HDL Modeling in Encounter® RTL Compiler

Product Version 14.2
August 2015

Contents

List of Figures

List of Examples

List of Tables

Preface

About This Manual

This manual describes HDL modeling in RTL Compiler. The RTL Compiler software accepts both VHDL entities and Verilog design modules.

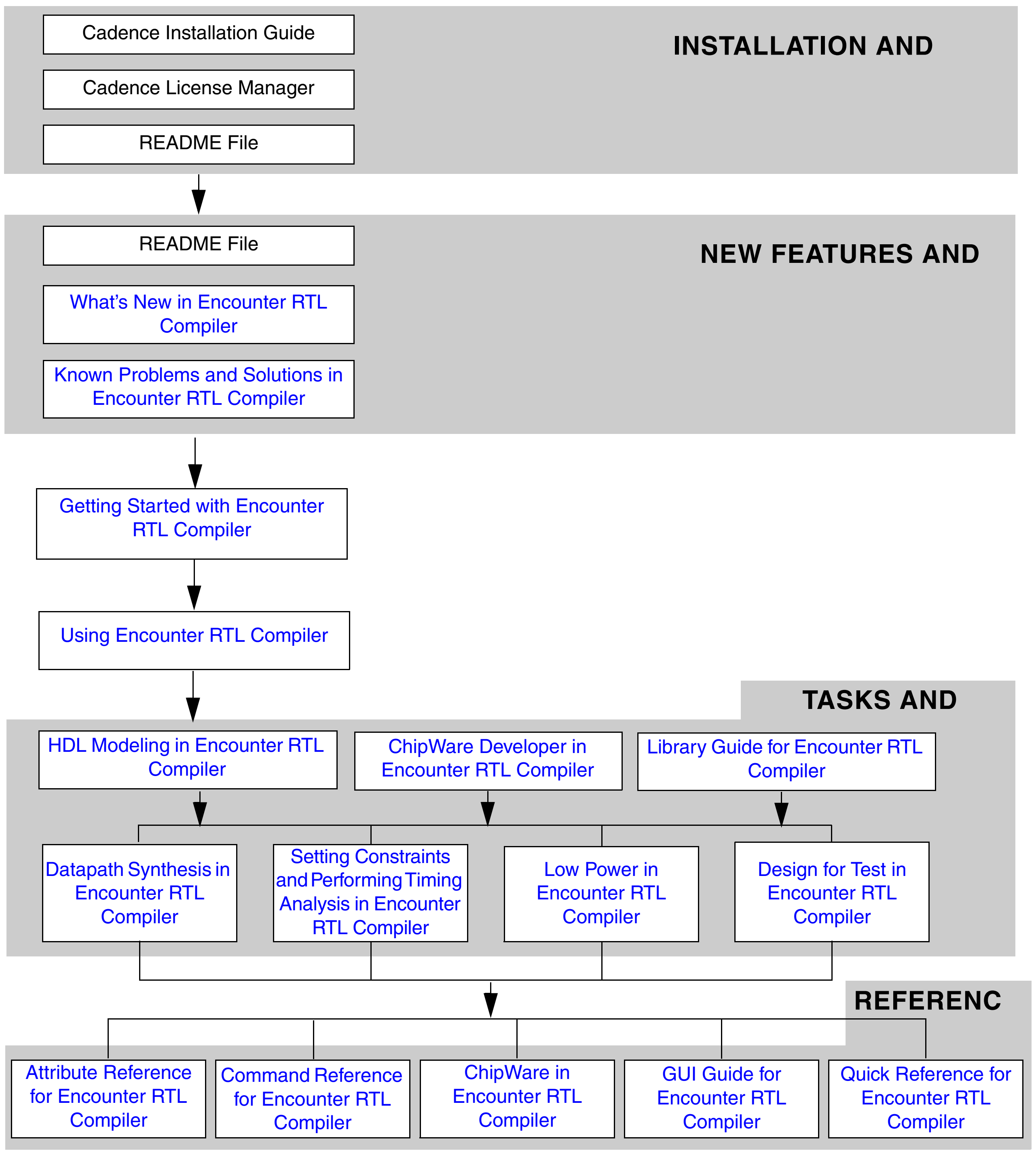
Additional References

The following sources are helpful references, but are not included with the product documentation:

- TclTutor, a computer aided instruction package for learning the Tcl language: http://www.msen.com/~clif/TclTutor.html.
- TCL Reference, *Tcl and the Tk Toolkit*, John K. Ousterhout, Addison-Wesley Publishing Company
- IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language (IEEE Std.1364-1995)
- IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language (IEEE Std. 1364-2001)
- IEEE Standard VHDL Language Reference Manual (IEEE Std. 1076-1987)
- IEEE Standard VHDL Language Reference Manual (IEEE Std. 1076-1993)

Note: For information on purchasing IEEE specifications go to http://shop.ieee.org/store/ and click on *Standards.*

How to Use the Documentation Set


Cadence Installation Guide
Cadence License Manager
README File
INSTALLATION AND
README File
What's New in Encounter RTL Compiler
Known Problems and Solutions in Encounter RTL Compiler
NEW FEATURES AND
Getting Started with Encounter RTL Compiler
Using Encounter RTL Compiler
TASKS AND
HDL Modeling in Encounter RTL Compiler
ChipWare Developer in Encounter RTL Compiler
Library Guide for Encounter RTL Compiler
Datapath Synthesis in Encounter RTL Compiler
Setting Constraints and Performing Timing Analysis in Encounter RTL Compiler
Low Power in Encounter RTL Compiler
Design for Test in Encounter RTL Compiler
REFERENC
Attribute Reference for Encounter RTL Compiler
Command Reference for Encounter RTL Compiler
ChipWare in Encounter RTL Compiler
GUI Guide for Encounter RTL Compiler
Quick Reference for Encounter RTL Compiler

Reporting Problems or Errors in Manuals

The Cadence® Help online documentation lets you view, search, and print Cadence product documentation. You can access Cadence help by typing `cdnshelp` from your Cadence tools hierarchy.

Contact Cadence Customer Support to file a PCR if you find

- An error in the manual
- Any missing information in a manual
- A problem using the Cadence Help documentation system

Customer Support

Cadence offers live and online support, as well as customer education and training programs.

Cadence Online Support

The Cadence® online support website offers answers to your most common technical questions. It lets you search more than 40,000 FAQs, notifications, software updates, and technical solutions documents that give you step-by-step instructions on how to solve known problems. It also gives you product-specific e-mail notifications, software updates, case tracking, up-to-date release information, full site search capabilities, software update ordering, and much more.

For more information on Cadence online support go to:

http://support.cadence.com

Other Support Offerings

- **Support centers**—Provide live customer support from Cadence experts who can answer many questions related to products and platforms.
- **Software downloads**—Provide you with the latest versions of Cadence products.
- **Education services**—Offers instructor-led classes, self-paced Internet, and virtual classroom.
- **University software program support**—Provides you with the latest information to answer your technical questions.

For more information on these support offerings go to:

http://www.cadence.com/support

Messages

From within RTL Compiler there are two ways to get information about messages.

- Use the `report messages` command.

  For example:

  ```
  rc:/> report messages
  ```

  This returns the detailed information for each message output in your current RTL Compiler run. It also includes a summary of how many times each message was issued.

- Use the `man` command.

  Note: You can only use the `man` command for messages within RTL Compiler

  For example, to get more information about the “TIM-11”message, type the following command:

  ```
  rc:/> man TIM-11
  ```

If you do not get the details that you need or do not understand a message, either contact Cadence Customer Support to file a PCR or email the message ID you would like improved to:

rc_pubs@cadence.com

Man Pages

In addition to the Command and Attribute References, you can also access information about the commands and attributes using the man pages in RTL Compiler. Man pages contain the same content as the Command and Attribute References. To use the man pages from the UNIX shell:

1. Set your environment to view the correct directory:

   ```
   setenv MANPATH $CDN_SYNTH_ROOT/share/synth/man
   ```

2. Enter the name of the command or attribute that you want either in RTL Compiler or within the UNIX shell. For example:

   ❑ `man check_dft_rules`

   ❑ `man cell_leakage_power`

3. Enter the name of the command or attribute that you want. For example:

   ❑ `man check_dft_rules`

   ❑ `man cell_leakage_power`

You can also use the `more` command, which behaves like its UNIX counterpart. If the output of a manpage is too small to be displayed completely on the screen, use the `more` command to break up the output. Use the spacebar to page forward, like the UNIX `more` command.

```
rc:/> more man synthesize
```

Command-Line Help

You can get quick syntax help for commands and attributes at the RTL Compiler command-line prompt. There are also enhanced search capabilities so you can more easily search for the command or attribute that you need.

Note: The command syntax representation in this document does not necessarily match the information that you get when you type `help` *`command_name`*. In many cases, the order of the arguments is different. Furthermore, the syntax in this document includes all of the dependencies, where the help information does this only to a certain degree.

If you have any suggestions for improving the command-line help, please e-mail them to:

rc_pubs@cadence.com

Getting the Syntax for a Command

➤ Type the `help` command followed by the command name. For example:

```
rc:/> help path_delay
```

This returns the syntax for the `path_delay` command.

Getting the Syntax for an Attribute

➤ Type the following:

```
rc:/> get_attribute attribute name * -help
```

For example:

```
rc:/> get_attribute max_transition * -help
```

This returns the syntax for the `max_transition` attribute.

Searching for Attributes

➤ Get a list of all the available attributes by typing the following command:

```
rc:/> get_attribute * * -help
```

➤ Type a sequence of letters after the `set_attribute` command and press `Tab` to get a list of all attributes that contain those letters. For example:

```
rc:/> set_attr li
```

```
ambiguous "li": lib_lef_consistency_check_enable lib_search_path libcell
liberty_attributes libpin library library_domain line_number
```

Searching For Commands When You Are Unsure of the Name

You can use help to find a command if you only know part of its name, even as little as one letter.

■ You can type a single letter and press `Tab` to get a list of all commands that start with that letter.

  For example:

```
rc:/> c <Tab>
```

  This returns the following commands:

```
ambiguous "c": cache_vname calling_proc case catch cd cdsdoc change_names
check_dft_rules chipware clear clock clock_gating clock_ports close cmdExpand
command_is_complete concat configure_pad_dft connect_scan_chains continue
cwd_install ..
```

■ You can type a sequence of letters and press `Tab` to get a list of all commands that start with those letters.

  For example:

```
rc:/> path_<Tab>
```

  This returns the following commands:

```
ambiguous command name "path_": path_adjust path_delay path_disable path_group
```

Documentation Conventions

Text Command Syntax

The list below defines the syntax conventions used for the RTL Compiler text interface commands.

`literal`	Nonitalic words indicate keywords you enter literally. These keywords represent command or option names.
`arguments and options`	Words in italics indicate user-defined arguments or information for which you must substitute a name or a value.
|	Vertical bars (OR-bars) separate possible choices for a single argument.
`[ ]`	Brackets indicate optional arguments. When used with OR-bars, they enclose a list of choices from which you can choose one.
`{ }`	Braces indicate that a choice is required from the list of arguments separated by OR-bars. Choose one from the list. `{argument1 | argument2 | argument3}`
{ }	Braces, used in Tcl commands, indicate that the braces must be typed in.
`...`	Three dots (...) indicate that you can repeat the previous argument. If the three dots are used with brackets (that is, `[argument]...)`, you can specify zero or more arguments. If the three dots are used without brackets (`argument...)`, you must specify at least one argument.
#	The pound sign precedes comments in command files.

1

Modeling HDL Designs

Overview of HDL Modeling

Perform RTL synthesis after loading the timing and power libraries. For information on reading Verilog files and libraries, see Chapter 5, "Loading Files" in *Using Encounter RTL Compiler*.

