
               Z => Z1
              );
```

Component Name Restrictions

You cannot name components with keywords, identifiers from any Synopsys or IEEE package, or the GTECH_ prefix.

Component Sources

A declared component can come from

- The same VHDL source file
- A different VHDL source file
- Another format, such as EDIF, state table, or programmable logic arrays (PLAs)
- A component from a logic library
- A Verilog source file

A component that is not in one of the current VHDL source files must already have been compiled by HDL Compiler and must reside either in memory or in a .db file in the search path. HDL Compiler searches for previously compiled components by name, in the following order:

1. In the current design in memory.
2. In the directories and files identified in the Design Compiler link path (`link_library` variable). These files can include previously compiled designs or libraries of technology-specific components.
3. In the directories and files identified in the HDL Compiler search path (`search_path` variable).

Component Port Consistency

HDL Compiler checks for consistent port mapping between all loaded designs. For RTL designs that are not VHDL or Verilog, the port names are taken from the original design description, as follows:

- For PLAs or state tables, the port names are the input and output names.
- For components in a logic library, the port names are the input and output pin names.
- For EDIF designs, the port names are the EDIF port names.

  The bit-widths of each port must match.
- For VHDL components, HDL Compiler verifies matching.
- For components from other sources, HDL Compiler checks matching when linking the component to the VHDL description.

Instantiating Technology-Independent Components

You can directly instantiate GTECH components in your RTL. The GTECH library contains the following technology-independent logical components:

- AND, OR, and NOR gates (2, 3, 4, 5, and 8)
- 1-bit adders and half adders
- 2-of-3 majority
- Multiplexers
- Flip-flops and latches
- Multiple-level logic gates, such as AND-NOT, AND-OR, and AND-OR-INVERT

Note:

Instantiating GTECH components should be used with caution because it restricts the optimization of logic and might result in a degradation of design quality of results (QoR).

When you instantiate GTECH components, you can set the `map_only` attribute to prevent HDL Compiler from ungrouping the GTECH component and selecting a similar cell from the target library. When this attribute is applied, HDL Compiler does not optimize the gates; instead, the gates are only mapped to the target technology. The `set_map_only` command in Example 52 sets the `map_only` attribute on each cell returned by the `find` command (all cells in the design RIPPLE_CARRY that reference a GTECH_ADD_ABC cell). If you use your own library with attributes already set in that library, you do not have to set the `map_only` attribute.

Example 52   GTECH Component Instantiation

```
library GTECH;
library ieee;
use IEEE.STD_LOGIC_1164.all;
use gtech.gtech_components.all;

entity RIPPLE_CARRY is
  generic(N: NATURAL);

  port(a, b : in std_logic_vector(n-1 downto 0);
      carry_in: in std_logic;
      sum : out std_logic_vector(n-1 downto 0);
      carry_out: out std_logic);
end RIPPLE_CARRY;

architecture TECH_INDEP of RIPPLE_CARRY is

  signal CARRY: std_logic_vector(N downto 0);

-- synopsys dc_tcl_script_begin
-- set_map_only [get_cells * -filter "ref_name==GTECH_ADD_ABC"]
-- synopsys dc_tcl_script_end
begin
  CARRY(0) <= CARRY_IN;

  GEN: for I in 0 to N-1 generate
    U1: GTECH_ADD_ABC port map(
             A          => A(I),
             B          => B(I),
             C      => CARRY(I),
             S       => SUM(I),
             COUT      => CARRY(I+1)
            );

  end generate GEN;

  CARRY_OUT <= CARRY(N);
end TECH_INDEP;
```

To link this design in HDL Compiler, you must have the GTECH.db library in your `link_library` variable.

Component Architecture

HDL Compiler uses the following two rules to select which entity and architecture to associate with a component instantiation:

1. Each component declaration must have an entity—a VHDL entity, a design entity from another source or format, or a library component—with the same name. This entity is used for each component instantiation associated with the component declaration.
2. If a VHDL entity has more than one architecture, HDL Compiler uses the last architecture analyzed. You can override this selection by using configurations. For more information on configuration, see Configuration Support.

Package Names

Synopsys supports different packages with the same name if they exist in different libraries.

Procedures and Functions as Design Components

Procedures and functions are represented by gates and cannot exist as entities (components), unless you use the directive `map_to_entity`, which causes HDL Compiler to implement a function or a procedure as a component instantiation. Procedures and functions that use `map_to_entity` are represented as components in designs where they are called.

When you add a `map_to_entity` directive to a subprogram definition, HDL Compiler assumes the existence of an entity with the identified name and the same interface.

HDL Compiler does not check this assumption until they link the parent design. The matching entity must have the same input and output port names. If the subprogram is a function, you must also provide a `return_port_name` directive where the matching entity has an output port of the same name.

These two directives are called component implication directives:

```
-- synopsys map_to_entity    entity_name
-- synopsys return_port_name port_name
```

Insert these directives after the function or procedure definition, as in the following example:

```
  function MUX_FUNC(A,B: in TWO_BIT; C: in BIT)
      return TWO_BIT is

-- synopsys map_to_entity MUX_ENTITY
-- synopsys return_port_name Z
...
```

The behavioral description of the subprogram is not checked against the functionality of the entity overloading it. If there are differences in functionality between them, pre-synthesis and post-synthesis simulation results might not match.

Overloaded as used here refers to an entity that can have more than one function or definition. In this case, the `MUX_FUNC` function is overloaded by the MUX_ENTITY entity. The Synopsys `map_to_entity` directive causes HDL Compiler to ignore the contents of the `MUX_FUNC` function and directly synthesize the contents of the MUX_ENTITY entity. However, simulation uses the contents of the function. The function is overloaded because it has two definitions: function contents and the entity. These definitions should be the same to avoid a mismatch between synthesis and simulation results.

The matching entity (*entity_name*) does not need to be written in VHDL. It can be in any format HDL Compiler supports.

When HDL Compiler encounters the `map_to_entity` directive, it parses but ignores the contents of the subprogram definition.

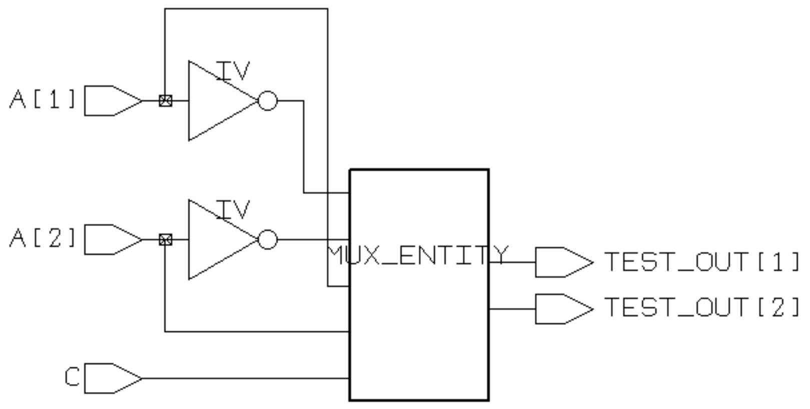
Example 53 shows a function that uses component implication directives. Figure 4 illustrates the corresponding design.

Example 53 Using Component Implication Directives on a Function

```
package MY_PACK is
 subtype TWO_BIT is BIT_VECTOR(1 to 2);
 function MUX_FUNC(A,B: in TWO_BIT; C: in BIT) return
  TWO_BIT;
end;

package body MY_PACK is

 function MUX_FUNC(A,B: in TWO_BIT; C: in BIT) return
  TWO_BIT is

 -- synopsys map_to_entity MUX_ENTITY
 -- synopsys return_port_name Z

 -- contents of this function are ignored but should match the
 -- functionality of the entity MUX_ENTITY, so pre- and post
 -- simulation will match
 begin
  if(C = '1') then
   return(A);
  else
   return(B);
  end if;
 end;
end;

use WORK.MY_PACK.ALL;
entity TEST is
 port(A: in TWO_BIT; C: in BIT; TEST_OUT: out TWO_BIT);
```

```
end;

architecture ARCH of TEST is
begin
 cal_func: process (a, c)
 begin
  TEST_OUT <= MUX_FUNC(not A, A, C);
                                -- Component implication call
 end process;
end;

use WORK.MY_PACK.ALL;

-- the following entity 'overloads' the function MUX_FUNC above

entity MUX_ENTITY is
 port(A, B: in TWO_BIT; C: in BIT; Z: out TWO_BIT);
end;

architecture ARCH of MUX_ENTITY is
begin
 process (a, b)
 begin
  case C is
   when '1' => Z <= A;
   when '0' => Z <= B;
  end case;
 end process;
end;
```

Figure 4 Schematic Design With Component Implication Directives



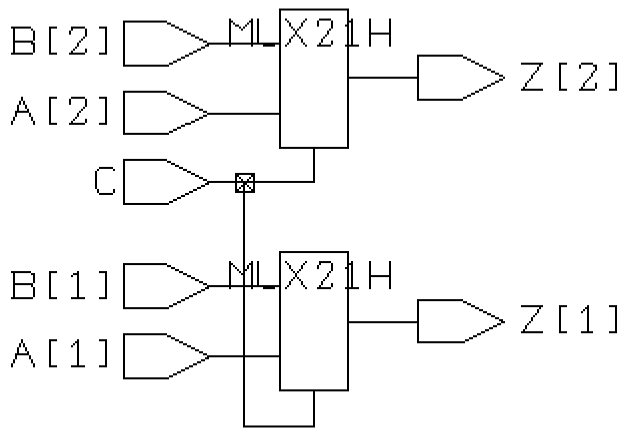
Example 54 shows the same design as Example 53, except that an entity is not created for the function. The component implication directives have been removed. Figure 5 illustrates the corresponding design.

Example 54 Using Gates to Implement a Function

```
package MY_PACK is
 subtype TWO_BIT is BIT_VECTOR(1 to 2);
 function MUX_FUNC(A,B: in TWO_BIT; C: in BIT)
```

```
  return TWO_BIT;
end;

package body MY_PACK is

 function MUX_FUNC(A,B: in TWO_BIT; C: in BIT)
  return TWO_BIT is
 begin
  if(C = '1') then
   return(A);
  else
   return(B);
  end if;
 end;
end;

use WORK.MY_PACK.ALL;
entity TEST is
 port(A: in TWO_BIT; C: in BIT; Z: out TWO_BIT);
end;

architecture ARCH of TEST is
begin
 process (a, c)
 begin
  Z <= MUX_FUNC(not A, A, C);
 end process;
end;
```

Figure 5 Schematic Design Without Component Implication Directives



Data Types and Data Objects

Data type and data object requirements specific to HDL Compiler are discussed in the following subsections:

- Globally Static Expressions in Port Maps
- Aliases
- Deferred Constants
- Aggregates in Constant Record Declarations
- Enumerated Types in the for and for-generate Constructs
- Groups
- Integer Data Types
- Overloading an Enumeration Literal
- Enumeration Encoding
- Constant Floating-Point Support
- math_real Package Support

Globally Static Expressions in Port Maps

HDL Compiler supports globally static expressions in port maps as shown in Example 55.

Example 55 HDL Compiler Supports Globally Static Expressions in Port Maps

```
component C is
  port (A, B: in BIT; Z: out BIT);
 end component;
 . . .
 signal X, Y: BIT
 . . .
 U1: C port map (X, '1', Y);
```

Aliases

HDL Compiler supports all alias types except labels, loop parameters, and generate parameters—these cannot be aliased per the VHDL language reference manual. Example 56 shows alias code that is supported in HDL Compiler.

Example 56 Support for Alias in HDL Compiler

```
entity e is
  port (a, c: in bit;
        z: out bit);
end;

architecture a of e is
  alias b is c;
begin
  z <= a and b;
end;
```

HDL Compiler supports the following alias types:

- Aliases without an explicit subtype indication as shown in Example 57
- Aliases with an explicit subtype indication
- Aliases to non-objects (that is, types) as shown in Example 58
- Aliases to subprograms as shown in Example 59

Example 57 Alias Without an Explicit Subtype Indication

```
signal S: BIT_VECTOR (0 to 7);
 ...
 alias A is S;
```

In this case, A has the same type as S.

Example 58 Alias to a Type

```
alias SLV is STD_LOGIC_VECTOR;
```

Example 59 shows an alias to a subprogram. This is a convenient way to refer to a specific subprogram in a package.

Example 59 Aliases to Subprograms

```
library IEEE;
 use IEEE.STD_LOGIC_1164.all;
 use IEEE.STD_LOGIC_SIGNED.all;
 use IEEE.STD_LOGIC_UNSIGNED.all;
 . . .
 alias    SLV_TO_SINT is
              STD_LOGIC_SIGNED.CONV_INTEGER;
 alias    SLV_TO_UINT is
              STD_LOGIC_UNSIGNED.CONV_INTEGER;
```

Subprogram aliases can also contain a signature. This makes it possible to distinguish among the various interpretations of an overloaded subprogram name as shown in Example 60.

Example 60 Subprogram Alias Containing a Signature

```
library IEEE;
 use IEEE.STD_LOGIC_1164.all;
 . . .
 alias   BV_TO_SLV is To_StdLogicVector
              [BIT_VECTOR return STD_LOGIC_VECTOR];

 . . .
 x <= BV_TO_SLV ("1101");
```

Deferred Constants

Constants defined in packages are useful for declaring a set of global design parameters that can be shared by multiple design entities. Example 61 shows how a constant declared in a global package DEFS is used to define the active edge for the clocks in the design. The value of “1” means that the flip-flops is clocked on the rising edge of the clock.

Example 61 shows a deferred constant declaration in a package.

Example 61 Deferred Constants

```
-- defs_pkg.vhd
-- ============
library ieee;
use ieee.std_logic_1164.all;

package defs is
constant CLOCK_ACTIVE_EDGE: std_logic := '1';
end;

design1.vhd
===========
library ieee, mylib;
use ieee.std_logic_1164.all;
use mylib.defs.all;

...
  process (clk) is
  begin
     if (clk'event and clk = CLOCK_ACTIVE_EDGE) then
        Q <= D;
     end if;
  end process;
...
```

One of the limitations of normal constant declarations in packages is that if the constant value in the package is changed, then all the designs making use of that package must be reanalyzed to use the new constant value. In the previous example, this means that if you want to change to a negative clock edge, you need to modify the CLOCK_ACTIVE_EDGE from “1” to “0” in defs_pkg.vhd and reanalyze all the files that reference this package.

With deferred constants, the constant is declared in the package without initializing its value. The initialization of the constant is deferred to the package body declaration. Now if the constant value is changed in the package body, only the package body needs to be reanalyzed, followed by a re-elaboration of the top-level design. Example 62 shows how a deferred constant declaration can be used to define the active edge for the clocks in the design.

Example 62 Deferred Constant Declaration

```
defs_pkg.vhd
============
package DEFS is
constant CLOCK_ACTIVE_EDGE: std_logic;
end DEFS;


defs_pkg_body.vhd
=================
package body DEFS is
constant CLOCK_ACTIVE_EDGE: std_logic := '1';
end;
```

Now if you want to change from a falling-edge-triggered to a rising-edge-triggered behavior, you only need to modify and reanalyze the package body, in defs_pkg_body.vhd, and, re-elaborate the top-level design to implement the change. This allows for a more flexible and manageable design flow.

Aggregates in Constant Record Declarations

HDL Compiler supports the use of aggregates in constant record declarations as shown in Example 63.

Example 63 Aggregates in Constant Record Declarations

```
type rec_type is
        record
            A   : bit_vector(1 downto 0);
            B   : bit_vector(1 downto 0);
        end record;

constant reset_rec   : rec_type := (
                                     A => (others => '0'),
                                     B => (others => '0')
                                  );
```

Enumerated Types in the for and for-generate Constructs

HDL Compiler supports the use of enumerated types as indexes in the for and for-generate constructs. Example 64 uses an enumerated type as an index in a for loop.

Example 64 Enumerated Types As Index in for Construct

```
package Defs is
type color is (RED, GREEN, BLUE);
subtype col_val is bit_vector (7 downto 0);
type pixel is array (color range RED to BLUE) of col_val;
function pix_fn (A1, A2: col_val) return col_val;
end Defs;

package body Defs is
  function pix_fn (A1, A2: col_val) return col_val is
  begin
    return (A1 xor A2);
  end pix_fn;
end Defs;

use work.Defs.all;
entity pix is
  port (A, B: in pixel;
        Z: out pixel
       );
end pix;

architecture rtl of pix is
begin
  process (A, B)
  begin
    for I in RED to BLUE loop    -- enumerated type used here
      Z(I) <= pix_fn(A(I),B(I));
    end loop;
  end process;
end rtl;
```

Groups

HDL Compiler supports VHDL-93 group declarations as shown in Example 65. This feature allows you to create groups of named entities. One useful application of this feature is that you can apply attributes to the group as a whole instead of referencing individual signals.

Example 65 Group Declarations

```
package Defs is
group sig_grp is (signal<>);
end Defs;

library Synopsys;
use Synopsys.attributes.all;
use work.Defs.all;

entity top is
port (A, B: in bit;
      Z: out bit
     );
end top;

architecture RTL of top is

group sig3_grp is (signal,signal,signal);

group inputs: sig_grp (A, B);
group all_ports: sig3_grp (A, B, Z);

-- input delay of 1.5 will be applied to A & B signals
attribute ARRIVAL of inputs: group is 1.5;

begin
  Z <= A or B;
end RTL;
```

Integer Data Types

Multidigit numbers in VHDL can include underscores (_) to make them easier to read.

HDL Compiler encodes an integer value as a bit vector whose length is the minimum necessary to hold the defined range and encodes integer ranges that include negative numbers as 2's-complement bit vectors.

Overloading an Enumeration Literal

You can overload an enumeration literal by including it in the definition of two or more enumeration types. When you use such an overloaded enumeration literal, HDL Compiler is usually able to determine the literal's type. However, under certain circumstances, determination might be impossible. In such cases, you must qualify the literal by explicitly stating its type. Example 66 shows how you can qualify an overloaded enumeration literal.

Example 66   Enumeration Literal Overloading

```
type COLOR is (RED, GREEN, YELLOW, BLUE, VIOLET);
type PRIMARY_COLOR is (RED, YELLOW, BLUE);
signal A : COLOR;
...
A <= COLOR'(RED);
```

Enumeration Encoding

Enumeration literals are synthesized into the binary equivalent of their positional value. By default, the first enumeration literal is assigned the value 0, the next enumeration literal is assigned the value 1, and so forth.

HDL Compiler automatically encodes enumeration values into bit vectors that are based on each value's position. The length of the encoding bit vector is the minimum number of bits required to encode the number of enumerated literals. For example, an enumeration type with five values would have a 3-bit encoding vector.

Example 67 shows the default encoding of an enumeration type with five values.

Example 67   Automatic Enumeration Encoding

```
type COLOR is (RED, GREEN, YELLOW, BLUE, VIOLET);
```

The enumeration values are encoded as follows:

```
RED    = "000"
GREEN  = "001"
YELLOW = "010"
BLUE   = "011"
VIOLET = "100"
```

The colors can be compared according to their encoded values; the results of a comparison are

RED < GREEN < YELLOW < BLUE < VIOLET.

You can override the automatic enumeration encodings and specify your own enumeration encodings with the `ENUM_ENCODING` attribute. This interpretation is specific to HDL Compiler, and overriding might result in a simulation/synthesis mismatch. For more information, see ENUM_ENCODING Attribute.

Constant Floating-Point Support

This section describes constant floating-point support, in the following subsections:

- Syntax and Declarations
- Operators and Expressions
- Guidelines

Syntax and Declarations

Floating-point syntax:

```
constant identifier_list : real_subtype [:= expression] ;
```

You can declare constant floating-point objects in

- Entities (except for generic maps)
- Architectures

- Processes
- Blocks
- Functions (as an argument, a return value, or a declarative part)
- Procedures (as an argument or a declarative part)

The following types can consist of constant floating-point objects:

- Scalar
- Array
- Record

Example 68 shows various constant floating-point declarations.

Example 68 Constant Floating-Point Declarations

```
-- real scalar
        constant my_const1: real  := 4.3 ;
        constant my_const2: real  := aa + 1.2 ;

-- real array(subscript op)
        type REAL_ARRAY_T is array (3 downto 0) of real;
        constant my_const3: REAL_ARRAY_T := (4.4, 3.3, 2.2, 1.1);

-- real array with range(vector op)
        type REAL_ARRAY_T2 is array (integer range<>) of real;
        constant my_const4: REAL_ARRAY_T2(3 downto 0) :=
        (4.4, 3.3, 2.2,1.1);
        constant my_const5: REAL_ARRAY_T2(1 downto 0) := aa(2 downto 1);

-- real record(field op)
        type RECORD_T is record
            rec1: integer;
            rec2: string(3 downto 1);
            rec3: real;
        end record;
        constant my_rec: RECORD_T := (5, "mmm", 3.14);
-- constant floating point can also be argument and return value
-- of functions and procedures.
        function func1(aa : real;
                           a, b : in bit) return bit;
        function func2(aa, bb : real) return real;
        procedure proc1(aa : real;
                            a, b : in bit;
                            z    : out bit);
```

Operators and Expressions

The following operators are supported for constant floating-point type:

- Adding, signing, multiplying, and absolute value operators
- ** (power) operator (<real_data> ** <integer_data>)
- Relational operators (>, <, =, /=, >=, <=)
- Subscript and vector operators (array)
- Field operator (record)

In the following example, my_const1 and my_const2 are constant floating-point types; my_rec is a constant record type and rec1 is an element of my_rec.

```
my_const1 + 3.14
my_const1/(-my_const2)
abs(-my_const1)
my_const1 ** 5
my_const1 >= my_const2
my_const1(3), my_const1(2 downto 0)
my_rec.rec1
```

Expressions can contain floating-point numbers and constants, but these expressions are only allowed at the following locations:

- Constant floating-point initialization Comparison (relational operation)

The value of an expression that contains a constant floating point must be resolvable at elaboration time.

In the following example, the floating-point constant, my_const1 (3.14), is smaller than 5.6, so the condition in the if statement is true. This causes HDL Compiler to elaborate the first clause of the if statement and ignore the second clause. HDL Compiler assigns bb to false, because my_const1 (equaling 3.14) is known at elaboration time.

```
constant my_const1 : real :=3.14;
    if (my_const1 <= 5.6) then
      state <= a;
    else
      state <= b;
    endif;
    bb <= (my_const1/=3.14);  -- bb is BOOLEAN type
    ...
```

Guidelines

- Floating-point signal and variable objects are not supported, and cannot be synthesized; only constant floating point objects are supported.
- Floating-point objects in the generic map of an entity are not supported.
- The floating-point range is -1.0e38 to 1.0e38 inclusive, the same as the float type in C.
- The floating-point object initialization expression must be present and its value must be resolvable at the time of elaboration, or an error is reported.

math_real Package Support

This section describes HDL Compiler support for the IEEE standard VHDL math_real package, which defines arithmetic functions using REAL type arguments.

This section contains the following:

- Unsupported Constructs and Operators
- Using the math_real Package
- Arithmetic Functions
- Usage Examples

Unsupported Constructs and Operators

HDL Compiler does not support the following components:

- Is_X() function is a simulation rather than synthesis construct; it is ignored in synthesis
- REAL signals
- REAL types with ranges

Using the math_real Package

The math_real package is typically installed in the Synopsys root directory. Access it with the following statement in your VHDL code:

```
Library IEEE;
Use IEEE.math_real.all;
```

Note:

Operations on REAL type data is only supported for constant evaluation.

Arithmetic Functions

The math_real package provides arithmetic functions for use with the REAL data type. These functions can be used in synthesis for constant calculations. Example 69 shows the declaration of these functions.

Example 69 Declarations of Arithmetic Functions

```
function "**" (X : in INTEGER; Y : in REAL) return REAL;
function "**" (X : in REAL; Y : in REAL) return REAL;
function "MOD" (X, Y : in REAL) return REAL;
function ARCCOS (X : in REAL) return REAL;
function ARCCOSH (X : in REAL) return REAL;
function ARCSIN (X : in REAL) return REAL;
function ARCSINH (X : in REAL) return REAL;
function ARCTAN (Y : in REAL) return REAL;
function ARCTAN (Y : in REAL; X : in REAL) return REAL;
function ARCTANH (X : in REAL) return REAL;
function CBRT (X : in REAL) return REAL;
function CEIL (X : in REAL) return REAL;
function COS (X : in REAL) return REAL;
function COSH (X : in REAL) return REAL;
function EXP (X : in REAL) return REAL;
function FLOOR (X : in REAL) return REAL;
function LOG (X : in REAL) return REAL;
function LOG (X : in REAL; BASE : in REAL) return REAL;
function LOG10 (X : in REAL) return REAL;
function LOG2 (X : in REAL) return REAL;
function REALMAX (X, Y : in REAL) return REAL;
function REALMIN (X, Y : in REAL) return REAL;
function ROUND (X : in REAL) return REAL;
function SIGN (X : in REAL) return REAL;
function SIN (X : in REAL) return REAL;
function SINH (X : in REAL) return REAL;
function SQRT (X : in REAL) return REAL;
function TAN (X : in REAL) return REAL;
function TANH (X : in REAL) return REAL;
function TRUNC (X : in REAL) return REAL;
procedure UNIFORM(variable SEED1, SEED2 : inout POSITIVE; variable X :
out REAL);
```

Usage Examples

See Example 70 and Example 71.

Example 70 Constant Evaluation of Parameters

```
library IEEE;
use IEEE.math_real.all;

entity test1 is
  generic (
            param1 : real := 1.1;
```

```
                param2 : real := 2.2;
                param3 : real := 3.3;
                param4 : real := 4.4
              );
…

architecture rtl of test1 is
  constant p1 : real    := REALMAX(param1, param2);
  constant p2 : real    := REALMIN(param3, param4);
  constant p3 : real    := ARCTAN(param2);
  constant rp4 : real   := ROUND(param4);
constant i1 : integer := integer(param4);
```

Example 71 User-Defined Functions

```
function realtodb(val   : real ) return real is
begin
return 20.0*(log10(val));
end function;

CONSTANT Ar      : real     := 1.375;
CONSTANT Ar_db   : real     := realtodb(Ar);
```

Operands

Operand requirements specific to HDL Compiler are discussed in the following subsections:

- Operand Bit-Width
- Array Slice Names
- Variable Array Slice Operations
- Computable and Noncomputable Operands
- Indexed Name Targets

Operand Bit-Width

HDL Compiler uses the bit-width of the largest operand to determine the bit-width needed to implement an operator in a circuit. For example, an INTEGER operand is 32 bits wide by default. An addition of two INTEGER operands causes HDL Compiler to build a 32-bit adder.

To use hardware resources efficiently, always indicate the bit-width of numeric operands. For example, use a subrange of INTEGER when declaring types, variables, or signals.

```
type     ENOUGH:  INTEGER range 0 to 255;
variable WIDE:    INTEGER range -1024 to 1023;
signal   NARROW:  INTEGER range 0 to 7;
```

Array Slice Names

Slice names identify a sequence of consecutive elements of an array variable or signal. The syntax is

```
identifier (expression direction expression)
```

identifier

The identifier is the name of a signal or variable of an array type. Each *expression* must return a value within the array's index range and must be computable. For more information, see Computable and Noncomputable Operands.

direction

The *direction* must be either to or downto. The direction of a slice must be the same as the direction of an identifier's array type. If the left and right expressions are equal, they define a single element.

The value returned to an operator is a subarray containing the specified array elements.

Variable Array Slice Operations

HDL Compiler supports *variable array slice* operations, which can perform assignment to (LHS) or selection of (RHS) a fixed-width group of bits at a variable base address of an expression.

Variable array slices must meet the following requirements:

- The slice width of the range must be constant.
- The upper and lower range expressions can use: *, +, -, (), one noncomputable variable, and any computable constants (numeric or nonnumeric).
- If any nonnumeric constants are used, the upper and lower expressions must be identical in format except for a single integer difference.

Assignment and Selection

Variable array slices on the LHS allow assignment to a variable group of bits:

```
  myArray(i downto i - 3) <= data(3 downto 0);
```

For assignment, if the calculated range is partially out of the destination range, the out-of-range bit assignments have no effect.

Variable array slices on the RHS allow selection of a variable group of bits:

```
data(3 downto 0) <= myArray(i downto i - 3);
```

Note:

For simplicity, subsequent examples in this section show only bare variable array slice expressions instead of assignment (LHS) or selection (RHS) statements.

Index Expression Variables

Index expressions can use the index variable, which itself is noncomputable, within any computable expression:

```
-- i is an integer port
myArray(i downto i - 3)

-- v is a bit vector port
myArray(v downto v - 3)
myArray(v(4 downto 0) downto v(4 downto 0) - 3)
myArray(my_func(v) downto my_func(v) - 3)
```

Index Expressions Using Only Numeric Constants

When only numeric constants are used, the upper and lower expressions can use any format if the slice width remains constant.

```
-- upper and lower expressions can differ in format
myArray(4 * (v - 1) downto 4 * v - 10)
myArray((v - 2) * 3 to (3 - 1) * v + v - 2)
```

Index Expressions Using Nonnumeric, Computable Constants

When nonnumeric but computable constants are used, the upper and lower expressions must share the same format except for a single integer difference to indicate the range.

```
-- upper and lower expressions must be identical in format

-- constant c1: integer := conv_integer(three);
-- constant c2: integer := conv_integer(four);
myArray((c1 + c2) * v downto (c1 + c2) * v - 3)   -- valid
myArray((c1 + c2) * v downto (c2 + c1) * v - 3)   -- invalid
--                            ^ constants swapped

-- generic ( WIDTH : integer := 2)
myArray((2 * WIDTH) * v - 3 to (2 * WIDTH) * v)   -- valid
myArray((2 * WIDTH) * v - 3 to (2 * WIDTH * v))   -- invalid
--                                         ^ parentheses used differently
```

Computable and Noncomputable Operands

A computable operand is one whose value can be determined by HDL Compiler at compile time; that is, the operand value is constant and does not depend on any inputs. Noncomputable operand values depend on inputs that are known only at runtime. Because the operand value varies according to inputs, HDL Compiler needs to build additional logic to determine what the value is at runtime.

Following are examples of computable operands:

- Literal values
- for...loop parameters, when the loop's range is computable
- Variables assigned a computable expression
- Aggregates that contain only computable expressions
- Function calls whose return value is computable
- Expressions with computable operands
- Qualified expressions when the expression is computable
- Type conversions when the expression is computable
- The value of the and or nand operators when one of the operands is a computable '0'
- The value of the or operator or the nor operator when one of the operands is a computable '1'

Additionally, a variable is given a computable value if it is an OUT or INOUT parameter of a procedure that assigns it a computable value.

Typically, the following are noncomputable operands:

- Signals
- Ports
- Variables assigned different computable values that depend on a noncomputable condition
- Variables assigned noncomputable values

Example 72 shows some definitions and declarations, followed by several computable and noncomputable expressions.

Example 72 Computable and Noncomputable Expressions

```
signal S: BIT;
. . .
```

```
function MUX(A, B, C: BIT) return BIT is
begin
  if (C = '1') then
    return(A);
  else
    return(B);
  end if;
end;

procedure COMP(A: BIT; B: out BIT) is
begin
  B := not A;
end;

process(S)
  variable V0, V1, V2: BIT;
  variable V_INT:      INTEGER;

  subtype MY_ARRAY is BIT_VECTOR(0 to 3);
  variable V_ARRAY:    MY_ARRAY;
begin
  V0 := '1';               -- Computable (value is '1')
  V1 := V0;                -- Computable (value is '1')
  V2 := not V1;            -- Computable (value is '0')

  for I in 0 to 3 loop
    V_INT := I;            -- Computable (value depends on iteration)
  end loop;

  V_ARRAY := MY_ARRAY'(V1, V2, '0', '0');
                           -- Computable ("1000")
  V1 := MUX(V0, V1, V2); -- Computable (value is '1')
  COMP(V1, V2);
  V1 := V2;                -- Computable (value is '0')
  V0 := S and '0';         -- Computable (value is '0')
  V1 := MUX(S, '1', '0');-- Computable (value is '1')
  V1 := MUX('1', '1', S);-- Computable (value is '1')

  if (S = '1') then
    V2 := '0';             -- Computable (value is '0')
  else
    V2 := '1';             -- Computable (value is '1')
  end if;
  V0 := V2;              -- Noncomputable; V2 depends on S
  V1 := S;               -- Noncomputable; S is a nonfixed signal
  V2 := V1;              -- Noncomputable; V1 is no longer computable
end process;
```

Indexed Name Targets

The syntax for an assignment to an indexed name (identifier) target is

```
identifier(index_expression) := expression; -- Variable assignment
identifier(index_expression) <= expression; -- Signal assignment
```

The `identifier` is the name of an array type signal or variable.

The `index_expression` must evaluate to an index value for the identifier array's index type and bounds. It does not have to be computable, but more logic is synthesized if it is not. For more information, see Computable and Noncomputable Operands.

The assigned `expression` must have the array's element type.

Modeling Considerations

Modeling requirements specific to HDL Compiler are discussed in the following subsections:

- Concatenation
- Unconstrained Type Ports
- Input Ports Associated With the Keyword open
- Multiple Events in a Single Process
- Multiple if Statements in a Process
- Keeping Signal Names
- Controlling Structure
- Resolution Functions
- Asynchronous Designs
- Using Don't Care Values
-
- Multibit Inference

Concatenation

HDL Compiler supports both the 87 and 93 VHDL language reference manual (LRM) definitions of the concatenation operator. The default support is for the 93 LRM definition. To enable the 87 LRM definition, set the `hdlin_vhdl93_concat` variable to `false`, changing it from its default of `true`. To understand the difference between the two definitions, consider Example 73. In this example, the k3 and k4 constants are defined by using concatenation with the k1 and k2 constants. The values of k3 and k4 differ according to what VHDL language standard you use: VHDL-87 or VHDL-93.

If you use VHDL-93, the value of k3'left to k3'right is the same as k4'left to k4'right. To determine this value, the tool counts from 0 to 3 and the value does not depend on the k1 and k2 start positions.

If you use VHDL-87, the value of k3'left to k3'right, which is from 0 to 3, is different from k4'left to k4'right, which is from 1 down to -2. The difference is that concatenation in VHDL-87 defines the position count from the start position of the left operand k1 position (1) and the procedure (downto) instead of starting from the 0 position as is the case in the VHDL-93 language standard.

Example 73 Understanding Concatenation in VHDL-93 and VHDL-87

```
constant c1 : bit_vector(0 to 3) := "1101";
       constant c2 : bit_vector(0 to 3) := "0010";

       -- value of "c3" is "11010010"
       constant c3 : bit_vector(0 to 7) := c1 & c2;

       -- value of "c4" is "11101"
       constant c4 : bit_vector(0 to 4) := '1' & c1;

       -- value of "c5" is "01"
       constant c5 : bit_vector(0 to 1) := '0' & '1';

type r is 0 to 7;
       type r_vector is array (r <> range) of bit;
       constant k1 : r_vector(1 downto 0) := "10";
       constant k2 : r_vector(0 to 1) := "01";
       constant k3 : r_vector := k2 & k1;
       constant k4 : r_vector := k1 & k2;
```

Example 74 shows the values of the k2, k3, and k5 constants interpreted by using VHDL-87 and VHDL-93 definitions.

Example 74 Another Example of Concatenation

```
type r1 is range 0 to 7;
type r2 is range 7 downto 0;
type t1 is array (r1 range <>) of bit;
type t2 is array (r2 range <>) of bit;

subtype s1 is t1(r1);
subtype s2 is t2(r2);

constant k2: t1 := k1(1 to 3) & k1(3 to 4);
          -- 93 concat: k2'left = 0 and k2'right = 4
          -- 87 concat: k2'left = 1 and k2'right = 5

constant k3: t1 := k1(5 to 7) & k1(1 to 2);
          -- 93 concat: k3'left = 0 and k3'right = 4
          -- 87 concat: k3'left = 5 and k3'right = 9
```

```
constant k5: t2 := k4(3 downto 1) & k1(3 to 4);
          -- 93 concat: k5'left = 0 and k5'right = 4
          -- 87 concat: k5'left = 3 and k5'right = -1
```

Unconstrained Type Ports

HDL Compiler supports the usage of unconstrained type ports when the type of the ports can be deduced. In these cases, you must use the `analyze` and `elaborate` commands to read your design. The `read_vhdl` command does not support type conversion on formal ports.

Input Ports Associated With the Keyword open

If you associate an input port with the reserved keyword open, you must initialize it with a default expression, or the `analyze` command reports an error. HDL Compiler connects the open port with the default expression after elaboration.

Multiple Events in a Single Process

HDL Compiler supports multiple events in a single process as shown in the following example:

```
process
begin
  wait until CLOCK'event and CLOCK = '1';
  if (CONDITION) then
    X <= A;
  else
    wait until CLOCK'event and CLOCK = '1';
  end if;
end process;
```

Multiple if Statements in a Process

You can specify the same condition for multiple `if` statements in a process. The following examples show the supported, unsupported, and illegal coding styles:

Supported Coding Styles

Example 75   Multiple if Statements With the Same Condition

```
process (clk)
begin
   if rising_edge(clk) then
      o1 <= i(1);
   end if;
```

```
      if rising_edge(clk) then
         o2 <= i(2);
      end if;
   end process;
```

Example 76 Multiple if Statements With or Without a Set or Reset Condition

```
   process (clk, rst)
   begin
      if rst='1' then
         o1 <= '1';
      elsif rising_edge(clk) then
         o1 <= i(1);  -- with reset
      end if;

      if rising_edge(clk) then
         o2 <= i(2);  -- without reset
      end if;
   end process;
```

Example 77 Multiple Sequential and Combinational Blocks

```
   process (clk) begin
      if rising_edge(clk) then -- sequential block
         o1 <= i(1);
      end if;

      if rising_edge(clk) then -- sequential block
         o2 <= i(2);
      end if;

      o3 <= i(1) and i(2);     -- combinational block
   end process;
```

You must specify sequential assignments before combinational assignments, as shown in Example 78.

Example 78 Sequential Assignments Before Combinational Assignments

```
   process (clk, rst)
   begin
      if rst='1' then
         o1 <= '1';
      elsif rising_edge(clk) then
         o1 <= i(1);
      end if;

      if rising_edge(clk) then
         o2 <= i(2); -- o2, sequential
      end if;

      o2 <= i(1);    -- o2, combinational
   end process;
```

Example 79   Multiple if Statements With the Same Variable

```
process (clk, rst)
variable t : std_logic;
begin
   if rst='1' then
      o1 <= '1';
   elsif rising_edge(clk) then
      t := i(1);         -- variable t is assigned
      o1 <= i(1);
   end if;

   if rst = '1' then
      o2 <= '0';
   elsif rising_edge(clk) then
      o2 <= t and i(2); -- variable t is used
   end if;
end process;
```

Unsupported Coding Style

You must use the same clock edge as a condition for multiple `if` statements, that is, either a positive edge or negative edge. If you use both clock edges in multiple `if` statements, as shown in Example 80, the tool issues an ELAB-2040 error message.

Example 80   Different Clock Edges for Multiple if Statements

```
process (clk)
begin
   if rising_edge(clk) then     -- positive edge
      o1 <= i(1);
   elsif falling_edge(clk) then -- negative edge
      o2 <= i(2);
   end if;
end process;
```

Illegal Coding Style

If a combinational assignment comes before a sequential assignment, as shown in Example 81, the tool issues an ELAB-113 error message.

Example 81   Combinational Assignments Before Sequential Assignments

```
process (clk, rst)
begin
   if rst='1' then
      o1 <= '1';
   elsif rising_edge(clk) then
      o1 <= i(1);
   end if;
   o2 <= i(1);    -- o2, combinational

   if rising_edge(clk) then
```

```
      o2 <= i(2); -- o2, sequential
   end if;
end process;
```

In Example 82, the o2 signal has two different assignments under the same condition. The tool issues an ELAB-112 error message for this coding style.

Example 82   Conflicting Assignments to a Signal

```
process (clk, rst)
begin
   if rst='1' then
      o1 <= '1';
   elsif rising_edge(clk) then
      o1 <= i(1);
   end if;

   if rising_edge(clk) then
      o2 <= i(2); -- assignment to o2
   end if;

   if rising_edge(clk) then
      o2 <= i(3); -- assignment to o2
   end if;
end process;
```

Keeping Signal Names

When a signal is in a path to an output port, HDL Compiler usually keeps the signal's name if the signal is not removed during optimizations, such as removing redundant code. You can give HDL Compiler guideline information for keeping a signal name by using the `hdlin_keep_signal_name` variable and the `keep_signal_name` directive. The default is `all_driving`. Table 6 describes the settings.

Table 6   hdlin_keep_signal_name Variable Options

Option	**Description**
`user`	This option works with the `keep_signal_name` directive. HDL Compiler attempts to preserve a signal if the signal isn't removed by optimizations and that signal is labeled with the `keep_signal_name` directive. Both dangling and driving nets are considered. Although not guaranteed, HDL Compiler typically keeps the specified signal for this configuration.
`all_driving` (default)	HDL Compiler attempts to preserve a signal if the signal isn't removed by optimizations and the signal is in an output path. Only driving nets are considered. This option does not guarantee a signal is kept.

Feedback

Note:

When a signal has no driver, the tool assumes logic 0 (ground) for the driver.

To prevent signals from being removed during optimizations, use the `keep_signal_name` directive, as shown in Example 83 and Example 84. This directive works together with the `hdlin_keep_signal_name` variable. For the examples, the value is set to `user`. Review the possible values and use the setting for your specific requirements before reading your design.

Example 83 Keep Signal tmp

```
entity bus_name is
   port (
     in1 : in bit_vector (1 downto 0) ;
     in2 : in bit_vector (1 downto 0) ;
     z : out bit_vector (1 downto 0));
end bus_name ;

architecture imp of bus_name is
-- synopsys keep_signal_name "tmp"
  signal tmp : bit_vector (1 downto 0);
begin
  process(in1, in2)
  begin
    tmp <= in1 and in2;
    z <= in1;
  end process ;
end imp;
```

Example 84 Keep Signal tmp1 and tmp2

```
entity test is
 port (a, b : in bit; z: out bit );
end;

architecture imp of test is
-- synopsys keep_signal_name "tmp1,   tmp2"
 signal tmp1 : bit;
 signal tmp2 : bit;

begin
 process (a, b)
 begin
     tmp1 <= a and b;
     tmp2 <= a or b;
     z <= b;
 end process;
end  imp;  ;
```

Controlling Structure

You can use parentheses to force the synthesis of parallel hardware. For example, (A + B) + (C + D) builds an adder for A+B, an adder for C+D, and an adder to add the result. HDL Compiler preserves the subexpressions dictated by the parentheses, but this restriction on HDL Compiler optimizations might lead to less-than-optimum area and timing results.

Parentheses can also be helpful in coding for late-arriving signals. For example, if you are adding three signals—A, B, and C—and A is late arriving, then A+(B+C) can be useful in handling the late-arriving signal A. HDL Compiler also tries to create a structure to allow the late-arriving signal to meet timing. Any restriction on HDL Compiler optimizations might lead to less-than-optimum area and timing results.

Resolution Functions

The resolution function and the coding style determine the choice of wired logic. Synthesis neither checks for nor resolves possible data collisions on a synthesized three-state bus. You must ensure that the three-state enablers for a common bused line are not active at the same time.

Asynchronous Designs

If you use asynchronous design techniques—that is, nonclocked designs—synthesis and simulation results might not agree. Because HDL Compiler does not issue warning messages for asynchronous designs, you are responsible for verifying the correctness of your circuit. See the *Synopsys Timing Constraints and Optimization User Guide* for additional information.

Using Don’t Care Values

HDL Compiler always evaluates comparisons to don’t care values to false. This behavior is different from simulation behavior. To prevent a synthesis/simulation mismatch, always use the IEEE 1076.3-1997 STD_MATCH function when using don’t care values in comparisons. For more information, see Don’t Care Values in Comparisons.

Multibit Inference

The HDL Compiler tool can infer registers that have identical structures for each bit as multibit components. These components reduce area and power consumption in a design, but their primary benefits are the creation of a more uniform structure for layout during place and route.

To direct HDL Compiler to infer multibit components, do one of the following tasks:

- Embed the `infer_multibit` attribute in the HDL source code.

  The attribute gives you control over individual case statements. Set the `infer_multibit` attribute to `true` on signals and variables to infer them as multibit components. See Example 85.

- Use the `hdlin_infer_multibit` variable.

  This variable controls multibit inference for all bused registers in the design. Valid values for this variable are:

  - `default_none`

    Infers multibit components for signals that have the `infer_multibit` attribute set to `true` in the VHDL RTL. This is the default.

  - `default_all`

    Infers multibit components for all bused registers. To disable multibit mapping for specific signals, set the `infer_multibit` attribute to `false` on those signals. See Example 87.

  - `never`

    Does not infer multibit components regardless of the attributes or directives in the HDL source.

In Example 85, the `infer_multibit` attribute is set on the q_0 signal.

Example 86 shows the inference report. In this report, the MB column indicates that the component is inferred as a flip-flop multibit component.

Example 85   Inferring a Multibit Flip-Flop With the infer_multibit Attribute

```
-- This example is run under the default conditions, that is,
-- the hdlin_infer_multibit variable is set to the default_none value.
[RTL]
library IEEE, Synopsys;
use IEEE.std_logic_1164.all;
use Synopsys.attributes.all;

entity test_multibit is
  port (d0, d1, d2 : in std_logic_vector(7 downto 0);
        clk, rst : in std_logic;
        q0, q1, q2 : out std_logic_vector(7 downto 0) );
end test_multibit;

architecture rtl of test_multibit is
attribute infer_multibit of q0 : signal is "true";
begin
```

```
process (clk, rst) begin
  if (rst = '0') then
    q0 <= "00000000";
    q1 <= "00000000";
    q2 <= "00000000";
  elsif (clk'event and clk = '1') then
    q0 <= d0;
    q1 <= d1;
    q2 <= d2;
  end if;
end process;
end rtl;
```

Example 86   Inference Report Showing q0_reg Inferred as a Multibit Flip-Flop

```
Inferred memory devices in process
        in routine test_multibit line 16 in file
                '/.../test.vhd'.
===========================================================================
| Register Name |    Type    | Width | Bus | MB |  Set  | Reset | ST | Line |
===========================================================================
|    q2_reg     | Flip-flop |   8    |  Y  | N  | None | None  | N  |  17  |
|    q0_reg     | Flip-flop |   8    |  Y  | Y  | None | None  | N  |  17  |
|    q1_reg     | Flip-flop |   8    |  Y  | N  | None | None  | N  |  17  |
============================================================================
Presto compilation completed successfully.
```

Example 87 shows the same VHDL code but illustrates how to prevent multibit inference of the q_0 signal when the `hdlin_infer_multibit` variable is set to the `default_all` value. Example 88 shows the inference report. In this report, the MB column indicates that the component is not inferred as a multibit component.

Example 87   Inferring Multibit Flip-Flops With the hdlin_infer_multibit Variable

```
-- In this example, the hdlin_infer_multibit variable is set to the
-- default_all value.
[RTL]
library IEEE, Synopsys;
use IEEE.std_logic_1164.all;
use Synopsys.attributes.all;

entity test_multibit is
  port (d0, d1, d2 : in std_logic_vector(7 downto 0);
        clk, rst : in std_logic;
        q0, q1, q2 : out std_logic_vector(7 downto 0) );
end test_multibit;

architecture rtl of test_multibit is
attribute infer_multibit of q0 : signal is "false";

begin
process (clk, rst) begin
  if (rst = '0') then
    q0 <= "00000000";
```

```
        q1 <= "00000000";
        q2 <= "00000000";
      elsif (clk'event and clk = '1') then
        q0 <= d0;
        q1 <= d1;
        q2 <= d2;
      end if;
    end process;
    end rtl;
```

Example 88   Inference Report Showing q1_reg and q2_reg Inferred as Multibit Components

```
Inferred memory devices in process
        in routine test_multibit line 16 in file
                '/remote/multibit/basic/test.vhd'.
==========================================================================
| Register Name |   Type    | Width | Bus | MB |  Set | Reset | ST | Line |
==========================================================================
|   q2_reg      | Flip-flop |   8   |  Y  | N  | None | None  | N  |  17  |
|   q0_reg      | Flip-flop |   8   |  Y  | Y  | None | None  | N  |  17  |
|   q1_reg      | Flip-flop |   8   |  Y  | N  | None | None  | N  |  17  |
==========================================================================
Presto compilation completed successfully.
```

For information on how HDL Compiler handles multibit components in a mapped design, see the HDL Compiler documentation.

Simulation/Synthesis Mismatch Issues

This following sections describe simulation/synthesis mismatch issues:

- Type Mismatches
- Set and Reset Signals
- Z Values in Expressions
- Don’t Care Values in Comparisons
- Ordering of Enumerated Types Using the ENUM_ENCODING Attribute
- Sensitivity Lists
- Delay Specifications

Type Mismatches

The numeric_std package and the std_logic_arith package have overlapping operations. Use of these two packages simultaneously during analysis could cause type mismatches.

Set and Reset Signals

A simulation/synthesis mismatch can occur if the set/reset signal is masked by an X during initialization in simulation. Use of the `sync_set_reset` directive reduces mismatches. For examples, see D Flip-Flop With Synchronous Set and D Flip-Flop With Synchronous Reset.

Z Values in Expressions

The use of the `z` value in an expression always evaluates to false and can cause a simulation/synthesis mismatch. For details, see Understanding the Limitations of Three-State Inference.

Don’t Care Values in Comparisons

To prevent simulation/synthesis mismatch, do not use don’t care values in comparisons unless you use the IEEE 1076.3-1997 STD_MATCH function.

Don’t care types are treated differently in simulation than they are in synthesis. To a simulator, a don’t care value is a distinct value, different from a 1 or a 0. In synthesis, however, a don’t care value becomes a 0 or a 1. When a don’t care value is used in a comparison, HDL Compiler always evaluates the comparison to false. Because of this difference in treatment, there is the potential for a simulation/synthesis mismatch whenever a comparison is made with a don't care value.

For example, if X = ’-’ then is synthesized as if FALSE then.

The following case statement causes a synthesis/simulation mismatch because the simulator evaluates 1- to match 11 and 10 but the synthesis tool evaluates 1- to false; the same hold true for the 0- evaluation.

```
case (A)
  1- : ....  --  you want 1- to match 11 and 10 but
             --  HDL Compiler always evaluates this comparison
             --  to false
  0- : ....  --  you want 0- to match 00 and 01 but
             --  HDL Compiler always evaluates this comparison
             --  to false
  default : ....
endcase
```

To fix this mismatch problem, always use the STD_MATCH function; for example, rewrite the code above by using if statements, as follows:

```
if (STD_MATCH (A, "1,-"))
...
elseif (STD_MATCH (A, "0,-"))
```

```
...
else
```

HDL Compiler issues a warning similar to the following when it synthesizes such comparisons:

```
Warning: A partial don't-care value was read in routine test
line 24 in file 'test.vhd' This can cause simulation to
disagree with synthesis. (HDL-171)
```

Ordering of Enumerated Types Using the ENUM_ENCODING Attribute

If you set the encoding of your enumerated types using the `ENUM_ENCODING` attribute, the ordering operators compare your encoded value ordering, not the declaration ordering. Because this interpretation is specific to HDL Compiler, it might cause a mismatch with the VHDL simulator, which uses the declaration's order of enumerated types. See Enumeration Encoding and ENUM_ENCODING Attribute.

Sensitivity Lists

HDL Compiler generates a warning if all the signals read by the process are not listed in the sensitivity list. The circuit HDL Compiler synthesizes is sensitive to all signals the process reads. To guarantee the same results from a VHDL simulator, follow these guidelines when developing the sensitivity list:

- For sequential logic, include the clock signal and all asynchronous control signals in the sensitivity list.
- For combinational logic, all inputs must be in the sensitivity list.

HDL Compiler checks sensitivity lists for completeness and issues a warning message for any signal that is read inside a process but is not in the sensitivity list. An error message is issued if a clock signal is read as data in a process.

Note:

The IEEE VHDL Standard does not allow a sensitivity list if the process has a `wait` statement. If your code has this condition, the HDL Compiler tool issues a warning and ignores the code.

Delay Specifications

Delays are ignored for synthesis. They can cause a synthesis or simulation mismatch.

3 Modeling Combinational Logic

This chapter describes coding guidelines specific to HDL Compiler that are useful in combinational logic synthesis.

This chapter contains the following sections:

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 6.

A synthetic library contains definitions for synthetic operators, synthetic modules, and bindings. It also contains declarations that associate synthetic modules with their implementations.

A synthetic library contains definitions for synthetic operators, synthetic modules, and bindings. It also contains declarations that associate synthetic modules with their

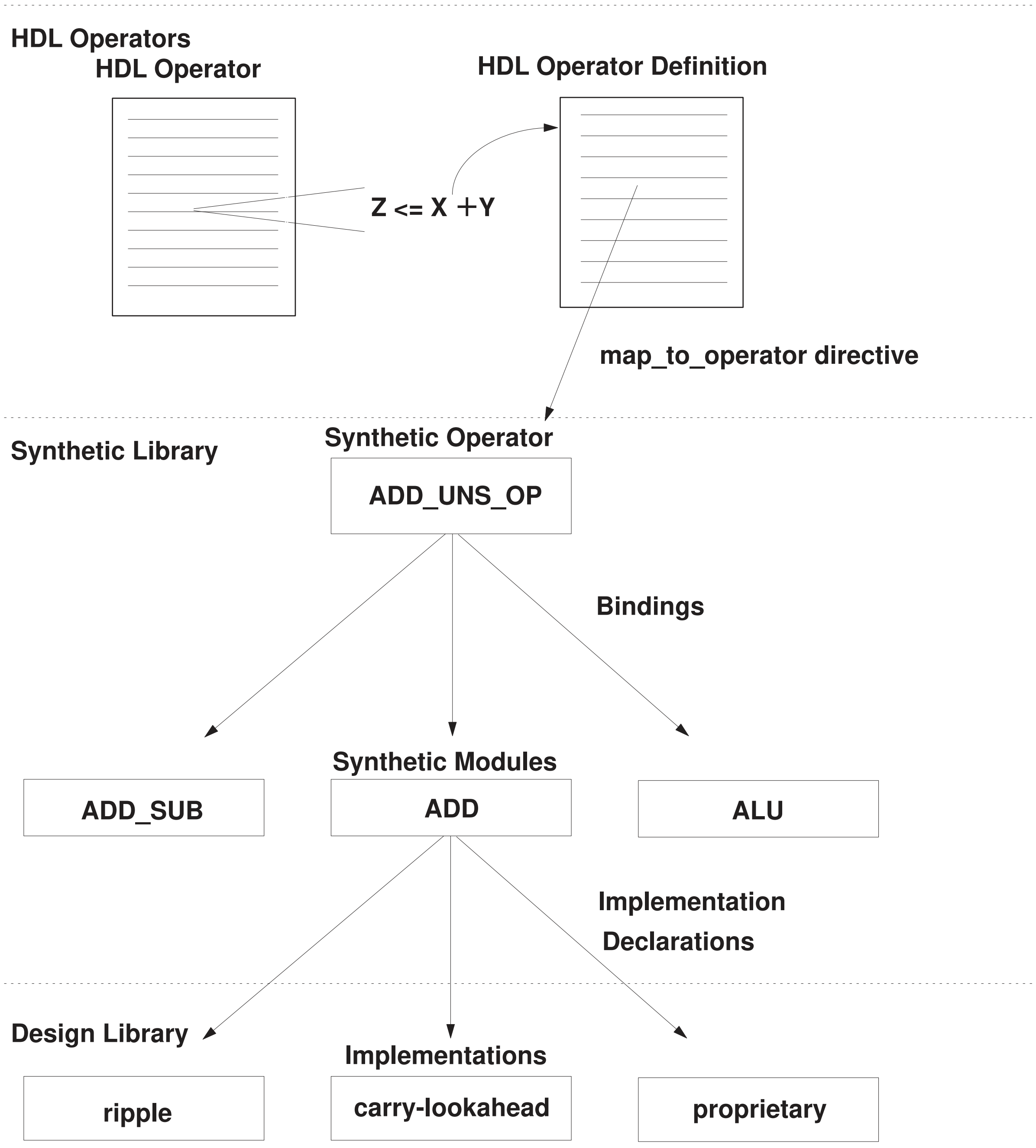
implementations. To display information about the standard synthetic library that is included with the HDL Compiler license, use the `report_synlib` command.

For example,

```
report_synlib standard.sldb
```

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

Figure 6 DesignWare Hierarchy



Logic and Arithmetic Operator Implementation

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.

A HDL Compiler license includes a DesignWare-Basic license that enables the DesignWare synthetic modules listed in Table 7. These modules support common logic and arithmetic HDL operators. By default, adders and subtracters must be more than 4 bits wide to be mapped to these modules. If they are smaller, the operators are mapped to combinational logic.

Table 7 Operators Supported by a DesignWare-Basic License

HDL Operator	Linked to DesignWare Synthetic Module
Comparison (> or <)	DW01_cmp2
Absolute value (abs)	DW01_absval
Addition (+)	DW01_add
Subtraction (-)	DW01_sub
Addition or Subtraction (+ or -)	DW01_addsub
Incrementer (+)	DW01_inc
Decrementer (-)	DW01_dec
Incrementer or decrementer (+ or -)	DW01_incdec
Multiplier (*)	DW02_mult

Synopsys creates numerous DesignWare synthetic modules in addition to the basic modules. The DesignWare Building Block IP (formally called Foundation Library) is a collection of reusable intellectual property blocks that are integrated into the Synopsys synthesis environment. This library contains high-performance implementations of Basic Library IP plus many IP that implement more advanced arithmetic and sequential logic functions. For more information about DesignWare synthetic modules, see the DesignWare documentation.

Propagating Constants

Constant propagation is the compile-time evaluation of expressions containing constants. HDL Compiler uses constant propagation to reduce the amount of hardware required to implement operators. For example, a "+" operator with a constant 1 as an input causes an incrementer, rather than a general adder, to be built. If both adder arguments are constants, no hardware is constructed, because the expression's value is calculated by HDL Compiler and inserted directly in the circuit.

Other operators that benefit from constant propagation include comparators and shifters. Shifting a vector by a constant amount requires no logic to implement; it requires only a reshuffling (rewiring) of bits.

Multiplexing Logic

Multiplexers are commonly modeled with if and case statements. To implement this logic, HDL Compiler uses SELECT_OP cells, which HDL Compiler maps to combinational logic or multiplexers in the logic library. If you want HDL Compiler to preferentially map multiplexing logic to multiplexers—or multiplexer trees—in your logic library, you must infer MUX_OP cells.

The following sections describe multiplexer inference:

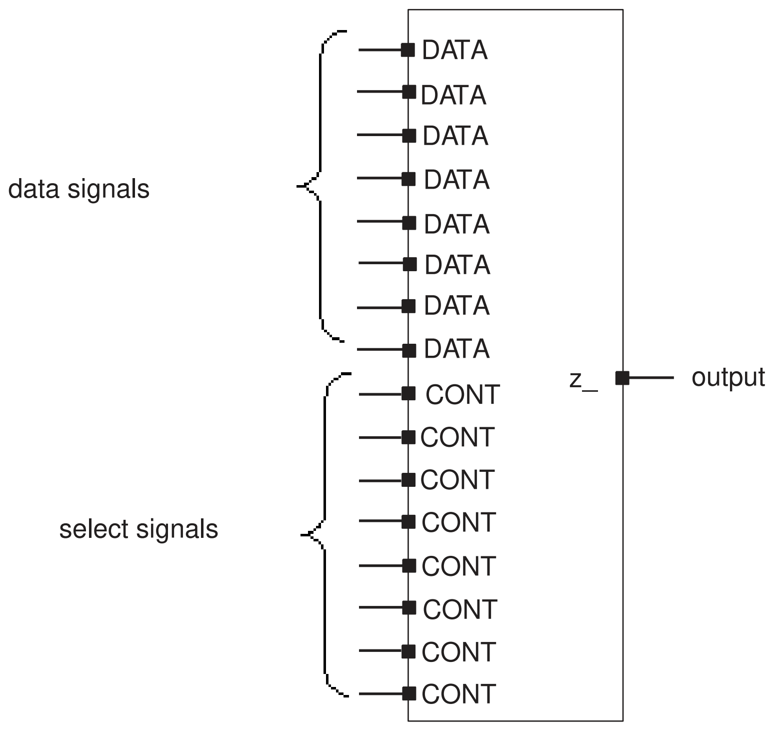
- SELECT_OP Inference
- MUX_OP Inference
- Variables That Control MUX_OP Inference
- MUX_OP Inference Examples
- MUX_OP Inference Limitations

SELECT_OP Inference

By default, HDL Compiler uses SELECT_OP components to implement conditional operations implied by if and case statements. An example SELECT_OP cell implementation for an 8-bit data signal is shown in Figure 7.

Feedback

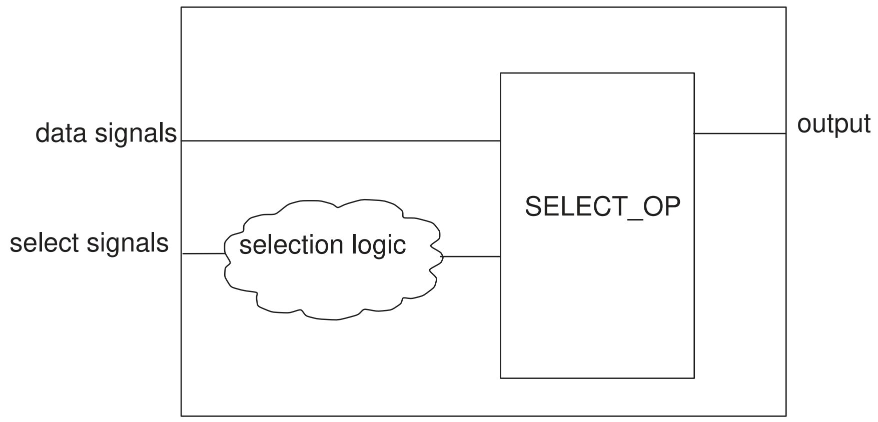
Figure 7 SELECT_OP Implementation for an 8-bit Data Signal



Note that for an 8-bit data signal, 8 selection bits are needed - this is called a one-hot implementation.

SELECT_OPs behave like one-hot multiplexers; the control lines are mutually exclusive, and each control input allows the data on the corresponding data input to pass to the output. To determine which data signal is chosen, HDL Compiler generates selection logic, as shown in Figure 8.

Figure 8 HDL Compiler Output—SELECT_OP and Selection Logic



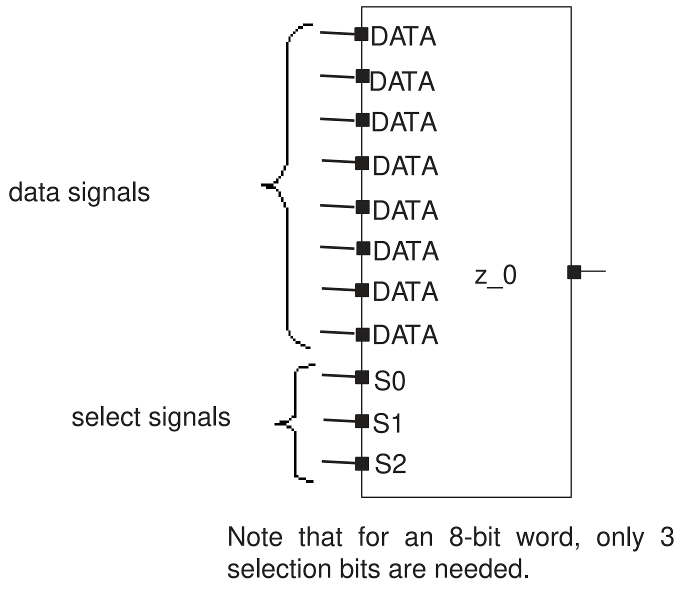
Depending on the design constraints, HDL Compiler implements the `SELECT_OP` with either combinational logic or multiplexer cells from the logic library.

MUX_OP Inference

If you want HDL Compiler to preferentially map multiplexing logic in your RTL to multiplexers—or multiplexer trees—in your logic library, you need to infer MUX_OP cells. These cells are hierarchical generic cells optimized to use the minimum number of select signals. They are typically faster than the SELECT_OP cell, which uses a one-hot implementation. Although MUX_OP cells improve design speed, they also might increase area. During optimization, HDL Compiler preferentially maps MUX_OP cells to multiplexers—or multiplexer trees—from the logic library, unless the area costs are prohibitive, in which case combinational logic is used. See the *Synopsys Timing Constraints and Optimization User Guide* for information about how HDL Compiler maps MUX_OP cells to multiplexers in the target logic library.

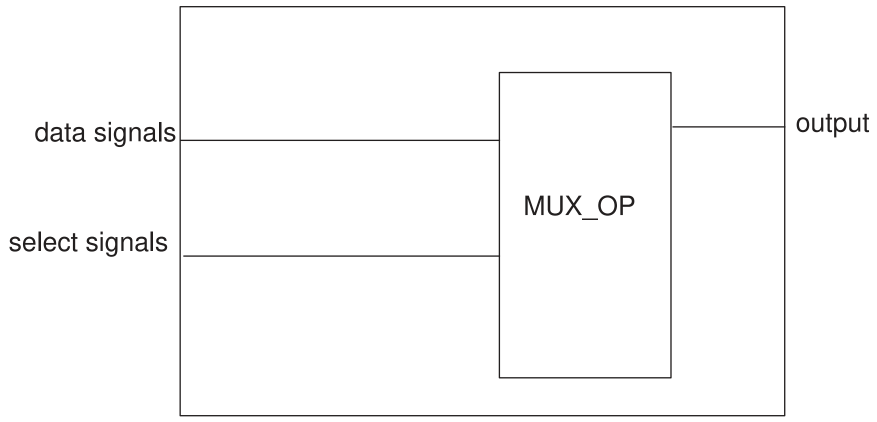
Figure 9 shows a MUX_OP cell for an 8-bit data signal. Notice that the MUX_OP cell only needs three control lines to select an output; compare this with the SELECT_OP which needed eight control lines.

Figure 9 MUX_OP Generic Cell for an 8-bit Data Signal



The MUX_OP cell contains internal selection logic to determine which data signal is chosen; HDL Compiler does not need to generate any selection logic, as shown in Figure 10.

Figure 10 HDL Compiler Output—MUX_OP Generic Cell for 8-Bit Data



Use the following methods to infer MUX_OP cells:

- To infer MUX_OP cells for a specific case or if statement, use the `infer_mux` attribute or the `-- synopsys infer_mux` directive in the VHDL description.
  - Attach the `infer_mux` attribute to a case statement, by using the following syntax: `case var is -- synopsys infer_mux`
  - Attach the `infer_mux` directive, as follows:

    ```
    case SEL3 is -- synopsys infer_mux
       when "00"    => DOUT3 <= DIN3(0);
       when "01"    => DOUT3 <= DIN3(1);
       when "10"    => DOUT3 <= DIN3(2);
       when "11"    => DOUT3 <= DIN3(3);
    ```

- To infer MUX_OP cells for a specific case or if statement regardless of the settings of the `hdlin_infer_mux`, `hdlin_mux_oversize_ratio`, `hdlin_mux_size_limit`, and `hdlin_mux_size_min` variables, use the `infer_mux_override` directive.

  For example,

  ```
  case SEL3 is -- synopsys infer_mux_override
     when "00"    => DOUT3 <= DIN3(0);
     when "01"    => DOUT3 <= DIN3(1);
     when "10"    => DOUT3 <= DIN3(2);
     when "11"    => DOUT3 <= DIN3(3);
  ```

  The tool marks the MUX_OP cells inferred by this directive with the `size_only` attribute to prevent logic decomposition during optimization. Specifying this directive infers MUX_OP cells even if the cells cause loss of resource sharing.

- To generate MUX_OP cells for all case statements in a block, set the `infer_mux` attribute on the block.
  - Attach the `infer_mux` attribute to a process, by using the following syntax:

    ```
    attribute infer_mux of process_label : label is "true";
    ```

- To generate MUX_OP cells for all case and if statements, use the `hdlin_infer_mux` variable.

Variables That Control MUX_OP Inference

The variables that control MUX_OP cell inference are listed in Table 8.

Table 8   MUX_OP Inference Variables

Variable	Description
`hdlin_infer_mux`	Controls MUX_OP inference for all designs you input in the same HDL Compiler session. Options: • `default` – Infers MUX_OPs for case and if statements in processes that have the `infer_mux` directive or attribute attached. • `none` – Does not infer MUX_OPs, regardless of the directives set in the VHDL description. HDL Compiler generates a warning if `hdlin_infer_mux = none` and `infer_mux` are used in the RTL. • `all` – Infers MUX_OPs for every case and if statement in your design for which one can be used. This can negatively affect the quality of results, because it might be more efficient to implement the MUX_OPs as random logic instead of using a specialized multiplexer structure.
`hdlin_mux_size_limit`	Sets the maximum size of a MUX_OP that HDL Compiler can infer. The default is 32. If you set this variable to a value greater than 32, HDL Compiler might take an unusually long elaboration time. If the number of branches in a case statement exceeds the maximum size specified by this variable, HDL Compiler generates the following message: `Warning: A mux was not inferred because case statement %s has a very large branching factor. (HDL-383)`
`hdlin_mux_size_min`	Sets the minimum number of data inputs for a MUX_OP inference. The default is 2.
`hdlin_mux_oversize_ratio`	Defines the ratio of the number of MUX_OP inputs to the unique number of data inputs. When this ratio is exceeded, a MUX_OP will not be inferred and the circuit will be generated with SELECT_OPs. The default is 100.

Table 8 MUX_OP Inference Variables (Continued)

Variable	Description
`hdlin_mux_size_only`	To ensure that MUX_OP cells are mapped to MUX technology cells, you must apply a `size_only` attribute to the cells to prevent logic decomposition in later optimization steps. You can set the `size_only` attribute on each MUX_OP manually or allow the tool to set it automatically. The automatic behavior can be controlled by the `hdlin_mux_size_only` variable. Options:

MUX_OP Inference Examples

In Example 89, two MUX_OPs and one SELECT_OP are inferred, as follows:

- In the process proc1, a MUX_OP is inferred for the case statement, because the `infer_mux` attribute is placed on proc1.
- In the process proc2, there are two case statements.
    - For the first case statement, a SELECT_OP is inferred. This is the default inference.
    - However, the second case statement in proc2 has the `infer_mux` compiler directive set on it which causes HDL Compiler to infer the MUX_OP cell.

Example 90 shows the inference report for the MUX_OPs. Figure 11 shows a representation of the HDL Compiler implementation.

Example 89 Two MUX_OPs and One SELECT_OP Inferred

```
library ieee, synopsys;
use ieee.std_logic_1164.all;
use synopsys.attributes.all;

entity test is
  port (DIN1  : in  std_logic_vector (7 downto 0);
        DIN2  : in  std_logic_vector (7 downto 0);
        DIN3  : in  std_logic_vector (3 downto 0);
        SEL1  : in  std_logic_vector (2 downto 0);
        SEL2  : in  std_logic_vector (2 downto 0);
        SEL3  : in  std_logic_vector (1 downto 0);
        DOUT1 : out std_logic;
        DOUT2 : out std_logic;
        DOUT3 : out std_logic
      );
end test;

architecture rtl of test is

attribute infer_mux of proc1 : label is "TRUE";

begin

  -- A MUX_OP for DOUT1 will be inferred from the
  -- infer_mux attribute set on proc1

  proc1 : process (SEL1, DIN1)
  begin
     case SEL1 is
       when "000"  => DOUT1 <= DIN1(0);
       when "001"  => DOUT1 <= DIN1(1);
       when "010"  => DOUT1 <= DIN1(2);
       when "011"  => DOUT1 <= DIN1(3);
       when "100"  => DOUT1 <= DIN1(4);
       when "101"  => DOUT1 <= DIN1(5);
       when "110"  => DOUT1 <= DIN1(6);
       when "111"  => DOUT1 <= DIN1(7);
       when others => DOUT1 <= DIN1(0);
    end case;
  end process;


  proc2 : process (SEL2, SEL3, DIN2, DIN3)
  begin

  -- A SELECT_OP will be generated for DOUT2
  -- in the absence of an infer_mux attribute

     case SEL2 is
       when "000"  => DOUT2 <= DIN2(0);
       when "001"  => DOUT2 <= DIN2(1);
       when "010"  => DOUT2 <= DIN2(2);
       when "011"  => DOUT2 <= DIN2(3);
       when "100"  => DOUT2 <= DIN2(4);
       when "101"  => DOUT2 <= DIN2(5);
```

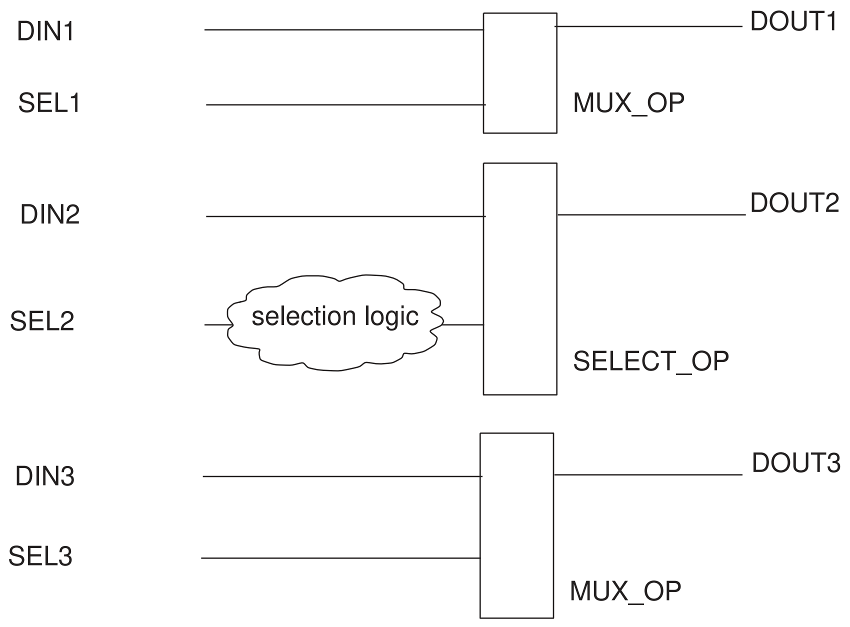
```
        when "110"  => DOUT2 <= DIN2(6);
        when "111"  => DOUT2 <= DIN2(7);
        when others => DOUT2 <= DIN2(0);
      end case;


    -- A MUX_OP will be inferred for DOUT3 from the
    -- infer_mux pragma placed on this case statement

      case SEL3 is -- synopsys infer_mux
        when "00"     => DOUT3 <= DIN3(0);
        when "01"     => DOUT3 <= DIN3(1);
        when "10"     => DOUT3 <= DIN3(2);
        when "11"     => DOUT3 <= DIN3(3);
        when others   => DOUT3 <= DIN3(0);
      end case;
    end process;

  end rtl;
```

Example 90 shows the MUX_OP inference report for the code in Example 89. The tool displays inference reports by default. If you do not want these reports displayed, you can turn them off using the `hdlin_reporting_level` variable. For more information about the `hdlin_reporting_level` variable, see Customizing Elaboration Reports.

Example 90 Inference Report for the MUX_OPs

```
 Statistics for MUX_OPs
===================================================================
|     block name/line       |  Inputs    |  Outputs    | # sel inputs |
===================================================================
|         proc1/24          |     8      |     1       |      3       |
|         proc2/55          |     4      |     1       |      2       |
===================================================================
```

Figure 11 HDL Compiler Implementation



MUX_OP Inference Limitations

HDL Compiler does not infer MUX_OP cells for

- Case statements in while loops
- Case statements embedded in if-then-else statements, unless the case statement appears in an if (CLK’event...) or in an elsif (CLK’event...) branch in the VHDL description

MUX_OP cells are inferred for incompletely specified case statements, such as case statements that

- Contain an if statement or an others clause that covers more than one value
- Have a missing case statement branch or a missing assignment in a case statement branch
- Contain don’t care values (X or "-")
- Are in an elsif (CLK’event...) branch

However, the logic might be nonoptimum, because other optimizations are disabled when you infer MUX_OP cells under these conditions. For example, HDL Compiler optimizes default branches by default. If the `infer_mux` attribute is on the case statement, this optimization is not done.

When inferring a MUX_OP for an incompletely specified case statement, HDL Compiler generates the following ELAB-304 warning:

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

Bit-Truncation Coding for DC Ultra Datapath Extraction

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

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

Note:

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

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

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

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 is one in which you specify the bit range for truncation.
	The following code indicates explicit upper-bit truncation of operand A:
	`signal A = std_logic_vector (i downto 0);`
	`z <= A (j downto 0);      -- where j < i`

Truncation type	Description

Implicit bit truncation	An implicit upper-bit truncation is one that occurs through assignment. Unlike explicit upper-bit truncation, here you do not explicitly define the range for truncation.
	The following code indicates implicit upper-bit truncation of operand Y:
	`signal A,B = std_logic_vector (7 downto 0);`
	`signal C,Y = std_logic_vector (8 downto 0);`
	`Y = A + B + C;`
	Because A and B are each 8 bits wide, the return value of A+B are 8 bits wide. However, because Y, which is 9 bits wide, is assigned to be the 9-bit wide addition (A+B)+C, the most significant bit (MSB) of the addition (A+B) is implicitly truncated. In this example, the MSB is the carryout.

To see how bit truncation affects datapath extraction, consider the code in Example 91. In this example, d has the upper bit truncated, but e is only 8-bits so this code is extracted.

Example 91 Design test1: Truncated Operand Is Extracted

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity test1 is
  port (a,b,c : in std_logic_vector(7 downto 0);
        e : out std_logic_vector(7 downto 0));  -- e is 8-bits wide
end test1;
architecture rtl of test1 is
  signal d : std_logic_vector(15 downto 0);  -- d is 16-bits wide
begin
  d <= a * b;
  e <= c + d (7 downto 0); -- explicit upper bit truncation
end rtl;
```

In Example 92, d is truncated to 8-bits and in an expression assigned to e which is 9-bits. This code is not extracted.