This chapter is organized for mixed Verilog and VHDL language usage and describes how to use RTL Compiler to synthesize hardware models described in Verilog and VHDL. Use these styles as a guideline to achieve the best synthesis results from RTL Compiler. See "Reading Designs with Mixed Verilog and VHDL Files" in *Using Encounter RTL Compiler* for more information.

See Supported Verilog Modeling Constructs on page 181 and VHDL Constructs on page 208 for a list of language constructs supported by RTL Compiler.

If you want to only see the Verilog-specific or the VHDL-specific information, refer to Chapter 4, "Synthesizing Verilog Designs", and Chapter 5, "Synthesizing VHDL Designs", respectively.

By default, RTL Compiler automatically generates a generic netlist from a RTL design. Use synthesis pragmas to control the synthesis process. See Chapter 2, "Specifying Synthesis Pragmas" for detailed information. See Supported Pragmas on page 82 for a list of Synopsys synthesis pragmas supported by RTL Compiler.

Chapter 3, "Using HDL Commands and Attributes" summarizes the commands and attributes used by RTL Compiler to synthesize generic netlists from Verilog and VHDL RTL designs.

The synthesizable subset of Verilog is based on the *IEEE 1364 - 1995 Standard*, the *1364 - 2001 Standard*, and the *Accellera SystemVerilog 3.1a for Verilog Register Transfer Level Synthesis*.

The synthesizable subset of VHDL is based on the *IEEE 1076.6-1999 Standard for VHDL Register Transfer Level Synthesis*. For detailed information on the VHDL syntax and semantics, refer to the following IEEE Standard VHDL Language Reference Manuals:

- *ANSI/IEEE Std 1076-1987* (for VHDL87)
- *ANSI/IEEE Std 1076-1993* (for VHDL93)
- *ANSI/IEEE Std 1076-2008* (for VHDL08)

VHDL designs have the following restrictions:

- Reads an entity before any of the entity's architectures and packages.
- Reads package bodies before reading any other packages, entities, or architectures that refer to them.

Modeling Flip-Flops

A register is either a level-sensitive latch or an edge-triggered flip-flop memory element. RTL Compiler identifies registers from the HDL syntax and generates the appropriate sequential logic.

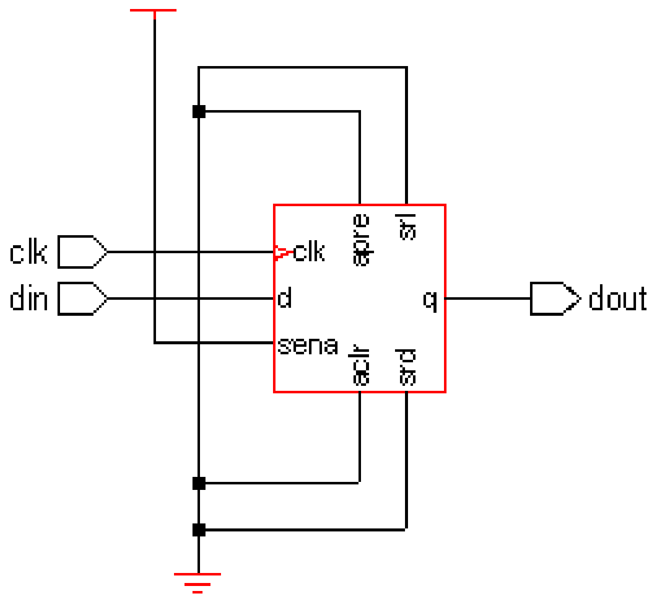
Modeling Flip-Flops in Verilog

When an assignment is conditioned upon a rising or falling transition on a signal, an edge-triggered flip-flop is inferred to implement the variable on the left side of the assignment, as shown in Example 1-1. Figure 1-1 shows the corresponding schematic for Example 1-1.

Example 1-1 Modeling a Rising Edge Triggered Flip-Flop (Verilog)

```
module sync_flop (clk, din, dout);
    input clk;
    input din;
    output dout;
    reg dout;
    always @(posedge clk)
    begin
        dout <= din;
    end
endmodule
```

Figure 1-1 Rising Edge Triggered Flip-Flop Schematic (Verilog)



The following examples show how a flip-flop with an asynchronous operation can be inferred. The `always` block is triggered when a rising edge is detected on `clk` or a rising edge on `rst`.

- Example 1-2 on page 33 uses `if` and `else` constructs
- Example 1-3 on page 33 uses the conditional operator (?:)

Figure 1-2 on page 34 shows the corresponding schematic for both examples.

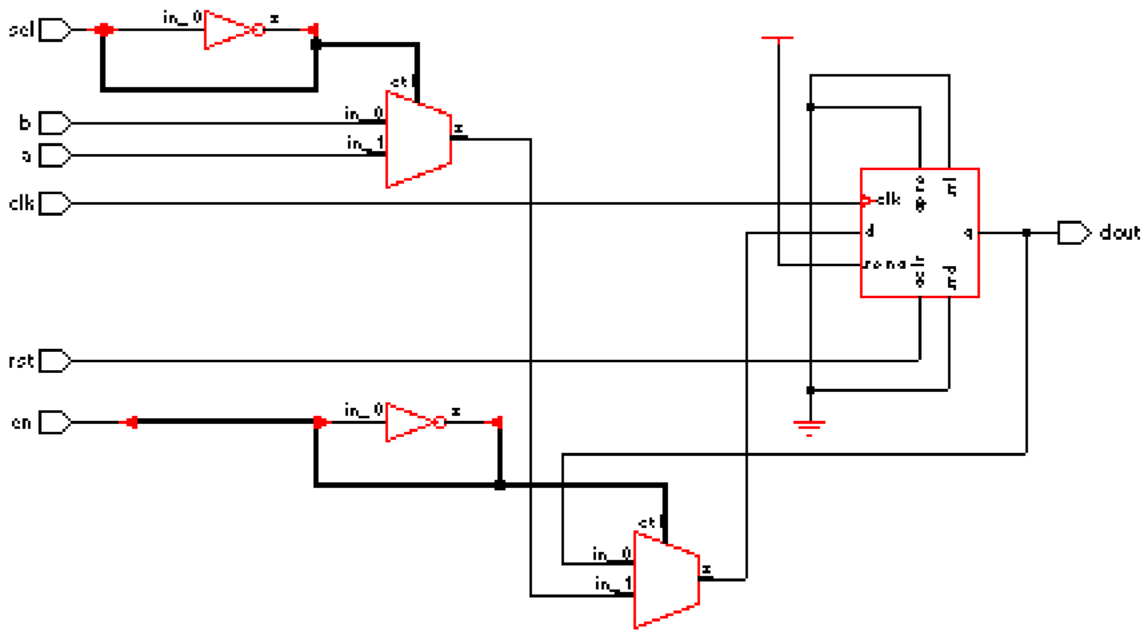
Example 1-2 Modeling an Active High Asynchronous Reset Flip-Flop (Verilog)

```
module ff_ar(dout,clk,rst,en,sel,a,b);
    input clk,rst,en,sel,a,b;
    output dout;
    reg dout;
    always @(posedge clk or posedge rst)
        begin
        if (rst)
            dout = 1'b0;
        else if (en) begin
            if (sel)
                dout = a;
            else
                dout = b;
        end
    end
endmodule
```

Example 1-3 Modeling an Active High Asynchronous Reset Flip-Flop (Verilog) using conditional operator

```
module ff_ar(dout,clk,rst,en,sel,a,b);
    input clk,rst,en,sel,a,b;
    output dout;
    reg dout;
    always @(posedge clk or posedge rst)

        dout = rst ? 1’b0 : (en ? (sel ? a : b) : dout);

endmodule
```

Figure 1-2 Active High Asynchronous Reset Flip-Flop Schematic (Verilog)



If `rst` is active low, then the event is in the sensitivity list, and the condition in the `if` statement should be negated.

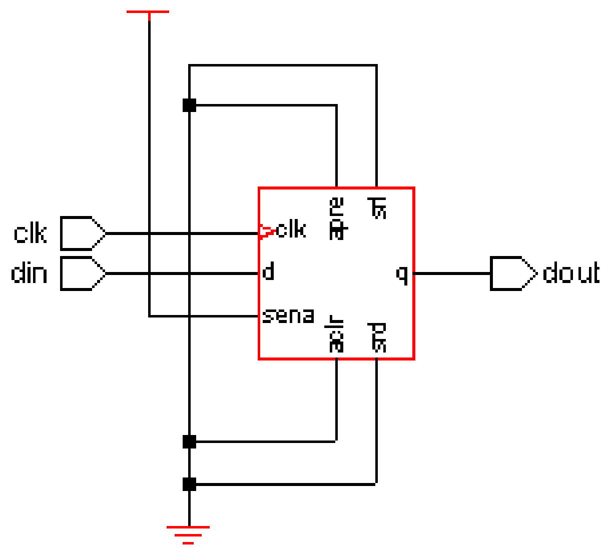
Modeling Flip-Flops in VHDL

When a process is triggered by a rising edge or a falling edge transition on a signal (typically a clock signal), the variable or signal on the left side of a procedural assignment is inferred as a flip-flop, as shown in Example 1-4. Figure 1-3 shows the corresponding schematic.

Example 1-4 Modeling a Rising Edge Triggered Flip-Flop (VHDL)

```
library ieee;
use ieee.std_logic_1164.all;
entity dff1 is
    port(
        din, clk: in std_logic;
        dout : out std_logic);
end;

architecture rtl of dff1 is begin
    dout <= din when rising_edge(clk);
end;
```

Figure 1-3 Elaborated Netlist Schematic for Example 1-4 (VHDL)



In VHDL93, the same flip-flop is modeled by using a procedural conditional signal assignment:

```
process(clk) begin
    if clk'event and clk = '1' then
        dout <= din;
    end if;
end process;
```

Note: Example 1-4 uses the standard `rising_edge` function, which is defined in the `IEEE.STD_LOGIC_1164` and `IEEE.NUMERIC_BIT` packages, to specify a positive edge on the `clk` signal.

Modeling Flip-Flop Clocks

- Using an `if` statement:

```
process (clk)
begin
    if (clk'event and clk = '1') then
        dout <= din;
    end if;
end process
```

- Using a `wait` statement:

```
process
begin
    wait until (clk'event and clk = '1');
    dout <= din;
end process;
```

- Using a `conditional signal` assignment statement in VHDL93:

```
dout <= din when (clk'event and clk = '1');
```

Use the model, as shown in Example 1-5, to synthesize a flip-flop with synchronous `set` and `reset` connections.