Example 92 Design test2: Truncated Operand Is Not Extracted

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity test2 is
  port (a,b,c : in std_logic_vector(7 downto 0);
        e : out std_logic_vector(8 downto 0));  -- e is 9-bits wide
end test2;
architecture rtl of test2 is
  signal d : std_logic_vector(15 downto 0);  -- d is 16-bit wide
begin
  d <= a * b;
  e <= '0'&c + d(7 downto 0); -- explicit upper bit truncation
end rtl;
```

In Example 93, the expression assigned to e contains implicit upper-bit truncation and the width of e is greater than the width of the implicitly truncated operand, so the code is not extracted.

Example 93 Design test3: Truncated Operand Is Not Extracted

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity test3 is
  port (a,b : in std_logic_vector(7 downto 0);
        e : out std_logic_vector(7 downto 0)); -- e is 8-bits wide

end test3;

architecture rtl of test3 is
  signal d : std_logic_vector(15 downto 0);
  begin
  d <= a * b;
  e <= a + b + d(7 downto 0);   -- implicit upper bit truncation
end rtl;
```

In Example 94, there is lower-bit truncation but no upper-bit truncation so this code is extracted.

Example 94 Design test4: Truncated Operand is Extracted

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity test4 is
  port (a,b : in std_logic_vector(7 downto 0);
        e : out std_logic_vector(7 downto 0));
end test4;
architecture rtl of test4 is
  signal d : std_logic_vector(15 downto 0);  -- d is 16-bit wide
begin
  d <= a * b;            -- no implicit upper bit truncation of d
  e <= d(15 downto 8);   -- explicit lower bit truncation of d
end rtl;
```

Unintended Latches and Feedback Paths in Combinational Logic

HDL Compiler infers a latch when a signal or variable in a combinational process (one without a wait or if signal'event statement) is not fully specified in the VHDL description. A variable or signal is fully specified when it is assigned under all possible conditions. A variable or signal is not fully specified when a condition exists under which the variable is not assigned.

Example 95 shows several variables. A, B, and C are fully specified; X is not.

Example 95 Variable X Is Not Fully Specified

```
process (COND1)
  variable A, B, C, X : BIT;
begin
  A := '0'      -- A is fully specified
  C := '0'      -- C is fully specified

  if (COND1) then
    B := '1';     -- B is assigned when COND1 is TRUE
    C := '1';     -- C is already fully specified
    X := '1';     -- X is assigned when COND1 is TRUE
  else
    B := '0';     -- B is assigned when COND1 is FALSE
  end if;

  -- B is assigned under all branches of if (COND1),
  --   that is, both when COND1 is TRUE and when
  --   COND1 is FALSE, so B is fully specified.

  -- C is assigned regardless of COND1, so C is fully
  -- specified.  (The second assignment to C does
  -- not change this.)

  -- X is not assigned under all branches of
  --   if (COND1), namely, when COND1 is FALSE,
  --   so X is not fully specified.
end process;
...
```

The conditions of each if and else statement are considered independent in Example 95.

In Example 96, variable A is not fully specified.

Example 96 Variable A Is Not Fully Specified

```
if (COND1) then
  A <= '1';
end if;

if (not COND1) then
  A <= '0';
end if;
```

A variable or signal that is not fully specified is considered conditionally specified, and HDL Compiler infers a latch to store the variable value. You can conditionally assign a variable, but you cannot read a conditionally specified variable. You can, however, both conditionally assign and read a signal.

If a fully specified variable is read before its assignment statements, combinational feedback might exist. For example, the following fragment synthesizes combinational feedback for VAL.

```
process(D, LOAD)
  variable VAL: BIT;
begin
  if (LOAD = '1') then
    VAL := D;
  else
    VAL := VAL;
  end if;
  VAL_OUT <= VAL;
end process;
```

In a combinational process, you can ensure that a variable or signal is fully specified, by providing an initial (default) assignment to the variable at the beginning of the process. This default assignment ensures that the variable is always assigned a value, regardless of conditions. Subsequent assignment statements can override the default. A default assignment is made to variables A and C in Example 95.

Another way to ensure that you do not imply combinational feedback is to use a sequential process (one with a wait or if signal'event statement). In such a case, variables and signals are registered. The registers break the combinational feedback loop.

HDL Compiler infers latches for incompletely specified case statements that use an others clause, where the others clause covers more than one value. To avoid latch inference, use a default statement before the case statements.

4

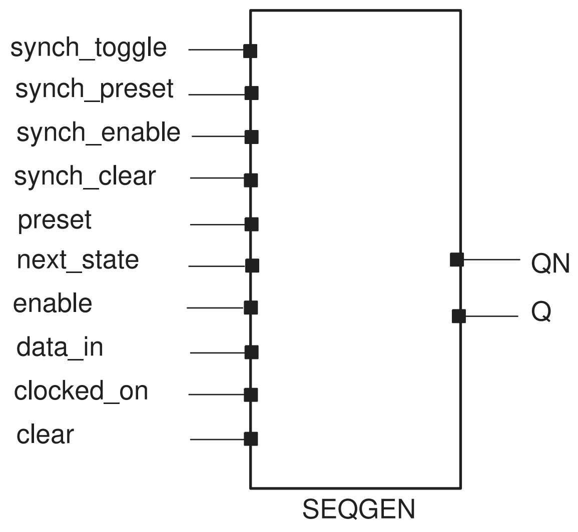
Modeling Sequential Logic

This chapter contains the following sections, which describe how to infer latches and flip-flops:

Generic Sequential Cells (SEQGENs)

When HDL Compiler reads a design, it uses generic sequential cells (SEQGENs), shown in Figure 12, to represent inferred flip-flops and latches.

Figure 12 SEQGEN Cell and Pin Assignments



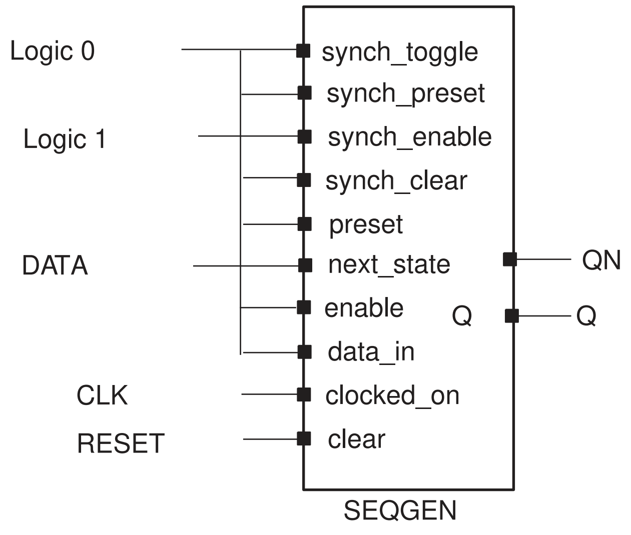
To illustrate how HDL Compiler uses SEQGENs to implement a flip-flop, consider Example 97. This code infers a D flip-flop with an asynchronous reset.

Example 97 D Flip-Flop With Asynchronous Reset

```
library IEEE;
use IEEE.std_logic_1164.all;

entity dff_async_reset is
  port (DATA, CLK, RESET : in std_logic;
        Q : out std_logic );
end dff_async_reset;

architecture rtl of dff_async_reset is
begin
process ( CLK, RESET) begin
  if (RESET = '1') then
    Q <= '0';
  elsif (CLK'event and CLK = '1') then
    Q <= DATA;
  end if;
end process;

end rtl;
```

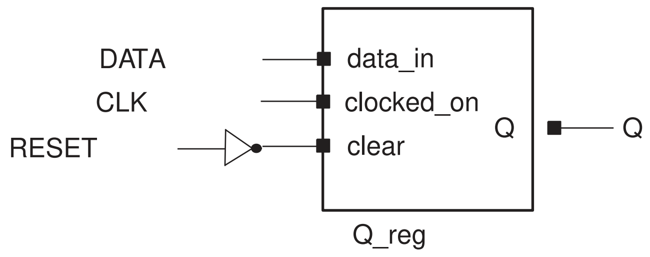
Figure 13 shows the SEQGEN implementation.

Figure 13 D Flip-flop With an Asynchronous Reset: HDL Compiler SEQGEN Implementation



After HDL Compiler compiles the design, SEQGENs are mapped to the appropriate latch or flip-flop in the logic library. Figure 14 shows an example implementation after compile.

Figure 14 D Flip-flop With an Asynchronous Reset: HDL Compiler Implementation



Note:

If the logic library does not contain the specific inferred flip-flop or latch, HDL Compiler creates combinational logic for the missing function, if possible. For example, if you infer a D flip-flip with a synchronous set but your target logic library does not contain this type of flip-flop, HDL Compiler creates combinational logic for the synchronous set function. HDL Compiler cannot create logic to duplicate an asynchronous preset/reset. Your library must contain the sequential cell with the asynchronous control pins.

Inference Reports for Flip-Flops and Latches

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 98. This basic inference report shows only which type of register was inferred.

Example 98 Inference Report for a D Flip-Flop With Asynchronous Reset

```
============================================================================
| Register Name   |   Type    | Width | Bus | MB | Set  | Reset | ST | Line |
============================================================================
|  Q_reg          | Flip-flop |   1   |  N  | N  | None | None  | N  |  17  |
============================================================================
```

In the report, the columns are abbreviated as follows:

- MB represents multibit cell
- ST represents synchronous toggle
- Line represents the RTL line from which the register is inferred

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 99 shows a verbose inference report.

Example 99 Verbose Inference Report for a D Flip-Flop With Asynchronous Reset

```
============================================================================
| Register Name   |   Type    | Width | Bus | MB | Set  | Reset | ST | Line |
============================================================================
|  Q_reg          | Flip-flop |   1   |  N  | N  | None | None  | N  |  17  |
============================================================================

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`.

Register Inference Variables

The variables in Table 9 control register inference. These are set before the design is read and apply to all applicable cells in the design. Use of these variables can have unintended consequences. For example, when the `hdlin_ff_always_sync_set_reset` variable is set to true, HDL Compiler treats every signal in every process as though the `sync_set_reset` directive is attached to it. Therefore, it checks all processes for all constant (0 or 1) assignments for a register input. The control for these constant-assigned signals becomes part of the set/reset logic.

Table 9 Variables That Control Register Inference

Variable	Description
`hdlin_keep_feedback` (Default is false)	When this variable is true, HDL Compiler keeps all flip-flop feedback loops. When this variable is false, HDL Compiler removes all flip-flop feedback loops. For example, HDL Compiler removes feedback loops inferred from a statement such as Q=Q. Removing the state feedback from a simple D flip-flop creates a synchronous loaded flip-flop.
`hdlin_ff_always_sync_set_reset` (Default is false)	When this variable is true, HDL Compiler attempts to infer synchronous set and reset conditions for all flip-flops.
`hdlin_ff_always_async_set_reset`(Default is true)	When this variable is true, HDL Compiler attempts to infer asynchronous set and reset conditions for all flip-flops.

Register Inference Attributes

Use the attributes in Table 10 to direct HDL Compiler to the type of sequential device you want inferred. Attributes are added to the RTL on specific processes.

Table 10 Attributes for Controlling Register Inference

Attribute	Description
`async_set_reset`	When a single-bit signal has this attribute set to true, 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. See the examples in Inferring SR Latches, D Latch With Asynchronous Set, and D Latch With Asynchronous Reset. Attach this attribute to 1-bit signals by using the following syntax: `attribute async_set_reset of` *`signal_name_list`* `: signal is "true";`

Table 10 Attributes for Controlling Register Inference (Continued)

Attribute	Description
`async_set_reset_local`	HDL Compiler treats listed signals in the specified process as if they have the `async_set_reset` attribute set to `true`. Attach this attribute to a process label by using the following syntax: `attribute async_set_reset_local of process_label : label is "signal_name_list";`
`async_set_reset_local_all`	HDL Compiler treats all signals in the specified processes as if they have the `async_set_reset` attribute set to `true`. Attach this attribute to process labels by using the following syntax: `attribute async_set_reset_local_all of process_label_list : label is "true";`
`sync_set_reset`	When a single-bit signal has this attribute set to `true`, HDL Compiler checks the signal to determine whether it synchronously sets or resets a register in the design. See the examples in D Flip-Flop With Synchronous Set and D Flip-Flop With Synchronous Reset. Attach this attribute to 1-bit signals by using the following syntax: `attribute sync_set_reset of` *`signal_name_list`* `:  signal is "true";`
`sync_set_reset_local`	HDL Compiler treats listed signals in the specified process as if they have the `sync_set_reset` attribute set to `true`. Attach this attribute to a process label by using the following syntax: `attribute sync_set_reset_local of process_label : label is "signal_name_list";`
`sync_set_reset_local_all`	HDL Compiler treats all signals in the specified processes as if they have the `sync_set_reset` attribute set to `true`. Attach this attribute to process labels by using the following syntax: `attribute sync_set_reset_local_all of process_label_list : label is "true";`

`one_cold` or `one_hot`	These attributes prevent HDL Compiler from implementing priority-encoding logic for the set and reset signals and are useful if you know your design has a one-hot or one-cold implementation. See the examples in D Latch With Asynchronous Set and Reset, D Flip-Flop With Asynchronous Set and Reset, and JK Flip-Flop With Asynchronous Set and Reset.
	Attach the attributes to set or reset signals on sequential devices by using the following syntax:
	`attribute one_cold` *`signal_name_list`* `: signal is "true";`
	or
	`attribute one_hot` *`signal_name_list`* `: signal is "true";`
	You might want to add an assertion to the VHDL code to ensure that the group of signals has a one-cold or one-hot implementation. HDL Compiler does not produce any logic to check this assertion.

Table 10 Attributes for Controlling Register Inference (Continued)

Attribute	Description
`clocked_on_also`	This attribute is set in an embedded HDL Compiler script and used for master-slave inference. For more information, see Inferring Master-Slave Latches.

Inferring D and Set/Reset (SR) Latches

This section describes how to infer SR and D latches, in the following subsections:

- Inferring SR Latches
- Inferring D Latches
- Limitations of D Latch Inference

Inferring SR Latches

Use SR latches with caution, because they are difficult to test. HDL Compiler does not ensure that the logic driving the inputs is hazard-free, so you must verify that the inputs are hazard-free and do not glitch. Example 100 provides the VHDL code that implements the SR latch described in the truth table in Table 11. Example 101 shows the inference report generated by HDL Compiler.

Table 11 SR Latch Truth Table (NAND Type)

Set	Reset	y
0	0	Not stable
0	1	1
1	0	0
1	1	y

Example 100 SR Latch

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity sr_latch is
  port (SET, RESET : in std_logic;
        Q : out std_logic );
  attribute async_set_reset of SET, RESET :
    signal is "true";
end sr_latch;

architecture rtl of sr_latch is
begin
process (SET, RESET) begin
  if (SET = '0') then
    Q <= '1';
  elsif (RESET = '0') then
    Q <= '0';
  end if;
end process;

end rtl;
```

Example 101 Inference Report for an SR Latch

```
============================================================================
| Register Name  |   Type    | Width | Bus | MB | Set  | Reset | ST | Line |
============================================================================
|  Q_reg         | Flip-flop |   1   |  N  | N  | None | None  | N  |  17  |
============================================================================

Sequential Cell (Q_reg)
        Cell Type: Latch
        Multibit Attribute: N
        Clock: 0
        Async Clear: RESET'
        Async Set: SET'
        Async Load: 0
```

Inferring D Latches

The following sections provide code examples and inference reports for D latches:

- Overview—Latch Inference
- Basic D Latch
- D Latch With Asynchronous Set
- D Latch With Asynchronous Reset
- D Latch With Asynchronous Set and Reset

Overview—Latch Inference

When you do not specify a variables value under all conditions, such as an incompletely specified if statement, HDL Compiler infers a D latch.

For example, the if statement in Example 102 infers a D latch, because there is no else clause. The resulting value for output Q is specified only when input enable has a logic 1 value. As a result, output Q becomes a latched value.

Example 102 Latch Inference

```
process(DATA, GATE) begin
  if (GATE = '1') then
    Q <= DATA;
  end if;
end process;
```

To avoid latch inference, assign a value to the signal under all conditions, as shown in Example 103.

Example 103 Fully Specified Signal: No Latch Inference

```
process(DATA, GATE) begin
  if (GATE = '1') then
    Q <= DATA;
  else
    Q <= '0';
  end if;
end process;
```

Variables declared locally within a subprogram do not hold their value over time, because each time a subprogram is called, its variables are reinitialized. Therefore, HDL Compiler does not infer latches for variables declared in subprograms. In Example 104, HDL Compiler does not infer a latch for output Q.

Example 104 Function: No Latch Inference

```
function MY_FUNC(DATA, GATE : std_logic) return std_logic is
     variable STATE: std_logic;
begin
  if (GATE = '1') then
    STATE := DATA;
  end if;
  return STATE;
end;
. . .
Q <= MY_FUNC(DATA, GATE);
```

Basic D Latch

When you infer a D latch, make sure you can control the gate and data signals from the top-level design ports or through combinational logic. Controllable gate and data signals ensure that simulation can initialize the design.

Example 105 provides the VHDL template for a D latch. HDL Compiler generates the verbose inference report shown in Example 106.

Example 105 Basic D Latch

```
library IEEE;
use IEEE.std_logic_1164.all;

entity d_latch is
  port (GATE, DATA: in std_logic;
        Q : out std_logic );
end d_latch;

architecture rtl of d_latch is
begin
process (GATE, DATA) begin
  if (GATE = '1') then
    Q <= DATA;
  end if;
end process;

end rtl;
```

Example 106 Verbose Inference Report for a D Latch

```
===========================================================================
| Register Name  |   Type    | Width | Bus | MB | Set   | Reset | ST | Line |
===========================================================================
|  Q_reg         | Flip-flop |   1   |  N  | N  | None | None  | N  |  17  |
===========================================================================

Sequential Cell (Q_reg)
        Cell Type: Latch
        Multibit Attribute: N
        Clock: 0
        Async Clear: 0
        Async Set: 0
        Async Load: GATE
```

D Latch With Asynchronous Set

Use the `async_set_reset` attribute to specify the asynchronous set or reset controls. HDL Compiler examines the polarity of the constants assigned to the signals with the `async_set_reset` attribute to determine if the signal is an AR ('0') or an AS ('1').

Example 107 provides the VHDL template for a D latch with an asynchronous set. HDL Compiler generates the verbose inference report shown in Example 108.

Example 107 D Latch With Asynchronous Set

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity d_latch_async_set is
  port (GATE, DATA, SET : in std_logic;
        Q : out std_logic );
  attribute async_set_reset of SET :
    signal is "true";
end d_latch_async_set;

architecture rtl of d_latch_async_set is
begin
process (GATE, DATA, SET) begin
  if (SET = '0') then
    Q <= '1';
  elsif (GATE = '1') then
    Q <= DATA;
  end if;
end process;

end rtl;
```

Example 108 Verbose Inference Report for a D Latch With Asynchronous Set

```
============================================================================
| Register Name   |    Type    | Width | Bus | MB | Set   | Reset | ST | Line |
============================================================================
|  Q_reg          | Flip-flop |   1    |  N   | N  | None | None   | N  |  17  |
============================================================================

Sequential Cell (Q_reg)
        Cell Type: Latch
        Multibit Attribute: N
        Clock: 0
        Async Clear: 0
        Async Set: SET'
        Async Load: GATE
```

D Latch With Asynchronous Reset

Use the `async_set_reset` attribute to specify asynchronous set or reset controls.

Example 109 provides the VHDL template for a D latch with an asynchronous reset. HDL Compiler generates the verbose inference report shown in Example 110.

Example 109 D Latch With Asynchronous Reset

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity d_latch_async_reset is
  port (GATE, DATA, RESET : in std_logic;
        Q : out std_logic );
  attribute async_set_reset of RESET :
    signal is "true";
end d_latch_async_reset;

architecture rtl of d_latch_async_reset is
begin
process (GATE, DATA, RESET) begin
  if (RESET = '0') then
```

```
        Q <= '0';
      elsif (GATE = '1') then
        Q <= DATA;
      end if;
    end process;
    end rtl;
```

Example 110 Inference Report for D Latch With Asynchronous Reset

```
===========================================================================
| Register Name   |   Type    | Width | Bus | MB | Set   | Reset | ST | Line |
===========================================================================
|  Q_reg          | Flip-flop |   1   |  N  | N  | None | None  | N  |  17  |
===========================================================================

Sequential Cell (Q_reg)
        Cell Type: Latch
        Multibit Attribute: N
        Clock: 0
        Async Clear: RESET'
        Async Set: 0
        Async Load: GATE
```

D Latch With Asynchronous Set and Reset

Example 111 provides the VHDL template for a D latch with an active-low asynchronous set and reset. This template uses the `async_set_reset` attribute to direct HDL Compiler to the asynchronous signals in the process.

The template in Example 111 uses the `one_cold` attribute to prevent priority encoding of the set and reset signals. If you do not specify the `one_cold` attribute, the set signal has priority, because it is used as the condition for the if clause. Use `one_cold` for active-low signals and `one_hot` for active-high signals. Example 112 shows the verbose inference report.

Example 111 D Latch With Asynchronous Set and Reset

```
    library IEEE, synopsys;
    use IEEE.std_logic_1164.all;
    use synopsys.attributes.all;
    entity d_latch_async is
      port (GATE, DATA, SET, RESET :in std_logic;
            Q : out std_logic );
      attribute one_cold of SET, RESET :
        signal is "true";
    end d_latch_async;
    architecture rtl of d_latch_async is
      attribute async_set_reset of SET, RESET :
        signal is "true";
    begin
    process (GATE, DATA, SET, RESET) begin
      if (SET = '0') then
        Q <= '1';
      elsif (RESET = '0') then
        Q <= '0';
      elsif (GATE = '1') then
        Q <= DATA;
      end if;
    end process;
    end rtl;
```

Example 112 Inference Report for D Latch With Asynchronous Set and Reset

```
============================================================================
| Register Name  |   Type    | Width | Bus | MB | Set   | Reset | ST | Line |
============================================================================
|  Q_reg         | Flip-flop |   1   |  N  | N  | None | None  | N  |  17  |
============================================================================

Sequential Cell (Q_reg)
        Cell Type: Latch
        Multibit Attribute: N
        Clock: 0
        Async Clear: RESET'
        Async Set: SET'
        Async Load: GATE
```

Limitations of D Latch Inference

A variable must always have a value before it is read. As a result, you cannot read a conditionally assigned variable after the if statement in which it is assigned. A conditionally assigned variable is assigned a new value under some, but not all, conditions. Example 113 shows an invalid use of the conditionally assigned variable VALUE.

Example 113 Invalid Use of a Conditionally Assigned Variable

```
signal X, Y : std_logic;
. . .
process
  variable VALUE : std_logic;
begin

  if (condition) then
    VALUE := X;
  end if;

  Y <= VALUE;  -- Invalid read of variable VALUE
end;
```

Inferring D Flip-Flops

The following subsections describe various types of D flip-flop inference:

- Overview—Inferring D Flip-Flops
- Enabling Conditions in if Statements

Overview—Inferring D Flip-Flops

HDL Compiler infers a D flip-flop whenever the condition of a wait or if statement uses an edge expression. Use the following syntax to describe a rising edge:

```
SIGNAL'event and SIGNAL = '1'
```

Use the following syntax to describe a falling edge:

```
SIGNAL'event and SIGNAL = '0'
```

If you are using the IEEE std_logic_1164 package, you can use the following syntax to describe a rising edge and a falling edge:

```
if (rising_edge (CLK)) then

if (falling_edge (CLK)) then
```

You can use the following syntax for a bused clock. You can also use a member of a bus as a signal.

```
sig(3)'event and sig(3) = '1'

rising_edge (sig(3))
```

If possible, use the if statement, because it provides greater control over the inferred registers. Only one wait statement per process is allowed.

In a process that models sequential logic, HDL Compiler allows statements to precede or to follow the if statement if no statement following the if statement tries to write a value that is assigned within the if statement. See Example 114.

Example 114 HDL Compiler Supports Statements Preceding and Following if ck'EVENT

```
P: process (ck)
  variable X, Y: BIT;
 begin
  Y := not D; -- assignment before the if statement
  if ck'EVENT and ck = '1' then
   X := D;
  end if;
  Q <= X and Y;  -- assignment after the if statement
 end process;
```

There are cases in which statements appearing before the if statement would make the code cannot be synthesized. Specifically, when the statements preceding the if statement writes to a variable that was also written to within the if body, as shown in Example 115, the code would not be synthesizable.

Example 115 Code Cannot Be Synthesized—if ck'EVENT Statement Writes to a Variable Written to in Body

```
P: process (ck)
  variable X: BIT;
 begin
  X := D1;
  if ck'EVENT and CK = '1' then
   X := D2; -- conflicts with previous assignment
  end if;
  Q <= X;
 end process;
```

Enabling Conditions in if Statements

HDL Compiler allows conditions in the test of the if statement that are not part of the clock edge test. When other conditions appear in the test, HDL Compiler synthesizes them by assuming they are enable conditions. HDL Compiler also recognizes permutations of the conditions in the if statement. Example 116 shows the coding style supported by HDL Compiler.

Example 116 HDL Compiler Supports Enabling Expressions in if Statements

```
process (ck)
begin
  if (ck = '1' and en = '1' and ck'EVENT) then
    --sequential_statements
  end if;
end process;
```

Rising-Edge-Triggered 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 asynchronous reset or set, or with synchronous reset or set.

Example 117 uses the 'event attribute and Example 118 uses the rising_edge function to code a rising-edge-triggered D flip-flop. Example 119 shows the verbose inference report.

Example 117 Rising-Edge-Triggered D Flip-Flop Using ‘event Attribute

```
library IEEE ;
use IEEE.std_logic_1164.all;

entity dff_pos is
  port (DATA, CLK : in std_logic;
        Q : out std_logic );
end dff_pos;

architecture rtl of dff_pos is
begin
process (CLK) begin
  if (CLK'event and CLK = '1') then
    Q <= DATA;
  end if;
end process;
end rtl;
```

Example 118 Rising-Edge-Triggered D Flip-Flop Using rising_edge

```
library IEEE ;
use IEEE.std_logic_1164.all;

entity dff_pos is
  port (DATA, CLK : in std_logic;
        Q : out std_logic );
end dff_pos;
architecture rtl of dff_pos is
begin
process (CLK) begin
  if (rising_edge (CLK)) then
    Q <= DATA;
  end if;
end process;
end rtl;
```

Example 119 Inference Report for Rising-Edge-Triggered D Flip-Flop

```
============================================================================
| Register Name  |   Type    | Width | Bus | MB | Set  | Reset | ST | Line |
============================================================================
|  Q_reg         | Flip-flop |   1   |  N  | N  | None | None  | N  |  17  |
============================================================================

Sequential Cell (Q_reg)
 Cell Type: Flip-Flop
 Multibit Attribute: N
 Clock: CLK
 Async Clear: 0
 Async Set: 0
 Async Load: 0
 Sync Clear: 0
 Sync Set: 0
 Sync Toggle: 0
 Sync Load: 1
```

Falling-Edge-Triggered D Flip-Flop

Example 120 uses the ‘event attribute and Example 121 uses the falling_edge function to code a falling-edge-triggered D flip-flop.

HDL Compiler generates the verbose inference report shown in Example 122.

Example 120 Falling-Edge-Triggered D Flip-Flop Using ‘event

```
library IEEE;
use IEEE.std_logic_1164.all;

entity dff_neg is
  port (DATA, CLK : in std_logic;
        Q : out std_logic );
end dff_neg;

architecture rtl of dff_neg is
begin
process (CLK) begin
  if (CLK'event and CLK = '0') then
    Q <= DATA;
  end if;
end process;
end rtl;
```

Example 121 Falling-Edge-Triggered D Flip-Flop Using falling_edge

```
library IEEE;
use IEEE.std_logic_1164.all;

entity dff_neg is
  port (DATA, CLK : in std_logic;
        Q : out std_logic );
end dff_neg;

architecture rtl of dff_neg is
begin
process (CLK) begin
  if (falling_edge (CLK)) then
```

```
    Q <= DATA;
  end if;
end process;

end rtl;
```

Example 122 Inference Report for Falling-Edge-Triggered D Flip-Flop

```
============================================================================
| Register Name  |    Type    | Width | Bus  | MB  | Set   | Reset  | ST  | Line |
============================================================================
|  Q_reg         | Flip-flop |   1   |  N   | N   | None | None   | N   |  17  |
============================================================================

Sequential Cell (Q_reg)
 Cell Type: Flip-Flop
 Multibit Attribute: N
 Clock: CLK'
 Async Clear: 0
 Async Set: 0
 Async Load: 0
 Sync Clear: 0
 Sync Set: 0
 Sync Toggle: 0
 Sync Load: 1
```

D Flip-Flop With Asynchronous Set

Example 123 provides the VHDL template for a D flip-flop with an asynchronous set. HDL Compiler generates the verbose inference report shown in Example 124.

Example 123 D Flip-Flop With Asynchronous Set

```
library IEEE;
use IEEE.std_logic_1164.all;

entity dff_async_set is
  port (DATA, CLK, SET : in std_logic;
        Q : out std_logic );
end dff_async_set;

architecture rtl of dff_async_set is
begin
process (CLK, SET) begin
  if (SET = '0') then
    Q <= '1';
  elsif (CLK'event and CLK = '1') then
    Q <= DATA;
  end if;
end process;
end rtl;
```

Example 124 Inference Report for a D Flip-Flop With Asynchronous Set

```
=============================================================================
| Register Name  |    Type    | Width | Bus | MB | Set   | Reset | ST | Line |
=============================================================================
|  Q_reg         | Flip-flop |   1   |  N  | N  | None | None  | N  |  17  |
=============================================================================

Sequential Cell (Q_reg)
 Cell Type: Flip-Flop
 Multibit Attribute: N
 Clock: CLK
 Async Clear: 0
 Async Set: SET'
 Async Load: 0
 Sync Clear: 0
 Sync Set: 0
 Sync Toggle: 0
 Sync Load: 1
```

D Flip-Flop With Asynchronous Reset

Example 125 provides the VHDL template for a D flip-flop with an asynchronous reset. HDL Compiler generates the verbose inference report shown in Example 126.

Example 125 D Flip-Flop With Asynchronous Reset

```
library IEEE;
use IEEE.std_logic_1164.all;

entity dff_async_reset is
  port (DATA, CLK, RESET : in std_logic;
        Q : out std_logic );
end dff_async_reset;

architecture rtl of dff_async_reset is
begin
process ( CLK, RESET) begin
  if (RESET = '1') then
    Q <= '0';
  elsif (CLK'event and CLK = '1') then
    Q <= DATA;
  end if;
end process;
end rtl;
```

Example 126 Inference Report for a D Flip-Flop With Asynchronous Reset

```
============================================================================
| Register Name  |    Type    | Width | Bus | MB | Set  | Reset | ST | Line |
============================================================================
|  Q_reg         | Flip-flop |   1   |  N  | N  | None | None  | N  |  17  |
============================================================================

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
```

D Flip-Flop With Asynchronous Set and Reset

Example 127 provides the VHDL template for a D flip-flop with active-high asynchronous set and reset pins.

The template in Example 127 uses the `one_hot` attribute to prevent priority encoding of the set and reset signals. If you do not specify the `one_hot` attribute, the reset signal has priority, because it is used as the condition for the if clause. The `one_cold` attribute would be used instead of the `one_hot` if you had active-low signals. HDL Compiler generates the verbose inference report shown in Example 128.

Example 127 D Flip-Flop With Asynchronous Set and Reset

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;
entity dff_async is
  port (DATA, CLK, SET, RESET : in std_logic;
        Q : out std_logic );
  attribute one_hot of SET, RESET : signal is "true";
end dff_async;

architecture rtl of dff_async is
begin
process (CLK, SET, RESET) begin
  if (RESET = '1') then
    Q <= '0';
  elsif (SET = '1') then
    Q <= '1';
  elsif (CLK'event and CLK = '1') then
    Q <= DATA;
  end if;
end process;
end rtl;
```

Example 128 Inference Report for a D Flip-Flop With Asynchronous Set and Reset

```
=============================================================================
| Register Name  |    Type    | Width | Bus | MB | Set  | Reset | ST | Line |
=============================================================================
|  Q_reg         | Flip-flop |   1    |  N  | N  | None | None  | N  |  17  |
=============================================================================

Sequential Cell (Q_reg)
 Cell Type: Flip-Flop
 Multibit Attribute: N
 Clock: CLK
 Async Clear: RESET
 Async Set: SET
 Async Load: 0
 Sync Clear: 0
 Sync Set: 0
 Sync Toggle: 0
 Sync Load: 1
```

D Flip-Flop With Synchronous Set

Use the `sync_set_reset` compiler directive to infer a D flip-flop with a synchronous set/reset. When you compile your design, the SEQGEN inferred by HDL Compiler is either

- Mapped to a flip-flop in the logic library with a synchronous set/reset pin or
- Mapped to a regular D flip-flop. In this case, HDL Compiler builds 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` compiler directive to label the set/reset signal. This compiler directive 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.

Example 129 shows the recommended coding style for a synchronous set/reset flip-flop, using the `sync_set_reset` compiler directive. HDL Compiler generates the verbose inference report shown in Example 130

Example 129 D Flip-Flop With Synchronous Set

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity dff_sync_set is
  port (DATA, CLK, SET : in std_logic;
        Q : out std_logic );
  attribute sync_set_reset of SET : signal is "true";
end dff_sync_set;
architecture rtl of dff_sync_set is
begin
process (CLK) begin
```

```
      if (CLK'event and CLK = '1') then
        if (SET = '1') then
          Q <= '1';
        else
          Q <= DATA;
        end if;
      end if;
    end process;
    end rtl;
```

Example 130 Inference Report for a D Flip-Flop With Synchronous Set

```
============================================================================
| Register Name  |    Type    | Width | Bus  | MB  | Set   | Reset  | ST | Line |
============================================================================
|  Q_reg         | Flip-flop  |   1   |  N   | N   | None | None   | N  |  17  |
============================================================================

Sequential Cell (Q_reg)
 Cell Type: Flip-Flop
 Multibit Attribute: N
 Clock: CLK
 Async Clear: 0
 Async Set: 0
 Async Load: 0
 Sync Clear: 0
 Sync Set: SET
 Sync Toggle: 0
 Sync Load: 1
```

D Flip-Flop With Synchronous Reset

Use the `sync_set_reset` compiler 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` compiler directive to label the set/reset signal. This compiler directive 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.

Example 131 shows the recommended coding style for a synchronous set/reset flip-flop using the `sync_set_reset` compiler directive. HDL Compiler generates the verbose inference report shown in Example 132.

Example 131 D Flip-Flop With Synchronous Reset

```
    library IEEE, synopsys;
    use IEEE.std_logic_1164.all;
    use synopsys.attributes.all;

    entity dff_sync_reset is
```

```
  port (DATA, CLK, RESET : in std_logic;
        Q : out std_logic );
  attribute sync_set_reset of RESET :
    signal is "true";
end dff_sync_reset;

architecture rtl of dff_sync_reset is
begin
process (CLK) begin
  if (CLK'event and CLK = '1') then
    if (RESET = '0') then
      Q <= '0';
    else
      Q <= DATA;
    end if;
  end if;
end process;

end rtl;
```

Example 132 Inference Report for a D Flip-Flop With Synchronous Reset

```
=============================================================================
| Register Name  |   Type    | Width | Bus  | MB  | Set   | Reset  | ST  | Line |
=============================================================================
|  Q_reg         | Flip-flop |   1   |  N   | N   | None | None   | N   |  17  |
=============================================================================

Sequential Cell (Q_reg)
 Cell Type: Flip-Flop
 Multibit Attribute: N
 Clock: CLK
 Async Clear: 0
 Async Set: 0
 Async Load: 0
 Sync Clear: RESET'
 Sync Set: 0
 Sync Toggle: 0
 Sync Load: 1
```

D Flip-Flop With Complex Set/Reset Signals

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

- Apply the appropriate set/reset attribute (`sync_set_reset` or `async_set_reset`) to the set/reset signal. For example,

```
entity data is
  port (DATA : in std_logic;
        CLK : in std_logic;
        RESET: in std_logic;
        ENABLE : in std_logic;
```

```
          Q : out std_logic );
      attribute async_set_reset of RESET :
        signal is "true";
    end data;
```

- 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, the following coding styles are not supported: using a subscripted value as reset, 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.

Examples:

```
process(...)
  if (rst='1' OR logic_expression)
    q <= 0;

  else ...
  else ...

...

a <= rst OR NOT( a | b & c);
process(...)
if (a)
q = 0;
else ...;
else ...;

...
process(...)
if ( NOT rst OR NOT (a OR b OR c))
q = 0;
else ...
else ...
```

D Flip-Flop With Synchronous and Asynchronous Load

To infer a component with both synchronous and asynchronous controls, you must check the asynchronous conditions before you check the synchronous conditions.

Example 133 provides the VHDL template for a D flip-flop with a synchronous load (called SLOAD) and an asynchronous load (called ALOAD). HDL Compiler generates the verbose inference report shown in Example 134.

Example 133 D Flip-Flop With Synchronous and Asynchronous Load

```
library IEEE;
use IEEE.std_logic_1164.all;

entity dff_a_s_load is
port(SLOAD, ALOAD, ADATA, SDATA,CLK : in std_logic; Q : out std_logic );
end dff_a_s_load;

architecture rtl of dff_a_s_load is

signal asyn_rst, asyn_set :std_logic ;
begin
asyn_set <= ALOAD AND  (ADATA);
asyn_rst <= ALOAD AND  NOT(ADATA);

process (CLK,asyn_set, asyn_rst)
begin
if (asyn_set ='1') then
Q <= '1';
elsif (asyn_rst ='1') then
q <= '0';
elsif (clk'event and clk ='1' and SLOAD = '1') then Q <= SDATA ; end if;
end process;

end rtl;
```

Example 134 Inference Report for a D Flip-Flop With Synchronous and Asynchronous Load

```
============================================================================
| Register Name  |   Type    | Width | Bus | MB | Set  | Reset | ST | Line |
============================================================================
|  Q_reg         | Flip-flop |   1   |  N  | N  | None | None  | N  |  17  |
============================================================================

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
```

Multiple Flip-Flops: Asynchronous and Synchronous Controls

If a signal is synchronous in one process but asynchronous in another, set both the `sync_set_reset` and `async_set_reset` attributes on the signal.

In Example 135, the `infer_sync` process uses the reset signal as a synchronous reset and the `infer_async` process uses the reset signal as an asynchronous reset. Example 136 shows the verbose inference report.

Example 135 Multiple Flip-Flops: Asynchronous and Synchronous Controls

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity multi_attr is
  port (DATA1, DATA2, CLK, RESET, SLOAD : in std_logic;
        Q1, Q2 : out std_logic );
end multi_attr;

architecture rtl of multi_attr is
  attribute async_set_reset of RESET :
    signal is "true";
  attribute sync_set_reset of RESET :
    signal is "true";
begin

infer_sync: process (CLK) begin
  if (CLK'event and CLK = '1') then
    if (RESET = '0') then
      Q1 <= '0';
    elsif (SLOAD = '1') then
      Q1 <= DATA1;
    end if;
  end if;
end process infer_sync;

infer_async: process (CLK, RESET) begin
  if (RESET = '0') then
    Q2 <= '0';
  elsif (CLK'event and CLK = '1' and SLOAD = '1') then
    Q2 <= DATA2;
  end if;
end process infer_async;

end rtl;
```

Example 136 Verbose Inference Reports

```
Inferred memory devices in process
 in routine multi_attr line 17 in file
  '/remote/vhdl_example/multi_attr.vhd'.
=============================================================================
| Register Name  |   Type    | Width | Bus | MB | Set  | Reset | ST | Line |
=============================================================================
|  Q_reg         | Flip-flop |   1   |  N  | N  | None | None  | N  |  17  |
=============================================================================

Inferred memory devices in process
```

```
 in routine multi_attr line 27 in file
  '/remote/vhdl_example/multi_attr.vhd'.
===========================================================================
| Register Name   |   Type    | Width | Bus | MB | Set   | Reset | ST | Line |
===========================================================================
|  Q_reg          | Flip-flop |   1   |  N  | N  | None | None   | N  |  17  |
===========================================================================
Sequential Cell (Q1_reg)
 Cell Type: Flip-Flop
 Multibit Attribute: N
 Clock: CLK
 Async Clear: 0
 Async Set: 0
 Async Load: 0
 Sync Clear: RESET'
 Sync Set: 0
 Sync Toggle: 0
 Sync Load: SLOAD

Sequential Cell (Q2_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: SLOAD
```

Inferring JK Flip-Flops

This section contains code examples and inference reports for the following types of JK flip-flops:

- Basic JK Flip-Flop
- JK Flip-Flop With Asynchronous Set and Reset

Basic JK Flip-Flop

When you infer a JK flip-flop, make sure you can control the J, K, and clock signals from the top-level design ports to ensure that simulation can initialize the design.

Example 137 provides the VHDL code that implements the JK flip-flop described in the truth table in Table 12.

In the JK flip-flop, the J and K signals act as active-high synchronous set and reset. Use the `sync_set_reset` attribute to indicate that the J and K signals are the synchronous set and reset for the design.

Example 138 shows the verbose inference report generated by HDL Compiler.

Table 12 Truth Table for JK Flip-Flop

J	K	CLK	Qn+1
0	0	Rising	Qn
0	1	Rising	0
1	0	Rising	1
1	1	Rising	not (Qn)
X	X	Falling	Qn

Example 137 JK Flip-Flop

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity jk is
  port(J, K, CLK : in std_logic;
       Q_out : out std_logic );
  attribute sync_set_reset of J, K :
    signal is "true";
end jk;

architecture rtl of jk is
  signal Q : std_logic;
begin
process
  variable JK : std_logic_vector ( 1 downto 0);
begin
  wait until (CLK'event and CLK = '1');
  JK := (J & K);
  case JK is
    when "01" => Q <= '0';
    when "10" => Q <= '1';
    when "11" => Q <= not (Q);
    when "00" => Q <= Q;
    when others => Q <= 'X';
  end case;
end process;

Q_out <= Q;
end rtl;
```

Example 138 Inference Report for a JK Flip-Flop

```
===========================================================================
| Register Name  |    Type     | Width | Bus | MB | Set   | Reset | ST | Line |
============================================================================
|  Q_reg           | Flip-flop |   1    |  N   | N   | None | None   | N  |  17  |
============================================================================

Sequential Cell (Q_reg)
 Cell Type: Flip-Flop
 Multibit Attribute: N
 Clock: CLK
 Async Clear: 0
 Async Set: 0
 Async Load: 0
 Sync Clear: J' K
 Sync Set: J K'
 Sync Toggle: 0
 Sync Load: K
```

JK Flip-Flop With Asynchronous Set and Reset

Example 139 provides the VHDL template for a JK flip-flop with asynchronous set and reset. Use the `sync_set_reset` attribute to indicate the JK function. Use the `one_hot` attribute to prevent priority encoding of the set and reset signals. HDL Compiler generates the verbose inference report shown in Example 140.

Example 139 JK Flip-Flop With Asynchronous Set and Reset

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all;

entity jk_async_sr is
  port (SET, RESET, J, K, CLK : in std_logic;
        Q_out : out std_logic );
  attribute sync_set_reset of J, K :
    signal is "true";
  attribute one_hot of SET,RESET : signal is "true";
end jk_async_sr;

architecture rtl of jk_async_sr is
  signal Q : std_logic;
begin
process (CLK, SET, RESET)
  variable JK : std_logic_vector (1 downto 0);
begin
  if (RESET = '1') then
    Q <= '0';
  elsif (SET = '1') then
    Q <= '1';
  elsif (CLK'event and CLK = '1') then
    JK := (J & K);
    case JK is
      when "01" => Q <= '0';
      when "10" => Q <= '1';
      when "11" => Q <= not(Q);
      when "00" => Q <= Q;
      when others => Q <= 'X';
    end case;
  end if;
end process;
```

```
Q_out <= Q;
end rtl;
```

Example 140 Inference Report for a JK Flip-Flop With Asynchronous Set and Reset

```
=============================================================================
| Register Name  |    Type    | Width | Bus | MB | Set   | Reset  | ST | Line |
=============================================================================
|   Q_reg        | Flip-flop |   1    |  N  | N  | None | None   | N  |  17  |
=============================================================================

Sequential Cell (Q_reg)
 Cell Type: Flip-Flop
 Multibit Attribute: N
 Clock: CLK
 Async Clear: RESET
 Async Set: SET
 Async Load: 0
 Sync Clear: J' K
 Sync Set: J K'
 Sync Toggle: 0
 Sync Load: K
```

Inferring Master-Slave Latches

This section contains the following subsections:

- Master-Slave Latch Overview
- Master-Slave Latch: Single Master-Slave Clock Pair
- Master-Slave Latch: Multiple Master-Slave Clock Pairs
- Master-Slave Latch: Discrete Components

Master-Slave Latch Overview

HDL Compiler infers master-slave latches by using the `clocked_on_also` attribute.

In your VHDL description, describe the master-slave latch as a flip-flop, using only the slave clock. Specify the master clock as an input port, but do not connect it. In addition, set the `clocked_on_also` attribute on the master clock port (called MCK in these examples).

This coding style requires that cells in the target logic library contain slave clocks defined with the `clocked_on_also` attribute. The `clocked_on_also` attribute defines the slave clocks in the cell's state declaration. For more information about defining slave clocks in the target logic library, see the *Library Compiler User Guide*.

If HDL Compiler does not find any master-slave latches in the target logic library, the tool leaves the master-slave generic cell (MSGEN) unmapped. HDL Compiler does not use D flip-flops to implement the equivalent functionality of the cell.

Note:

Although the vendor's component behaves as a master-slave latch, Library Compiler supports only the description of a master-slave flip-flop.

Master-Slave Latch: Single Master-Slave Clock Pair

Example 141 provides the VHDL template for a master-slave latch.

See dc_tcl_script_begin and dc_tcl_script_end for more information on the `dc_tcl_script_begin` and `dc_tcl_script_end` compiler directives. HDL Compiler generates the verbose inference report shown in Example 142.

Example 141 Master-Slave Latch

```
library IEEE;
use IEEE.std_Logic_1164.all;

entity mslatch is
  port(MCK, SCK, DATA : in std_logic;
       Q : out std_logic );
end mslatch;

architecture rtl of mslatch is
begin

-- 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

process(SCK, DATA) begin
  if (SCK'event and SCK= '1') then
    Q <= DATA;
  end if;
end process;

end rtl;
```

Example 142 Inference Report for a Master-Slave Latch

```
============================================================================
| Register Name  |   Type    | Width | Bus | MB | Set  | Reset | ST | Line |
============================================================================
|  Q_reg         | Flip-flop |   1   |  N  | N  | None | None  | N  |  17  |
============================================================================
```

Master-Slave Latch: 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. Example 143 illustrates the use of `clocked_on_also` with the `associated_clock` option. Example 144 shows the verbose inference reports.

Example 143 Inferring Master-Slave Latches With Two Pairs of Clocks

```
library IEEE;
use IEEE.std_Logic_1164.all;

entity mslatch2 is
  port(SCK1, MCK1, DATA1, SCK2, MCK2, DATA2 : in std_logic;
  Q1, Q2 : out std_logic );
end mslatch2;

architecture rtl of mslatch2 is
begin

-- 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

process (SCK1, DATA1) begin
  if (SCK1'event and SCK1 = '1') then
    Q1 <= DATA1;
  end if;
end process;

process (SCK2, DATA2) begin
  if (SCK2'event and SCK2 = '1') then
    Q2 <= DATA2;
  end if;
end process;

end rtl;
```

Example 144 Inference Reports for Master-Slave Latch: Multiple Clock Pairs

```
Inferred memory devices in process
            in routine mslatch2 line 21 in file
                        '../rtl/ch4.ex4.47.master.slave.latch.2clks.vhd'
============================================================================
| Register Name |    Type    | Width | Bus | MB | Set  | Reset | ST | Line |
============================================================================
| Q_reg         | Flip-flop |   1   |  N  | N  | None | None  | N  |  17  |
============================================================================
Inferred memory devices in process
            in routine mslatch2 line 27 in file
                        '../rtl/ch4.ex4.47.master.slave.latch.2clks.vhd'
=============================================================================
| Register Name  |    Type    | Width | Bus | MB | Set  | Reset | ST | Line |
=============================================================================
|  Q_reg         | Flip-flop |   1   |  N  | N  | None | None  | N  |  17  |
=============================================================================
```

Master-Slave Latch: Discrete Components

If your target logic library does not contain master-slave latch components, you can infer two-phase systems using two D latches. Example 145 shows a simple two-phase system with clocks MCK and SCK. Example 146 shows the verbose inference reports.

Example 145 Two-Phase Clocks

```
library IEEE;
use IEEE.std_Logic_1164.all;

entity LATCH_VHDL is
  port(MCK, SCK, DATA: in std_logic;
       Q : out std_logic );
end LATCH_VHDL;

architecture rtl of LATCH_VHDL is
  signal TEMP : std_logic;
begin
process (MCK, DATA) begin
  if (MCK = '1') then
    TEMP <= DATA;
  end if;
end process;

process (SCK, TEMP) begin
  if (SCK = '1') then
    Q <= TEMP;
  end if;
end process;

end rtl;
```

Example 146 Inference Reports for Two-Phase Clocks

```
Inferred memory devices in process
        in routine LATCH_VHDL line 10 in file
                '/remote7vhdl_example/latch_vhdl.vhd'.

=========================================================================
| Register Name |  Type  | Width | Bus | MB | Set  | Reset | ST | Line |
=========================================================================
| TEMP_reg      |  Latch |   1   |  N  | N  | None | None  | N  |  17  |
=========================================================================

Inferred memory devices in process
        in routine LATCH_VHDL line 15 in file
                '/remote7vhdl_example/latch_vhdl.vhd'.
========================================================================
| Register Name |  Type | Width | Bus | MB | Set  | Reset | ST | Line |
========================================================================
| TEMP_reg      | Latch  |   1   |  N  | N  | None | None  | N  |  17 |
========================================================================

Sequential Cell (TEMP_reg)
```

```
        Cell Type: Latch
        Multibit Attribute: N
        Clock: 0
        Async Clear: 0
        Async Set: 0
        Async Load: MCK

Sequential Cell (Q_reg)
        Cell Type: Latch
        Multibit Attribute: N
        Clock: 0
        Async Clear: 0
        Async Set: 0
        Async Load: SCK
```

Limitations of Register Inference

For best results when inferring registers, restrict each process to inferring a single type of memory cell, use the templates provided in this chapter, and understand the following inference limitations.

HDL Compiler cannot infer the following components. You must instantiate them in your VHDL description.

- Flip-flops and latches with three-state outputs
- Flip-flops with bidirectional pins
- Flip-flops with multiple clock inputs
- Multiport latches
- Register banks

  Note:

  Although you can instantiate flip-flops with bidirectional pins, HDL Compiler interprets these cells as black boxes.

If you use an if statement to infer D flip-flops, your design must meet the following requirements:

- The edge expression, such as CLK’event rising_edge (CLK), must be the only condition of an if or an elsif clause.

  The following if statement is invalid, because it has multiple conditions in the if clause:

  ```
  if (edge and RST = '1')
  ```

- You can have only one edge expression in an if clause, and the if clause must not have an else clause.

The following if statement is invalid, because you cannot include an else clause when using an edge expression as the if or elsif condition:

```
if X > 5 then
  sequential_statement;
elsif edge then
  sequential_statement;
else
  sequential_statement;
end if;
```

- An edge expression cannot be part of another logical expression or be used as an argument.

  The following function call is invalid, because you cannot use the edge expression as an argument:

  ```
  any_function(edge);
  ```

- If you are using only wait statements for sequential inferencing, only one wait statement is allowed in a process. Coding styles using multiple wait statements, such as FSMs using multiple wait statements, are not supported. The tool generates the following error message if you use multiple wait statements in a process:

  ```
  HDLC does not support processes with multiple event
  statements. (ELAB-336)
  ```

Unloaded Sequential Cell Preservation

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`.

Caution:

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

Example 147 has `hdlin_preserve_sequential` set to `all` 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 the design as shown in Example 148.

Example 147 Preserves an Unloaded Cell (sum2) and Two Adders

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


entity seq_cell_ex2 is
  port(
       in1, in2, in3 : in   std_logic_vector(1 downto 0);
       out_z          : out  std_logic_vector(1 downto 0);
       clk           : in   std_logic);
end seq_cell_ex2;

architecture rtl of seq_cell_ex2 is
  signal sum1, sum2  : std_logic_vector(1 downto 0);
  signal save : std_logic_vector(1 downto 0);
begin
process (clk) begin
  if (clk'event and clk = '1') then
    sum1 <= in1 + in2;
    sum2 <= in1 + in2 + in3;  -- sum2 reg is saved

  end if;
end process;


out_z <= not sum1;

end rtl;
```

Example 148 preserves the sum2 register and all combinational logic before it.

Example 148 Preserves an Unloaded Cell (save) and Three Adders

```
set hdlin_preserve_sequential = all
.
.
.
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
```

```
entity seq_cell_ex3 is
  port(
       in1, in2, in3 : in   std_logic_vector(1 downto 0);
       out_z         : out  std_logic_vector(1 downto 0);
       clk           : in   std_logic);
end seq_cell_ex3;

architecture rtl of seq_cell_ex3 is

signal sum1, sum2 : std_logic_vector(1 downto 0);
signal save : std_logic_vector(1 downto 0);
begin
process (clk) begin
  if (clk'event and clk = '1') then
    sum1 <= in1 + in2;
    sum2 <= in1 + in2 + in3;  -- this combinational logic
                              -- is saved
  end if;
end process;

out_z <= not sum1;

process (clk) begin
  if (clk'event and clk = '1') then
    save  <= sum1 + sum2; -- this combinational logic is saved
  end if;
end process;
end rtl;
```

Note:

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

Note:

The tool does not distinguish between unloaded cells (those not connected to any output ports) and feedthroughs. See Example 149 for a feedthrough.

Example 149 Feedthrough Example

```
entity reg1 is
  port (
    d0, clk : in bit;
    q0: out bit);
end entity reg1;
```

```
architecture behave of reg1 is
begin  -- behave
  storage: process (clk)
    variable temp1, temp2 : bit;
      begin
        if clk'event and clk = '1' then
          temp1 := d0;
          temp2 := temp1;
        end if;
        q0 <= temp2;
      end process storage;
end behave;
```

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 150 Feedthrough Register temp1

```
============================================================================
| Register Name  |    Type    | Width | Bus | MB | Set   | Reset | ST | Line |
============================================================================
|   temp1_reg     | Flip-flop |   1    |  N  | N  | None | None  | N  |  17  |
|   temp2_reg     | Flip-flop |   1    |  N  | N  | None | None  | N  |  17  |
============================================================================
```

5

Modeling Three-State Buffers

HDL Compiler infers a three-state buffer when you assign the value of Z to a signal or variable. The Z value represents the high-impedance state. HDL Compiler infers one three-state buffer per process. You can assign high-impedance values to single-bit or bused signals (or variables). HDL Compiler does not provide any variables, attributes, or directives to control the inference.

This chapter includes the following sections:

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 151 shows a three-state inference report:

Example 151 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.

Inferring a Basic Three-State Driver

Example 152 provides the VHDL template for a basic three-state buffer. HDL Compiler generates the inference report shown in Example 153. Figure 15 shows the compiled output.

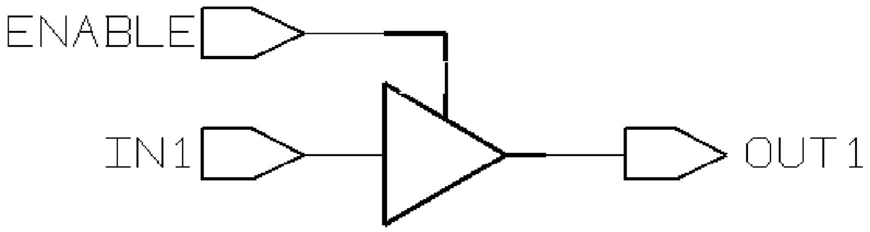
Example 152 Basic Three-State Buffer

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
entity three_state_basic is
port(IN1, ENABLE : in std_logic;
     OUT1 : out std_logic );
end;

architecture rtl of three_state_basic is
begin

process (IN1, ENABLE) begin
  if (ENABLE = '1') then
    OUT1 <= IN1;
  else
    OUT1 <= 'Z';   -- assigns high-impedance state
  end if;
end process;
end rtl;
```

Example 153 Inference Report for a Basic Three-State Buffer

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

Figure 15 A Basic Three-State Buffer



Inferring One Three-State Buffer From a Single Process

Example 154 provides an example of placing all high-impedance assignments in a single process. In this case, the data is gated and HDL Compiler infers a single three-state buffer. Example 155 shows the inference report.

Example 154 Inferring One Three-State Buffer From a Single Process

```
library IEEE;
use IEEE.std_logic_1164.all;

entity three_state is
  port ( A, B, SELA, SELB : in std_logic ;
         T : out std_logic );
end three_state;

architecture rtl of three_state is
begin
infer : process (SELA, A, SELB, B) begin
  T <= 'Z';
  if (SELA = '1') then
    T <= A;
  elsif (SELB = '1') then
    T <= B;
  end if;
end process infer;

end rtl;
```

Example 155 Single Process Inference Report

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

Inferring Two Three-State Buffers

Example 156 provides an example of placing each high-impedance assignment in a separate process. In this case, HDL Compiler infers multiple three-state buffers. Example 157 shows the inference report. Figure 16 shows the design.

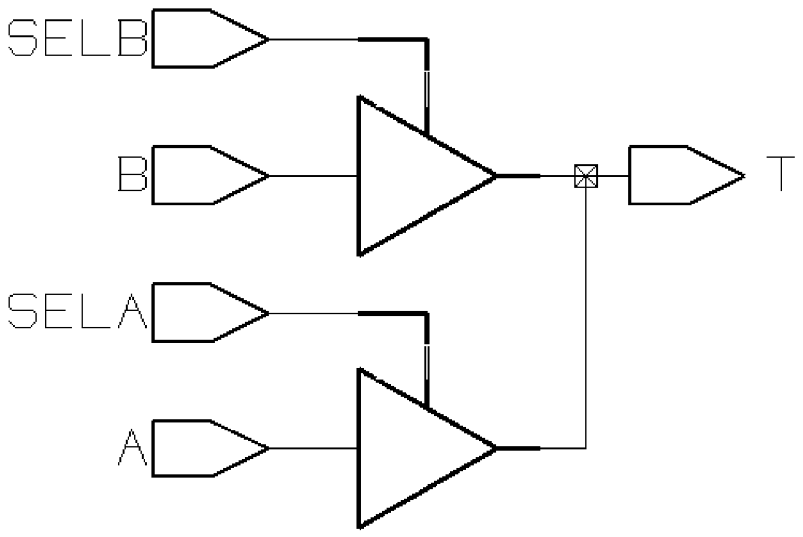
Example 156 Inferring Two Three-State Buffers

```
library IEEE;
use IEEE.std_logic_1164.all;

entity three_state is
  port ( A, B, SELA, SELB : in std_logic ;
         T : out std_logic );
end three_state;

architecture rtl of three_state is
begin
infer1 : process (SELA, A) begin
  if (SELA = '1') then
    T <= A;
  else
    T <= 'Z';
  end if;
end process infer1;

infer2 : process (SELB, B) begin
  if (SELB = '1') then
    T <= B;
  else
    T <= 'Z';
  end if;
end process infer2;

end rtl;
```

Example 157 Inference Report for Two Three-State Buffers

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

============================================
| Register Name |       Type       | Width |
============================================
|   T_tri2      | Tri-State Buffer |   1   |
============================================
```

Figure 16 Two Three-State Buffers



Three-State Buffer With Registered Enable

When a variable, such as THREE_STATE in Example 158, is assigned to a register and defined as a three-state buffer within the same process, HDL Compiler also registers the enable pin of the three-state gate. Example 158 shows an example of this type of code, and Example 159 shows the inference report. Figure 17 shows the schematic generated by the code, a three-state buffer with a register on its enable pin.

Example 158 Inferring a Three-State Buffer With Registered Enable

```
library IEEE;
use IEEE.std_logic_1164.all;

entity three_state is
   port ( DATA, CLK, THREE_STATE : in std_logic ;
          OUT1 : out std_logic );
end three_state;

architecture rtl of three_state is
begin
infer : process (THREE_STATE, CLK) begin
   if (THREE_STATE = '0') then
      OUT1 <= 'Z';
   elsif (CLK'event and CLK = '1') then
      OUT1 <= DATA;
   end if;
end process infer;
```

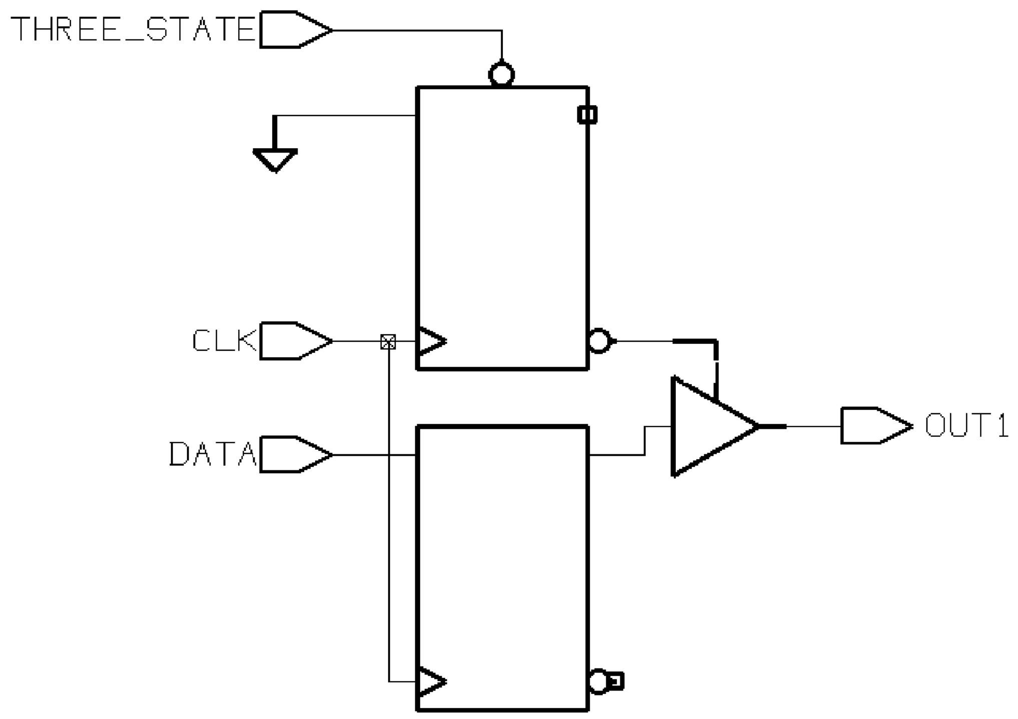
Feedback

```
    end rtl;
```

Example 159 Inference Report for a Three-State Buffer With Registered Enable

```
===========================================================================
|Register Name       |   Type    | Width | Bus | MB | Set   | Reset | ST | Line
 |
============================================================================
===
|OUT1_reg            | Flip-flop |   1   |  N  | N  | None | None   | N  |  17
  |
|OUT1_tri_enable_reg | Flip-flop |   1   |  N  | N  | None | None   | N  |  17
  |
=============================================================================
===

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

Figure 17 Three-State Buffer With Registered Enable



Feedback

Three-State Buffer With Registered Data

Example 160 uses two processes to instantiate a three-state buffer, with a flip-flop on the input pin. Example 161 shows the inference report. Figure 18 shows the schematic generated by the code.

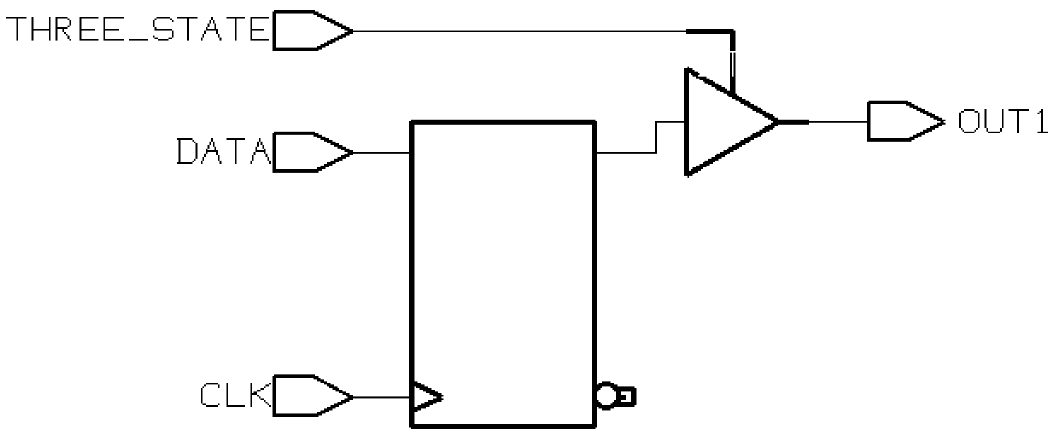
Example 160 Three-State Buffer With Registered Data

```
library IEEE;
use IEEE.std_logic_1164.all;

entity ff_3state2 is
   port ( DATA, CLK, THREE_STATE : in std_logic ;
          OUT1 : out std_logic );
end ff_3state2;

architecture rtl of ff_3state2 is
   signal TEMP : std_logic;
begin

process (CLK) begin
   if (CLK'event and CLK = '1') then
      TEMP <= DATA;
   end if;
end process;

process (THREE_STATE, TEMP) begin
   if (THREE_STATE = '0') then
      OUT1 <= 'Z';
   else
      OUT1 <= TEMP;
   end if;
end process;

end rtl;
```

Example 161 Inference Report for a Three-State Buffer With Registered Data

```
============================================================================
| Register Name  |    Type    | Width | Bus | MB | Set   | Reset | ST | Line |
============================================================================
|  TEMP_reg      | Flip-flop |   1   |  N  | N  | None | None   | N  |  17  |
============================================================================

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

Figure 18 Three-State Buffer With Registered Data



Understanding the Limitations of Three-State Inference

You can use the Z value as

- A signal assignment
- A variable assignment
- A function call argument
- A return value
- An aggregate definition

You cannot use the `z` value in an expression, except for concatenation and comparison with `z`, such as in

```
if (IN_VAL = 'Z') then y<=0 endif;
```

This is an example of permissible use of the `z` value in an expression, but it always evaluates to false. Therefore, it is also a simulation and synthesis mismatch.

This code

```
OUT_VAL <= ('Z' and IN_VAL);
```

is an example of an incorrect use of the `z` value in an expression. It is incorrect because it is not a comparison expression. This code generates an error because HDL Compiler cannot compute any expressions that use ’Z’ as an input.

Be careful when using expressions that compare with the `z` value. HDL Compiler always evaluates these expressions to false, and the pre-synthesis and post-synthesis simulation results might differ. For this reason, HDL Compiler issues a warning when it synthesizes such comparisons.

6

Directives, Attributes, and Variables

This chapter describes the compiler directives, attributes, and HDL read variables supported by HDL Compiler in the following sections:

Compiler Directives

HDL Compiler synthesis directives are special comments which affect how synthesis processes the RTL. These comments are ignored by other VHDL tools.

These directives begin as a VHDL comment (`--`) followed by a *pragma* prefix (`pragma`, `synopsys`, or `synthesis`) and then the directive. Whitespace is permitted (but not required) before and after the two hyphens.

Note:

Not all directives support all pragma prefixes; see Directive Support by Pragma Prefix for details.

The following topics describe the supported directives:

- keep_signal_name
- template
- translate_off and translate_on
- resolution_method
- rp_group and rp_endgroup
- rp_place
- rp_fill
- rp_array_dir

- map_to_entity and return_port_name
- dc_tcl_script_begin and dc_tcl_script_end
- Directive Support by Pragma Prefix

keep_signal_name

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

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

template

The `template` directive is used to read a design with a generic given that the generic default is specified. For more information, see Parameterized Models (Generics).

translate_off and translate_on

The code contained within these directives is ignored and treated as comments.

resolution_method

Resolution directives determine the resolution function associated with resolved signals. HDL Compiler does not support arbitrary resolution functions. It only supports the following three resolution methods:

```
-- synopsys resolution_method wired_and
-- synopsys resolution_method wired_or
-- synopsys resolution_method three_state
```

For more information, see Resolution Functions.

rp_group and rp_endgroup

Note:

HDL Compiler supports relative placement directives for RTL designs only.

The `rp_group` and `rp_endgroup` directives allow you to specify a relative placement group. All cell instances declared between the `rp_group` and `rp_endgroup` directives are members of the specified group. These directives are available for RTL designs and netlist designs.

The VHDL syntax for RTL and netlist designs is as follows:

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

For more information and an example, see .

rp_place

The `rp_place` directive allows you to specify a subgroup at a specific hierarchy, a keepout region, or an instance to be placed in the current relative placement group. When you use the `rp_place` directive to specify a subgroup at a specific hierarchy, you must instantiate the subgroup's instances outside of any group declarations in the module. This directive is available for RTL designs and netlist designs.

The VHDL syntax for RTL and netlist designs is as follows:

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

For more information and an example, see .

Note:

HDL Compiler supports relative placement directives for RTL designs only.

rp_fill

The `rp_fill` directive automatically places the cells at the location specified by a pointer. Each time a new instance is declared that is not explicitly placed, it is inserted into the grid at the location indicated by the current value of the pointer. After the instance is placed, the pointer is updated incrementally and the process is ready to be repeated. This directive is available for RTL designs and netlist designs.

Note:

HDL Compiler supports relative placement directives for RTL designs only.

The `rp_fill` arguments define how the pointer is updated. The *`col`* and *`row`* parameters specify the initial coordinates of the pointer. These parameters can represent absolute row or column locations in the group's grid or locations that are relative to the current pointer value. To represent locations relative to the current pointer, enclose the column and row values in angle brackets (<>). For example, assume the current pointer location is (3,4). In this case, specifying `rp_fill <1> 0` initializes the pointer to (4,0) and that is where the next instance is placed. Absolute coordinates must be nonnegative integers; relative coordinates can be any integer.

The VHDL syntax for RTL designs is as follows:

```
-- synopsys rp_fill ( {col row} {pattern pat} )
```

For more information, see .

rp_array_dir

The `rp_array_dir` directive specifies whether the elements of an array are placed upward, from the least significant bit to the most significant bit, or downward, from the most significant bit to the least significant bit.

The VHDL syntax for RTL designs is as follows:

```
-- synopsys rp_array_dir ( up|down )
```

For more information and an example, see .

Note:

HDL Compiler supports relative placement directives for RTL designs only.

map_to_entity and return_port_name

Component implication directives map VHDL subprograms onto existing components or VHDL entities.

Synopsys supports the following component implication directives:

```
  -- synopsys map_to_entity entity_name
  -- synopsys return_port_name port_name
```

For more information, see Procedures and Functions as Design Components. Other directives, such as `map_to_operator`, are used to drive inference of HDL operators such as *, +, and –. See the *DesignWare Developer Guide* for more information about synthetic comments.

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 162.

Example 162 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
...
```

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.

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 13 shows how each directive is handled by each pragma prefix.

Table 13 Directive Support by Pragma Prefix

Directive	-- synopsys	-- pragma	-- synthesis
`translate_off` `translate_on`	Used	Used	Used

`dc_tcl_script_begin`
`dc_tcl_script_end`
`dc_script_begin`
`dc_script_end`
	Used	Used	Used

`keep_signal_name`	Used	Ignored	Ignored
`template`	Used	Ignored	Ignored
`resolution_method`	Used	Ignored	Ignored

`rp_group`
`rp_endgroup`
`rp_place`
`rp_fill`
`rp_array_dir`
	Used	Ignored	Ignored

`map_to_entity` `return_port_name`	Used	Ignored	Ignored
Any unknown directive	Error	Ignored	Ignored

Attributes

This section describes the following:

- Synopsys Defined Attributes
- IEEE Predefined Attributes

Synopsys Defined Attributes

The Synopsys defined attributes are listed in Table 14. When you use these attributes, insert the following line in your VHDL description, just before the entity declaration.

```
use SYNOPSYS.ATTRIBUTES.all;
```

These attributes are included in the ATTRIBUTES package.

Table 14 Attributes Supported by HDL Compiler

Attribute	Description
`arrival`	See Table 15.
`async_set_reset`	See the table in Register Inference Attributes.
`async_set_reset_local`	See the table in Register Inference Attributes.
`async_set_reset_local_all`	See the table in Register Inference Attributes.
`dont_touch`	See Table 15.
`dont_touch_network`	See Table 15.
`drive_strength`	See Table 15.
`enum_encoding`	See ENUM_ENCODING Attribute.
`equal`	See Table 15.
`fall_arrival`	See Table 15.
`fall_drive`	See Table 15.
`infer_multibit`	See Multibit Inference.
`infer_mux`	See MUX_OP Inference.
`load`	See Table 15.
`logic_one`	See Table 15.

Table 14 Attributes Supported by HDL Compiler (Continued)

Attribute	Description
`logic_zero`	See Table 15.
`max_area`	See Table 15.
`max_delay`	See Table 15.
`max_fall_delay`	See Table 15.
`max_rise_delay`	See Table 15.
`max_transition`	See Table 15.
`min_delay`	See Table 15.
`min_fall_delay`	See Table 15.
`min_rise_delay`	See Table 15.
`one_cold`	See the table in Register Inference Attributes.
`one_hot`	See the table in Register Inference Attributes.
`opposite`	See Table 15.
`rise_arrival`	See Table 15.
`rise_drive`	See Table 15.
`sync_set_reset`	See the table in Register Inference Attributes.
`sync_set_reset_local`	See the table in Register Inference Attributes.
`sync_set_reset_local_all`	See the table in Register Inference Attributes.
`unconnected`	See Table 15.

The design attributes are described in Table 15.