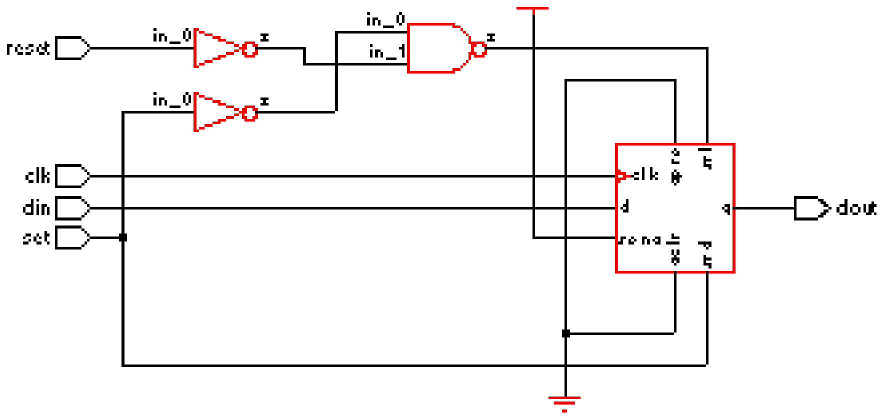
Example 1-5 Synthesizing Synchronous Set and Reset Signals On a Flip-Flop (VHDL)

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

entity sync_sr1 is
    port(
        din, clk, set, reset: in std_logic;
        dout : out std_logic);
end;

architecture rtl of sync_sr1 is begin
    process(clk) begin
        if clk'event and clk = '1' then
            if set = '1' then
                dout <= '1';
            elsif reset = '1' then
                dout <= '0';
            else
                dout <= din;
            end if;
        end if;
    end process;
end;
```

The process is triggered only on the rising edge of `clk`, but the assignment to `dout` is controlled by `set` and `reset` signals; `dout` is assigned the value of `din` only when `set` and `reset` are inactive. Only single-bit `set` and `reset` signals are supported. See Specifying Synthesis Pragmas on page 79 for more information on controlling the `set` and `reset` connections for a flip-flop.

Figure 1-4 shows the corresponding schematic for Example 1-5.

Figure 1-4 Synchronous Set and Reset Signals On a Flip-Flop Schematic (VHDL)



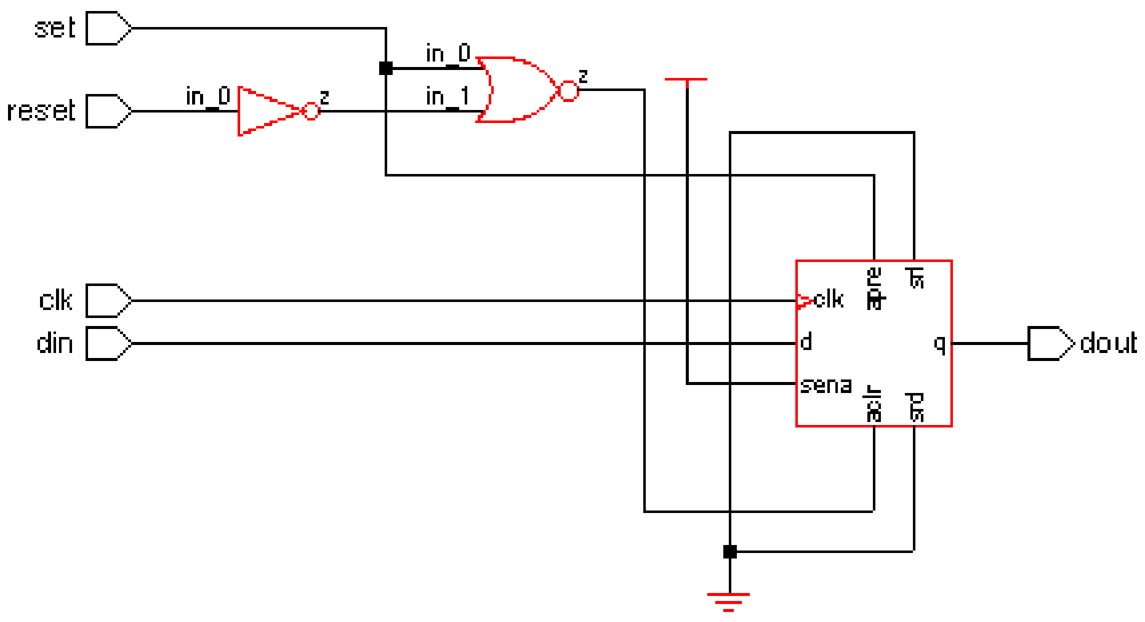
Use the model, as shown in Example 1-6, to synthesize a flip-flop with asynchronous `set` and `reset` connections.

Example 1-6 Synthesizing Asynchronous Set and Reset Signals on a Flip-Flop (VHDL)

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

entity async_sr1 is
    port(
        din, clk, set, reset: in std_logic;
        dout : out std_logic);
end;

architecture rtl of async_sr1 is begin
    process (clk, set, reset) begin
        if set = '1' then
            dout <= '1';
        elsif reset = '1' then
            dout <= '0';
        elsif clk'event and clk = '1' then
            dout <= din;
        end if;
    end process;
end;
```

The process is triggered when a rising edge is detected on `clk` or a change is detected on `set` or `reset`. Figure 1-5 shows the corresponding schematic for Example 1-6.

Figure 1-5 Asynchronous Set and Reset Signals On a Flip-Flop Schematic (VHDL)



If `set` or `reset` is active low, then the condition in the `if` statement is canceled. For example:

```
process(clk, set, ...)
begin
    if set = '0' then
        dout <= '0';
```

Specifying Clock Signals for Flip-Flops

Specify the rising edge of the clock signal in the following ways:

- For `bit` clock signals:
  - `clk'event and clk = '1'`
  - `not clk'stable and clk = '1'`
- For `boolean` clock signals:
  - `clk'event and clk = TRUE`
  - `not clk'stable and clk = TRUE`

- For `std_ulogic` and `std_logic` clock signals:
    - `rising_edge(clk)`
    - `clk'event and clk = '1'`
    - `not clk'stable and clk = '1'`

Specify the falling edge of the clock signal in the following ways:

- For `bit` clock signals:
    - `clk'event and clk = '0'`
    - `not clk'stable and clk = '0'`
- For `boolean` clock signals:
    - `clk'event and clk = FALSE`
    - `not clk'stable and clk = FALSE`
- For `std_ulogic` and `std_logic` clock signals:
    - `falling_edge(clk)`
    - `clk'event and clk = '0'`
    - `not clk'stable and clk = '0'`

Use these clock-edge expressions in `if`, `wait`, and `conditional` signal assignment statements.

In addition, use the following expressions in `wait` statements to specify rising and falling edges respectively:

- `wait until (clk = '1'); -- rising clock edge`
- `wait until (clk = '0'); -- falling clock edge`

Modeling Latches

Modeling Latches in Verilog

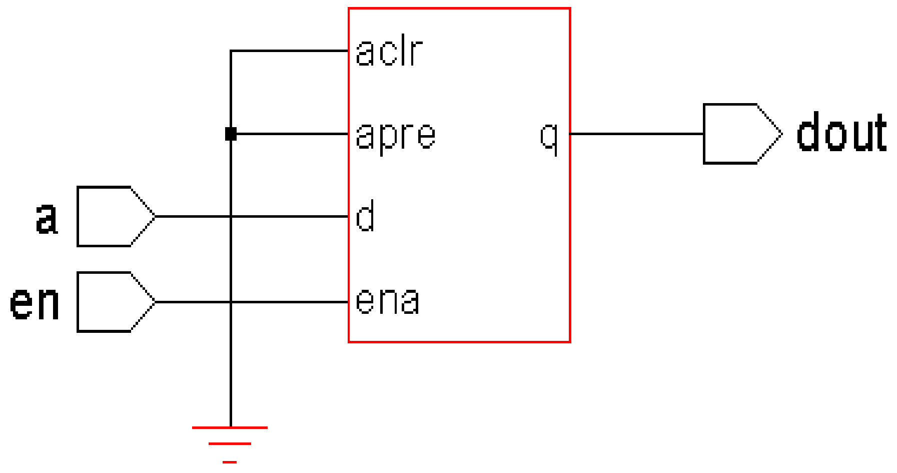
RTL Compiler infers a latch for a variable if it is updated whenever any of the variables that contribute to its value change when the enable signal is valid, as shown in Example 1-7. The `dout` signal is updated when `en` is high, otherwise the `dout` signal retains its previous value. RTL Compiler infers a latch to implement the `dout` variable.

Example 1-7 Modeling a Latch in Verilog

```
module latch(dout,en,a);
    input en,a;
    output dout;
    reg dout;

always @(en or a)
    begin
        if (en)
            dout = a;
    end
endmodule
```

Figure 1-6 shows the corresponding schematic for Example 1-7.

Figure 1-6 Latch Schematic (Verilog)



Modeling Latches in VHDL

RTL Compiler infers a latch for a variable that is incompletely assigned and that is updated whenever any of the variables that contribute to its value change, as shown in Example 1-8.

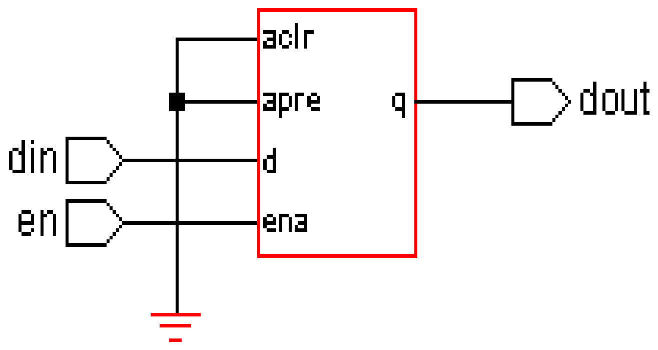
Example 1-8 Modeling a Latch (VHDL)

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

entity latch1 is
    port(
        din, en: in std_logic;
        dout : out std_logic);
end;

architecture rtl of latch1 is begin
    process(din, en) begin
        if en = '1' then
            dout <= din;
        end if;
    end process;
end;
```

Figure 1-7 shows the corresponding schematic.

Figure 1-7 Elaborated Netlist Schematic for Example 1-8 (VHDL)

.



In VHDL93, the same latch is inferred by using a concurrent conditional signal assignment:

```
    dout <= din when (en = '1');
```

Modeling Combinational Logic

Modeling Combinational Logic in Verilog

Much of logic design involves connecting simple, easily understood circuits to construct a larger circuit that performs a much more complicated function. Combinational logic is probably the easiest circuitry to design.

Use combinational logic to design circuits, such as multiplexers, decoders, and 1-bit adders. The outputs from a combinational logic circuit depend only on the current inputs.

Continuous assignments and procedural assignments are the main styles for modeling combinational logic.

Modeling Combinational Logic Using Continuous Assignments

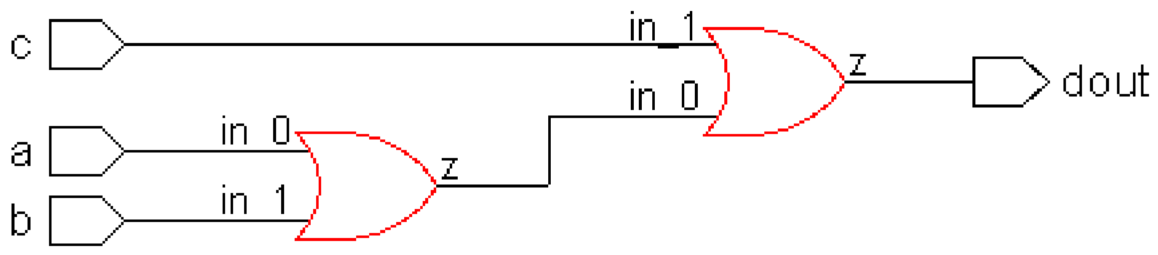
Continuous assignments are introduced by the *assign* keyword. Combinational logic is inferred for any variable assigned with continuous assignments, as shown in Example 1-9.

Example 1-9 Modeling Combinational Logic Using Continuous Assignments (Verilog)

```
module comb_or(dout,a,b,c);
    input a,b,c;
    output dout;

    assign dout = a | b | c;

endmodule
```

Figure 1-8 shows the corresponding schematic for Example 1-9.

Figure 1-8 Combinational Logic Using Continuous Assignments (Verilog)



Modeling Combinational Logic Using Procedural Assignments

Procedural assignments are introduced by `always` blocks, tasks, and functions and are used to assign values to variables declared as registers. Use a procedural assignment statement in a sequential block of an `always` statement to describe the composition of intermediate values within a combinational block.

Combinational logic is inferred for any variable assigned using procedural assignments under all possible conditions whenever any of the variables in the right-side expression change.

Variables used on the left side of a procedural assignment are declared as `reg`, which is a storage data type. However, not all variables declared as a `reg` data type need to be implemented in hardware with a memory element, such as a latch or flip-flop.

RTL Compiler synthesizes combinational logic to implement a variable under the following conditions:

- The variable is unconditionally assigned a value before it is used.
- Whenever any of the variables on the right-side expression change.

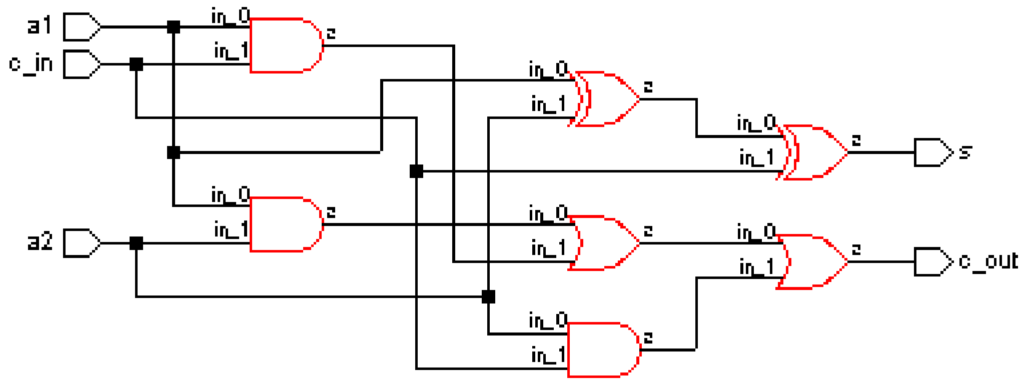
Combinational logic is synthesized to implement the `c_out` variable in Example 1-10.

Example 1-10 Modeling Combinational Logic Using Procedural Assignments (Verilog)

```
module comb_full_adder (a1, a2, c_in, s, c_out);
        input  a1, a2, c_in;
        output s, c_out;
        reg s, c_out;
        always @(a1 or a2 or c_in)
        begin
        s = a1 ^ a2 ^ c_in;
        c_out = (a1 & a2) | (a1 & c_in) | (a2 & c_in);
    end
endmodule
```

Figure 1-9 shows the corresponding schematic for Example 1-10.

Figure 1-9 Combinational Logic Using Procedural Assignments (Verilog)



Modeling Clock Gating Using Conditional Statements

Registers that are conditionally loaded can be considered by low power (LP) for clock gating.

In Example 1-11 and Example 1-12 signal `en` is used for gating `clk`. Example 1-11 shows an incomplete conditional statement, while Example 1-12 shows a complete conditional statement. Low Power can use both conditions to insert clock-gating logic.

Example 1-11 Modeling Incomplete Conditional Statements (Verilog)

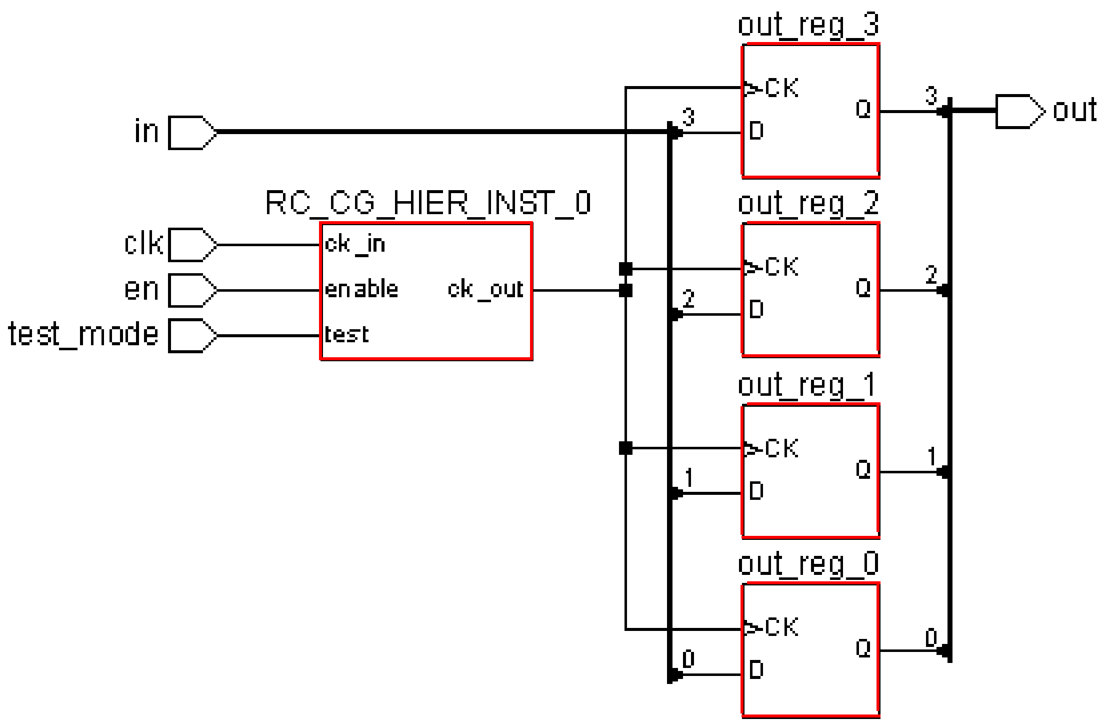
```
module ex1 (in, out, en, clk);
    input clk, en;
    input [3:0] in;
    output [3:0] out;
    reg [3:0] out;
    always @ (posedge clk) begin
        if (en)
            out <= in;
    end
endmodule
```

Example 1-12 Modeling Complete Conditional Statements (Verilog)

```
module ex1a (in, out, en, clk);
    input en, clk;
    input [3:0] in;
    output [3:0] out;
    reg [3:0] out;
    always @ (posedge clk) begin
        if (en)
            out <= in;
        else
            out <= out;

        end
endmodule
```

Figure 1-10 shows the mapped netlist for Example 1-11 and Example 1-12 when the `lp_insert_clock_gating` attribute is set to `true`.

Figure 1-10 Complete Conditional Statement (Verilog)



Modeling Combinational Logic in VHDL

The RTL Compiler software synthesizes combinational logic to implement a variable or signal under any of the following conditions:

- The variable or signal is unconditionally assigned a value before it is used and whenever any of the signals on the right side of the expression change. See Example 1-13 and the corresponding schematic shown in Figure 1-11.
- The variable or signal is conditionally assigned a value under all possible conditions whenever any of the signals in the right side of the expression change. See Example 1-14 and the corresponding schematic shown in Figure 1-12.

Example 1-13 Modeling Combinational Logic With an Unconditional Assignment (VHDL)

```
library ieee;
use ieee.numeric_std.all;

entity comb1 is
    port(
        a, b: in unsigned(3 downto 0);
        z : out unsigned(3 downto 0));
end;

architecture rtl of comb1 is begin
    process(a, b)
        begin
        z <= a + b;
    end process;
end;
```

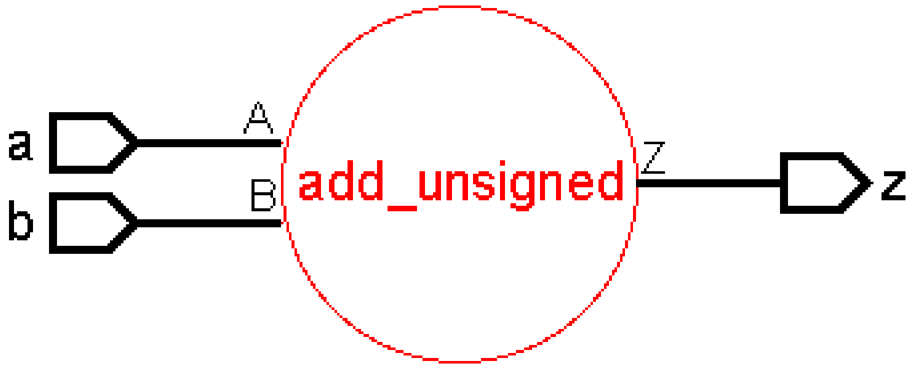
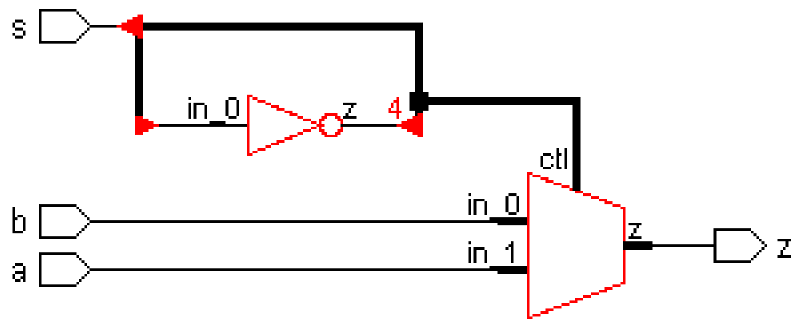


Figure 1-11 Elaborated Netlist for Example 1-13 (VHDL)

Example 1-14 Synthesizing Combinational Logic with a Conditional Assignment (VHDL)

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

entity comb2 is
    port(
        a, b, s: in std_logic;
        z : out std_logic);
end;

architecture rtl of comb2 is begin
    process(a, b, s) begin
        if (s = '1') then
            z <= a;
        else
            z <= b;
        end if;
    end process;
end;
```

Figure 1-12 Combinational Logic with a Conditional Assignment (VHDL)



Modeling Arithmetic Components (Verilog and VHDL)

Using HDL operators, such as `+` (add) or `*`(multiply) to infer arithmetic components is functionally equivalent to explicitly instantiating the corresponding `CW_add` and `CW_mult` ChipWare components. However, this is not true for division-related HDL operators, such as `/` and `%` in Verilog HDL, and `mod` and `rem` in VHDL.The core division functionality is the same as the `CW_div` component, but the exception handling is not.

See *ChipWare in Encounter RTL Compiler* for detailed information on ChipWare components.

Modeling Adders

Modeling an Unsigned Adder in Verilog and VHDL

Example 1-15 Modeling an Unsigned Adder in Verilog

```
module unsigned_add (y, a, b);
    parameter w = 16;
    input [w-1:0] a, b;
    output [w-1:0] y;
    assign y = a + b;
endmodule
```

Example 1-16 Modeling an Unsigned Adder in VHDL

```
library ieee;
use ieee.numeric_std.all;
entity unsigned_add is
    generic (w : integer := 4);
    port (y : out unsigned (w-1 downto 0);
        a, b : in  unsigned (w-1 downto 0)  );
end unsigned_add;
architecture rtl of unsigned_add is
begin
    y <= a + b;
end rtl;
```

Example 1-17 Modeling an Unsigned Adder in VHDL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity unsigned_add is
    generic (w : integer := 4);
    port (y : out std_logic_vector (w-1 downto 0);
        a, b : in std_logic_vector (w-1 downto 0)  );
end unsigned_add;
architecture rtl of unsigned_add is
begin
    y <= a + b;
end rtl;
```

Modeling a Signed Adder in Verilog and VHDL

Example 1-18 Modeling a Signed Adder in Verilog

```
module signed_add (y, a, b);
    parameter w = 16;
    input signed [w-1:0] a, b;
    output signed [w-1:0] y;
    assign y = a + b;
endmodule
```

Example 1-19 Modeling a Signed Adder in VHDL

```
library ieee;
use ieee.numeric_std.all;
entity signed_add is
    generic (w : integer := 16);
    port (y : out signed (w-1 downto 0);
        a, b : in signed (w-1 downto 0)  );
end signed_add;
architecture rtl of signed_add is
begin
    y <= a + b;
end rtl;
```

Example 1-20 Modeling a Signed Adder in VHDL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
entity signed_add is
    generic (w : integer := 16);
    port (y : out std_logic_vector (w-1 downto 0);
        a, b : in std_logic_vector (w-1 downto 0)  );
end signed_add;
architecture rtl of signed_add is
begin
    y <= a + b;
end rtl;
```