Table 15 Design Attributes

Attribute	Type	Description
Input port Attributes		
`MAX_AREA`	real	-- Maximum desired area, in logic library -- area units. `attribute MAX_AREA of EXAMPLE : entity is 500.0;`

Table 15 Design Attributes (Continued)

Attribute	Type	Description
`MAX_TRANSITION`	real	-- Maximum allowable transition time for any -- network in the design, in logic library -- time units. `attribute MAX_TRANSITION of EXAMPLE : entity is 3.0;`
`ARRIVAL`	real	-- Expected signal arrival time, in technology -- library time units. Sets both RISE_ARRIVAL and -- FALL_ARRIVAL.attribute ARRIVAL of A : signal is 1.5;
`DRIVE_STRENGTH`	real	-- Input signal's drive strength, in technology -- library load units. Sets both RISE_DRIVE and -- FALL_DRIVE.attribute DRIVE_STRENGTH of A, B : signal is 0.01;
`RISE_ARRIVAL`	real	-- Input signal's rise time. attribute RISE_ARRIVAL of C : signal is 1.5;
`FALL_ARRIVAL`	real	-- Input signal's fall time. attribute FALL_ARRIVAL of A, B : signal is 1.5;
`FALL_DRIVE`	real	-- Input signal's drive strength while falling.attribute FALL_DRIVE of B : signal is 0.01;
`RISE_DRIVE`	real	-- Input signal's drive strength while rising.attribute RISE_DRIVE of A : signal is 0.01;
`EQUAL`	Boolean	-- Applied to pairs of input ports; true -- if both ports are logically equal. attribute EQUAL of A, B : signal is TRUE;The attributes EQUAL and OPPOSITE are used only for pairs of single-bit ports (signals).
`OPPOSITE`	Boolean	-- Applied to pairs of input ports; true -- if the two ports are logically opposite.attribute OPPOSITE of A, B: signal is TRUE;
`LOGIC_ONE`	Boolean	-- True if the input port is always at logic 1.attribute LOGIC_ONE of A : signal is TRUE;
`LOGIC_ZERO`	Boolean	-- True if the input port is always at logic 0attribute LOGIC_ZERO of A, B: signal is TRUE;
`DONT_TOUCH_NETWORK`	Boolean	-- True if the network connected to the input -- port is to be excluded from optimization.attribute DONT_TOUCH_NETWORK of A : signal is TRUE;
Output Port Attributes		
`LOAD`	real	-- Loading on output port, in library load units.attribute LOAD of Y, Z : signal is 5.0;

Table 15 Design Attributes (Continued)

Attribute	Type	Description
UNCONNECTED	Boolean	-- Can be set to true if the output port is not -- connected to external circuitry.attribute UNCONNECTED of X : signal is TRUE;
MIN_RISE_DELAY	real	-- Minimum allowable delay time before -- the output port's signal rises.attribute MIN_RISE_DELAY of X : signal is 5.0;
MIN_FALL_DELAY	real	-- Minimum allowable delay time before -- the output port's signal falls.attribute MIN_FALL_DELAY of Y : signal is 5.0;
MAX_DELAY	real	-- Maximum allowable delay time, from any -- input signal connected to the output -- port, in logic library time units. -- Sets both MAX_RISE_DELAY and MAX_FALL_DELAY.attribute MAX_DELAY of X : signal is 20.0;
MAX_RISE_DELAY	real	-- Maximum allowable delay time before -- the output port's signal rises.attribute MAX_RISE_DELAY of Z : signal is 20.0;
MAX_FALL_DELAY	real	-- Maximum allowable delay time before -- the output port's signal falls.attribute MAX_FALL_DELAY of X, Y : signal is 20.0;
MIN_DELAY	real	-- Minimum allowable delay time, from any -- input signal connected to the output -- port, in logic library time units. -- Sets both MIN_RISE_DELAY and MIN_FALL_DELAYattribute MIN_DELAY of X, Z : signal is 5.0;
Cell attributes		
DONT_TOUCH	Boolean	-- True if the instance is not to be optimized. attribute DONT_TOUCH of INSTANCE : label is TRUE; A dont_touch attribute cannot be set to false.

ENUM_ENCODING Attribute

You can override the automatic enumeration encodings and specify your own enumeration encodings with the `ENUM_ENCODING` attribute. This interpretation is specific to HDL Compiler. This attribute allows HDL Compiler to interpret your logic correctly. Place the synthesis attribute `ENUM_ENCODING` on your primary logic type.

The `ENUM_ENCODING` attribute must be a string containing a series of vectors, one for each enumeration literal in the associated type. The encoding vector is specified by ’0’s, ’1’s, ’D’s, ’U’s, and ’Z’s, separated by blank spaces.

The possible encoding values for the `ENUM_ENCODING` attribute are ’0’, ’1’, ’D’, ’U’, and ’Z’ and are described in Table 16.

Table 16 Encoding Values for the ENUM_ENCODING Attribute

Encoding value	Description
’0’	Bit value ’0’.
’1’	Bit value ’1’.
’D’	Don’t care (can be either ’0’ or ’1’).
’U’	Unknown. If ’U’ appears in the encoding vector for an enumeration, you cannot use that enumeration literal except as an operand to the = and /= operators. You can read an enumeration literal encoded with a ’U’ from a variable or signal, but you cannot assign it. For synthesis, the = operator returns false and /= returns true when either of the operands is an enumeration literal whose encoding contains ’U’.
’Z’	High impedance.

The first vector in the attribute string specifies the encoding for the first enumeration literal, the second vector specifies the encoding for the second enumeration literal, and so on. The `ENUM_ENCODING` attribute must immediately follow the type declaration.

Example 163 illustrates how the default encodings from the example in Enumeration Encoding can be changed with the `ENUM_ENCODING` attribute.

Example 163 Using the ENUM_ENCODING Attribute

```
attribute ENUM_ENCODING: STRING;
  -- Attribute definition

type COLOR is (RED, GREEN, YELLOW, BLUE, VIOLET);
attribute ENUM_ENCODING of
  COLOR: type is "010 000 011 100 001";
  -- Attribute declaration
```

The enumeration values are encoded as follows:

```
RED    = "010"
GREEN  = "000"
YELLOW = "011"
BLUE   = "100"
VIOLET = "001"
```

The result is GREEN < VIOLET < RED < YELLOW < BLUE.

Note:

The interpretation of the `ENUM_ENCODING` attribute is specific to HDL Compiler. Other VHDL tools, such as simulators, use the standard encoding (ordering).

IEEE Predefined Attributes

See Names for the IEEE predefined attributes supported by HDL Compiler.

Variables

HDL Compiler read variables are described in Table 17.

Table 17 Variables

Name	Default	Description
hdlin_elab_errors_deep	false	Allows the elaboration of submodules even if the top-level module elaboration fails, enabling HDL Compiler to report more elaboration, link, and VER-37 errors and warnings in a hierarchical design during the first elaboration run. See Reporting Elaboration Errors.
hdlin_enable_configurations	False	Enables configuration support.
hdlin_generate_separator_style	_	Specifies the separator string for instances generated in multiple-nested loops. This is a VHDL only supported variable. Verilog generate naming follows the Verilog LRM standard, so this variable is not required.
hdlin_infer_function_local_latches	False	Allows latches to be inferred for function- and procedure-scope variables.
hdlin_keep_signal_name	all_driving	Attempts to keep a signal name if there is path from the signal to an output port. This includes preserving cells between the signal and the output port.
hdlin_mux_oversize_ratio	100	Defined as the ratio of the number of MUX inputs to the unique number of data inputs. When this ratio is exceeded, a MUX will not be inferred and the circuit will be generated with SELECT_OPs.
hdlin_mux_size_min	2	Sets the minimum number of data inputs for MUX inference.

Table 17 Variables (Continued)

Name	Default	Description
hdlin_mux_size_only	1	Controls which MUX_OP cells receive the `size_only` attribute. By default, MUX_OP cells that are generated with the RTL `infer_mux` compiler directive and that are on set/reset signals receive the `size_only` attribute. For more information and a complete list of options, see the table in Variables That Control MUX_OP Inference.
hdlin_preserve_sequential	none	Preserves unloaded sequential cells (latches or flip-flops) that would otherwise be removed during optimization by HDL Compiler. The following options are supported: • `none` or `false`–No unloaded sequential cells are preserved. This is the default behavior. • `all` or `true`–All unloaded sequential cells are preserved, excluding unloaded sequential cells that are used solely as loop variables. • `all+loop_variables` or `true+loop_variables`–All unloaded sequential cells are preserved, including unloaded sequential cells that are used solely as loop variables. • `ff`–Only flip-flop cells are preserved, excluding unloaded sequential cells that are used solely as loop variables. • `ff+loop_variables`–Only flip-flop cells are preserved, including unloaded sequential cells that are used solely as loop variables. • `latch`–Only unloaded latch cells are preserved, excluding unloaded sequential cells that are used solely as loop variables. • `latch+loop_variables`–Only unloaded latch cells are preserved, including unloaded sequential cells that are used solely as loop variables. *Important*: To preserve unloaded cells through compile, you must set `compile_delete_unloaded_sequential_cells` to `false`. See Unloaded Sequential Cell Preservation.
hdlin_prohibit_nontri_multiple_drivers	`true`	Issues an error when a non-tri net is driven by more than one process or continuous assignment.
hdlin_vhdl_std	2008	Specifies the VHDL standard to enforce: 1987, 1993, or 2008.

Table 17 Variables (Continued)

Name	Default	Description
hdlin_vhdl_syntax_extensions	`false`	Enables the following features: • Deferred constant definition • Arrays of base type Boolean • Impure functions When you use these constructs, make sure you perform a thorough verification.

7
Writing Out Designs in VHDL Format

While using HDL Compiler, you can write out any design in a variety of formats, including VHDL. Existing gate-level netlists, sets of logic equations, or technology-specific circuits can be automatically converted to a VHDL description. The resulting VHDL description can serve as documentation of the original design, and you can use it as a starting point for reimplementation in a new technology. In addition, you can give the VHDL description to a VHDL simulator to provide circuit timing information.

The following sections discuss how to write out designs in VHDL format.

- Netlist Writer Variables
- Writing Out VHDL Files
- VHDL Write Variables
- Bit and Bit-Vector Variables
- Resolution Function Variables
- Types and Type Conversion Variables
- Architecture and Configuration Variables
- Preserving Port Types
- VHDL Netlister Coding Considerations

Netlist Writer Variables

The netlist writer variables discussed in this chapter are listed in Table 18.

Table 18 Variable Summary

Group	Attributes/Directives/Variables

Write variables	`vhdlout_dont_create_dummy_netsvhdlout_equationsvhdlout_follow_vector_directionvhdlout_separate_scan_invhdlout_local_attributesvhdlout_upcase`
	`vhdlout_use_packages`
	`vhdlout_write_architecture`
	`vhdlout_write_components`
	`vhdlout_write_entity`
	`vhdlout_write_top_configuration`

Bit and bit-vector variables	`vhdlout_three_state_namevhdlout_unknown_name`
	`vhdlout_zero_name`
	`vhdlout_bit_type`
	`vhdlout_bit_vector_type`
	`vhdlout_one_name`

Resolution function variables	`vhdlout_three_state_res_funcvhdlout_wired_and_res_funcvhdlout_wired_or_res_func`
Types and type conversion variables	`vhdlout_package_naming_stylevhdlout_preserve_hierarchical_typesvhdlout_single_bit`
Architecture and configuration variables	`vhdlout_top_configuration_arch_namevhdlout_top_configuration_entity_namevhdlout_top_configuration_name`

Writing Out VHDL Files

To write out VHDL design files, use the `write` command.

```
dc_shell> write -format vhdl -output my_file.vhdl
```

The `write -format vhdl` command is valid whether or not the current design originated as a VHDL source file. You can write out any design, regardless of initial format (equation, netlist, and so on), as a VHDL design.

For more information about the `write` command, see *Using Tcl With Synopsys Tools.*

VHDL Write Variables

Several application variables affect how designs are written out as VHDL files. These variables must be set before you write out the design. They can be set interactively or in your .synopsys_dc.setup file.

The following variables affect writing out VHDL (vhdlout_variables). To list them, enter

```
dc_shell> man vhdlio_variables
```

`vhdlout_dont_create_dummy_nets`

Controls whether the VHDL writer creates dummy nets for connecting unused pins or ports.

`vhdlout_equations`

When set to true, this variable determines that combinational logic is written with technology-independent Boolean equations, sequential logic is written with technology-independent wait and if statements, and three-state drivers are written with technology-independent code.

`vhdlout_follow_vector_direction`

Controls how the VHDL writer determines the array range direction.

`vhdlout_local_attributes`

This variable is obsolete. Use the `write_script` command instead (see the `write_script` man page).

`vhdlout_separate_scan_in`

Controls how the scan chain is written out in VHDL.

By default, this variable is false and the VHDL writer writes the scan chain in the same file as the design. In this case, the scan chain is not visible in the testbench and parallel-load simulation is not possible.

Set this variable to true to force the VHDL writer to write the scan chain as a separate package to enable parallel-load simulation.

`vhdlout_upcase`

Determines, when set to true, that identifiers are written out in uppercase to the VHDL file.

`vhdlout_use_packages`

This variable is a list of package names. A use clause is written into the VHDL file for each of these packages for all entities; library clauses are also written out as needed.

`vhdlout_write_architecture`

When this variable is set to true (the default), an architecture definition is written out to the VHDL file.

`vhdlout_write_components`

This variable controls whether component declarations for cells mapped to a logic library are written out (if set to true) or not (false).

Component declarations are required by VHDL. If you set this variable to false, make sure a package containing the necessary component declarations is listed in `vhdlout_use_packages`.

The default is true. See also the `vhdlout_use_packages` variable.

`vhdlout_write_entity`

When this variable is set to true (the default), an entity definition is written out to the VHDL file and to any conversion packages as necessary.

`vhdlout_write_top_configuration`

When this variable is set to true, a top-level configuration definition is written out to the VHDL file. The default is false.

Bit and Bit-Vector Variables

Bit and bit-vector variables, whose descriptions follow, define the names of bits, bit vectors, and the associated types.

`vhdlout_bit_type`

The name of the bit type used for writing out single-bit values, used with the following variables:

When writing a generic three-state model, HDL Compiler displays an error if `vhdlout_bit_type` is set to its default value of a bit. Set `vhdlout_bit_type` to a bit type that includes a high-impedance value (’Z’). For more information about inferred three-state devices, see Modeling Three-State Buffers.

`vhdlout_bit_vector_type`

The name of the bit vector type used for writing multiple-bit values, used with the `vhdlout_bit_type`, `vhdlout_one_name`, and `vhdlout_zero_name` variables.

`vhdlout_one_name`

The name of the enumeration literal that represents a logic 1.

`vhdlout_three_state_name`

The name of the high-impedance bit value used for three-state device values.

`vhdlout_unknown_name`

The value used to drive a signal to the unknown state, usually a character literal or an enumeration name.

`vhdlout_zero_name`

The name of the enumeration literal that represents a logic 0.

Resolution Function Variables

The resolution function variables, whose descriptions follow, name resolution functions that are written out.

`vhdlout_three_state_res_func`

Names a three-state resolution function to use instead of the default function. You must supply this function in a package listed in `vhdlout_use_packages`.

`vhdlout_wired_and_res_func`

Names a wired AND resolution function to use instead of the default function. You must supply this function in a package listed in `vhdlout_use_packages`.

`vhdlout_wired_or_res_func`

Names a wired OR resolution function to use instead of the default. You must supply this function in a package listed in `vhdlout_use_packages`.

Types and Type Conversion Variables

The following types and type conversion variables define type conversion functions and how the VHDL writer writes out types.

`vhdlout_package_naming_style`

This variable controls how packages of conversion functions are named. The default is "CONV_PACK_%d", where %d is a number that is incremented as necessary to produce a unique name. By default, the package name and the number are separated by underscores (_).

`vhdlout_preserve_hierarchical_types`

This variable affects how ports on lower-level designs are written out. Top-level design ports are controlled by `vhdlout_single_bit`. (A design is considered lower-level if it is instantiated by any of the designs being written out.)

When this variable is set to USER, all ports on lower-level designs are written with their original data types. This option affects only designs that are read in VHDL format.

When set to VECTOR, all ports on lower-level designs are written with their ports bused; ports keep their names. These bused ports contrast to ports that are bit-blasted. Bit-blasting is the term for breaking down a bus to its individual bus members. The port types are defined by `vhdlout_bit_vector_type` or by `vhdlout_bit_type`, in the case of single-bit ports. This setting is likely to give the most efficient description for simulation. The default is VECTOR. You must ensure that `vhdlout_bit_vector_type` is an array type whose elements are of `vhdlout_bit_type`.

When this variable is set to BIT, typed ports are bit-blasted. If the type of a port is N bits wide, it is written to the VHDL file as N separate ports. Each port is given the type defined by `vhdlout_bit_type`. This variable has no effect if you set `vhdlout_single_bit to BIT`. `vhdlout_preserve_hierarchical_types` is then ignored, and the whole design hierarchy is written out bit-blasted.

This variable cannot take on a higher value than the current setting of `vhdlout_single_bit`. The descending order is {USER, VECTOR, BIT}. Thus, the combination of `vhdlout_single_bit set to VECTOR` and `vhdlout_preserve_hierarchical_types set to USER` is not possible.

`vhdlout_single_bit`

This variable affects how ports on the top-level design are written out. Lower-level design ports are controlled by `vhdlout_preserve_hierarchical_types`. A design is considered lower-level if it is instantiated by any of the designs being written out.

To determine the current value of this variable, use the `list vhdlout_single_bit` command.

Architecture and Configuration Variables

The following architecture and configuration variables control the names of the architectures, configurations, and entities written to the VHDL file.

`vhdlout_top_configuration_arch_name`

Determines the architecture name that is written out in a configuration definition. The default is "A".

`vhdlout_top_configuration_entity_name`

Determines the entity name that is written out in a configuration definition. The default is "E".

`vhdlout_top_configuration_name`

Determines the configuration name that is written out in a configuration definition. The default is "CFG_TB_E".

Preserving Port Types

Example 164 shows how to write out the current design in VHDL format with port types (vector or record types) preserved.

Example 164 Preserving Port Types When Writing VHDL

```
-- The design must originate in VHDL format
dc_shell> read_vhdl my_design.vhdl

-- Set the variable that causes the port types to be
   preserved
dc_shell> set vhdlout_single_bit user

-- Now write the current design in VHDL format
dc_shell> write -format vhdl -output design_out.vhdl
```

Example 165 shows a VHDL input file. Example 166 and Example 167 shows the corresponding output files.

Example 165 Original VHDL Input File

```
library IEEE;
use IEEE.std_logic_1164.all;

entity test_vhdl is
```

```
port ( a: in std_logic_vector (3 downto 0);
       b: out std_logic_vector ( 3 downto 0));
end test_vhdl;

architecture structural of test_vhdl is
begin
     b <= not a;
end structural;
```

The output file in Example 166 use the default values of the vhdlout_variables (described in VHDL Write Variables) to generate the test_vhdl output file.

Example 166 TEST_VHDL Written Out in Default VHDL Format

```
library IEEE;

use IEEE.std_logic_1164.all;

package CONV_PACK_test_vhdl is

-- define attributes
attribute ENUM_ENCODING : STRING;

end CONV_PACK_test_vhdl;

library IEEE;

use IEEE.std_logic_1164.all;

use work.CONV_PACK_test_vhdl.all;

entity test_vhdl is

   port( a : in std_logic_vector (3 downto 0);  b : out std_logic_vector
(3 downto 0));

end test_vhdl;

architecture SYN_structural of test_vhdl is

   component GTECH_NOT
      port( A : in std_logic;  Z : out std_logic);
   end component;

begin

   I_0 : GTECH_NOT port map( A => a(3), Z => b(3));
   I_1 : GTECH_NOT port map( A => a(2), Z => b(2));
   I_2 : GTECH_NOT port map( A => a(1), Z => b(1));
   I_3 : GTECH_NOT port map( A => a(0), Z => b(0));

end SYN_structural;
```

If you set vhdlout_single_bit to bit, the output file generated is shown in Example 167.

Example 167 TEST_VHDL Written Out With Port Types in VHDL Format

```
library IEEE;

use IEEE.std_logic_1164.all;

entity test_vhdl is

   port( a_3_port, a_2_port, a_1_port, a_0_port : in std_logic;
b_3_port,
         b_2_port, b_1_port, b_0_port : out std_logic);

end test_vhdl;

architecture SYN_structural of test_vhdl is

   component GTECH_NOT
      port( A : in std_logic;  Z : out std_logic);
   end component;

begin

   I_0 : GTECH_NOT port map( A => a_3_port, Z => b_3_port);
   I_1 : GTECH_NOT port map( A => a_2_port, Z => b_2_port);
   I_2 : GTECH_NOT port map( A => a_1_port, Z => b_1_port);
   I_3 : GTECH_NOT port map( A => a_0_port, Z => b_0_port);

end SYN_structural;
```

VHDL Netlister Coding Considerations

To understand how the VHDL netlister writes out designs, you need to be familiar with the following coding considerations:

- Built-In Type Conversion Function
- How the Netlister Handles Custom Types
- Case Sensitivity

These issues are discussed in the nest sections.

Built-In Type Conversion Function

The VHDL netlister does not use packages and does not check for type equivalence. If you do not provide your own type conversion functions, the VHDL netlister translates only the logic values 0 and 1. Example 168 shows the VHDL netlister's built-in type conversion function that converts from type std_logic_vector to type my_bit.

Example 168 Type Conversion Function

```
-- User-defined type declaration
attribute ENUM_ENCODING : STRING;
type my_bit is (A, B, C) ;
attribute ENUM_ENCODING of my_bit : type is "00 01 11";

-- std_logic_vector to enum type function
function std_logic_vector_to_my_bit(arg : in std_logic_vector ( 1 to 2 ))
return my_bit is
-- synopsys built_in SYN_FEED_THRU;
begin
 case arg is
  when "00" => return A;
  when "01" => return B;
  when "11" => return C;
  when others => assert FALSE -- this should not happen.
   report "un-convertible value"
   severity warning;
   return A;
 end case;
end;
```

How the Netlister Handles Custom Types

All types you use should be resolved. If types are not resolved, the VHDL netlister uses built-in resolution functions to resolve conflicts between multiple drivers on the same signal. Use the following functions to specify your own resolution function to the VHDL netlister:

```
vhdlout_three_state_res_func
vhdlout_wired_and_res_func
vhdlout_wired_or_res_func
```

Example 169 shows the resolution function the VHDL netlister writes out. This resolution function is used to resolve the value for multiple sources driving a signal, port, or pin.

Example 169 VHDL Resolution Function

```
function X( inputs : in vhdlout_bit_vector_type ) return vhdlout_bit_type is
-- synopsys resolution_method three_state
variable retval: vhdlout_bit_type;
begin
  retval := vhdlout_three_state_name;
 for i in inputs'range loop
         if inputs(i) /= vhdlout_three_state_name then
                 if ( retval = vhdlout_three_state_name ) then
    retval := inputs(i);
   else
                         retval := vhdlout_unknown_name
    exit;
   end if;
  end if;
 end loop;
return retval;
end X;
```

Example 170 shows a simplified description of the process flow for the resolution function in Example 169.

In this example, the `vhdlout_three_state_name` and `vhdlout_unknown_name` variables use the default values z and x, respectively, for brevity. You can set the values for both of these variables.

Example 170 Pseudocode of VHDL Resolution Function

```
if the only logic values are 'z'
 return 'z'
if there are 'z's and another logic value
 return the other logic value
if there are non-'z' logic values that are different
 return 'x'
else
 return the common logic value
```

Case Sensitivity

The VHDL netlist writer is case insensitive. For example, \A and \a are considered to be unique identifiers; however, the VHDL netlist writer considers them to be the same identifier.

Note:

The VHDL netlist reader is case-sensitive and supports the VHDL 93 standard.

Feedback

8

VHDL-2008 Language Support

The following topics describe the VHDL-2008 Language support in the HDL Compiler tool:

- VHDL-2008 Setting
- fixed_generic_pkg Package

VHDL-2008 Setting

By default, the HDL Compiler tool is set to read VHDL-2008 language constructs.

Note:

If you set the VHDL-2008 language constructs, it enables VHDL-2008 parsing semantics. The language features supported for VHDL-2008 are Simplified Sensitivity List and fixed_generic_pkg.

To specify which VHDL language version to use during the read process, set the `hdlin_vhdl_std` variable. The valid values are `1987`, `1993`, and `2008`, corresponding to the 1987, 1993, and 2008 VHDL LRM releases respectively.

Simplified Sensitivity List

When the VHDL version is set to `2008`, you can use the all keyword in the sensitivity list. Instead of listing all inputs to a combinational logic process, use the `all` keyword to make the process sensitive to all the input signals. This simplifies the sensitivity list, reducing mismatches between simulation and synthesis.

fixed_generic_pkg Package

The HDL Compiler tool supports the VHDL-2008 fixed_generic_pkg package, which is used to implement fixed-point arithmetic.

The HDL Compiler tool supports a compatible version of the VHDL-2008 fixed-point package. The VHDL-2008 fixed-point package declares generics to control the rounding and overflow behaviors, while the compatible version, fixed_generic_pkg package, defines these elements as constants with the following constant values:

```
package fixed_generic_pkg is
   constant fixed_round_style : FIXED_ROUND_STYLE_TYPE := fixed_round;
   constant fixed_overflow_style : FIXED_OVERFLOW_STYLE_TYPE :=
fixed_saturate;
   constant fixed_guard_bits: NATURAL := 3;
   constant no_warning : BOOLEAN := false
...
```

Because the compatible fixed-point package uses constants rather than generics, you do not need to instantiate it. This package is installed in the Synopsys root directory. To access it, include the following statement in your VHDL code:

```
library IEEE;
use IEEE.fixed_generic_pkg.all
```

To learn the fixed-point package and the functions it provides, see

- fixed_generic_pkg Data Type
- Conversion Functions
- Resize Functions
- Arithmetic Functions
- Comparison Functions
- Logical Operator Functions
- Shift and Rotate Functions

fixed_generic_pkg Data Type

The fixed_generic_pkg package defines the UNRESOLVED_UFIXED and UNRESOLVED_SFIXED data types for unsigned and signed representations respectively.

```
type UNRESOLVED_UFIXED is array (INTEGER range <>) of STD_ULOGIC;
type UNRESOLVED_SFIXED is array (INTEGER range <>) of STD_ULOGIC;
```

To be compatible with the VHDL-2008 data types, the fixed_generic_pkg package defines the next subtypes:

```
subtype U_UFIXED is UNRESOLVED_ufixed;
subtype U_SFIXED is UNRESOLVED_sfixed;
subtype UFIXED is UNRESOLVED_ufixed;
subtype SFIXED is UNRESOLVED_sfixed;
```

Note:

Synthesis neither checks for nor resolves possible data collisions on a synthesized three-state bus. You should ensure that the three-state enablers for a common bus are not active at the same time.

The fixed-point data types define the location of the binary point using a negative index, and the binary point resides between index 0 and -1. In the following example, Y represents a signed fixed-point number, where bits 7 through 0 are designated to the integer part of the number and bits -1 through -6 are designated to the fractional part.

```
signal Y : SFIXED (7 downto -6)
```

You must declare fixed-point data types with an index range in descending order; an index range in ascending order is not allowed. For example, if a number is declares as SFIXED (-5 to 6), the tool issues an error message. For more information, see the IEEE Std 1076-2008.

Conversion Functions

The fixed-point package provides functions to convert INTEGER, REAL, SIGNED, UNSIGNED, and literals. To call a conversion function, you can specify the upper and lower index bounds or pass a parameter that uses the upper and lower index bounds. In the same way, the package provides functions to convert fixed-point values to UNSIGNED, SIGNED, INTEGER, and REAL. For example,

```
signal r : SFIXED (7 downto -6);
signal i : INTEGER;
signal s : SIGNED (13 downto 0);

i <= to_integer (r);    -- Directly specify the index bounds
s <= to_signed  (r, s); -- s pass a reference to set the output length
```

You can optionally specify the overflow_style, round_style, and guard_bits arguments for the conversion functions. If you do not specify an argument, the tool uses the default.

The following example shows how to use the to_sfixed conversion function:

```
signal p : SFIXED (7 downto -6);
signal q : SFIXED (3 downto -7);

-- -3.125 literal is sized to be assigned to "p"
p <= to_sfixed (-3.125, 7, -6);     -- Directly specify the index bounds
p <= to_sfixed (-3.125, p'high, p'low); -- Use the "p" attributes
p <= to_sfixed (-3.125, p);         -- Pass reference to set the index

-- "p" sized to fix "q" index bounds
q <= to_sfixed (p, q);
```

Resize Functions

The fixed-point package data widths are designed to prevent overflows. The index range of each operation result is defined in the following table.

Operation	Result range
A + B	Max(A'left, B'left) + 1 downto Min(A'right, B'right)
A – B	Max(A'left, B'left) + 1 downto Min(A'right, B'right)
A * B	A'left + B'left + 1 downto A'right + B'right
A rem B	Min(A'left, B'left) downto Min(A’right, B'right)
Signed A/B	A'left – B'right + 1 downto A'right – B'left
Signed A mod B	Min(A'left, B'left) downto Min(A'right, B'right)
Signed reciprocal(A)	–A'right downto –A'left – 1
abs A	A'left + 1 downto A'right
– A	A'left + 1 downto A'right
Unsigned A/B	A'left – B'right downto A'right – B'left – 1
Unsigned A mod B	B'left downto Min(A'right, B'right)
Unsigned reciprocal(A	–A'right + 1 downto –A'lef

The package provides the sfixed_high, ufixed_high, sfixed_low, and ufixed_low functions to calculate index ranges and to resize output results. You specify the operands and operator for each function to calculate the index range for each operation. For example,

```
signal a : SFIXED (5 downto -3);
signal b : SFIXED (7 downto -9);
signal rslt : SFIXED (sfixed_high (a'high, a'low, '/', b'high, b'low)
              downto sfixed_low  (a'high, a'low, '/', b'high, b'low));
begin
   rslt <= a / b; -- signed fixed-point divide ...
```

The valid operators include +, -, *, /, r or R (rem), m or M (mod), 1 (reciprocal), a or A (abs), and n or N (unary). You can also pass the operands as a parameter to obtain the upper and lower bounds of the index range automatically. For example,

```
signal rslt : SFIXED (sfixed_high (a'/', b)
              downto sfixed_low (a', '/', b));
```

You can use the resize functions to fix an output size following the rounding and saturation rules. For example,

```
rslt <= resize ( a / b,            -- arg
                 rslt'high,        -- left index
                 rslt'low,         -- right index
                 fixed_truncate,   -- round style
                 fixed_wrap);      -- overflow style
```

Alternatively, you can resize an output by passing the resulting variable as a reference to calculate the index bounds. For example,

```
rslt <= resize ( a / b,            -- arg
                 rslt,             -- result reference
                 fixed_truncate,   -- round style
                 fixed_wrap);      -- overflow style
```

To use the round_style and overflow_style values, you need to include the fixed_float_types package in your VHDL code, as shown in the following statement:

•

```
library IEEE;
use IEEE.fixed_generic_pkg.all
use IEEE.fixed_float_types.all
```

The fixed_float_types package defines the round_style_type and overflow_style_type data types.

```
type FIXED_ROUND_STYLE_TYPE is (fixed_round, fixed_truncate);
type FIXED_OVERFLOW_STYLE_TYPE is (fixed_saturate, fixed_wrap);
```

If you do not specify the round_style and overflow_style values, the fixed_round and fixed_saturate values are used by default.

Arithmetic Functions

You can use the following arithmetic functions defined in the fixed-point package with the UFIXED, SFIXED, NATURAL, and INTEGER data types and literals (including REAL):

- Binary arithmetic functions: +, -, *, /, rem, mod, divide, remainder, and modulo
- Unary arithmetic functions: -, abs, and reciprocal

As shown in the following example, the fixed-point package supports the scalb function:

```
constant half: UFIXED (2 downto -2) := "00010";   --000.10
signal two UFIXED (5 downto 0);
two  <=  scalb (half, 2);   -- "00010."
```

The fixed-point package also supports the IS_NEGATIVE function, which returns true when the SFIXED argument is negative.

Comparison Functions

You can use the comparison functions (>, <, <=, >=, =, /=, and std_match) to compare the SFIXED (or UFIXED) data type with the SFIXED (or UFIXED), NATURAL, INTEGER, and REAL data types. The returned data type is BOOLEAN for these functions.

You can use the matching operators (?=,?/=,?>, and ?<) to compare the SFIXED (or UFIXED) data type with the SFIXED (or UFIXED), NATURAL, INTEGER, and REAL data types. The returned data type is STD_ULOGIC.

The maximum and minimum functions perform a comparison operation and return the appropriate values.

Logical Operator Functions

The fixed-point package provides functions to define all the logical operators, including NOT, AND, NAND, OR, NOR, XOR, and XNOR. These logical operator functions can operate on the UFIXED (or SFIXED) and STD_ULOGIC data types and return the UFIXED (or SFIXED) data type.

Operator	**L**	**R**	**Return type**
nor	UFIXED	--	UFIXED
nor	SFIXED	--	SFIXED
and, nand, or, nor, xor, xnor	UFIXED	UFIXED	UFIXED
and, nand, or, nor, xor, xnor	SFIXED	SFIXED	SFIXED
and, nand, or, nor, xor, xnor	STD_ULOGIC	SFIXED	SFIXED
and, nand, or, nor, xor, xnor	SFIXED	STD_ULOGIC	SFIXED
and, nand, or, nor, xor, xnor	STD_ULOGIC	UFIXED	UFIXED
and, nand, or, nor, xor, xnor	UFIXED	STD_ULOGIC	UFIXED

The fixed-point package provides reduction functions to perform reduction operations on all bits of a vector (UFIXED or SFIXED), returning an STD_ULOGIC data type (and_reduce, nand_reduce, or_reduce, nor_reduce, xor_reduce, and xnor_reduce).

Note:

As a compatibility version of VHDL-2008, the fixed-point package adds a postfix of _reduce to all reduction functions. For example, and_reduce and nand_reduce.

Shift and Rotate Functions

The fixed-point package provides functions to shift bits for UFIXED and SFIXED data types. The following table shows the shift function declarations.

Operator	Arg	Count	Return type
sll	UFIXED	INTEGER	UFIXED
srl	UFIXED	INTEGER	UFIXED
rol	UFIXED	INTEGER	UFIXED
ror	UFIXED	INTEGER	UFIXED
sla	UFIXED	INTEGER	UFIXED
sra	UFIXED	INTEGER	UFIXED
sll	SFIXED	INTEGER	SFIXED
srl	SFIXED	INTEGER	SFIXED
rol	SFIXED	INTEGER	SFIXED
ror	SFIXED	INTEGER	SFIXED
sla	SFIXED	INTEGER	SFIXED
sra	SFIXED	INTEGER	SFIXED
shift_left	UFIXED	NATURAL	UFIXED
shift_right	UFIXED	NATURAL	UFIXED
shift_left	SFIXED	NATURAL	SFIXED
shift_right	SFIXED	NATURAL	SFIXED

A
Examples

Source files for examples demonstrating the use of VHDL are typically in the /synopsys/syn/examples/vhdl directory. These examples are included in the following sections:

Read-Only Memory

Example 171 shows how you can define a read-only memory in VHDL. The ROM is defined as an array constant, ROM. Each line of the constant array specification defines the contents of one ROM address. To read from the ROM, index into the array.

The number of ROM storage locations and bit-width is easy to change. The subtype ROM_RANGE specifies that the ROM contains storage locations 0 to 7. The constant ROM_WIDTH specifies that the ROM is 5 bits wide.

After you define a ROM constant, you can index into that constant many times to read many values from the ROM. If the ROM address is computable, no logic is built and the appropriate data value is inserted. If the ROM address is not computable, logic is built for each index into the value. For this reason, consider resource sharing when using a ROM. In Example 171, ADDR is not computable, so logic is synthesized to compute the value.

HDL Compiler does not actually instantiate a typical array-logic ROM, such as those available from ASIC vendors. Instead, it creates the ROM from random logic gates (AND, OR, NOT, and so on). This type of implementation is preferable for small ROMs and for ROMs that are regular. For very large ROMs, consider using an array-logic implementation supplied by your ASIC vendor.

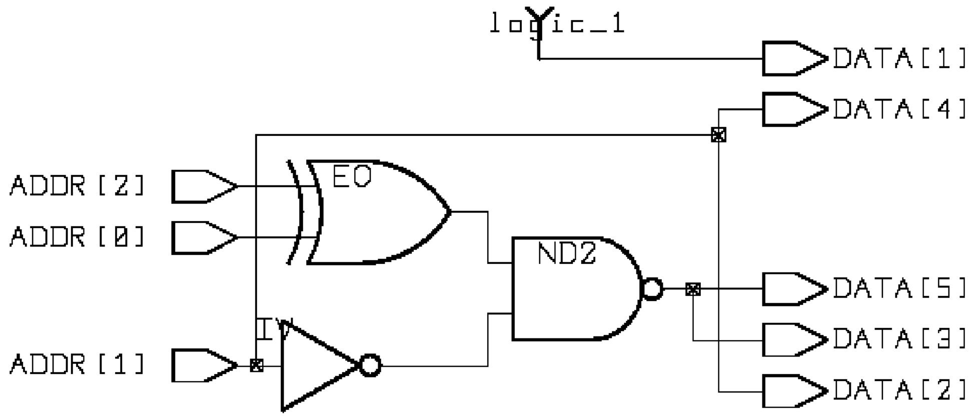
Example 171 shows the VHDL source code, and Figure 19 shows the synthesized circuit schematic.