Modeling Subtractors

Modeling an Unsigned Subtractor in Verilog and VHDL

Example 1-21 Modeling an Unsigned Subtractor in Verilog

```
module unsigned_subtract (y, a, b);
    parameter w = 16;
    input [w-1:0] a, b;
    output [w-1:0] y;
    assign y = a - b;
endmodule
```

Example 1-22 Modeling an Unsigned Subtractor in VHDL

```
library ieee;
use ieee.numeric_std.all;

entity unsigned_subtract is
    generic (w : integer := 16);
    port (y : out unsigned (w-1 downto 0);
        a, b : in unsigned (w-1 downto 0)  );
end unsigned_subtract;
architecture rtl of unsigned_subtract is
begin
    y <= a - b;
    end rtl;
```

Example 1-23 Modeling an Unsigned Subtractor in VHDL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity unsigned_subtract is
    generic (w : integer := 16);
    port (y    : out std_logic_vector (w-1 downto 0);
        a, b : in std_logic_vector (w-1 downto 0)  );
end unsigned_subtract;
architecture rtl of unsigned_subtract is
begin
    y <= a - b;
end rtl;
```

Modeling a Signed Subtractor in Verilog and VHDL

Example 1-24 Modeling a Signed Subtractor in Verilog

```
module signed_subtract (y, a, b);
    parameter w = 16;
    input signed [w-1:0] a, b;
    output signed [w-1:0] y;
    assign y = a - b;
endmodule
```

Example 1-25 Modeling an Signed Subtractor in VHDL

```
library ieee;
use ieee.numeric_std.all;
entity signed_subtract is
    generic (w : integer := 16);
    port (y : out signed (w-1 downto 0);
        a, b : in signed (w-1 downto 0)  );
end signed_subtract;
architecture rtl of signed_subtract is
begin
    y <= a - b;
end rtl;
```

Example 1-26 Modeling an Signed Subtractor in VHDL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
entity signed_subtract is
    generic (w : integer := 16);
    port (y : out std_logic_vector (w-1 downto 0);
        a, b : in std_logic_vector (w-1 downto 0)  );
end signed_subtract;
architecture rtl of signed_subtract is
begin
    y <= a - b;
end rtl;
```

Modeling a Negation Subtractor in Verilog and VHDL

Example 1-27 Modeling a Negation Subtractor in Verilog

```
module unary_minus (y, a);
    parameter w = 16;
    input signed [w-1:0] a;
    output signed [w:0] y;
    assign y = -a;
endmodule
```

Example 1-28 Modeling a Negation Subtractor in VHDL

```
library ieee;
use ieee.numeric_std.all;
entity unary_minus is
    generic (w : integer := 16);
    port (y : out signed (w-1 downto 0);
        a : in signed (w-1 downto 0)  );
end unary_minus;
architecture rtl of unary_minus is
begin
    y <= -a;
end rtl;
```

Example 1-29 Modeling a Negation Subtractor in VHDL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
entity unary_minus is
    generic (w : integer := 16);
    port (y : out std_logic_vector (w-1 downto 0);
    a : in std_logic_vector (w-1 downto 0)  );
end unary_minus;
architecture rtl of unary_minus is
begin
    y <= -a;
end rtl;
```

Modeling an Absolute Value in VHDL

Example 1-30 Modeling an Absolute Value in VHDL

```
library ieee;
use ieee.numeric_std.all;
entity absolute_value is
    generic (w : integer := 16);
    port (y : out signed (w-1 downto 0);
        a : in signed (w-1 downto 0)  );
end absolute_value;
architecture rtl of absolute_value is
begin
    y <= abs(a);
end rtl;
```

Example 1-31 Modeling an Absolute Value in VHDL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
entity absolute_value is
    generic (w : integer := 16);
    port (y : out std_logic_vector (w-1 downto 0);
        a : in std_logic_vector (w-1 downto 0)  );
end absolute_value;
architecture rtl of absolute_value is
begin
    y <= abs(a);
end rtl;
```

Modeling Multipliers

Modeling an Unsigned Multiplier in Verilog and VHDL

Example 1-32 Modeling an Unsigned Multiplier in Verilog

```
module unsigned_multiply (y, a, b);
    parameter wA = 16, wB = 16;
    input [wA-1:0] a;
    input [wB-1:0] b;
    output [wA+wB-1:0] y;
    assign y = a * b;
endmodule
```

Example 1-33 Modeling an Unsigned Multiplier in VHDL

```
library ieee;
use ieee.numeric_std.all;
entity unsigned_multiply is
    generic (wA : integer := 16;
             wB : integer := 16);
    port (y : out unsigned (wA+wB-1 downto 0);
          a : in unsigned (wA-1 downto 0);
          b : in unsigned (wB-1 downto 0) );
end unsigned_multiply;

architecture rtl of unsigned_multiply is
begin
    y <= a * b;
end rtl;
```

Example 1-34 Modeling an Unsigned Multiplier in VHDL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity unsigned_multiply is
    generic (wA : integer := 16;
             wB : integer := 16);
    port (y : out std_logic_vector (wA+wB-1 downto 0);
          a : in std_logic_vector (wA-1 downto 0);
          b : in std_logic_vector (wB-1 downto 0) );
end unsigned_multiply;
architecture rtl of unsigned_multiply is
begin
    y <= a * b;
end rtl;
```

Modeling a Signed Multiplier in Verilog and VHDL

Example 1-35 Modeling a Signed Multiplier in Verilog

```
module signed_multiply (y, a, b);
    parameter wA = 16, wB = 16;
    input signed [wA-1:0] a;
    input signed [wB-1:0] b;
    output signed [wA+wB-1:0] y;
    assign y = a * b;
endmodule
```

Example 1-36 Modeling a Signed Multiplier in VHDL

```
library ieee;
use ieee.numeric_std.all;
entity signed_multiply is
    generic (wA : integer := 16;
             wB : integer := 16);
    port (y : out signed (wA+wB-1 downto 0);
          a : in signed (wA-1 downto 0) ;
          b : in signed (wB-1 downto 0) );
end signed_multiply;

architecture rtl of signed_multiply is
begin
    y <= a * b;
end rtl;
```

Example 1-37 Modeling a Signed Multiplier in VHDL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
entity signed_multiply is
    generic (wA : integer := 16;
             wB : integer := 16);
    port (y : out std_logic_vector (wA+wB-1 downto 0);
          a : in std_logic_vector (wA-1 downto 0);
          b : in std_logic_vector (wB-1 downto 0) );
end signed_multiply;
architecture rtl of signed_multiply is
begin
    y <= a * b;
end rtl;
```

Modeling Dividers

Modeling an Unsigned Divider in Verilog and VHDL

Example 1-38 Modeling an Unsigned Divider in Verilog

```
module unsigned_divide (y, a, b);
    parameter wA = 16, wB = 6;
    input [wA-1:0] a;
    input [wB-1:0] b;
    output [wA-1:0] y;
    assign y = a / b;
endmodule
```

Example 1-39 Modeling an Unsigned Divider in VHDL

```
library ieee;
use ieee.numeric_std.all;
entity unsigned_divide is
    generic (wA : integer := 16;
             wB : integer := 6);
    port (y : out unsigned (wA-1 downto 0);
          a : in unsigned (wA-1 downto 0);
          b : in unsigned (wB-1 downto 0) );
end unsigned_divide;

architecture rtl of unsigned_divide is
begin
    y <= a / b;
end rtl;
```

Modeling a Signed Divider in Verilog and VHDL

Example 1-40 Modeling a Signed Divider in Verilog

```
module signed_divide (y, a, b);
    parameter wA = 16, wB = 6;
    input signed [wA-1:0] a;
    input signed [wB-1:0] b;
    output signed [wA-1:0] y;
    assign y = a / b;
endmodule
```

Example 1-41 Modeling a Signed Divider in VHDL

```
library ieee;
use ieee.numeric_std.all;
entity signed_divide is
    generic (wA : integer := 16;
             wB : integer := 6);
    port (y : out signed (wA-1 downto 0);
          a : in signed (wA-1 downto 0);
          b : in signed (wB-1 downto 0) );
end signed_divide;

architecture rtl of signed_divide is
begin
    y <= a / b;
end rtl;
```

Modeling an Unsigned Modulus in Verilog and VHDL

Example 1-42 Modeling an Unsigned Modulus in Verilog

```
module unsigned_modulus (y, a, b);
    parameter wA = 16, wB = 6;
    input [wA-1:0] a;
    input [wB-1:0] b;
    output [wB-1:0] y;
    assign y = a % b;
    endmodule
```

Example 1-43 Modeling an Unsigned Modulus in VHDL

```
library ieee;
use ieee.numeric_std.all;
entity unsigned_modulus is
    generic (wA : integer := 16;
             wB : integer := 6);
    port (y : out unsigned (wB-1 downto 0);
          a : in unsigned (wA-1 downto 0);
          b : in unsigned (wB-1 downto 0) );
end unsigned_modulus;

architecture rtl of unsigned_modulus is
begin
    y <= a mod b;
end rtl;
```

Modeling a Signed Modulus in Verilog and VHDL

Example 1-44 Modeling a Signed Modulus in Verilog

```
module signed_modulus (y, a, b);
    parameter wA = 16, wB = 6;
    input signed [wA-1:0] a;
    input signed [wB-1:0] b;
    output signed [wB-1:0] y;
    assign y = a % b;
endmodule
```

Example 1-45 Modeling a Signed Modulus in VHDL

```
library ieee;
use ieee.numeric_std.all;
entity signed_modulus is
    generic (wA : integer := 16;
             wB : integer := 6);
    port (y : out signed (wB-1 downto 0);
          a : in signed (wA-1 downto 0);
          b : in signed (wB-1 downto 0) );
end signed_modulus;

architecture rtl of signed_modulus is
begin
    y <= a mod b;
end rtl;
```

Modeling an Unsigned and Signed Remainder in VHDL

Example 1-46 Modeling an Unsigned Remainder in VHDL

```
library ieee;
use ieee.numeric_std.all;
entity unsigned_remainder is
    generic (wA : integer := 16;
             wB : integer := 6);
    port (y : out unsigned (wB-1 downto 0);
          a : in unsigned (wA-1 downto 0);
          b : in unsigned (wB-1 downto 0) );
end unsigned_remainder;

architecture rtl of unsigned_remainder is
begin
    y <= a rem b;
end rtl;
```

Example 1-47 Modeling a Signed Remainder in VHDL

```
library ieee;
use ieee.numeric_std.all;
entity signed_remainder is
    generic (wA : integer := 16;
             wB : integer := 6);
    port (y : out signed (wB-1 downto 0);
          a : in signed (wA-1 downto 0);
          b : in signed (wB-1 downto 0));
end signed_remainder;

architecture rtl of signed_remainder is
begin
    y <= a rem b;
end rtl;
```

Using case Statements for Multi-Way Branching

Use a `case` statement for multi-way branching in a functional description. When a `case` statement is used as a decoder to assign one of several different values to a variable, the ensuing logic is implemented as combinational or sequential logic based on whether the variable is assigned a value in all branches of the `case` statement. RTL Compiler automatically determines whether a `case` statement is `full` or `parallel`. A `case` statement is `full` if all possible `case` items are specified. A `case` statement is `parallel` if none of the `case` statement conditions overlap and are mutually exclusive. If automatic determination of `full` or `parallel case` is not possible, use the `full` and `parallel` case pragmas (see Specifying the full_case Pragma on page 90, and Specifying the parallel_case Pragma on page 92).

Using case Statements in Verilog

The following sections describe the impact on synthesis for different use models and types of `case` statements.

Using an Incomplete case Statement to Infer a Latch

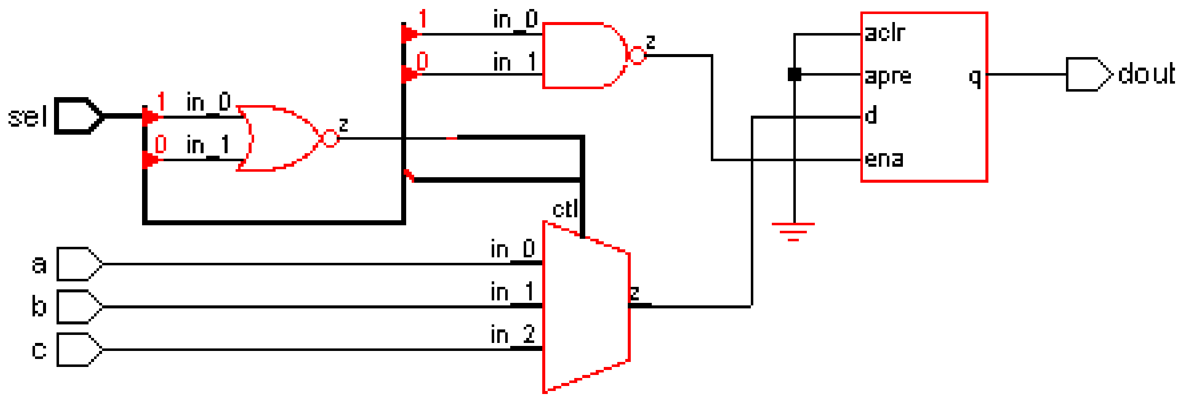
When a `case` statement does not specify all possible `case` condition values, a latch is inferred. If RTL Compiler determines that the case is not `full`, then it uses a latch to implement a state transition table, as shown in Example 1-48.

Example 1-48 Modeling a State Transition Table to Infer a Latch (Verilog)

```
module case_latch(dout,sel,a,b,c);
    input [1:0] sel;
    input a,b,c;
    output dout;
    reg dout;

    always @(a or b or c or sel)
        begin
            case (sel)
                2'b00 : dout = a;
                2'b01 : dout = b;
                2'b10 : dout = c;
            endcase
        end
endmodule
```

Figure 1-13 shows the corresponding schematic for Example 1-48.

Figure 1-13 State Transition Table to Infer a Latch Schematic (Verilog)



Using a Fully Specified case Statement to Prevent a Latch

Use one of the following methods to assign a default value to `dout`.

- Initialize the `dout` variable to a default value, then use a `case` statement to modify it, as shown in the Example 1-49.

Example 1-49 Preventing a Latch by Assigning a Default Value (Verilog)

```
module case_latch(dout,sel,a,b,c);
    input [1:0] sel;
    input a,b,c;
    output dout;
    reg dout;

    always @(a or b or c or sel)
        begin
        dout = 1'b0;
        case (sel)
            2'b00 : dout = a;
            2'b01 : dout = b;
            2'b10 : dout = c;
        endcase
    end
endmodule
```

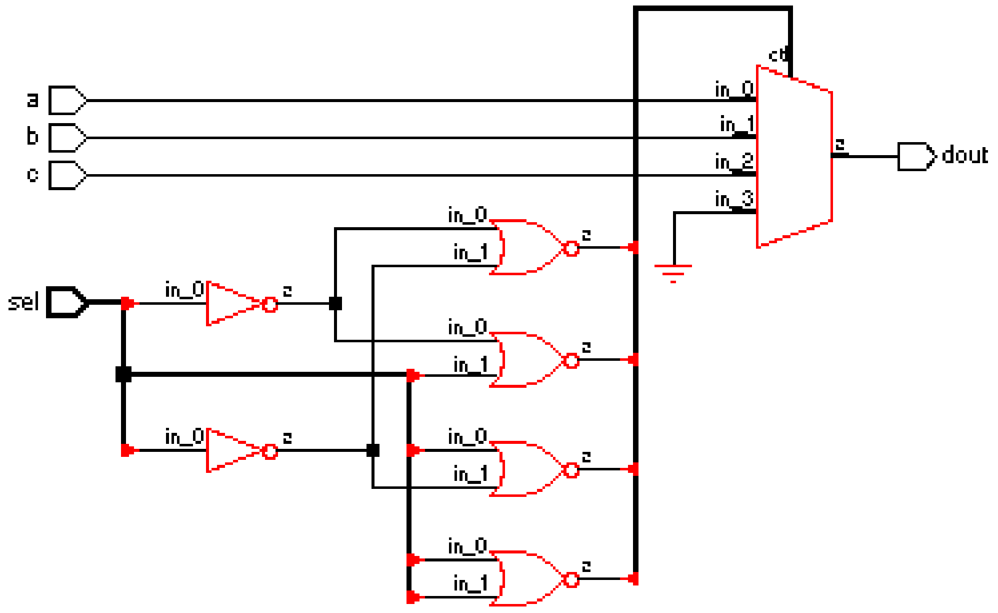
- Use the default case in the `case` statement to capture all the remaining cases where the next state variable is assigned a value, as shown in Example 1-50.

Example 1-50 Preventing a Latch Using the Default Case in a Case Statement (Verilog)

```
module case_default(dout,sel,a,b,c);
    input [1:0] sel;
    input a,b,c;
    output dout;
    reg dout;

    always @(a or b or c or sel) begin
        case (sel)
            2'b00 : dout = a;
            2'b01 : dout = b;
            2'b10 : dout = c;
            default : dout = 1'b0;
        endcase
    end
endmodule
```

Figure 1-14 shows the corresponding schematic for Example 1-49 and Example 1-50.

Figure 1-14 Preventing a Latch Using the Default Case Schematic (Verilog)



You can also use the `full_case` synthesis pragma. If the `full_case` synthesis pragma is incorrectly used, RTL simulation and gate-level simulation results in a mismatch. When an unspecified case occurs during the simulation, the RTL model will preserve the value of the variable because it is a `reg` type variable. The gate-level simulation uses the implemented combinational logic, possibly generating an incorrect output. The simulation results between functional and gate level models may mismatch if this synthesis pragma is used.

Using casez and casex Statements in Verilog to Treat x, z and ? Like Don’t Cares

Use `casex` and `casez` statements to treat `x`, `z` and `?` values like don’t care conditions when comparing for the matching `case`. These statements are treated like `case` statements with the following differences:

- Use a `casez` statement to treat `z` and `?` as a don’t care condition.
- Use a `casex` statement to treat `x`, `z` and `?` as a don’t care condition.

Example 1-51 shows a `casez` statement using don’t care conditions to mask three of the four bits in the decoding select line (`input sel`).

Example 1-51 Modeling Don’t Care Conditions in a Casez Statement (Verilog)

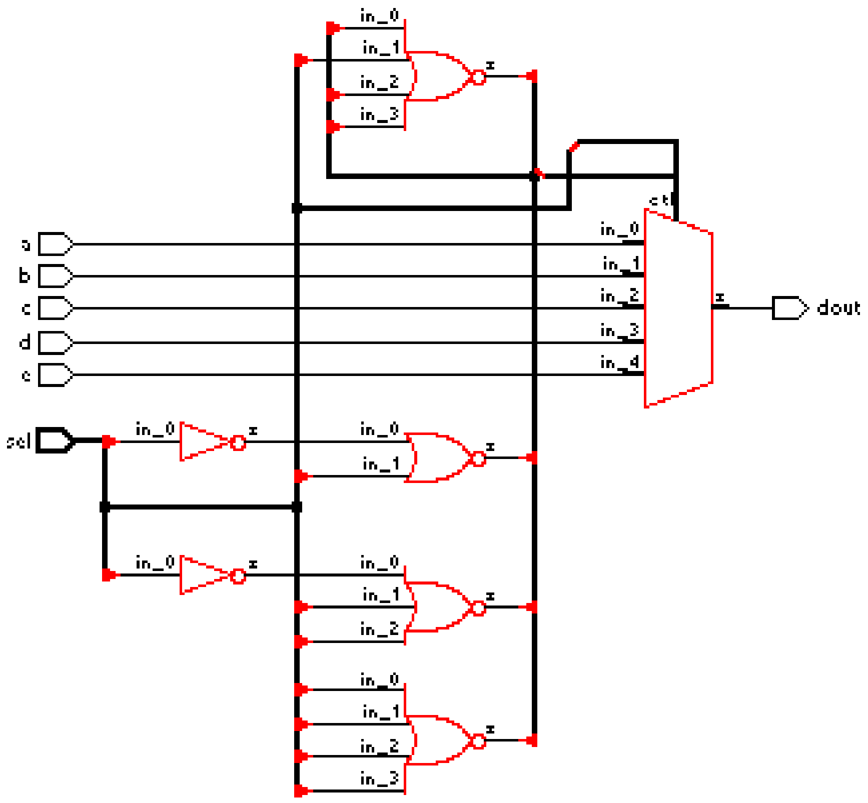
```
module case_z(dout,sel,a,b,c,d,e);
    input [3:0] sel;
    input a,b,c,d,e;
    output dout;
    reg dout;

    always @(a or b or c or d or e or sel) begin
        casez (sel)
        4'b0000 : dout = a;
        4'b???1 : dout = b;
        4'b??1? : dout = c;
        4'b?1?? : dout = d;
        4'b1??? : dout = e;
    endcase
    end
endmodule
```

In the example, `dout` is set to `b` if sel[0] = 1, regardless of the values of sel[3], sel[2] and sel[1]; `dout` is set to `c` only if sel[0] = 0 and sel[1] = 1, regardless of the values of sel[3] and sel[2].

One or more `case` items overlap (not parallel) and a priority encoder is required to implement the equivalent hardware.

Figure 1-15 shows the corresponding schematic for Example 1-51.

Figure 1-15 Don’t Care Conditions in a casez Statement Schematic (Verilog)



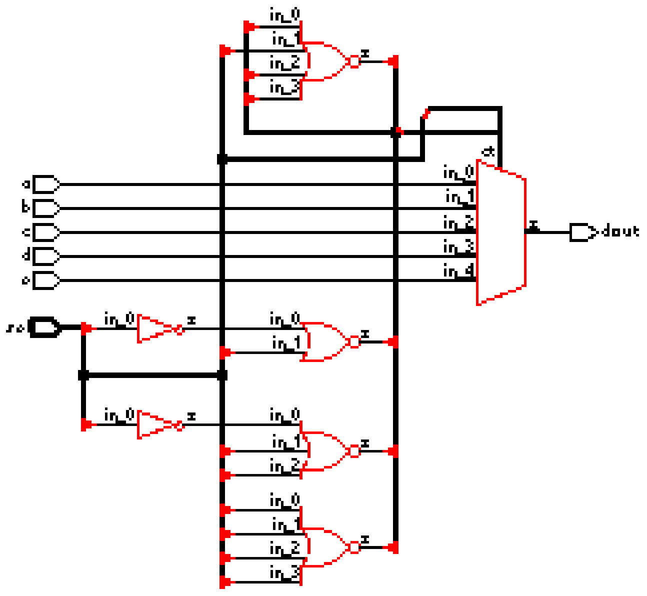
Example 1-52 shows a `casex` statement using don’t care conditions in the same manner as the `casez` statement. The difference between the two models is that the `casex` statement masks three bits of the select line that would match `x`, `z`, or `?`, but the `casez` statement will not mask `x` in the select line.

Example 1-52 Modeling Don’t Care Conditions in a Casex Statement (Verilog)

```
module case_x(dout,sel,a,b,c,d,e);
    input [3:0] sel;
    input a,b,c,d,e;
    output dout;
    reg dout;
    always @(a or b or c or d or e or sel)
        begin
        casex (sel)
        4'bxxx1 : dout = a;
        4'bxx1x : dout = b;
        4'bx1xx : dout = c;
        4'b1xxx : dout = d;
        default : dout = e;
        endcase
    end
endmodule
```

Figure 1-16 shows the corresponding schematic for Example 1-52.