Example 171 Implementation of a ROM in Random Logic

```
package ROMS is
  -- declare a 5x8 ROM called ROM
  constant ROM_WIDTH: INTEGER := 5;
  subtype ROM_WORD is BIT_VECTOR (1 to ROM_WIDTH);
  subtype ROM_RANGE is INTEGER range 0 to 7;
  type ROM_TABLE is array (0 to 7) of ROM_WORD;
  constant ROM: ROM_TABLE := ROM_TABLE'(
      ROM_WORD'("10101"),              -- ROM contents
      ROM_WORD'("10000"),
      ROM_WORD'("11111"),
      ROM_WORD'("11111"),
      ROM_WORD'("10000"),
      ROM_WORD'("10101"),
      ROM_WORD'("11111"),
      ROM_WORD'("11111"));
end ROMS;
use work.ROMS.all;   -- Entity that uses ROM
entity ROM_5x8 is
  port(ADDR: in ROM_RANGE;
       DATA: out ROM_WORD);
end;
architecture BEHAVIOR of ROM_5x8 is
begin
  DATA <= ROM(ADDR);      -- Read from the ROM
end BEHAVIOR;
```

Feedback

Figure 19 Synthesized Circuit of the ROM



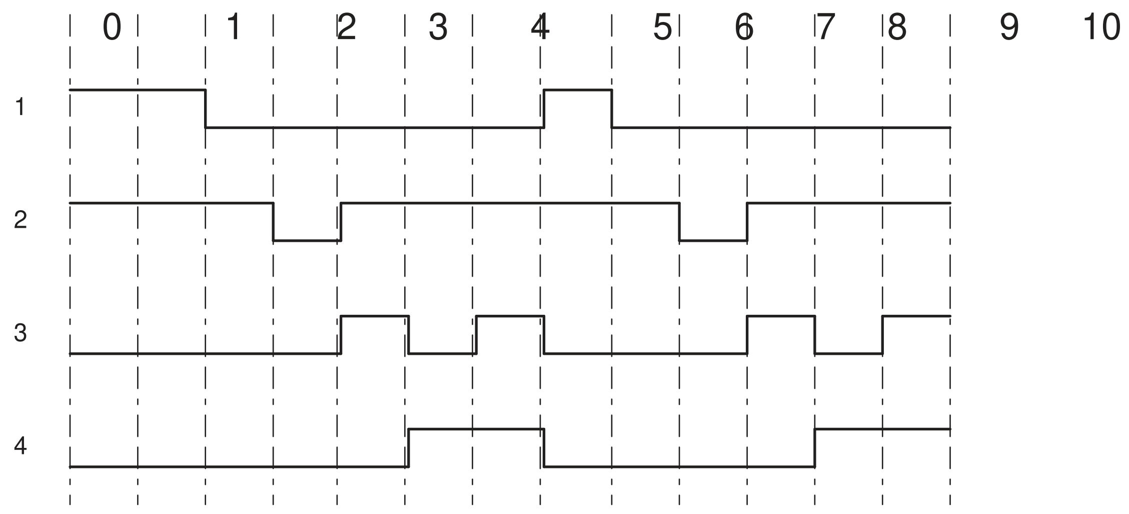
Waveform Generator

The waveform generator example shows how to use the previous ROM example to implement a waveform generator.

Assume that you want to produce the waveform output shown in Figure 20.

1. First, declare a ROM wide enough to hold the output signals (4 bits) and deep enough to hold all time steps (0 to 12, for a total of 13).
2. Next, define the ROM so that each time step is represented by an entry in the ROM.
3. Finally, create a counter that cycles through the time steps (ROM addresses), generating the waveform at each time step.

Figure 20 Waveform Example



Example 172 shows an implementation for the waveform generator. It consists of a ROM, a counter, and some simple reset logic.

Example 172 Implementation of a Waveform Generator

```
package ROMS is
  -- a 4x13 ROM called ROM that contains the waveform
  constant ROM_WIDTH: INTEGER := 4;
  subtype ROM_WORD is BIT_VECTOR (1 to ROM_WIDTH);
  subtype ROM_RANGE is INTEGER range 0 to 12;
  type ROM_TABLE is array (0 to 12) of ROM_WORD;
  constant ROM: ROM_TABLE := ROM_TABLE'(
      "1100",   -- time step 0
      "1100",   -- time step 1
      "0100",   -- time step 2
      "0000",   -- time step 3
      "0110",   -- time step 4
      "0101",   -- time step 5
      "0111",   -- time step 6
      "1100",   -- time step 7
      "0100",   -- time step 8
      "0000",   -- time step 9
      "0110",   -- time step 10
      "0101",   -- time step 11
      "0111");  -- time step 12
end ROMS;

use work.ROMS.all;
entity WAVEFORM is            -- Waveform generator
```

```
  port(CLOCK: in BIT;
       RESET: in BOOLEAN;
       WAVES: out ROM_WORD);
end;

architecture BEHAVIOR of WAVEFORM is
  signal STEP: ROM_RANGE;
begin
  TIMESTEP_COUNTER: process   -- Time stepping process
  begin
    wait until CLOCK'event and CLOCK = '1';
    if RESET then             -- Detect reset
      STEP <= ROM_RANGE'low;  -- Restart
    elsif STEP = ROM_RANGE'high then  -- Finished?
      STEP <= ROM_RANGE'high;  -- Hold at last value
   -- STEP <= ROM_RANGE'low;   -- Continuous wave
    else
      STEP <= STEP + 1;        -- Continue stepping
    end if;
  end process TIMESTEP_COUNTER;

  WAVES <= ROM(STEP);
end BEHAVIOR;
```

When the counter STEP reaches the end of the ROM, STEP stops, generates the last value, then waits until a reset. To make the sequence automatically repeat, remove the following statement:

```
STEP <= ROM_RANGE'high;  -- Hold at last value
```

Use the following statement instead (commented out in Example 172):

```
STEP <= ROM_RANGE'low;   -- Continuous wave
```

Definable-Width Adder-Subtracter

VHDL lets you create functions for use with array operands of any size. This example shows an adder-subtracter circuit that, when called, is adjusted to fit the size of its operands.

Example 173 shows an adder-subtracter defined for two unconstrained arrays of bits (type BIT_VECTOR) in a package named MATH. When an unconstrained array type is used for an argument to a subprogram, the actual constraints of the array are taken from the actual parameter values in a subprogram call.

Example 173 MATH Package

```
package MATH is
  function ADD_SUB(L, R: BIT_VECTOR; ADD: BOOLEAN)
      return BIT_VECTOR;
    -- Add or subtract two BIT_VECTORs of equal length
```

```
end MATH;

package body MATH is
    function ADD_SUB(L, R: BIT_VECTOR; ADD: BOOLEAN)
        return BIT_VECTOR is
      variable CARRY: BIT;
      variable A, B, SUM:
          BIT_VECTOR(L'length-1 downto 0);
    begin
      if ADD then
          -- Prepare for an "add" operation
          A := L;
          B := R;
          CARRY := '0';
      else

          -- Prepare for a "subtract" operation
          A := L;
          B := not R;
          CARRY := '1';
      end if;

      -- Create a ripple carry chain; sum up bits
      for i in 0 to A'left loop
        SUM(i) := A(i) xor B(i) xor CARRY;
        CARRY := (A(i) and B(i)) or
                 (A(i) and CARRY) or
                 (CARRY and B(i));
      end loop;
      return SUM;        -- Result
    end;
end MATH;
```

Within the function ADD_SUB, two temporary variables, A and B, are declared. These variables are declared to be the same length as L (and necessarily, R) but have their index constraints normalized to L'length-1 downto 0. After the arguments are normalized, you can create a ripple carry adder by using a for loop.

No explicit references to a fixed array length are in the function ADD_SUB. Instead, the VHDL array attributes 'left and 'length are used. These attributes allow the function to work on arrays of any length.

Example 174 shows how to use the adder-subtracter defined in the MATH package. In this example, the vector arguments to functions ARG1 and ARG2 are declared as BIT_VECTOR(1 to 6). This declaration causes ADD_SUB to work with 6-bit arrays.

Example 174 Implementation of a 6-Bit Adder-Subtracter

```
use work.MATH.all;

entity EXAMPLE is
    port(ARG1, ARG2: in BIT_VECTOR(1 to 6);
         ADD: in BOOLEAN;
         RESULT : out BIT_VECTOR(1 to 6));
end EXAMPLE;

architecture BEHAVIOR of EXAMPLE is
begin
  RESULT <= ADD_SUB(ARG1, ARG2, ADD);
end BEHAVIOR;
```

Count Zeros—Combinational Version

The count zeros—combinational example, Example 175, illustrates a design problem in which an 8-bit-wide value is given and the circuit determines two things:

- That no more than one sequence of zeros is in the value.
- The number of zeros in that sequence (if any). This computation must be completed in a single clock cycle.

The circuit produces two outputs: the number of zeros found and an error indication.

A valid input value can have at most one consecutive series of zeros. A value consisting entirely of ones is defined as a valid value. If a value is invalid, the zero counter resets to 0. For example, the value 00000000 is valid and has eight zeros; value 11000111 is valid and has three zeros; value 00111100 is invalid.

Example 175 shows the VHDL description for the circuit. It consists of a single process with a for loop that iterates across each bit in the given value. At each iteration, a temporary INTEGER variable (TEMP_COUNT) counts the number of zeros encountered. Two temporary Boolean variables (SEEN_ZERO and SEEN_TRAILING), initially false, are set to true when the beginning and end of the first sequence of zeros are detected.

If a zero is detected after the end of the first sequence of zeros (after SEEN_TRAILING is true), the zero count is reset (to 0), ERROR is set to true, and the for loop is exited.

Example 175 shows a combinational (parallel) approach to counting the zeros. The next example shows a sequential (serial) approach.

Example 175 Count Zeros—Combinational

```
entity COUNT_COMB_VHDL is
  port(DATA:  in  BIT_VECTOR(7 downto 0);
       COUNT: out INTEGER range 0 to 8;
       ERROR: out BOOLEAN);
end;

architecture BEHAVIOR of COUNT_COMB_VHDL is
begin
  process(DATA)
    variable TEMP_COUNT : INTEGER range 0 to 8;
```

```
      variable SEEN_ZERO, SEEN_TRAILING : BOOLEAN;
    begin
      ERROR <= FALSE;
      SEEN_ZERO := FALSE;
      SEEN_TRAILING := FALSE;
      TEMP_COUNT := 0;
      for I in 0 to 7 loop
        if (SEEN_TRAILING and DATA(I) = '0') then
          TEMP_COUNT := 0;
          ERROR <= TRUE;
          exit;
        elsif (SEEN_ZERO and DATA(I) = '1') then
          SEEN_TRAILING := TRUE;
        elsif (DATA(I) = '0') then
          SEEN_ZERO := TRUE;
          TEMP_COUNT := TEMP_COUNT + 1;
        end if;
      end loop;

      COUNT <= TEMP_COUNT;
    end process;
  end BEHAVIOR;
```

Count Zeros—Sequential Version

The count zeros—sequential example, Example 176, shows a sequential (clocked) variant of the preceding design (Count Zeros—Combinational Version).

The circuit now accepts the 8-bit data value serially, 1 bit per clock cycle, by using the DATA and CLK inputs. The other two inputs are

- RESET, which resets the circuit
- READ, which causes the circuit to begin accepting data bits

The circuit's three outputs are

- IS_LEGAL, which is true if the data was a valid value
- COUNT_READY, which is true at the first invalid bit or when all 8 bits have been processed
- COUNT, the number of zeros (if IS_LEGAL is true)

Note:

The output port COUNT is declared with mode BUFFER so that it can be read inside the process. OUT ports can only be written to, not read in.

Example 176 Count Zeros—Sequential

```
entity COUNT_SEQ_VHDL is
  port(DATA, CLK: in BIT;
       RESET, READ: in BOOLEAN;
       COUNT: buffer INTEGER range 0 to 8;
       IS_LEGAL: out BOOLEAN;
       COUNT_READY: out BOOLEAN);
end;
architecture BEHAVIOR of COUNT_SEQ_VHDL is
begin
  process
    variable SEEN_ZERO, SEEN_TRAILING: BOOLEAN;
    variable BITS_SEEN: INTEGER range 0 to 7;
  begin
    wait until CLK'event and CLK = '1';

    if(RESET) then
      COUNT_READY    <= FALSE;
      IS_LEGAL    <= TRUE;   -- signal assignment
      SEEN_ZERO    := FALSE;   -- variable assignment
      SEEN_TRAILING := FALSE;
      COUNT    <= 0;
      BITS_SEEN    := 0;
    else
      if (READ) then
        if (SEEN_TRAILING and DATA = '0') then
          IS_LEGAL <= FALSE;
          COUNT <= 0;
          COUNT_READY <= TRUE;
        elsif (SEEN_ZERO and DATA = '1') then
          SEEN_TRAILING := TRUE;
        elsif (DATA = '0') then
          SEEN_ZERO := TRUE;
          COUNT <= COUNT + 1;
        end if;

        if (BITS_SEEN = 7) then
          COUNT_READY <= TRUE;
        else
          BITS_SEEN :=  BITS_SEEN + 1;
        end if;

      end if;    -- if (READ)
    end if;      -- if (RESET)
  end process;
end BEHAVIOR;
```

Soft Drink Machine—State Machine Version

The soft drink machine—state machine example, Example 177, is a control unit for a soft drink vending machine.

The circuit reads signals from a coin input unit and sends outputs to a change dispensing unit and a drink dispensing unit.

Here are the design parameters for Example 177 and the example in Soft Drink Machine —Count Nickels Version:

- This example assumes that only one kind of soft drink is dispensed.
- This is a clocked design with CLK and RESET input signals.
- The price of the drink is 35 cents.
- The input signals from the coin input unit are NICKEL_IN (nickel deposited), DIME_IN (dime deposited), and QUARTER_IN (quarter deposited).
- The output signals to the change dispensing unit are NICKEL_OUT and DIME_OUT.
- The output signal to the drink dispensing unit is DISPENSE (dispense drink).

The first VHDL description for this design uses a state machine description style. The second VHDL description is in the example in Soft Drink Machine—Count Nickels Version.

Example 177 Soft Drink Machine—State Machine

```
library synopsys; use synopsys.attributes.all;
entity DRINK_STATE_VHDL is
  port(NICKEL_IN, DIME_IN, QUARTER_IN, RESET: BOOLEAN;
       CLK: BIT;
       NICKEL_OUT, DIME_OUT, DISPENSE: out BOOLEAN);
end;

architecture BEHAVIOR of DRINK_STATE_VHDL is
  type STATE_TYPE is (IDLE, FIVE, TEN, FIFTEEN,
                      TWENTY, TWENTY_FIVE, THIRTY, OWE_DIME);
  signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;
  attribute STATE_VECTOR : STRING;
  attribute STATE_VECTOR of BEHAVIOR : architecture is
     "CURRENT_STATE";

attribute sync_set_reset of reset : signal is "true";
begin
  process(NICKEL_IN, DIME_IN, QUARTER_IN,
          CURRENT_STATE, RESET, CLK)
  begin
    -- Default assignments
    NEXT_STATE <= CURRENT_STATE;
    NICKEL_OUT <= FALSE;
    DIME_OUT <= FALSE;
    DISPENSE <= FALSE;

    -- Synchronous reset
    if(RESET) then
      NEXT_STATE <= IDLE;
    else
      -- State transitions and output logic
      case CURRENT_STATE is
        when IDLE =>
          if(NICKEL_IN) then
            NEXT_STATE <= FIVE;
          elsif(DIME_IN) then
            NEXT_STATE <= TEN;
          elsif(QUARTER_IN) then
            NEXT_STATE <= TWENTY_FIVE;
          end if;

        when FIVE =>
          if(NICKEL_IN) then
            NEXT_STATE <= TEN;
          elsif(DIME_IN) then
            NEXT_STATE <= FIFTEEN;
          elsif(QUARTER_IN) then
            NEXT_STATE <= THIRTY;
          end if;

        when TEN =>
          if(NICKEL_IN) then
            NEXT_STATE <= FIFTEEN;
          elsif(DIME_IN) then
            NEXT_STATE <= TWENTY;
          elsif(QUARTER_IN) then
            NEXT_STATE <= IDLE;
            DISPENSE <= TRUE;
          end if;

        when FIFTEEN =>
          if(NICKEL_IN) then
            NEXT_STATE <= TWENTY;
          elsif(DIME_IN) then
            NEXT_STATE <= TWENTY_FIVE;
          elsif(QUARTER_IN) then
            NEXT_STATE <= IDLE;
            DISPENSE <= TRUE;
            NICKEL_OUT <= TRUE;
          end if;

        when TWENTY =>
          if(NICKEL_IN) then
            NEXT_STATE <= TWENTY_FIVE;
          elsif(DIME_IN) then
            NEXT_STATE <= THIRTY;
          elsif(QUARTER_IN) then
            NEXT_STATE <= IDLE;
            DISPENSE <= TRUE;
            DIME_OUT <= TRUE;
          end if;

        when TWENTY_FIVE =>
          if(NICKEL_IN) then
            NEXT_STATE <= THIRTY;
          elsif(DIME_IN) then
            NEXT_STATE <= IDLE;
            DISPENSE <= TRUE;
          elsif(QUARTER_IN) then
            NEXT_STATE <= IDLE;
            DISPENSE <= TRUE;
            DIME_OUT <= TRUE;
            NICKEL_OUT <= TRUE;
          end if;

        when THIRTY =>
          if(NICKEL_IN) then
            NEXT_STATE <= IDLE;
            DISPENSE <= TRUE;
          elsif(DIME_IN) then
            NEXT_STATE <= IDLE;
            DISPENSE <= TRUE;
            NICKEL_OUT <= TRUE;
          elsif(QUARTER_IN) then
            NEXT_STATE <= OWE_DIME;
            DISPENSE <= TRUE;
            DIME_OUT <= TRUE;
          end if;

        when OWE_DIME =>
          NEXT_STATE <= IDLE;
          DIME_OUT <= TRUE;
      end case;
    end if;
  end process;

  -- Synchronize state value with clock
  -- This causes it to be stored in flip-flops
  process
  begin
    wait until CLK'event and CLK = '1';
    CURRENT_STATE <= NEXT_STATE;
  end process;
end BEHAVIOR;
```

Soft Drink Machine—Count Nickels Version

The soft drink machine—count nickels example, Example 178, uses the same design parameters as the preceding Example 177 (Soft Drink Machine—State Machine Version), with the same input and output signals. In this version, a counter counts the number of nickels deposited. This counter is incremented by 1 if the deposit is a nickel, by 2 if it is a dime, and by 5 if it is a quarter.

Example 178 Soft Drink Machine—Count Nickels

```
entity DRINK_COUNT_VHDL is
  port(NICKEL_IN, DIME_IN, QUARTER_IN, RESET: BOOLEAN;
       CLK: BIT;
       NICKEL_OUT, DIME_OUT, DISPENSE: out BOOLEAN);
end;

architecture BEHAVIOR of DRINK_COUNT_VHDL is
  signal CURRENT_NICKEL_COUNT,
         NEXT_NICKEL_COUNT: INTEGER range 0 to 7;
  signal CURRENT_RETURN_CHANGE, NEXT_RETURN_CHANGE : BOOLEAN;
begin

  process(NICKEL_IN, DIME_IN, QUARTER_IN, RESET, CLK,
          CURRENT_NICKEL_COUNT, CURRENT_RETURN_CHANGE)
    variable TEMP_NICKEL_COUNT: INTEGER range 0 to 12;
  begin
    -- Default assignments
    NICKEL_OUT <= FALSE;
    DIME_OUT <= FALSE;
    DISPENSE <= FALSE;
    NEXT_NICKEL_COUNT <= 0;
    NEXT_RETURN_CHANGE <= FALSE;

    -- Synchronous reset
    if (not RESET) then
      TEMP_NICKEL_COUNT := CURRENT_NICKEL_COUNT;

      -- Check whether money has come in
      if (NICKEL_IN) then
        -- NOTE:  This design will be flattened, so
        --    these multiple adders will be optimized
        TEMP_NICKEL_COUNT := TEMP_NICKEL_COUNT + 1;
      elsif(DIME_IN) then
        TEMP_NICKEL_COUNT := TEMP_NICKEL_COUNT + 2;
      elsif(QUARTER_IN) then
        TEMP_NICKEL_COUNT := TEMP_NICKEL_COUNT + 5;
      end if;

      -- Enough deposited so far?
      if(TEMP_NICKEL_COUNT >= 7) then
        TEMP_NICKEL_COUNT := TEMP_NICKEL_COUNT - 7;
        DISPENSE <= TRUE;
      end if;

      -- Return change
      if(TEMP_NICKEL_COUNT >= 1 or
         CURRENT_RETURN_CHANGE) then
        if(TEMP_NICKEL_COUNT >= 2) then
```

```
          DIME_OUT <= TRUE;
          TEMP_NICKEL_COUNT := TEMP_NICKEL_COUNT - 2;
          NEXT_RETURN_CHANGE <= TRUE;
        end if;
        if(TEMP_NICKEL_COUNT = 1) then
          NICKEL_OUT <= TRUE;
          TEMP_NICKEL_COUNT := TEMP_NICKEL_COUNT - 1;
        end if;
      end if;

      NEXT_NICKEL_COUNT <= TEMP_NICKEL_COUNT;
    end if;
  end process;

  -- Remember the return-change flag and
  -- the nickel count for the next cycle
  process
  begin
    wait until CLK'event and CLK = '1';
    CURRENT_RETURN_CHANGE <= NEXT_RETURN_CHANGE;
    CURRENT_NICKEL_COUNT <= NEXT_NICKEL_COUNT;
  end process;

end BEHAVIOR;
```

Carry-Lookahead Adder

This example uses concurrent procedure calls to build a 32-bit carry-lookahead adder. The adder is built by partitioning of the 32-bit input into eight slices of 4 bits each. Each of the eight slices computes propagate and generate values by using the PG procedure.

Propagate (output P from PG) is ’1’ for a bit position if that position propagates a carry from the next-lower position to the next-higher position. Generate (output G) is ’1’ for a bit position if that position generates a carry to the next-higher position, regardless of the carry-in from the next lower position. The carry-lookahead logic reads the carry-in, propagate, and generate information computed from the inputs. The logic computes the carry value for each bit position and makes the addition operation an XOR of the inputs and the carry values.

Carry Value Computations

The carry values are computed by a three-level tree of 4-bit carry-lookahead blocks.

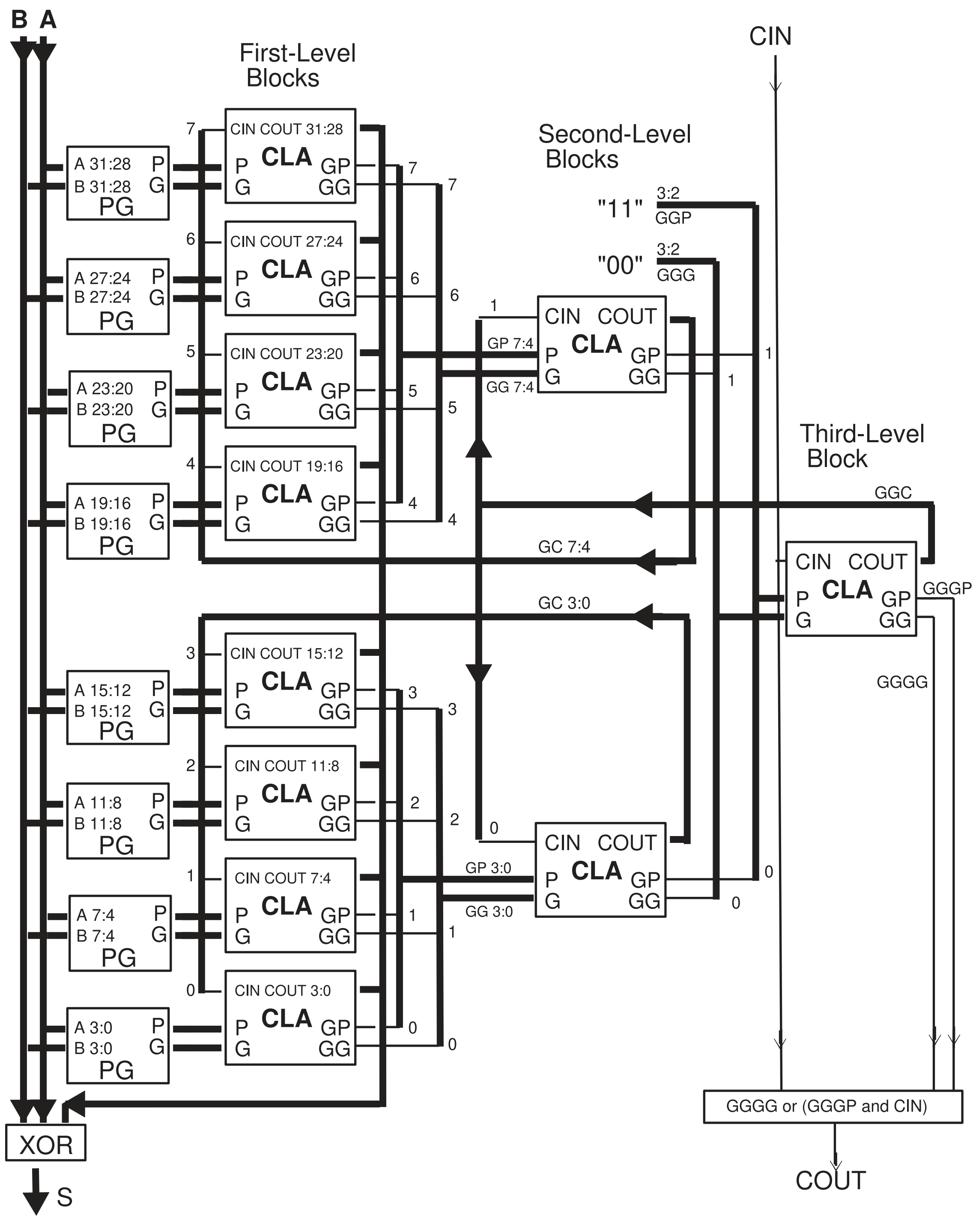
- The first level of the tree computes the 32 carry values and the eight group-propagate and generate values. Each of the first-level group-propagate and generate values tells if that 4-bit slice propagates and generates carry values from the next-lower group to the next-higher group. The first-level lookahead blocks read the group carry computed at the second level.
- The second-level lookahead blocks read the group-propagate and generate information from the four first-level blocks and then compute their own group-propagate and

generate information. The second-level lookahead blocks also read group carry information computed at the third level to compute the carries for each of the third-level blocks.

- The third-level block reads the propagate and generate information of the second level to compute a propagate and generate value for the entire adder. It also reads the external carry to compute each second-level carry. The carry-out for the adder is ’1’ if the third-level generate is ’1’ or if the third-level propagate is ’1’ and the external carry is ’1’.

  The third-level carry-lookahead block is capable of processing four second-level blocks. But because there are only two second-level blocks, the high-order 2 bits of the computed carry are ignored; the high-order 2 bits of the generate input to the third-level are set to 0, "00", and the propagate high-order bits are set to "11". These settings cause the unused portion to propagate carries but not to generate them. Figure 21 shows the overall structure for the carry-lookahead adder.

Figure 21 Carry-Lookahead Adder Block Diagram



The VHDL implementation of the design in Figure 21 is accomplished with the following procedures:

CLA

Names a 4-bit carry-lookahead block.

PG

Computes first-level propagate and generate information.

SUM

Computes the sum by adding the XOR values to the inputs with the carry values computed by CLA.

BITSLICE

Collects the first-level CLA blocks, the PG computations, and the SUM. This procedure performs all the work for a 4-bit value except for the second- and third-level lookaheads.

Example 179 shows a VHDL description of the adder.

Example 179 Carry-Lookahead Adder

```
package LOCAL is
  constant N:   INTEGER := 4;

  procedure BITSLICE(
      A, B: in BIT_VECTOR(3 downto 0);
      CIN: in BIT;
      signal S: out BIT_VECTOR(3 downto 0);
      signal GP, GG: out BIT);
  procedure PG(
      A, B: in BIT_VECTOR(3 downto 0);
      P, G: out BIT_VECTOR(3 downto 0));
  function SUM(A, B, C: BIT_VECTOR(3 downto 0))
      return BIT_VECTOR;
  procedure CLA(
      P, G: in BIT_VECTOR(3 downto 0);
      CIN: in BIT;
      C: out BIT_VECTOR(3 downto 0);
      signal GP, GG: out BIT);
end LOCAL;

package body LOCAL is
  -----------------------------------------------
  -- Compute sum and group outputs from a, b, cin
  -----------------------------------------------

  procedure BITSLICE(
      A, B: in BIT_VECTOR(3 downto 0);
      CIN: in BIT;
      signal S: out BIT_VECTOR(3 downto 0);
      signal GP, GG: out BIT) is

    variable P, G, C: BIT_VECTOR(3 downto 0);
  begin
    PG(A, B, P, G);
    CLA(P, G, CIN, C, GP, GG);
    S <= SUM(A, B, C);
  end;

  -------------------------------------------------
  -- Compute propagate and generate from input bits
  -------------------------------------------------

  procedure PG(A, B: in BIT_VECTOR(3 downto 0);
                 P, G: out BIT_VECTOR(3 downto 0)) is

  begin
    P := A or B;
    G := A and B;
  end;

  --------------------------------------------------
  -- Compute sum from the input bits and the carries
  --------------------------------------------------

  function SUM(A, B, C: BIT_VECTOR(3 downto 0))
      return BIT_VECTOR is

  begin
    return(A xor B xor C);
  end;

  ------------------------------
  -- 4-bit carry-lookahead block
  ------------------------------

  procedure CLA(
      P, G: in BIT_VECTOR(3 downto 0);
      CIN: in BIT;
      C: out BIT_VECTOR(3 downto 0);
      signal GP, GG: out BIT) is
    variable TEMP_GP, TEMP_GG, LAST_C: BIT;
  begin
    TEMP_GP := P(0);
    TEMP_GG := G(0);
    LAST_C := CIN;
    C(0) := CIN;

    for I in 1 to N-1 loop
      TEMP_GP := TEMP_GP and P(I);
      TEMP_GG := (TEMP_GG and P(I)) or G(I);
      LAST_C := (LAST_C and P(I-1)) or G(I-1);
      C(I) := LAST_C;
    end loop;

    GP <= TEMP_GP;
    GG <= TEMP_GG;
  end;
end LOCAL;

use WORK.LOCAL.ALL;

---------------------------------
-- A 32-bit carry-lookahead adder
```

```
---------------------------------

entity ADDER is
  port(A, B: in BIT_VECTOR(31 downto 0);
       CIN: in BIT;
       S: out BIT_VECTOR(31 downto 0);
       COUT: out BIT);
end ADDER;
architecture BEHAVIOR of ADDER is

  signal GG,GP,GC: BIT_VECTOR(7 downto 0);
    -- First-level generate, propagate, carry
  signal GGG, GGP, GGC: BIT_VECTOR(3 downto 0);
    -- Second-level gen, prop, carry
  signal GGGG, GGGP: BIT;
    -- Third-level gen, prop

begin
  -- Compute Sum and 1st-level Generate and Propagate
  -- Use input data and the 1st-level Carries computed
  -- later.
  BITSLICE(A( 3 downto  0),B( 3 downto  0),GC(0),
           S( 3 downto  0),GP(0), GG(0));
  BITSLICE(A( 7 downto  4),B( 7 downto  4),GC(1),
           S( 7 downto  4),GP(1), GG(1));
  BITSLICE(A(11 downto  8),B(11 downto  8),GC(2),
           S(11 downto  8),GP(2), GG(2));
  BITSLICE(A(15 downto 12),B(15 downto 12),GC(3),
           S(15 downto 12),GP(3), GG(3));
  BITSLICE(A(19 downto 16),B(19 downto 16),GC(4),
           S(19 downto 16),GP(4), GG(4));
  BITSLICE(A(23 downto 20),B(23 downto 20),GC(5),
           S(23 downto 20),GP(5), GG(5));
  BITSLICE(A(27 downto 24),B(27 downto 24),GC(6),
           S(27 downto 24),GP(6), GG(6));
  BITSLICE(A(31 downto 28),B(31 downto 28),GC(7),
           S(31 downto 28),GP(7), GG(7));

  -- Compute first-level Carries and second-level
  -- generate and propagate.
  -- Use first-level Generate, Propagate, and
  -- second-level carry.
  process(GP, GG, GGC)
    variable TEMP: BIT_VECTOR(3 downto 0);
  begin
    CLA(GP(3 downto 0), GG(3 downto 0), GGC(0), TEMP,
        GGP(0), GGG(0));
    GC(3 downto 0) <= TEMP;
  end process;

  process(GP, GG, GGC)
    variable TEMP: BIT_VECTOR(3 downto 0);
  begin
    CLA(GP(7 downto 4), GG(7 downto 4), GGC(1), TEMP,
        GGP(1), GGG(1));
    GC(7 downto 4) <= TEMP;
  end process;

  -- Compute second-level Carry and third-level
  --   Generate and Propagate
  -- Use second-level Generate, Propagate and Carry-in
  --   (CIN)
  process(GGP, GGG, CIN)
    variable TEMP: BIT_VECTOR(3 downto 0);
  begin
    CLA(GGP, GGG, CIN, TEMP, GGGP, GGGG);
    GGC <= TEMP;
  end process;

  -- Assign unused bits of second-level Generate and
  --   Propagate
  GGP(3 downto 2) <= "11";
  GGG(3 downto 2) <= "00";

  -- Compute Carry-out (COUT)
  -- Use third-level Generate and Propagate and
  --   Carry-in (CIN).
  COUT <= GGGG or (GGGP and CIN);
end BEHAVIOR;
```

Implementation

In the carry-lookahead adder implementation, procedures perform the computation of the design. The procedures can also be in the form of separate entities and used by component instantiation, producing a hierarchical design. HDL Compiler does not collapse a hierarchy of entities, but it does collapse the procedure call hierarchy into one design.

The keyword *signal* is included before some of the interface parameter declarations. This keyword is required for the out formal parameters when the actual parameters must be signals.

The output parameter C from the CLA procedure is not declared as a signal; thus, it is not allowed in a concurrent procedure call. Only signals can be used in such calls. To overcome this problem, subprocesses are used, declaring a temporary variable TEMP. TEMP receives the value of the C parameter and assigns it to the appropriate signal (a generally useful technique).

Serial-to-Parallel Converter—Counting Bits

This example shows the design of a serial-to-parallel converter that reads a serial, bit-stream input and produces an 8-bit output.

The design reads the following inputs:

SERIAL_IN

  The serial input data.

RESET

  The input that, when it is ’1’, causes the converter to reset. All outputs are set to 0, and the converter is prepared to read the next serial word.

CLOCK

The value of RESET and SERIAL_IN, which is read on the positive transition of this clock. Outputs of the converter are also valid only on positive transitions.

The design produces the following outputs:

PARALLEL_OUT

The 8-bit value read from the SERIAL_IN port.

READ_ENABLE

The output that, when it is ’1’ on the positive transition of CLOCK, causes the data on PARALLEL_OUT to be read.

PARITY_ERROR

The output that, when it is ’1’ on the positive transition of CLOCK, indicates that a parity error has been detected on the SERIAL_IN port. When a parity error is detected, the converter halts until restarted by the RESET port.

Input Format

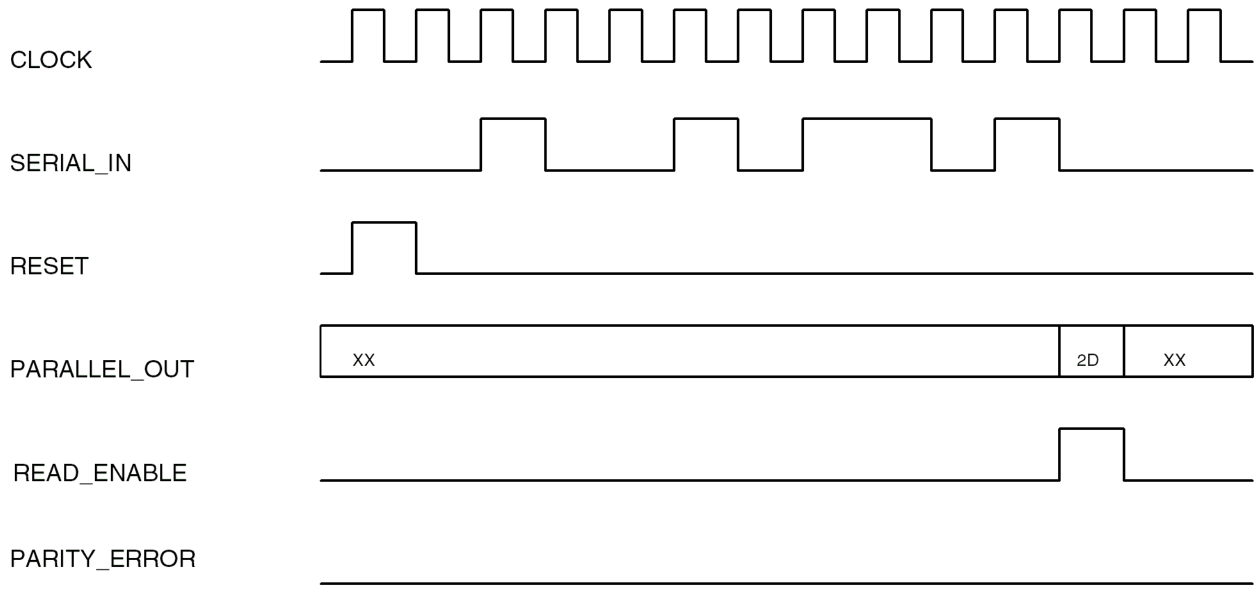
When no data is being transmitted to the serial port, keep it at a value of ’0’. Each 8-bit value requires 10 clock cycles to read it. On the 11th clock cycle, the parallel output value can be read.

In the first cycle, a ’1’ is placed on the serial input. This assignment indicates that an 8-bit value follows. The next 8 cycles transmit each bit of the value. The most significant bit is transmitted first. The 10th cycle transmits the parity of the 8-bit value. It must be ’0’ if an even number of ’1’ values are in the 8-bit data, and ’1’ otherwise. If the converter detects a parity error, it sets the PARITY_ERROR output to ’1’ and waits until the value is reset.

On the 11th cycle, the READ_ENABLE output is set to ’1’ and the 8-bit value can be read from the PARALLEL_OUT port. If the SERIAL_IN port has a ’1’ on the 11th cycle, another 8-bit value is read immediately; otherwise, the converter waits until SERIAL_IN goes to ’1’.

Figure 22 shows the timing of this design.

Figure 22 An Example of Waveform Through the Converter



Implementation Details

The implementation of the converter is as a four-state finite-state machine with synchronous reset. When a reset is detected, the converter enters a WAIT_FOR_START state. Description of the states follow.

WAIT_FOR_START

> Stay in this state until a ’1’ is detected on the serial input. When a ’1’ is detected, clear the PARALLEL_OUT registers and go to the READ_BITS state.

READ_BITS

> If the value of the current_bit_position counter is 8, all 8 bits have been read. Check the computed parity with the transmitted parity. If it is correct, go to the ALLOW_READ state; otherwise, go to the PARITY_ERROR state.
>
> If all 8 bits have not yet been read, set the appropriate bit in the PARALLEL_OUT buffer to the SERIAL_IN value, compute the parity of the bits read so far, and increment the current_bit_position.

ALLOW_READ

> This is the state where the outside world reads the PARALLEL_OUT value. When that value is read, the design returns to the WAIT_FOR_START state.

PARITY_ERROR_DETECTED

In this state, the PARITY_ERROR output is set to ’1’ and nothing else is done.

This design has four values stored in registers:

CURRENT_STATE

Remembers the state as of the last clock edge.

CURRENT_BIT_POSITION

Remembers how many bits have been read so far.

CURRENT_PARITY

Keeps a running XOR of the bits read.

CURRENT_PARALLEL_OUT

Stores each parallel bit as it is found.

The design has two processes: the combinational NEXT_ST containing the combinational logic and the sequential SYNCH that is clocked.

NEXT_ST performs all the computations and state assignments. The NEXT_ST process starts by assigning default values to all the signals it drives. This assignment guarantees that all signals are driven under all conditions. Next, the RESET input is processed. If RESET is not active, a case statement determines the current state and its computations. State transitions are performed by assignment of the next state’s value you want to the NEXT_STATE signal.

The serial-to-parallel conversion itself is performed by these two statements in the NEXT_ST process:

```
NEXT_PARALLEL_OUT(CURRENT_BIT_POSITION) <= SERIAL_IN;
NEXT_BIT_POSITION <= CURRENT_BIT_POSITION + 1;
```

The first statement assigns the current serial input bit to a particular bit of the parallel output. The second statement increments the next bit position to be assigned.

SYNCH registers and updates the stored values previously described. Each registered signal has two parts, NEXT_... and CURRENT_... :

NEXT_...

Signals hold values computed by the NEXT_ST process.

CURRENT_...

Signals hold the values driven by the SYNCH process. The CURRENT_... signals hold the values of the NEXT_... signals as of the last clock edge.

Example 180 shows a VHDL description of the converter.

Example 180 Serial-to-Parallel Converter—Counting Bits

```
-- Serial-to-Parallel Converter, counting bits

package TYPES is
  -- Declares types used in the rest of the design
  type STATE_TYPE is (WAIT_FOR_START,
                      READ_BITS,
                      PARITY_ERROR_DETECTED,
                      ALLOW_READ);
  constant PARALLEL_BIT_COUNT: INTEGER := 8;
  subtype PARALLEL_RANGE is INTEGER
      range 0 to (PARALLEL_BIT_COUNT-1);
  subtype PARALLEL_TYPE is BIT_VECTOR(PARALLEL_RANGE);
end TYPES;

use WORK.TYPES.ALL;     -- Use the TYPES package

entity SER_PAR is       -- Declare the interface
  port(SERIAL_IN, CLOCK, RESET: in BIT;
       PARALLEL_OUT: out PARALLEL_TYPE;
       PARITY_ERROR, READ_ENABLE: out BIT);
end;

architecture BEHAVIOR of SER_PAR is
  -- Signals for stored values
  signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;
  signal CURRENT_PARITY, NEXT_PARITY: BIT;
  signal CURRENT_BIT_POSITION, NEXT_BIT_POSITION:
      INTEGER range PARALLEL_BIT_COUNT downto 0;
  signal CURRENT_PARALLEL_OUT, NEXT_PARALLEL_OUT:
      PARALLEL_TYPE;
begin
NEXT_ST: process(SERIAL_IN, CURRENT_STATE, RESET,
                 CURRENT_BIT_POSITION, CURRENT_PARITY,
                 CURRENT_PARALLEL_OUT)
  -- This process computes all outputs, the next
  --   state, and the next value of all stored values
  begin
    PARITY_ERROR <= '0'; -- Default values for all
    READ_ENABLE <= '0';  --  outputs and stored values
    NEXT_STATE <= CURRENT_STATE;
    NEXT_BIT_POSITION <= 0;
    NEXT_PARITY <= '0';
    NEXT_PARALLEL_OUT <= CURRENT_PARALLEL_OUT;

    if (RESET = '1') then      -- Synchronous reset
      NEXT_STATE <= WAIT_FOR_START;
    else
      case CURRENT_STATE is   -- State processing
        when WAIT_FOR_START =>
          if (SERIAL_IN = '1') then
            NEXT_STATE <= READ_BITS;
            NEXT_PARALLEL_OUT <=
                PARALLEL_TYPE'(others=>'0');
          end if;
        when READ_BITS =>
          if (CURRENT_BIT_POSITION =
              PARALLEL_BIT_COUNT) then
            if (CURRENT_PARITY = SERIAL_IN) then
              NEXT_STATE <= ALLOW_READ;
              READ_ENABLE <= '1';
            else
              NEXT_STATE <= PARITY_ERROR_DETECTED;
            end if;
          else
            NEXT_PARALLEL_OUT(CURRENT_BIT_POSITION) <=
                SERIAL_IN;
            NEXT_BIT_POSITION <=
                CURRENT_BIT_POSITION + 1;
            NEXT_PARITY <= CURRENT_PARITY xor
                           SERIAL_IN;
          end if;
        when PARITY_ERROR_DETECTED =>
          PARITY_ERROR <= '1';
        when ALLOW_READ =>
          NEXT_STATE <= WAIT_FOR_START;
      end case;
    end if;
  end process;

  SYNCH: process
    -- This process remembers the stored values
    --    across clock cycles
  begin
    wait until CLOCK'event and CLOCK = '1';
    CURRENT_STATE <= NEXT_STATE;
    CURRENT_BIT_POSITION <= NEXT_BIT_POSITION;
    CURRENT_PARITY <= NEXT_PARITY;
    CURRENT_PARALLEL_OUT <= NEXT_PARALLEL_OUT;
  end process;

  PARALLEL_OUT <= CURRENT_PARALLEL_OUT;

end BEHAVIOR;
```

Serial-to-Parallel Converter—Shifting Bits

This example describes another implementation of the serial-to-parallel converter in the last example. This design performs the same function as the previous one but uses a different algorithm to do the conversion.

The previous implementation used a counter to indicate the bit of the output that was set when a new serial bit was read. In this implementation, the serial bits are shifted into place. Before the conversion occurs, a ’1’ is placed in the least-significant bit position. When that ’1’ is shifted out of the most significant position (position 0), the signal NEXT_HIGH_BIT is set to ’1’ and the conversion is complete.

Example 181 shows the listing of the second implementation. The differences are highlighted in bold. The differences relate to the removal of the ..._BIT_POSITION signals, the addition of ..._HIGH_BIT signals, and the change in the way NEXT_PARALLEL_OUT is computed.

Example 181 Serial-to-Parallel Converter—Shifting Bits

```
package TYPES is
  -- Declares types used in the rest of the design
  type STATE_TYPE is (WAIT_FOR_START,
                      READ_BITS,
                      PARITY_ERROR_DETECTED,
                      ALLOW_READ);
  constant PARALLEL_BIT_COUNT: INTEGER := 8;
  subtype PARALLEL_RANGE is INTEGER
      range 0 to (PARALLEL_BIT_COUNT-1);
  subtype PARALLEL_TYPE is BIT_VECTOR(PARALLEL_RANGE);
end TYPES;

use WORK.TYPES.ALL;       -- Use the TYPES package

entity SER_PAR is         -- Declare the interface
  port(SERIAL_IN, CLOCK, RESET: in BIT;
       PARALLEL_OUT: out PARALLEL_TYPE;
       PARITY_ERROR, READ_ENABLE: out BIT);
end;

architecture BEHAVIOR of SER_PAR is
  -- Signals for stored values
  signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;

  signal CURRENT_PARITY, NEXT_PARITY: BIT;
  signal CURRENT_HIGH_BIT, NEXT_HIGH_BIT: BIT;
  signal CURRENT_PARALLEL_OUT, NEXT_PARALLEL_OUT:
     PARALLEL_TYPE;
begin

NEXT_ST: process(SERIAL_IN, CURRENT_STATE, RESET,
                 CURRENT_HIGH_BIT, CURRENT_PARITY,
                 CURRENT_PARALLEL_OUT)
  -- This process computes all outputs, the next
  --   state, and the next value of all stored values
  begin
    PARITY_ERROR <= '0'; -- Default values for all
    READ_ENABLE <= '0';  --  outputs and stored values
    NEXT_STATE <= CURRENT_STATE;
    NEXT_HIGH_BIT <= '0';
    NEXT_PARITY <= '0';
    NEXT_PARALLEL_OUT <= PARALLEL_TYPE'(others=>'0');
    if(RESET = '1') then      -- Synchronous reset
      NEXT_STATE <= WAIT_FOR_START;
    else
      case CURRENT_STATE is   -- State processing
        when WAIT_FOR_START =>
          if (SERIAL_IN = '1') then
            NEXT_STATE <= READ_BITS;
            NEXT_PARALLEL_OUT <=
                PARALLEL_TYPE'(others=>'0');
          end if;
        when READ_BITS =>
          if (CURRENT_HIGH_BIT = '1') then
            if (CURRENT_PARITY = SERIAL_IN) then
              NEXT_STATE <= ALLOW_READ;
              READ_ENABLE <= '1';
            else
              NEXT_STATE <= PARITY_ERROR_DETECTED;
            end if;
          else
            NEXT_HIGH_BIT <= CURRENT_PARALLEL_OUT(0);
            NEXT_PARALLEL_OUT <=
                CURRENT_PARALLEL_OUT(
                    1 to PARALLEL_BIT_COUNT-1) &
                SERIAL_IN;
            NEXT_PARITY <= CURRENT_PARITY xor
                           SERIAL_IN;
          end if;
        when PARITY_ERROR_DETECTED =>
          PARITY_ERROR <= '1';
        when ALLOW_READ =>
          NEXT_STATE <= WAIT_FOR_START;
      end case;
    end if;
  end process;

  SYNCH: process
    -- This process remembers the stored values
    --    across clock cycles
  begin
    wait until CLOCK'event and CLOCK = '1';
    CURRENT_STATE <= NEXT_STATE;
    CURRENT_HIGH_BIT <= NEXT_HIGH_BIT;
    CURRENT_PARITY <= NEXT_PARITY;
    CURRENT_PARALLEL_OUT <= NEXT_PARALLEL_OUT;
  end process;

  PARALLEL_OUT <= CURRENT_PARALLEL_OUT;

end BEHAVIOR;
```

Note:

The synthesized schematic for the shifter implementation is much simpler than that of the previous count implementation in Example 180. It is simpler because the shifter algorithm is inherently easier to implement.

With the count algorithm, each of the flip-flops holding the PARALLEL_OUT bits needed logic that decoded the value stored in the BIT_POSITION flip-flops to see when to route in the value of SERIAL_IN. Also, the BIT_POSITION flip-flops needed an incrementer to compute their next value.

In contrast, the shifter algorithm requires neither an incrementer nor flip-flops to hold BIT_POSITION. Additionally, the logic in front of most PARALLEL_OUT bits needs to read- only the value of the previous flip-flop or ’0’. The value depends on whether bits are currently being read. In the shifter algorithm, the SERIAL_IN port needs to be connected only to the least significant bit (number 7) of the PARALLEL_OUT flip-flops.

These two implementations illustrate the importance of designing efficient algorithms. Both work properly, but the shifter algorithm produces a faster, more area-efficient design.

Programmable Logic Arrays

This example shows a way to build programmable logic arrays (PLAs) in VHDL. The PLA function uses an input lookup vector as an index into a constant PLA table and then returns the output vector specified by the PLA.

The PLA table is an array of PLA rows, where each row is an array of PLA elements. Each element is either a one, a zero, a minus, or a space (’1’, ’0’, ’–’, or ’ ’). The table is split between an input plane and an output plane. The input plane is specified by 0s, 1s, and minuses. The output plane is specified by 0s and 1s. The two planes’ values are separated by a space.

In the PLA function, the output vector is first initialized to be all 0s. When the input vector matches an input plane in a row of the PLA table, the 1s in the output plane are assigned to the corresponding bits in the output vector. A match is determined as follows:

- If a 0 or 1 is in the input plane, the input vector must have the same value in the same position.
- If a minus is in the input plane, it matches any input vector value at that position.

The generic PLA table types and the PLA function are defined in a package named LOCAL. An entity PLA_VHDL that uses LOCAL needs only to specify its PLA table as a constant, then call the PLA function.

The PLA function does not explicitly depend on the size of the PLA. To change the size of the PLA, change the initialization of the TABLE constant and the initialization of the constants INPUT_COUNT, OUTPUT_COUNT, and ROW_COUNT. In Example 182, these constants are initialized to a PLA equivalent to the ROM shown previously in the example from Read-Only Memory. Accordingly, the synthesized schematic is the same as that of the ROM, with one difference: in the example from Read-Only Memory, the DATA output port range is 1 to 5; in Example 182, the OUT_VECTOR output port range is 4 down to 0.

Example 182 shows the capabilities of VHDL. It is more efficient to define the PLA directly by using the PLA input format. See the *Synopsys Timing Constraints and Optimization User Guide* for more information about the PLA input format.

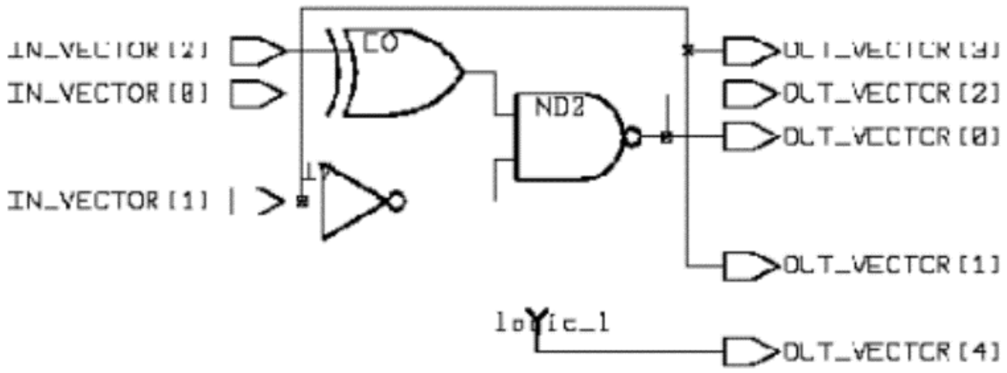
Example 182 Programmable Logic Array

```
package LOCAL is
  constant INPUT_COUNT: INTEGER := 3;
  constant OUTPUT_COUNT: INTEGER := 5;
  constant ROW_COUNT: INTEGER := 6;
  constant ROW_SIZE: INTEGER := INPUT_COUNT +
                               OUTPUT_COUNT + 1;
  type PLA_ELEMENT is ('1', '0', '-', ' ');
  type PLA_VECTOR is
      array (INTEGER range <>) of PLA_ELEMENT;
  subtype PLA_ROW is
```

```
        PLA_VECTOR(ROW_SIZE - 1 downto 0);
    subtype PLA_OUTPUT is
        PLA_VECTOR(OUTPUT_COUNT - 1 downto 0);
    type PLA_TABLE is
        array(ROW_COUNT - 1 downto 0) of PLA_ROW;

    function PLA(IN_VECTOR: BIT_VECTOR;
                 TABLE: PLA_TABLE)
        return BIT_VECTOR;
  end LOCAL;

  package body LOCAL is
    function PLA(IN_VECTOR: BIT_VECTOR;
                 TABLE: PLA_TABLE)
        return BIT_VECTOR is
      subtype RESULT_TYPE is
          BIT_VECTOR(OUTPUT_COUNT - 1 downto 0);
      variable RESULT: RESULT_TYPE;
      variable ROW: PLA_ROW;
      variable MATCH: BOOLEAN;
      variable IN_POS: INTEGER;

    begin
      RESULT := RESULT_TYPE'(others => BIT'( '0' ));
      for I in TABLE'range loop
        ROW := TABLE(I);

        MATCH := TRUE;
        IN_POS := IN_VECTOR'left;

        -- Check for match in input plane
        for J in ROW_SIZE - 1 downto OUTPUT_COUNT loop
          if(ROW(J) = PLA_ELEMENT'( '1' )) then
            MATCH := MATCH and
                     (IN_VECTOR(IN_POS) = BIT'( '1' ));
          elsif(ROW(J) = PLA_ELEMENT'( '0' )) then
            MATCH := MATCH and
                     (IN_VECTOR(IN_POS) = BIT'( '0' ));
          else
            null;     -- Must be minus ("don't care")
          end if;
          IN_POS := IN_POS - 1;
        end loop;

        -- Set output plane
        if(MATCH) then
          for J in RESULT'range loop
            if(ROW(J) = PLA_ELEMENT'( '1' )) then
              RESULT(J) := BIT'( '1' );
            end if;
          end loop;
        end if;
      end loop;
```

```
        return(RESULT);
      end;
    end LOCAL;

    use WORK.LOCAL.all;
    entity PLA_VHDL is
      port(IN_VECTOR: BIT_VECTOR(2 downto 0);
           OUT_VECTOR: out BIT_VECTOR(4 downto 0));
    end;

    architecture BEHAVIOR of PLA_VHDL is
      constant TABLE: PLA_TABLE := PLA_TABLE"(
           PLA_ROW'("--- 10000"),
           PLA_ROW'("-1- 01000"),
           PLA_ROW'("0-0 00101"),
           PLA_ROW'("-1- 00101"),
           PLA_ROW'("1-1 00101"),
           PLA_ROW'("-1- 00010"));

    begin
      OUT_VECTOR <= PLA(IN_VECTOR, TABLE);
    end BEHAVIOR;
```

Figure 23 Synthesized Circuit of the PLA



B

Predefined Libraries

This appendix describes the following packages that are included in an HDL Compiler installation:

- std_logic_1164
- std_logic_arith
- numeric_std
- std_logic_misc
- Standard Package
- Synopsys Package—ATTRIBUTES

std_logic_1164

The std_logic_1164 package is typically installed in the $SYNOPSYS/packages/IEEE/src/std_logic_1164.vhd subdirectory of the Synopsys root directory. The std_logic_1164.vhd file has been updated with Synopsys synthesis directives, such as the `built_in` HDL Compiler directive described below. HDL Compiler automatically uses the `built_in` compiler directives to improve performance. You can also write your own `built_in` compiler directives.

built_in HDL Compiler Directives

The Synopsys IEEE std_logic_1164 package contains the following `built_in` functions that enable HDL Compiler to quickly and easily interpret your code:

- SYN_AND
- SYN_OR
- SYN_NAND
- SYN_NOR
- SYN_XOR

- SYN_XNOR
- SYN_NOT
- SYN_BUF

These functions are automatically enabled by HDL Compiler for the respective operators in your code; you do not have to use them. If you create your own `built_in` functions, label them with a `built_in` compiler directive, as shown in Example 183. (HDL Compiler interprets a comment as a directive if the first word of the comment is “pragma”.) When you use a `built_in` compiler directive, HDL Compiler parses but ignores the body of the function and directly substitutes the appropriate logic for the function.

Example 183 shows the XOR `built_in` function.

Example 183 XOR built_in Function

```
function "XOR" (L, R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR is
  -- pragma built_in SYN_XOR
    begin
        if (L = '1') xor (R = '1') then
            return '1';
        else
            return '0';
        end if;
end "XOR";
```

Example 184 shows the SYN_AND built_in function.

Example 184 SYN_AND built_in Function

```
function "AND" (L, R: STD_LOGIC_VECTOR) return
STD_LOGIC_VECTOR is
  -- pragma built_in SYN_AND
    variable MY_L: STD_LOGIC_VECTOR (L'length-1 downto 0);
    variable MY_R: STD_LOGIC_VECTOR (L'length-1 downto 0);
    variable RESULT: STD_LOGIC_VECTOR (L'length-1 downto 0);
begin
    assert L'length = R'length;
    MY_L := L;
    MY_R := R;
    for i in RESULT'range loop
        if (MY_L(i) = '1') and (MY_R(i) = '1') then
            RESULT(i) := '1';
        else
            RESULT(i) := '0';
        end if;
    end loop;
    return RESULT;
end "AND";
```

Example 185 shows the SYN_NOT built_in function.

Example 185 SYN_NOT built_in Function

```
function "NOT" (L: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR is
  -- pragma built_in SYN_NOT
     variable MY_L: STD_LOGIC_VECTOR (L'length-1 downto 0);
     variable RESULT: STD_LOGIC_VECTOR (L'length-1 downto 0);
begin
    MY_L := L;
    for i in result'range loop
        if (MY_L(i) = '0' or MY_L(i) = 'L') then
            RESULT(i) := '1';
        elsif (MY_L(i) = '1' or MY_L(i) = 'H') then
            RESULT(i) := '0';
        else
            RESULT(i) := 'X';
        end if;
    end loop;
    return RESULT;
end "NOT";
```

Example 186 shows the SYN_FEED_THRU built_in function which performs type conversion between unrelated types. The synthesized logic from SYN_FEED_THRU wires the single input of a function to the return value.

Example 186 SYN_FEED_THRU built_in Function

```
type COLOR is (RED, GREEN, BLUE);
attribute ENUM_ENCODING : STRING;
attribute ENUM_ENCODING of COLOR : type is "01 10 11";
...

function COLOR_TO_BV (L: COLOR) return BIT_VECTOR is
  -- pragma built_in SYN_FEED_THRU
begin
    case L is
       when RED   => return "01";
       when GREEN => return "10";
       when BLUE  => return "11";
    end case;
end COLOR_TO_BV;
```

std_logic_arith

This section contains the following subsections:

- std_logic_arith Package Overview
- Modifying the std_logic_arith Package
- std_logic_arith Data Types
- UNSIGNED
- SIGNED

std_logic_arith Package Overview

The std_logic_arith package is typically installed in the $SYNOPSYS/packages/IEEE/src/std_logic_arith.vhd subdirectory of the Synopsys root directory. To use this package in a VHDL source file, include the following lines at the beginning of the source file:

```
library IEEE;
use IEEE.std_logic_arith.all;
```

Functions defined in the std_logic_arith package provide conversion to and from the predefined VHDL data type INTEGER, arithmetic, comparison, and BOOLEAN operations. This package lets you perform arithmetic operations and numeric comparisons on array data types. The package defines some arithmetic operators (+, -, *, ABS) and the relational operators (<, >, <=, >=, =, /=). (IEEE VHDL does not define arithmetic operators for arrays and defines the comparison operators in a manner inconsistent with an arithmetic interpretation of array values.)

The package also defines two major data types of its own: UNSIGNED and SIGNED (see std_logic_arith Data Types for details). The std_logic_arith package is legal VHDL; you can use it for both synthesis and simulation.

You can configure the std_logic_arith package to work on any array of single-bit types. You encode single-bit types in 1 bit with the ENUM_ENCODING attribute.

You can make the vector type (for example, std_logic_vector) synonymous with either SIGNED or UNSIGNED. This way, if you plan to use mostly UNSIGNED numbers, you do not need to convert your vector type to call UNSIGNED functions. The disadvantage of making your vector type synonymous with either UNSIGNED or SIGNED is that it causes redefinition of the standard VHDL comparison operators (=, /=, <, >, <=, >=).

Table 19 shows that the standard comparison functions for BIT_VECTOR do not match the SIGNED and UNSIGNED functions.

Table 19 UNSIGNED, SIGNED, and BIT_VECTOR Comparison Functions

ARG1	op	ARG2	UNSIGNED	SIGNED	BIT_VECTOR
"000"	=	"000"	true	true	true

Table 19 UNSIGNED, SIGNED, and BIT_VECTOR Comparison Functions (Continued)

ARG1	op	ARG2	UNSIGNED	SIGNED	BIT_VECTOR
"00"	=	"000"	true	true	false
"100"	=	"0100"	true	false	false
"000"	<	"000"	false	false	false
"00"	<	"000"	false	false	true
"100"	<	"0100"	false	true	false

Modifying the std_logic_arith Package

The std_logic_arith package is written in standard VHDL. You can modify or add to it. When you change the content, you must reanalyze the package.

For example, to convert a vector of multivalued logic to an INTEGER, you can write the function shown in Example 187. This MVL_TO_INTEGER function returns the integer value corresponding to the vector when the vector is interpreted as an unsigned (natural) number. If unknown values are in the vector, the return value is –1.

Example 187 New Function Based on a std_logic_arith Package Function

```
library IEEE;
use IEEE.std_logic_1164.all;

function MVL_TO_INTEGER(ARG : MVL_VECTOR)
  return INTEGER is
  -- pragma built_in SYN_FEED_THRU
  variable uns: UNSIGNED (ARG'range);
begin
    for i in ARG'range loop
        case ARG(i) is
            when '0' | 'L' => uns(i) := '0';
            when '1' | 'H' => uns(i) := '1';
            when others    => return -1;
        end case;
    end loop;
    return CONV_INTEGER(uns);
end;
```

Note the use of the CONV_INTEGER function in Example 187.

HDL Compiler performs almost all synthesis directly from the VHDL descriptions. However, several functions are hard-wired for efficiency. They can be identified by the following comment in their declarations:

```
-- pragma built_in
```

This statement marks functions as special, causing the body of the function to be ignored. Modifying the body does not change the synthesized logic unless you remove the built_in comment. If you want new functionality, write it by using the built_in function; this is more efficient than removing the built_in function and modifying the body of the function.

std_logic_arith Data Types

The std_logic_arith package defines two data types: UNSIGNED and SIGNED.

```
type UNSIGNED is array (natural range <>) of std_logic;
type SIGNED is array (natural range <>) of std_logic;
```

These data types are similar to the predefined VHDL type BIT_VECTOR, but the std_logic_arith package defines the interpretation of variables and signals of these types as numeric values.

UNSIGNED

The UNSIGNED data type represents an unsigned numeric value. HDL Compiler interprets the number as a binary representation, with the farthest-left bit being most significant. For example, the decimal number 8 can be represented as

```
UNSIGNED'("1000")
```

When you declare variables or signals of type UNSIGNED, a larger vector holds a larger number. A 4-bit variable holds values up to decimal 15, an 8-bit variable holds values up to 255, and so on. By definition, negative numbers cannot be represented in an UNSIGNED variable. Zero is the smallest value that can be represented.

Example 188 illustrates some UNSIGNED declarations. The most significant bit is the farthest-left array bound, rather than the high- or low-range value.

Example 188 UNSIGNED Declarations

```
variable VAR: UNSIGNED (1 to 10);
  -- 10-bit number
  -- VAR(VAR'left) = VAR(1) is the most significant bit

signal SIG: UNSIGNED (5 downto 0);
  -- 6-bit number
  -- SIG(SIG'left) = SIG(5) is the most significant bit
```

SIGNED

The SIGNED data type represents a signed numeric value. HDL Compiler interprets the number as a 2's-complement binary representation, with the farthest-left bit as the sign bit. For example, you can represent decimal 5 and –5 as

```
SIGNED'("0101")  -- represents +5
SIGNED'("1011")  -- represents -5
```

When you declare SIGNED variables or signals, a larger vector holds a larger number. A 4-bit variable holds values from –8 to 7; an 8-bit variable holds values from –128 to 127. A SIGNED value cannot hold as large a value as an UNSIGNED value with the same bit-width.

Example 189 shows some SIGNED declarations. The sign bit is the farthest-left bit, rather than the highest or lowest.

Example 189 SIGNED Declarations

```
variable S_VAR: SIGNED (1 to 10);
  -- 10-bit number
  -- S_VAR(S_VAR'left) = S_VAR(1) is the sign bit

signal S_SIG: SIGNED (5 downto 0);
  -- 6-bit number
  -- S_SIG(S_SIG'left) = S_SIG(5) is the sign bit
```

Conversion Functions

The std_logic_arith package provides three sets of functions to convert values between its UNSIGNED and SIGNED types and the predefined type INTEGER. This package also provides the std_logic_vector. Example 190 shows the declarations of these conversion functions, with BIT and BIT_VECTOR types.

Example 190 Conversion Functions

```
subtype SMALL_INT is INTEGER range 0 to 1;
function CONV_INTEGER(ARG: INTEGER)  return INTEGER;
function CONV_INTEGER(ARG: UNSIGNED) return INTEGER;
function CONV_INTEGER(ARG: SIGNED)   return INTEGER;
function CONV_INTEGER(ARG: STD_ULOGIC) return SMALL_INT;

function CONV_UNSIGNED(ARG: INTEGER;
                       SIZE: INTEGER) return UNSIGNED;
function CONV_UNSIGNED(ARG: UNSIGNED;
                       SIZE: INTEGER) return UNSIGNED;
function CONV_UNSIGNED(ARG: SIGNED;
                       SIZE: INTEGER) return UNSIGNED;
function CONV_UNSIGNED(ARG: STD_ULOGIC;
                       SIZE: INTEGER) return UNSIGNED;
```

```
function CONV_SIGNED(ARG: INTEGER;
                     SIZE: INTEGER)   return SIGNED;
function CONV_SIGNED(ARG: UNSIGNED;
                     SIZE: INTEGER)   return SIGNED;
function CONV_SIGNED(ARG: SIGNED;
                     SIZE: INTEGER)   return SIGNED;
function CONV_SIGNED(ARG: STD_ULOGIC;
                     SIZE: INTEGER)   return SIGNED;

function CONV_STD_LOGIC_VECTOR(ARG: INTEGER;
                     SIZE: INTEGER)   return STD_LOGIC_VECTOR;
function CONV_STD_LOGIC_VECTOR(ARG: UNSIGNED;
                     SIZE: INTEGER)   return STD_LOGIC_VECTOR;
function CONV_STD_LOGIC_VECTOR(ARG: SIGNED;
                     SIZE: INTEGER)   return STD_LOGIC_VECTOR;
function CONV_STD_LOGIC_VECTOR(ARG: STD_ULOGIC;
                     SIZE: INTEGER)   return STD_LOGIC_VECTOR;
```

There are four versions of each conversion function. The VHDL operator overloading mechanism determines the correct version from the function call's argument types.

The CONV_INTEGER functions convert an argument of type INTEGER, UNSIGNED, SIGNED, or STD_ULOGIC to an INTEGER return value. The CONV_UNSIGNED and CONV_SIGNED functions convert an argument of type INTEGER, UNSIGNED, SIGNED, or STD_ULOGIC to an UNSIGNED or SIGNED return value whose bit-width is SIZE.

The CONV_INTEGER functions have a limitation on the size of operands. VHDL defines INTEGER values as being between –2147483647 and 2147483647. This range corresponds to a 31-bit UNSIGNED value or a 32-bit SIGNED value. You cannot convert an argument outside this range to an INTEGER.

The CONV_UNSIGNED and CONV_SIGNED functions each require two operands. The first operand is the value converted. The second operand is an INTEGER that specifies the expected size of the converted result. For example, the following function call returns a 10-bit UNSIGNED value representing the value in sig.

```
ten_unsigned_bits := CONV_UNSIGNED(sig, 10);
```

If the value passed to CONV_UNSIGNED or CONV_SIGNED is smaller than the expected bit-width (such as representing the value 2 in a 24-bit number), the value is bit-extended appropriately. HDL Compiler places 0s in the more significant (left) bits for an UNSIGNED return value, and it uses sign extension for a SIGNED return value.

You can use the conversion functions to extend a number's bit-width even if conversion is not required. For example,

```
CONV_SIGNED(SIGNED'("110"), 8) -> "11111110"
```

An UNSIGNED or SIGNED return value is truncated when its bit-width is too small to hold the ARG value. For example,

```
CONV_SIGNED(UNSIGNED'("1101010"), 3) -> "010"
```

Arithmetic Functions

The std_logic_arith package provides arithmetic functions for use with combinations of the Synopsys UNSIGNED and SIGNED data types and the predefined types STD_ULOGIC and INTEGER. These functions produce adders and subtracters.

There are two sets of arithmetic functions: binary functions having two arguments, such as A+B or A*B, and unary functions having one argument, such as –A. Example 191 and Example 192 show the declarations for these functions.

Example 191 Binary Arithmetic Functions

```
function "+"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
function "+"(L: SIGNED;   R: SIGNED)   return SIGNED;
function "+"(L: UNSIGNED; R: SIGNED)   return SIGNED;
function "+"(L: SIGNED;   R: UNSIGNED) return SIGNED;
function "+"(L: UNSIGNED; R: INTEGER)  return UNSIGNED;
function "+"(L: INTEGER;  R: UNSIGNED) return UNSIGNED;
function "+"(L: SIGNED;   R: INTEGER)  return SIGNED;
function "+"(L: INTEGER;  R: SIGNED)   return SIGNED;
function "+"(L: UNSIGNED; R: STD_ULOGIC) return UNSIGNED;
function "+"(L: STD_ULOGIC; R: UNSIGNED) return UNSIGNED;
function "+"(L: SIGNED;   R: STD_ULOGIC) return SIGNED;
function "+"(L: STD_ULOGIC; R: SIGNED)   return SIGNED;

function "+"(L: UNSIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "+"(L: UNSIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: UNSIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
function "+"(L: INTEGER; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
function "+"(L: INTEGER; R: SIGNED) return STD_LOGIC_VECTOR;
function "+"(L: UNSIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
function "+"(L: STD_ULOGIC; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
function "+"(L: STD_ULOGIC; R: SIGNED) return STD_LOGIC_VECTOR;
function "-"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
function "-"(L: SIGNED;   R: SIGNED)   return SIGNED;
function "-"(L: UNSIGNED; R: SIGNED)   return SIGNED;
function "-"(L: SIGNED;   R: UNSIGNED) return SIGNED;
function "-"(L: UNSIGNED; R: INTEGER)  return UNSIGNED;
function "-"(L: INTEGER;  R: UNSIGNED) return UNSIGNED;
function "-"(L: SIGNED;   R: INTEGER)  return SIGNED;
function "-"(L: INTEGER;  R: SIGNED)   return SIGNED;
function "-"(L: UNSIGNED; R: STD_ULOGIC) return UNSIGNED;
function "-"(L: STD_ULOGIC; R: UNSIGNED) return UNSIGNED;
function "-"(L: SIGNED;   R: STD_ULOGIC) return SIGNED;
function "-"(L: STD_ULOGIC; R: SIGNED)   return SIGNED;

function "-"(L: UNSIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "-"(L: SIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "-"(L: UNSIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "-"(L: SIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "-"(L: UNSIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
function "-"(L: INTEGER; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "-"(L: SIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
function "-"(L: INTEGER; R: SIGNED) return STD_LOGIC_VECTOR;
function "-"(L: UNSIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
function "-"(L: STD_ULOGIC; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "-"(L: SIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
function "-"(L: STD_ULOGIC; R: SIGNED) return STD_LOGIC_VECTOR;

function "*"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
function "*"(L: SIGNED;   R: SIGNED)   return SIGNED;
function "*"(L: SIGNED;   R: UNSIGNED) return SIGNED;
function "*"(L: UNSIGNED; R: SIGNED)   return SIGNED;
```

Example 192 Unary Arithmetic Functions

```
function "+"(L: UNSIGNED) return UNSIGNED;
function "+"(L: SIGNED)   return SIGNED;
function "-"(L: SIGNED)   return SIGNED;
function "ABS"(L: SIGNED) return SIGNED;
```

The binary and unary arithmetic functions in Example 191 and Example 192 determine the width of their return values, as follows:

1. When only one UNSIGNED or SIGNED argument is present, the width of the return value is the same as the width of that argument.
2. When both arguments are either UNSIGNED or SIGNED, the width of the return value is the larger of the two argument widths. An exception is that when an UNSIGNED number is added to or subtracted from a SIGNED number that is the same size or smaller, the return value is a SIGNED number 1 bit wider than the UNSIGNED argument. This size guarantees that the return value is large enough to hold any (positive) value of the UNSIGNED argument.

Table 20 illustrates the number of bits returned by addition (+) and subtraction (–).

```
signal U4: UNSIGNED (3 downto 0);
signal U8: UNSIGNED (7 downto 0);
signal S4: SIGNED (3 downto 0);
signal S8: SIGNED (7 downto 0);
```

Table 20 Number of Bits Returned by Addition and Subtraction

+ or -	U4	U8	S4	S8
U4	4	8	5	8
U8	8	8	9	9
S4	5	9	4	8
S8	8	9	8	8

In some circumstances, you might need to obtain a carry-out bit from the addition or subtraction operation. To do this, extend the larger operand by 1 bit. The high bit of the return value is the carry, as illustrated in Example 193.

Example 193 Using the Carry-Out Bit

```
process
    variable a, b, sum: UNSIGNED (7 downto 0);
    variable temp: UNSIGNED (8 downto 0);
    variable carry: BIT;
begin
    temp  := CONV_UNSIGNED(a,9) + b;
    sum   := temp(7 downto 0);
    carry := temp(8);
end process;
```

Comparison Functions

The std_logic_arith package provides functions for comparing UNSIGNED and SIGNED data types with each other and with the predefined type INTEGER. HDL Compiler compares the numeric values of the arguments, returning a BOOLEAN value. For example, the following expression evaluates true.

```
UNSIGNED'("001") > SIGNED'("111")
```

The std_logic_arith comparison functions are similar to the built-in VHDL comparison functions. The only difference is that the std_logic_arith functions accommodate signed numbers and varying bit-widths. The predefined VHDL comparison functions perform bitwise comparisons and do not have the correct semantics for comparing numeric values (see Ordering of Enumerated Types Using the ENUM_ENCODING Attribute).

These functions produce comparators. The function declarations are listed in two groups: ordering operators ("<", "<=", ">", ">="), shown in Example 194, and equality functions ("=", "/="), shown in Example 195.