Figure 1-16 Don't Care Conditions in a Casex Statement Schematic (Verilog)



Using Case Statements in VHDL

Using an Incomplete case Statement to Infer a Latch

If a `case` statement specifies only some of the values that the `case` expression can possibly have, then a latch is inferred, as shown in Example 1-53.

Example 1-53 Modeling a State Transition Table to Infer a Latch (VHDL)

```
signal curr_state, next_state, modifier:std_logic_vector(2 downto 0);
process(curr_state, modifier)
begin
    case curr_state is
        when "000" => next_state <= "100" or modifier;
        when "001" => next_state <= "110" or modifier;
        when "010" => next_state <= "001" and modifier;
        when "100" => next_state <= "101" and modifier;
        when "101" => next_state <= "010" or modifier;
        when "110" => next_state <= "000" and modifier;
        when others => null;
    end case;
end process;
```

The `next_state` signal is assigned a new value if `curr_state` is any one of the six values specified. For the other two possible states, the `next_state` signal retains its previous value. This behavior causes RTL Compiler to infer a three bit latch for `next_state`.

Using a Complete case Statement to Prevent a Latch

If you do not want RTL Compiler to infer a latch, the `next_state` signal must be assigned a value under all situations. In other words, the `next_state` signal must have a default value. Assign the `next_state` signal a value unconditionally then modify it by a `case` statement, as shown in Example 1-54.

Example 1-54 Assigning the next_state Signal a Value to Prevent a Latch (VHDL)

```
    process(curr_state, modifier)
    begin
        next_state <= "000";
        case curr_state is
            when "000" => next_state <= "100" or modifier;
            when "001" => next_state <= "110" or modifier;
            when "010" => next_state <= "001" and modifier;
            when "100" => next_state <= "101" and modifier;
            when "101" => next_state <= "010" or modifier;
            when "110" => next_state <= "000" and modifier;
            when others => null;
    end case;
end process;
```

Use the `others` clause in the `case` statement to capture all the remaining cases where `next_state` is assigned a value, as shown in Example 1-55.

Example 1-55 Using the Others Clause in the Case Statement (VHDL)

```
signal curr_state,next_state,modifier:
    std_logic_vector(2 downto 0);
    process(curr_state, modifier)
    begin
        case curr_state is
            when "000" => next_state <= "100" or modifier;
            when "001" => next_state <= "110" or modifier;
            when "010" => next_state <= "001" and modifier;
            when "100" => next_state <= "101" and modifier;
            when "101" => next_state <= "010" or modifier;
            when "110" => next_state <= "000" and modifier;
            when others => next_state <= "000";
    end case;
end process;
```

Replacing a Nested if-else-if Statement With a Functionally Equivalent case Statement

Example 1-56 shows a nested if-else-if statement. In general, it is better to use a case statement to replace a functionally equivalent nested if-else-if statement, as shown in Example 1-57.

Example 1-56 Modeling a Nested if-else-if Statement (VHDL)

```
if (stat(23 downto 19) = 3 ) then result := 1;
    elsif (stat(23 downto 19) = 5) then result := 2;
    elsif (stat(23 downto 19) = 6) then result := 3;
    elsif (stat(23 downto 19) = 9) then result := 4;
    elsif (stat(23 downto 19) = 10)then result := 5;
    elsif (stat(23 downto 19) = 12)then result := 6;
    else
        result := 0;
end if;.
```

You can improve the QoS by changing the coding style to a functionally equivalent case statement, as shown in Example 1-57. Although RTL Compiler can automatically transform certain if-else-if statements into equivalent case-statements, it is better to model the RTL using a case statement whenever possible.

Example 1-57 Replacing a nested if-else-if Statement With a Functionally Equivalent Case Statement (VHDL)

```
case stat(23 downto 19) is
        when "00011" => result := 1;
        when "00101" => result := 2;
        when "00110" => result := 3;
        when "01001" => result := 4;
        when "01010" => result := 5;
        when "01100" => result := 6;
        when others" => result := 6;
end case;
```

Using a for Statement to Describe Repetitive Operations

Using a for Statement in Verilog

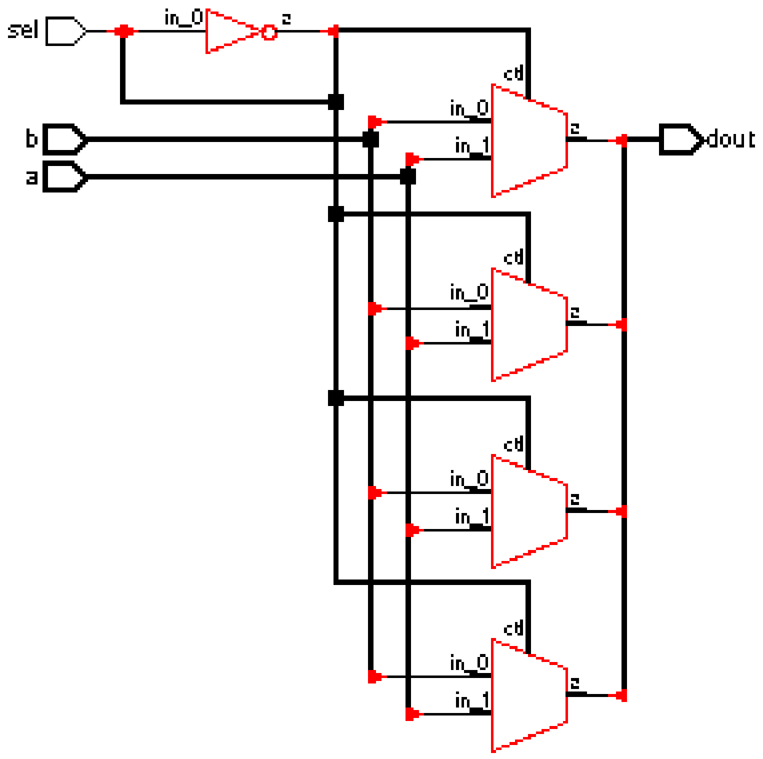
Example 1-58 uses the `for` statement where `i` is declared as an integer and `dout` is a 4-bit register. The `for` statement is expanded to repeat the operations over the range of the index.

Example 1-58 Modeling a for Statement to Describe Repetitive Operations (Verilog)

```
module for_loop(dout,sel,a,b,)
    input sel;
    input [3:0] a,b;
    output [3:0] dout;
    reg [3:0] dout;
    integer i;
    always @(a or b or sel)
        begin
        for (i = 0; i <= 3; i = i + 1)
        begin
            if (sel)
                dout[i] = a[3 -i];
            else
                dout[i] = b[i];
        end
    end
endmodule
```

Figure 1-17 shows the corresponding schematic for Example 1-58.

Figure 1-17 Using the for Statement to Describe Repetitive Operations Schematic (Verilog)



Supported Forms of the for Statement

```
for (index = low; index < high; index = index + step)
for (index = low; index <= high; index = index + step)
for (index = high; index > low; index = index - step)
for (index = high; index >= low; index = index - step)
```

The `index` is declared as an `integer` or a `reg`; `high`, `low` and `step` are integers, and `high` must be greater than or equal to `low`.

Note: `High`, `low`, and `step` must evaluate to constant numbers during synthesis. An error message is generated if one of them does not evaluate to a constant number.

A `for` statement can be nested inside another `for` statement, but it cannot contain any form of timing control or event control, as shown in Example 1-59.

Example 1-59 Illegal Use of the for Statement

```
for (i = 0; i <= 7; i = i + 1)
    @(posedge clk) out[7-i] <= in[i] ;
```

Using a for Statement in VHDL

Using a for loop Statement to Describe Repetitive Operations

The following are the supported `for loop` statement forms:

`for` *index* `in` *start_val* `to` *end_val* `loop`

`for` *index* `in` *start_val* `downto` *end_val* `loop`

`for` *index* `in` *discrete_subtype_indication* `loop`

Use a `for loop` statement to describe repetitive operations, as shown in Example 1-60.

Example 1-60 Using a for loop Statement to Describe Repetitive Operations (VHDL)

```
process(in_sig, out_sig)
begin
    for i in 0 to 7 loop
        out_sig(7-i) <= in_sig(i);
    end loop;
end process;
```

Where `i` is declared as `integer` and `out_sig` and `in_sig` are eight bit signals, the `for loop` is expanded to repeat the operations over the range of the index. Therefore, the `for` statement model shown in Example 1-60 is treated in an equivalent manner to the following operations:

```
out_sig(7) <= in_sig(0);
out_sig(6) <= in_sig(1);
out_sig(5) <= in_sig(2);
out_sig(4) <= in_sig(3);
out_sig(3) <= in_sig(4);
out_sig(2) <= in_sig(5);
out_sig(0) <= in_sig(6);
```

Use a `for loop` statement to store all the bits of a vector (`in_sig`) in reverse order, as shown in Example 1-61.

Example 1-61 Reversing and Assigning Bits of curr_state to next_state (VHDL)

```
signal curr_state: std_logic_vector(2 downto 0);
signal next_state: std_logic_vector(2 downto 0);
process(curr_state)
    subtype INT02 is integer range 0 to 2;
begin
    for I in INT02 loop
        next_state(2-I) <= curr_state(I);
    end loop;
end process;
```

Modeling Library Cell Instances with Complex Ports

RTL Compiler can map instances with complex HDL ports, such as SystemVerilog `struct` types, to library cells with complex bus ports.

Consider the following RTL file:

```
typedef struct packed {
logic [2:0] x;
logic [2:0] y;
} new_struct;

module test(input new_struct in_logic);
    lib_test i_new(.in_logic(in_logic));
endmodule;
```

Assume that the definition for `lib_test` in the .lib file has two buses defined as `bus (in_logic[x])` and `bus (in_logic[y])`, RTL Compiler can successfully map the `packed struct` type ports to the corresponding lib cells.

Modeling Logic Abstracts

A logic abstract refers to a skeletal description of a module that only specifies the name of the module and the name, width, and direction of the module's ports. A logic abstract does not describe the contents or the function of the module.

Inferring a Logic Abstract From the RTL in Verilog

You can infer a logic abstract in one of the following ways:

- Infer a logic abstract from an empty Verilog module description that lists the ports but has no other information, such as no concurrent statements or sequential blocks. Example 1-62 infers a logic abstract from the `my_sub_empty` module:

Example 1-62 Inferring a Logic Abstract From an Empty Verilog Module Description

```
module my_sub_empty (p, q, x);
    parameter w = 4;
    input [w-1:0] p, q;
    output [w-1:0] x;
endmodule
module my_top (a, b, c, y);
    parameter w = 4;
    input [w-1:0] a, b, c;
    wire [w-1:0] t;
    output [w-1:0] y;
    my_sub_empty #(w) u1 (.p(a), .q(b), .x(t));
    assign y = t | c;
endmodule
```

- Infer a logic abstract from a SystemVerilog external declaration of a module where the definition of the external module is not found in the input HDL. This is different from a typical unresolved reference since the input and output direction and bit-range of ports of the instantiated sub-module are known. It is unresolved since the definition of that sub-module is missing. The RTL coding style shown in Example 1-63, infers a logic abstract for the `my_sub_gray` module.

Example 1-63 Inferring a Logic Abstract for an External Module with Missing Module

```
extern module my_sub_gray #(parameter w = 4)
    (input [w-1:0] p, q, output [w-1:0] x);
module my_top (a, b, c, y);
    parameter w = 4;
    input [w-1:0] a, b, c;
    wire [w-1:0] t;
    output [w-1:0] y;
    my_sub_gray #(w) u1 (a, b, t);
    assign y = t | c;
endmodule
```

Inferring a Logic Abstract From the RTL in VHDL

You can infer a logic extract in VHDL in one of the following ways:

- Infer a logic abstract from a VHDL entity whose architecture is missing in the RTL.