Example 194 Ordering Functions

```
function "<"(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function "<"(L: SIGNED;     R: SIGNED)    return BOOLEAN;
function "<"(L: UNSIGNED;     R: SIGNED)    return BOOLEAN;
function "<"(L: SIGNED;     R: UNSIGNED)    return BOOLEAN;
function "<"(L: UNSIGNED;     R: INTEGER)    return BOOLEAN;
function "<"(L: INTEGER;     R: UNSIGNED) return BOOLEAN;
function "<"(L: SIGNED;     R: INTEGER)    return BOOLEAN;
function "<"(L: INTEGER;     R: SIGNED)    return BOOLEAN;

function "<="(L: UNSIGNED;     R: UNSIGNED) return BOOLEAN;
function "<="(L: SIGNED;     R: SIGNED)    return BOOLEAN;
function "<="(L: UNSIGNED;     R: SIGNED)    return BOOLEAN;
function "<="(L: SIGNED;     R: UNSIGNED)    return BOOLEAN;
function "<="(L: UNSIGNED;       R: INTEGER)    return BOOLEAN;
function "<="(L: INTEGER;       R: UNSIGNED)    return BOOLEAN;
function "<="(L: SIGNED;       R: INTEGER)    return BOOLEAN;
function "<="(L: INTEGER;       R: SIGNED)    return BOOLEAN;
function ">"(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function ">"(L: SIGNED;       R: SIGNED)    return BOOLEAN;
function ">"(L: UNSIGNED;       R: SIGNED)    return BOOLEAN;
function ">"(L: SIGNED;      R: UNSIGNED)    return BOOLEAN;
function ">"(L: UNSIGNED;       R: INTEGER)    return BOOLEAN;
function ">"(L: INTEGER;       R: UNSIGNED) return BOOLEAN;
function ">"(L: SIGNED;       R: INTEGER)    return BOOLEAN;
function ">"(L: INTEGER;       R: SIGNED)    return BOOLEAN;

function ">="(L: UNSIGNED;      R: UNSIGNED) return BOOLEAN;
function ">="(L: SIGNED;       R: SIGNED)    return BOOLEAN;
function ">="(L: UNSIGNED;      R: SIGNED)    return BOOLEAN;
function ">="(L: SIGNED;       R: UNSIGNED)    return BOOLEAN;
function ">="(L: UNSIGNED;       R: INTEGER)    return BOOLEAN;
function ">="(L: INTEGER;       R: UNSIGNED)    return BOOLEAN;
function ">="(L: SIGNED;       R: INTEGER)    return BOOLEAN;
function ">="(L: INTEGER;       R: SIGNED)    return BOOLEAN;
```

Example 195 Equality Functions

```
function "="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function "="(L: SIGNED;   R: SIGNED)   return BOOLEAN;
function "="(L: UNSIGNED; R: SIGNED)   return BOOLEAN;
function "="(L: SIGNED;   R: UNSIGNED) return BOOLEAN;
function "="(L: UNSIGNED; R: INTEGER)  return BOOLEAN;
function "="(L: INTEGER;  R: UNSIGNED) return BOOLEAN;
function "="(L: SIGNED;   R: INTEGER)  return BOOLEAN;
function "="(L: INTEGER;  R: SIGNED)   return BOOLEAN;

function "/="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function "/="(L: SIGNED;   R: SIGNED)   return BOOLEAN;
function "/="(L: UNSIGNED; R: SIGNED)   return BOOLEAN;
function "/="(L: SIGNED;   R: UNSIGNED) return BOOLEAN;
function "/="(L: UNSIGNED; R: INTEGER)  return BOOLEAN;
function "/="(L: INTEGER;  R: UNSIGNED) return BOOLEAN;
function "/="(L: SIGNED;   R: INTEGER)  return BOOLEAN;
function "/="(L: INTEGER;  R: SIGNED)   return BOOLEAN;
```

Shift Functions

The std_logic_arith package provides functions for shifting the bits in SIGNED and UNSIGNED numbers. These functions produce shifters. Example 196 shows the shift function declarations. For a list of shift and rotate operators, see Operators.

Example 196 Shift Functions

```
function SHL(ARG: UNSIGNED;
             COUNT: UNSIGNED)  return UNSIGNED;
```

```
function SHL(ARG: SIGNED;
             COUNT: UNSIGNED)  return SIGNED;

function SHR(ARG: UNSIGNED;
             COUNT: UNSIGNED)  return UNSIGNED;
function SHR(ARG: SIGNED;
             COUNT: UNSIGNED)  return SIGNED;
```

The SHL function shifts the bits of its argument ARG left by COUNT bits. SHR shifts the bits of its argument ARG right by COUNT bits.

The SHL functions work the same for both UNSIGNED and SIGNED values of ARG, shifting in zero bits as necessary. The SHR functions treat UNSIGNED and SIGNED values differently. If ARG is an UNSIGNED number, vacated bits are filled with 0s; if ARG is a SIGNED number, the vacated bits are copied from the ARG sign bit.

Example 197 shows some shift function calls and their return values.

Example 197 Shift Operations

```
variable U1, U2: UNSIGNED (7 downto 0);
variable S1, S2: SIGNED   (7 downto 0);
variable COUNT:  UNSIGNED (1 downto 0);
. . .
U1 := "01101011";
U2 := "11101011";

S1 := "01101011";
S2 := "11101011";

COUNT := CONV_UNSIGNED(ARG => 3, SIZE => 2);
. . .
SHL(U1, COUNT) = "01011000"
SHL(S1, COUNT) = "01011000"
SHL(U2, COUNT) = "01011000"
SHL(S2, COUNT) = "01011000"

SHR(U1, COUNT) = "00001101"
SHR(S1, COUNT) = "00001101"
SHR(U2, COUNT) = "00011101"
SHR(S2, COUNT) = "11111101"
```

Multiplication Using Shifts

You can use shift operations for simple multiplication and division of UNSIGNED numbers if you are multiplying or dividing by a power of 2.

For example, to divide the following UNSIGNED variable U by 4, use this syntax:

```
variable U: UNSIGNED (7 downto 0) := "11010101";
variable quarter_U: UNSIGNED (5 downto 0);
quarter_U := SHR(U, "01");
```

numeric_std

This section describes HDL Compiler support for the numeric_std, the IEEE Standard VHDL Synthesis Package, which defines numeric types and arithmetic functions.

This section contains the following:

Note:

The numeric_std package and the std_logic_arith package have overlapping operations. Use of these two packages simultaneously during analysis could cause type mismatches.

Unsupported Constructs and Operators

HDL Compiler does not support the following numeric_std package component:

- TO_01 function as a simulation construct

Using the numeric_std Package

The numeric_std package is typically installed in the Synopsys root directory. Access it with the following statement in your VHDL code:

```
library IEEE;
use IEEE.numeric_std.all;
```

numeric_std Data Types

The numeric_std package defines the following two data types in the same way that the std_logic_arith package does:

- USIGNED

  ```
  type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
  ```

  See UNSIGNED for more information.

- SIGNED

  ```
  type SIGNED is array (NATURAL range <>) of STD_LOGIC;
  ```

  See SIGNED for more information.

Conversion Functions

The numeric_std package provides functions to convert values between its UNSIGNED and SIGNED types. Table 21 shows the declarations of these conversion functions.

Table 21 numeric_std Conversion Functions

	Parameters		
Operator	**Arg**	**Size**	**Return type**
TO_INTEGER	UNSIGNED	--	NATURAL
TO_INTEGER	SIGNED	--	INTEGER
TO_UNSIGNED	INTEGER	NATURAL	UNSIGNED
TO_SIGNED	INTEGER	NATURAL	SIGNED

TO_INTEGER, TO_SIGNED, and TO_UNSIGNED are similar to CONV_INTEGER, CONV_SIGNED, and CONV_UNSIGNED in std_logic_arith (see Conversion Functions).

Resize Functions

The resize function numeric_std supports is shown in the declarations in Table 22.

Table 22 numeric_std Resize Functions

	Parameters		
Operator	**Arg**	**Size**	**Return type**
RESIZE	NATURAL	NATURAL	SIGNED
RESIZE	NATURAL	NATURAL	UNSIGNED

Arithmetic Functions

The numeric_std package provides arithmetic functions for use with combinations of Synopsys UNSIGNED and SIGNED data types and the predefined types STD_ULOGIC and INTEGER. These functions produce adders and subtracters.

There are two sets of arithmetic functions, which the numeric_std package defines in the same way the std_logic_arith package does (see Arithmetic Functions for more information):

- Binary functions having two arguments, such as

  A+B

  A*B

  Table 23 shows the declarations for these functions.

- Unary functions having one argument, such as

  –A

  abs A

Table 23 numeric_std Binary Arithmetic Functions

	Parameters		
Operator	**L**	**R**	**Return type**
+	UNSIGNED	UNSIGNED	UNSIGNED
+	SIGNED	SIGNED	SIGNED
+	UNSIGNED	NATURAL	UNSIGNED
+	NATURAL	UNSIGNED	UNSIGNED
+	INTEGER	SIGNED	SIGNED

	Parameters		
Operator	**L**	**R**	**Return type**
+	SIGNED	INTEGER	SIGNED
–	UNSIGNED	UNSIGNED	UNSIGNED
–	SIGNED	SIGNED	SIGNED
–	UNSIGNED	NATURAL	UNSIGNED
–	NATURAL	UNSIGNED	UNSIGNED
–	SIGNED	INTEGER	SIGNED
–	INTEGER	SIGNED	SIGNED
*	UNSIGNED	UNSIGNED	UNSIGNED
*	SIGNED	SIGNED	SIGNED
*	UNSIGNED	NATURAL	UNSIGNED
*	NATURAL	UNSIGNED	UNSIGNED
*	SIGNED	INTEGER	SIGNED
*	INTEGER	SIGNED	SIGNED

Table 24 shows the declarations for unary functions.

Table 24 numeric_std Unary Arithmetic Functions

Operator	Arg	Return type
abs	SIGNED	SIGNED
–	SIGNED	SIGNED

Comparison Functions

The numeric_std package provides functions to compare UNSIGNED and SIGNED data types with each other and with the predefined type INTEGER. HDL Compiler compares the numeric values of the arguments and returns a BOOLEAN value.

These functions produce comparators. The function declarations are listed in two groups:

- Ordering functions (<, <=, >, >=)
- Equality functions (=, /=)

Table 25 shows the ordering functions.

Table 25 numeric_std Ordering Functions

	Parameters		
Operator	**L**	**R**	**Return type**
>	UNSIGNED	UNSIGNED	BOOLEAN
>	SIGNED	SIGNED	BOOLEAN
>	NATURAL	UNSIGNED	BOOLEAN
>	INTEGER	SIGNED	BOOLEAN
>	UNSIGNED	NATURAL	BOOLEAN
>	SIGNED	INTEGER	BOOLEAN
<	UNSIGNED	UNSIGNED	BOOLEAN
<	SIGNED	SIGNED	BOOLEAN
<	NATURAL	UNSIGNED	BOOLEAN
<	INTEGER	SIGNED	BOOLEAN
<	UNSIGNED	NATURAL	BOOLEAN
<	SIGNED	INTEGER	BOOLEAN
<=	UNSIGNED	UNSIGNED	BOOLEAN
<=	SIGNED	SIGNED	BOOLEAN
<=	NATURAL	UNSIGNED	BOOLEAN
<=	INTEGER	SIGNED	BOOLEAN
<=	UNSIGNED	NATURAL	BOOLEAN
<=	SIGNED	INTEGER	BOOLEAN
>=	UNSIGNED	UNSIGNED	BOOLEAN
>=	SIGNED	SIGNED	BOOLEAN

Table 25 numeric_std Ordering Functions (Continued)

	Parameters		
Operator	**L**	**R**	**Return type**
>=	NATURAL	UNSIGNED	BOOLEAN
>=	INTEGER	SIGNED	BOOLEAN
>=	UNSIGNED	NATURAL	BOOLEAN
>=	SIGNED	INTEGER	BOOLEAN

Table 26 shows the equality functions.

Table 26 numeric_std Equality Functions

	Parameters		
Operator	**L**	**R**	**Return type**
=	UNSIGNED	UNSIGNED	BOOLEAN
=	SIGNED	SIGNED	BOOLEAN
=	NATURAL	UNSIGNED	BOOLEAN
=	INTEGER	SIGNED	BOOLEAN
=	UNSIGNED	NATURAL	BOOLEAN
=	SIGNED	INTEGER	BOOLEAN
/=	UNSIGNED	UNSIGNED	BOOLEAN
/=	SIGNED	SIGNED	BOOLEAN
/=	NATURAL	UNSIGNED	BOOLEAN
/=	INTEGER	SIGNED	BOOLEAN
/=	UNSIGNED	NATURAL	BOOLEAN
/=	SIGNED	INTEGER	BOOLEAN

Defining Logical Operators Functions

The numeric_std package provides functions that define all of the logical operators: NOT, AND, OR, NAND, NOR, XOR, and XNOR. These functions work just like similar functions in std_logic_1164, except that they operate on SIGNED and UNSIGNED values rather than on STD_LOGIC_VECTOR values. Table 27 shows these function declarations.

Table 27 numeric_std Logical Operators Functions

	Parameters		
Operator	**L**	**R**	**Return type**
not	UNSIGNED	--	UNSIGNED
and	UNSIGNED	UNSIGNED	UNSIGNED
or	UNSIGNED	UNSIGNED	UNSIGNED
nand	UNSIGNED	UNSIGNED	UNSIGNED
nor	UNSIGNED	UNSIGNED	UNSIGNED
xor	UNSIGNED	UNSIGNED	UNSIGNED
xnor	UNSIGNED	UNSIGNED	UNSIGNED
not	SIGNED	--	SIGNED
and	SIGNED	SIGNED	SIGNED
or	SIGNED	SIGNED	SIGNED
nand	SIGNED	SIGNED	SIGNED
nor	SIGNED	SIGNED	SIGNED
xor	SIGNED	SIGNED	SIGNED
xnor	SIGNED	SIGNED	SIGNED

Shift and Rotate Functions

The numeric_std package provides functions for shifting the bits in UNSIGNED and SIGNED numbers. These functions produce shifters. Table 28 shows the shift function declarations.

Table 28 numeric_std Shift and Rotate Functions

	Parameters		
Operator	**Arg**	**Count**	**Return type**
SHIFT_LEFT	UNSIGNED	NATURAL	UNSIGNED
SHIFT_RIGHT	UNSIGNED	NATURAL	UNSIGNED
SHIFT_LEFT	SIGNED	NATURAL	SIGNED
SHIFT_RIGHT	SIGNED	NATURAL	SIGNED
ROTATE_LEFT	UNSIGNED	NATURAL	UNSIGNED
ROTATE_RIGHT	UNSIGNED	NATURAL	UNSIGNED
ROTATE_LEFT	SIGNED	NATURAL	SIGNED
ROTATE_RIGHT	SIGNED	NATURAL	SIGNED

The SHIFT_LEFT function shifts the bits of its argument ARG left by COUNT bits. SHIFT_RIGHT shifts the bits of its argument ARG right by COUNT bits.

The SHIFT_LEFT functions work the same for both UNSIGNED and SIGNED values of ARG, shifting in zero bits as necessary. The SHIFT_RIGHT functions treat UNSIGNED and SIGNED values differently:

- If ARG is an UNSIGNED number, vacated bits are filled with 0s.
- If ARG is a SIGNED number, the vacated bits are copied from the ARG sign bit.

The example shown in std_logic_misc shows some shift function calls and their return values.

The ROTATE_LEFT and ROTATE_RIGHT functions are similar to the shift functions. The example shown in Shift and Rotate Operators shows some rotate function declarations.

Shift and Rotate Operators

The numeric_std package provides shift operators and rotate operators, which work in the same way that shift functions and rotate functions do. The shift operators are sll, srl, sla, and sra. Table 29 shows some shift and rotate operator declarations. Example 198 includes some shift and rotate operators.

Table 29 numeric_std Shift and Rotate Operators

	Parameters		
Operator	**Arg**	**Count**	**Return type**
sll	UNSIGNED	INTEGER	UNSIGNED
sll	SIGNED	INTEGER	SIGNED
srl	UNSIGNED	INTEGER	UNSIGNED
srl	SIGNED	INTEGER	SIGNED
rol	UNSIGNED	INTEGER	UNSIGNED
rol	SIGNED	INTEGER	SIGNED
ror	UNSIGNED	INTEGER	UNSIGNED
ror	SIGNED	INTEGER	SIGNED

Example 198 Some numeric_std Shift and Rotate Functions and Shift and Rotate Operators

```
Variable U1, U2: UNSIGNED (7 downto 0);
Variable S1, S2: SIGNED (7 downto 0);
Variable COUNT: NATURAL;
...
U1 := "01101011";
U2 := "11101011";
S1 := "01101011";
S2 := "11101011";
COUNT := 3;
...
SHIFT_LEFT (U1, COUNT) = "01011000"
SHIFT_LEFT (S1, COUNT) = "01011000"
SHIFT_LEFT (U2, COUNT) = "01011000"
SHIFT_LEFT (S2, COUNT) = "01011000"

SHIFT_RIGHT (U1, COUNT) = "00001101"
SHIFT_RIGHT (S1, COUNT) = "00001101"
SHIFT_RIGHT (U2, COUNT) = "00011101"
SHIFT_RIGHT (S2, COUNT) = "11111101"

ROTATE_LEFT (U1, COUNT) = "01011011"
ROTATE_LEFT (S1, COUNT) = "01011011"
ROTATE_LEFT (U2, COUNT) = "01011111"
ROTATE_LEFT (S2, COUNT) = "01011111"

ROTATE_RIGHT (U1, COUNT) = "01101101"
ROTATE_RIGHT (S1, COUNT) = "01101101"
ROTATE_RIGHT (U2, COUNT) = "01111101"
ROTATE_RIGHT (S2, COUNT) = "01111101"

U1 sll COUNT = "01011000"
S1 sll COUNT = "01011000"
U2 sll COUNT = "01011000"
S2 sll COUNT = "01011000"

U1 srl COUNT = "00001101"
S1 srl COUNT = "00001101"
U2 srl COUNT = "00011101"
S2 srl COUNT = "11111101"

U1 rol COUNT = "01011011"
S1 rol COUNT = "01011011"
U2 rol COUNT = "01011111"
S2 rol COUNT = "01011111"

U1 ror COUNT = "01101101"
S1 ror COUNT = "01101101"
U2 ror COUNT = "01111101"
S2 ror COUNT = "01111101"
```

std_logic_misc

The std_logic_misc package is typically installed in the $SYNOPSYS/packages/IEEE/src/std_logic_misc.vhd directory. It declares the primary data types that the Synopsys VSS tools support.

Boolean reduction functions take one argument (an array of bits) and return a single bit. For example, the AND reduction of "101" is "0", the logical AND of all 3 bits.

Several functions in the std_logic_misc package provide Boolean reduction operations for the predefined type STD_LOGIC_VECTOR. Example 199 shows the declarations of these functions.

Example 199 Boolean Reduction Functions

```
function AND_REDUCE  (ARG: STD_LOGIC_VECTOR) return UX01;
function NAND_REDUCE (ARG: STD_LOGIC_VECTOR) return UX01;
function OR_REDUCE   (ARG: STD_LOGIC_VECTOR) return UX01;
function NOR_REDUCE  (ARG: STD_LOGIC_VECTOR) return UX01;
function XOR_REDUCE  (ARG: STD_LOGIC_VECTOR) return UX01;
function XNOR_REDUCE (ARG: STD_LOGIC_VECTOR) return UX01;
function AND_REDUCE  (ARG: STD_ULOGIC_VECTOR) return UX01;
function NAND_REDUCE (ARG: STD_ULOGIC_VECTOR) return UX01;
function OR_REDUCE   (ARG: STD_ULOGIC_VECTOR) return UX01;
function NOR_REDUCE  (ARG: STD_ULOGIC_VECTOR) return UX01;
function XOR_REDUCE  (ARG: STD_ULOGIC_VECTOR) return UX01;
function XNOR_REDUCE (ARG: STD_ULOGIC_VECTOR) return UX01;
```

These functions combine the bits of the STD_LOGIC_VECTOR, as the name of the function indicates. For example, XOR_REDUCE returns the XOR of all bits in ARG. Example 200 shows some reduction function calls and their return values.

Example 200 Boolean Reduction Operations

```
AND_REDUCE("111")    = '1'
AND_REDUCE("011")    = '0'

OR_REDUCE("000")    = '0'
OR_REDUCE("001")    = '1'

XOR_REDUCE("100")    = '1'
XOR_REDUCE("101")    = '0'

NAND_REDUCE("111")   = '0'
NAND_REDUCE("011")  = '1'

NOR_REDUCE("000")    = '1'
NOR_REDUCE("001")    = '0'

XNOR_REDUCE("100") = '0'
XNOR_REDUCE("101") = '1'
```

Standard Package

The STANDARD package of data types is included in all VHDL source files by an implicit use clause.

HDL Compiler implements the synthesizable subset of the STANDARD package listed in Example 201.

Example 201 HDL Compiler STANDARD Package

```
package STANDARD is

  type BOOLEAN is (FALSE, TRUE);

  type BIT is ('0', '1');

  type CHARACTER is (
    NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL,
    BS,  HT,  LF,  VT,  FF,  CR,  SO,  SI,
    DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB,
    CAN, EM,  SUB, ESC, FSP, GSP, RSP, USP,

    ' ', '!', '"', '#', '$', '%', '&', ''',
    '(', ')', '*', '+', ',', '-', '.', '/',
    '0', '1', '2', '3', '4', '5', '6', '7',
    '8', '9', ':', ';', '<', '=', '>', '?',
    '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
    'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
    'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
    'X', 'Y', 'Z', '[', '\', ']', '^', '_',

    ''', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
    'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
    'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
    'x', 'y', 'z', '{', '|', '}', '~', DEL);

  type INTEGER is range -2147483647 to 2147483647;

  subtype NATURAL is INTEGER range 0 to 2147483647;

  subtype POSITIVE is INTEGER range 1 to 2147483647;

  type STRING is array (POSITIVE range <>)
       of CHARACTER;

  type BIT_VECTOR is array (NATURAL range <>)
       of BIT;
end STANDARD;
```

This section describes the following synthesizable data types:

- Data Type BOOLEAN
- Data Type BIT
- Data Type CHARACTER
- Data Type INTEGER
- Data Type NATURAL
- Data Type POSITIVE
- Data Type STRING
- Data Type BIT_VECTOR

Data Type BOOLEAN

The BOOLEAN data type is actually an enumerated type with two values, false and true, where false < true. Logical functions, such as equality (=) and comparison (<) functions, return a BOOLEAN value.

Convert a BIT value to a BOOLEAN value as follows:

```
BOOLEAN_VAR := (BIT_VAR = '1');
```

Data Type BIT

The BIT data type represents a binary value as one of two characters, ’0’ or ’1’. Logical operations such as “and” can take and return BIT values.

Convert a BOOLEAN value to a BIT value as follows:

```
if (BOOLEAN_VAR) then
  BIT_VAR := '1';
else
  BIT_VAR := '0';
end if;
```

Data Type CHARACTER

The CHARACTER data type enumerates the ASCII character set. Nonprinting characters are represented by a three-letter name, such as NUL for the null character. Printable characters are represented by themselves, in single quotation marks, as follows:

```
variable CHARACTER_VAR: CHARACTER;
. . .
CHARACTER_VAR := 'A';
```

Data Type INTEGER

The INTEGER data type represents positive and negative whole numbers.

Data Type NATURAL

The NATURAL data type is a subtype of INTEGER that is used for representing natural (nonnegative) numbers.

Data Type POSITIVE

The POSITIVE data type is a subtype of INTEGER that is used for representing positive (nonzero, nonnegative) numbers.

Data Type STRING

The STRING data type is an unconstrained array of characters. A STRING value is enclosed in double quotation marks, as follows:

```
variable STRING_VAR: STRING(1 to 7);
. . .
STRING_VAR := "Rosebud";
```

Data Type BIT_VECTOR

The BIT_VECTOR data type represents an array of BIT values.

Synopsys Package—ATTRIBUTES

The ATTRIBUTES package declares all supported synthesis attributes; the source code is typically installed in the Synopsys libraries $SYNOPSYS/packages/synopsys/src/attributes.vhd directory. Supported attributes include

- HDL Compiler constraint attributes, described in Synopsys Defined Attributes on page 180
- State vector attribute, described in
- Enumeration encoding attribute, described in Enumeration Encoding on page 84

Reference this package when you use synthesis attributes:

```
library SYNOPSYS;
use SYNOPSYS.ATTRIBUTES.all;
```

Feedback

C

VHDL Constructs

Many VHDL language constructs, although useful for simulation and other stages in the design process, are not relevant to synthesis. Because these constructs cannot be synthesized, HDL Compiler does not support them.

This appendix provides a list of synthesizable VHDL language constructs, with the level of support for each, followed by a list of VHDL reserved words.

This appendix includes the following sections:

- VHDL Construct Support
- Predefined Language Environment
- VHDL Reserved Words

VHDL Construct Support

A construct can be fully supported, ignored, or unsupported. Ignored and unsupported constructs are defined as follows:

- Ignored means that the construct is allowed in the VHDL source but is ignored by HDL Compiler.
- Unsupported means that the construct is not allowed in the VHDL source and that HDL Compiler flags it as an error. If errors are in a VHDL description, the description is not read.

The following subsections describe the constructs:

- Configurations
- Design Units
- Data Types
- Declarations
- Specifications
- Names

- Operators
- Operands and Expressions
- Sequential Statements
- Concurrent Statements
- Lexical Elements

Configurations

The HDL Compiler tool supports standalone, nested, and embedded configurations. For details, see Configuration Support.

Design Units

entity

The entity statement part is ignored.

Default values for ports are ignored.

generics

In addition to supporting integer-type generics, HDL Compiler adds support for the following types: bit, bit_vector, std_ulogic, std_ulogic_vector, std_logic, std_logic_vector, signed, and unsigned. HDL Compiler also supports integer arrays and strings as generics.

architecture

Multiple architectures are allowed. Global signal interaction between architectures is unsupported.

configuration

Configuration declarations and block configurations are supported, but only to specify the top-level architecture for a top-level entity. For more information, see Configuration Support.

The use clauses and attribute specifications are unsupported.

package

Packages are fully supported.

library

Libraries and separate compilation are supported.

subprogram

Default values for parameters are unsupported. Assigning to indexes and slices of unconstrained out parameters is unsupported, unless the actual parameter is an identifier.

Subprogram recursion is unsupported if the recursion is not bounded by a static value.

Resolution functions are supported for wired-logic and three-state functions only.

Subprograms can be declared only in packages and in the declaration part of an architecture.

Data Types

enumeration

Enumeration is fully supported.

real

Constant real data types are fully supported.

integer

Infinite-precision arithmetic is unsupported.

Integer types are automatically converted to bit vectors whose width is as small as possible to accommodate all possible values of the type’s range. The type’s range can be either in unsigned binary for nonnegative ranges or in 2’s-complement form for ranges that include negative numbers.

physical

Physical type declarations are ignored. The use of physical types is ignored in delay specifications.

floating

Floating-point type declarations are ignored. The use of floating-point types is unsupported except for floating-point constants used with Synopsys defined attributes.

array

Array ranges and indexes other than integers are unsupported.

By default, the tool infers MUX_OP cells for arrays during synthesis.

Multidimensional arrays are supported.

record

Record data types are fully supported.

access

Access type declarations are ignored, and the use of access types is unsupported.

file

File type declarations are ignored, and the use of file types is unsupported.

incomplete type declarations

Incomplete type declarations are unsupported.

Declarations

constant

Constant declarations are supported except for deferred constant declarations.

signal

Register and bus declarations are unsupported. Resolution functions are supported for wired and three-state functions only. Declarations other than from a globally static type are unsupported. Initial values are unsupported.

variable

Declarations other than from a globally static type are unsupported. Initial values are unsupported.

shared variable

Variable shared by different processes. Shared variables are fully supported.

file

File declarations are unsupported.

interface

Buffer and linkage are translated to out and inout, respectively.

alias

Alias declarations are supported.

component

Component declarations that list a name other than a valid entity name are unsupported. However, HDL Compiler allows components to be directly instantiated in the design without a component declaration. For more information, see Direct Instantiation of Components.

attribute

Attribute declarations are fully supported, but the use of user-defined attributes is unsupported.

group

HDL Compiler supports VHDL-93 group declarations. This allows you to create groups of named entities. One useful application of this feature is that you can apply attributes to the group as a whole instead of referencing individual signals. For more information, see Groups.

Specifications

attribute

HDL Compiler supports the `leftof, `rightof, pos, val, succ, and pred attributes for enum data types. HDL Compiler supports the pos, val, succ, and pred attributes for integer and range data types. HDL Compiler supports the 'high(n),'low(n),'left(n),'right(n), and ‘length(n) attributes on multidimensional arrays. Others and all are unsupported in attribute specifications. User-defined attributes can be specified, but the use of user-defined attributes is unsupported.

configuration

Configuration specifications are unsupported.

disconnection

Disconnection specifications are unsupported. Attribute declarations are fully supported, but the use of user-defined attributes is unsupported.

Names

simple

Simple names are fully supported.

selected

Selected (qualified) names outside a use clause are unsupported. Overriding the scopes of identifiers is unsupported.

operator symbol

Operator symbols are fully supported.

indexed

Indexed names are fully supported, with one exception: Indexing an unconstrained out parameter in a procedure is unsupported.

slice

Slice names are fully supported, with one exception: Using a slice of an unconstrained out parameter in a procedure is unsupported unless the actual parameter is an identifier.

attribute

Only the following predefined attributes are supported: `base`, `left`, `right`, `high`, `low`, `range`, `reverse_range`, `length`, and `ascending`. The `event` and `stable` attributes are supported only as described with the wait and if statements. User-defined attribute names are unsupported. The use of attributes with selected names (name.name'attribute) is unsupported.

Table 30 shows the values of some array attributes for the variable MY_VECTOR in Example 202.

Table 30

Operators

logical

Logical operators are fully supported.

relational

Relational operators are fully supported.

addition

Concatenation and arithmetic operators are fully supported. The default (and only) concatenation support is for the 93 LRM definition. For more information, see Concatenation.

signing

Signing operators are fully supported.

divide, mod, rem

The / (division), mod, and rem operators are fully supported in the std_logic_arith and the numeric_std packages.

multiply

The * multiply operator is fully supported.

exponentiation

The ** operator is supported only when both operands are constant or when the left operand is 2. HDL Compiler predefines the exponentiation operator for all integer types.

absolute value

The abs operator is fully supported. HDL Compiler predefines the absolute value operator for all integer types.

operator overloading

Operator overloading is fully supported.

short-circuit operation

The short-circuit behavior of operators is not supported.

Shift and rotate operators

You can define shift and rotate operators for any one-dimensional array type whose element type is either of the predefined types, BIT or Boolean. The right operand is always of type integer. The type of the result of a shift operator is the same as the type of the left operand. The shift and rotate operators are included in the list of VHDL reserved words listed in VHDL Reserved Words. There is more information about the shift and rotate operators that numeric_std supports in Shift and Rotate Functions. The shift operators are

sll	Shift left logical
srl	Shift right logical
sla	Shift left arithmetic
sra	Shift right arithmetic

The rotate operators are

rol	Rotate left logical
ror	Rotate right logical

Example 203 illustrates the use of shift and rotate operators.

Example 203 Use of Shift and Rotate Operators

```
architecture arch of shft_op is
begin
 a <= "01101";
 q1 <= a sll 1;     -- q1 = "11010"
 q2 <= a srl 3;     -- q2 = "00001"
 q3 <= a rol 2;     -- q3 = "10101"
 q4 <= a ror 1;     -- q4 = "10110"
 q5 <= a sla 2;     -- q5 = "10100"
 q6 <= a sra 1;     -- q6 = "00110"
end;
```

XNOR Operator

You can define the binary logical operator XNOR for predefined types BIT and Boolean as well as for any one-dimensional array type whose element type is BIT or Boolean. The operands must be the same type and length. The result also has the same type and length. The XNOR operator is included in the list of VHDL reserved words in VHDL Reserved Words.

Example 204 Showing Use of XNOR Operator

```
a <= "10101";
b <= "11100";
c <= a xnor b;       -- c = "10110"
```

Operands and Expressions

based literal

Based literals are fully supported.

null literal

Null slices, null ranges, and null arrays are supported.

physical literal

Physical literals are ignored.

string

Strings are fully supported.

aggregate

The use of types as aggregate choices is supported. Record aggregates are supported.

function call

Function calls are supported. Function conversions on input ports are supported, because type conversions on formal ports in a connection specification (port map) are supported. HDL Compiler supports the usage of unconstrained type ports when the type of the ports can be deduced. In these cases, you must use analyze/elaborate to read your design. The read command does not support type conversion on formal ports.

qualified expression

Qualified expressions are fully supported.

type conversion

Type conversion is fully supported.

allocator

Allocators are unsupported.

static expression

Static expressions are fully supported.

universal expression

Floating-point expressions are unsupported, except in a Synopsys-recognized attribute definition. Infinite-precision expressions are not supported. Precision is limited to 32 bits; all intermediate results are converted to integers.

Sequential Statements

wait

The wait statement is unsupported unless it is in one of the following forms:

```
wait until                     clock = VALUE;
wait until     clock'event  and clock = VALUE;
wait until not clock'stable and clock = VALUE;
```

Where, VALUE is ’0’, ’1’, or an enumeration literal whose encoding is 0 or 1. A wait statement in this form is interpreted to mean “wait until the falling (VALUE is ’0’) or rising (VALUE is ’1’) edge of the signal named clock.” You cannot use wait statements in subprograms or for-loop statements. If any path through the logic has a wait statement, all the paths must have a wait statement. HDL Compiler supports only one wait statement per process.

assert

Assert statements are treated like display statements, as shown in the following code snippet:

```
Assert (c) report "...";

-- is the same as
```

```
If (!c)
  $display ("...");
```

report

Report statements are ignored.

statement label

Statement labels are ignored.

signal

Guarded signal assignment is unsupported. The transport and after signals are ignored. Multiple waveform elements in signal assignment statements are unsupported.

variable

Variable statements are fully supported.

procedure call

Type conversion on formal parameters is unsupported. Assignment to single bits of vectored ports is unsupported.

if

The if statements are fully supported.

case

The case statements are fully supported.

loop

The for loops are supported, with two constraints: The loop index range must be globally static, and the loop body must not contain a wait statement. The while loops are supported, but the loop body must contain at least one wait statement. Combinational while loops are supported if the iterative bound is statically determinable. The loop statements with no iteration scheme (infinite loops) are supported, but the loop body must contain at least one wait statement.

next

Next statements are fully supported.

exit

Exit statements are fully supported.

return

Return statements are fully supported.

null

Null statements are fully supported.

Concurrent Statements

block

Guards on block statements are supported. Ports and generics in block statements are unsupported.

process

Sensitivity lists in process statements are ignored.

concurrent procedure call

Concurrent procedure call statements are fully supported.

concurrent assertion

Concurrent assertion statements are ignored.

concurrent signal assignment

The `guarded` keyword is supported. The `transport` keyword is ignored. Multiple waveforms are unsupported.

component instantiation

Type conversion on formal ports of a connection specification is supported. HDL Compiler supports the usage of unconstrained type ports when the type of the ports can be deduced. In these cases, you must use `analyze`/`elaborate` to read your design. The `read` command does not support type conversion on formal ports.

generate

The generate statements are fully supported.

Lexical Elements

An identifier in VHDL is a user-defined name for any of these: constant, variable, function, signal, entity, port, subprogram, parameter, or instance.

Specifics of Identifiers

The characteristics of identifiers are as follows:

- They can be composed of letters, digits, and the underscore character (_).
- Their first character must be a letter, unless it is an extended identifier (see Example 205).
- They can be of any length.
- They are case-insensitive.
- All of their characters are significant.

Specifics of Extended Identifiers

The characteristics of extended identifiers are as follows:

- Any of the following can be defined as one:
  - Identifiers that contain special characters
  - Identifiers that begin with numbers
  - Identifiers that have the same name as a keyword
- They start with a backslash character (\), followed by a sequence of characters, followed by another backslash (\).
- They are case-sensitive.

Example 205 shows some extended identifiers.

Example 205 Sample Extended Identifiers

```
\a+b\      \3state\
\type\      \(a&b)|c\
```

Predefined Language Environment

severity_level type

The severity_level type is unsupported.

time type

The time type is ignored if time variables and constants are used only in after clauses. In the following two code fragments, both the after clause and TD are ignored:

```
constant TD: time := 1.4 ns;
X <= Y after TD;

X <= Y after 1.4 ns;
```

now function

The now function is unsupported.

TEXTIO package

The TEXTIO package is unsupported. The TEXTIO package defines types and operations for communication with a standard programming environment (terminal and file I/O). This package is not needed for synthesis; therefore, HDL Compiler does not support it.

predefined attributes

These predefined attributes are supported: `base`, `left`, `right`, `high`, `low`, `range`, `reverse_range`, `ascending`, and `length`. The `event` and `stable` attributes are supported only in the if and wait statements.

VHDL Reserved Words

Table 31 lists the words that are reserved for the VHDL language and cannot be used as identifiers:

Table 31 VHDL Reserved Words

abs	access	after	alias	all	and
architecture	array	assert	attribute	begin	block
body	buffer	bus	case	component	configuration
constant	disconnect	downto	else	elsif	end
entity	exit	file	for	function	generate
generic	group	guarded	if	impure	in
inertial	inout	is	label	library	linkage
literal	loop	map	mod	nand	new
next	nor	not	null	of	on
open	or	others	out	package	port
postponed	procedure	process	pure	range	record

Table 31 VHDL Reserved Words (Continued)

register	reject	rem	report	return	rol
ror	select	severity	shared	signal	sla
sll	sra	srl	subtype	then	to
transport	type	unaffected	units	until	use
variable	wait	when	while	with	xnor
xnor					

Glossary

anonymous type
A predefined or underlying type with no name, such as a universal integer.

architecture body
The VHDL description of the internal organization or operation of a design entity.

ASIC
Application-specific integrated circuit.

behavioral view
The set of VHDL 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 *data flow view, sequential statement,* and *structural view*.

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 can be determined.

concurrent statements
VHDL statements that execute asynchronously in no defined relative order. Concurrent statements make up the data flow and structural views in VHDL.

configuration body
The VHDL description of how component instances are bound to design entities to form a complete, linked design.

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 VHDL 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 *behavioral view*, *concurrent statements,* and *structural view*.

design constraints

See *constraints*.

design entity

In VHDL, the combination of an entity declaration and one or more architectural bodies constitute a design entity.

flip-flop

An edge-sensitive memory device.

HDL

Hardware Description Language.

identifier

A sequence of letters, underscores, and numbers. An identifier cannot be a VHDL 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 of the design. HDL Compiler optimizes designs and tries to meet specified design constraints for area and speed.

package

A collection of declarations that is available to more than one design entity.

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 similar VHDL operations, such as +, 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
The set of VHDL 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 VHDL through either signal or port declarations.

signed value
A value that can be positive, 0, or negative.

structural view
The set of VHDL statements used to instantiate primitive and hierarchical components in a design. A VHDL 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.

translation
The mapping of high-level language constructs onto a lower-level form.

type
In VHDL, 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 0.

variable
A VHDL object local to a process or subprogram that has a single current value.

VHDL
VHSIC Hardware Description Language, used to describe discrete systems.

VHSIC
Very-high-speed integrated circuit, a high-technology program of the United States Department of Defense.