  The RTL coding style shown in Example 1-64, infers a logic abstract for the `my_sub_empty` component.

Example 1-64 Inferring a Logic Abstract From a VHDL Entity with Missing Architecture

```
library ieee;
use ieee.std_logic_1164.all;
entity my_sub_empty is
    generic (w : integer := 4);
    port (p, q : in std_logic_vector (w-1 downto 0);
        x : out std_logic_vector (w-1 downto 0) );
end my_sub_empty;

library ieee;
use ieee.std_logic_1164.all;
entity my_top is
    generic (w : integer := 4);
    port (a, b, c : in std_logic_vector (w-1 downto 0);
        y : out std_logic_vector (w-1 downto 0)    );
end my_top;
architecture rtl of my_top is
    signal t : std_logic_vector (w-1 downto 0);
    component my_sub_empty
        generic (w : integer := 4);
        port (p, q : in std_logic_vector (w-1 downto 0);
            x : out std_logic_vector (w-1 downto 0) );
    end component;
begin
    u1: my_sub_empty generic map (w => w)
        port map (p => a, q => b, x => t);
    y <= t or c;
end rtl;
```

- Infer a logic abstract from an empty VHDL architecture description that has ports but no other information, such as no concurrent statements or process blocks.

  The RTL coding style, shown in Example 1-65, infers a logic abstract from the `my_sub_empty` component.

Example 1-65 Inferring a Logic Abstract From an Empty VHDL Architecture

```
library ieee;
use ieee.std_logic_1164.all;
entity my_sub_empty is
    generic (w : integer := 4);
    port (p, q : in std_logic_vector (w-1 downto 0);
        x : out std_logic_vector (w-1 downto 0) );
end my_sub_empty;
architecture rtl of my_sub_empty is
begin
end rtl;
library ieee;
use ieee.std_logic_1164.all;
use work.my_sub_empty;
entity my_top is
    generic (w : integer := 4);
    port (a, b, c : in std_logic_vector (w-1 downto 0);
        y : out std_logic_vector (w-1 downto 0)    );
end my_top;
architecture rtl of my_top is
    signal t : std_logic_vector (w-1 downto 0);
begin
    u1: entity my_sub_empty generic map (w)
        port map (a, b, t);
    y <= t or c;
end rtl;
```

- Infer a logic abstract from a component instantiation where the component declaration statement exists as usual, but the entity and architecture definition of the declared component are not found in the input HDL code.

  This is different from a typical unresolved reference since the input and output direction and bit-range of ports of the instantiated component are known. It is unresolved since the entity and architecture of that component are missing. The RTL coding style, shown in Example 1-66 infers a logic abstract for the `my_sub_gray` component.

Example 1-66 Inferring a Logic Abstract From a Component Instantiation With Missing Entity and Architecture

```
library ieee;
use ieee.std_logic_1164.all;
entity my_top is
    generic (w : integer := 4);
    port (a, b, c : in std_logic_vector (w-1 downto 0);
        y : out std_logic_vector (w-1 downto 0)    );
end my_top;
architecture rtl of my_top is
    signal t : std_logic_vector (w-1 downto 0);
    component my_sub_gray
        generic (w : integer := 4);
        port (p, q : in std_logic_vector (w-1 downto 0);
            x : out std_logic_vector (w-1 downto 0) );
    end component;
begin
    u1: my_sub_gray generic map (w) port map (a, b, t);
    y <= t or c;
end rtl;
```

Interpreting a Logic Abstract in Verilog or VHDL

➤ In Verilog, use the `hdl_use_techelt_first` attribute when there is a user-defined module (empty or not) that shares the same name as a technology element in the library.

If this attribute is set to `false`, then RTL Compiler picks up the user module. If this attribute is set to `true`, then RTL Compiler picks up the tech element.

If a logic abstract is inferred from a SystemVerilog external module statement whose module is missing, as shown in Example 1-63, then it goes through the library look-up process. If a library cell of the same name is found, then it becomes an instance of that library cell. If not, then it becomes an unresolved reference in the design.

In Verilog, if a logic abstract is inferred from either an empty module, as shown in Example 1-62, or in VHDL, if a logic abstract is inferred from either an entity without an architecture, as shown in Example 1-64, or an entity whose architecture is empty, as shown in Example 1-65, then its interpretation is affected by the `hdl_use_techelt_first` and `hdl_infer_unresolved_from_logic_abstract` attributes in the following way:

- If either attribute is set to `true`, then the logic abstract goes through the library look-up process. If a library cell of the same name is found, then the logic abstract becomes an instance of that library cell. If not, the process continues.
- If the `hdl_infer_unresolved_from_logic_abstract` attribute is set to `true`, then the logic abstract becomes an unresolved reference in the design.
- If the `hdl_infer_unresolved_from_logic_abstract` attribute is set to `false`, then it remains at the level of a user-defined design hierarchy, although its function is unknown.

By default, the `hdl_infer_unresolved_from_logic_abstract` attribute is set to `true` for LEC compatibility.

Example 1-67 Inferring behavior based on the attribute `hdl_infer_unresolved_from_logic_abstract`

```
module top(a,b,c);
input a,b;
output c;
foo u0 (a,b,c);
endmodule
```

```
module foo (a,b,c);
input a,b;
output c;
endmodule
```

Here `foo` is an empty module definition.

If `hdl_infer_unresolved_from_logic_abstract` is true

The empty module `foo` will be considered as an unresolved module. It will be treated as any other blackbox and hence its internal will not be altered. That is, its output ports will not be treated as undriven and so will not be driven by a constan logic value.The `unresolved` attribute for instance `u0` will be `true` in this case.

If `hdl_infer_unresolved_from_logic_abstract` is false

In this case, `foo` will be treated like a regular module and during synthesis its ports will be treated as undriven (may be driven by constant logic value). Hence, they will be treated unlike a blackbox and optimizations can take place on `foo`. Also, the `unresolved` attribute on instance `u0` will be `false`.

Writing Out a Logic Abstract in Verilog

If a logic abstract is internally treated as an unresolved reference, then it can be written out as either an empty module or as an unresolved reference in a netlist generated by RTL Compiler using the following attribute.

➤ Set the `write_vlog_empty_module_for_logic_abstract` attribute to `true` to write out this type of unresolved reference as an empty module in the Verilog netlist.

  In the netlist it becomes a design hierarchy level with no known functionality, and it also becomes a resolved reference, as shown in Example 1-69.

For example, for the RTL code shown in Example 1-69, a component statement is provided but the entity and architecture of the instantiated component are missing. If you set the `write_vlog_empty_module_for_logic_abstract` attribute to `true`, as shown in Example 1-68, then Example 1-69 shows the resulting Verilog netlist.

Example 1-68 Writing an Unresolved Reference as an Empty Module

```
rc:/> set_attribute library tutorial.lbr
rc:/> set_attribute write_vlog_empty_module_for_logic_abstract true
rc:/> read_hdl test.v
rc:/> elaborate
rc:/> write_hdl
```

Example 1-69 Unresolved Reference as an Empty Module in a Verilog Netlist

```
module my_sub_gray_w_4 (p, q, x);
    input [3:0] p, q;
    output [3:0] x;
endmodule
module my_top (a, b, c, y);
    input [3:0] a, b, c;
    output [3:0] y;
    wire t_0, t_1, t_2, t_3;
    my_sub_gray_w_4 u1 (.p (a), .q (b), .x ({t_3, t_2, t_1, t_0}));
    or g1 (y[0], t_0, c[0]);
    or g2 (y[1], t_1, c[1]);
    or g3 (y[2], t_2, c[2]);
    or g4 (y[3], t_3, c[3]);
endmodule
```

➤ Set the `write_vlog_empty_module_for_logic_abstract` to `false` if you want this type of unresolved reference to remain unresolved in the netlist.

  It does not have an empty module in the Verilog netlist, as shown in Example 1-71.

If you set the `write_vlog_empty_module_for_logic_abstract` attribute to `false`, as shown in Example 1-70, then Example 1-71 shows the resulting Verilog netlist.

Example 1-70 Writing an Unresolved Reference That Remains Unresolved in Netlist

```
rc:/> set_attribute library tutorial.lbr
rc:/> set_attribute write_vlog_empty_module_for_logic_abstract false
rc:/> read_hdl tst.v
rc:/> elaborate
rc:/> write_hdl
```

Example 1-71 Unresolved Reference Remains Unresolved in Netlist (Verilog)

```
module my_top (a, b, c, y);
    input [3:0] a, b, c;
    output [3:0] y;
    wire t_0, t_1, t_2, t_3;
    my_sub_gray_w_4 u1 (.p (a), .q (b), .x ({t_3, t_2, t_1, t_0}));
    or g1 (y[0], t_0, c[0]);
    or g2 (y[1], t_1, c[1]);
    or g3 (y[2], t_2, c[2]);
    or g4 (y[3], t_3, c[3]);
endmodule
```

By default, the `write_vlog_empty_module_for_logic_abstract` attribute is set to `true` for LEC compatibility.

Note: The `write_vlog_empty_module_for_logic_abstract` attribute does not apply to an unresolved reference that is not a logic abstract.

Representing a Blackbox as an Empty Module

➤ If you want to use an empty module in the HDL code as a place-holder for an unresolved reference, such as for a hard macro, then set the following attributes to `true`:

```
rc:/> set_attribute hdl_infer_unresolved_from_logic_abstract true
rc:/> set_attribute write_vlog_empty_module_for_logic_abstract true
```

If RTL Compiler reads back a netlist that it previously wrote out, setting these attributes to `true` ensures that everything is interpreted in the same way as before.

Representing a Technology Cell as an Empty Module

➤ If you want to use empty modules in the HDL code to represent technology cells in the synthesis library, then set the following attributes:

```
rc:/> set_attribute hdl_infer_unresolved_from_logic_abstract true
rc:/> set_attribute write_vlog_empty_module_for_logic_abstract false
```

If RTL Compiler reads back a netlist that it previously wrote out, then this is consistent, because an empty module in the original RTL code becomes an unresolved reference in the netlist, therefore, it goes through the library look-up process when the netlist is read back in.

2
Specifying Synthesis Pragmas

Overview of Synthesis Pragmas

Synthesis pragmas are specially-formatted comments or attributes in the HDL that tell RTL Compiler how to synthesize the HDL.

- For Verilog, specify synthesis pragmas as comments using the format:

```
// pragma_keyword pragma_name [optional_pragma_value]
        or
/* pragma_keyword pragma_name [optional_pragma_value] */
```

- For VHDL, specify synthesis pragmas as either comments or attributes:

```
-- pragma_keyword pragma_name [optional_pragma_value]

attribute pragma_name of hdl_name : hdl_name_type is hdl_value
```

RTL Compiler supports synthesis pragmas that begin with the keywords `cadence`, `synopsys`, `ambit`, `pragma` and `synthesis`. If the HDL has other keywords, or if you want to prevent RTL Compiler from recognizing a pragma without changing the HDL, use the `input_pragma_keyword` attribute to add or remove supported keywords prior to reading in the HDL.

Note: You cannot remove `cadence` from the list of pragma keywords.

```
rc:/> get_attribute input_pragma_keyword

cadence synopsys ambit pragma synthesis

rc:/> set_attribute input_pragma_keyword "keyword_a keyword_b"

Cannot remove 'cadence' from attribute 'input_pragma_keyword'.

Setting attribute of root '/' : 'input_pragma_keyword' = cadence keyword_a
     keyword_b
```

This tells RTL Compiler that comments that begin with `cadence, keyword_a, or keyword_b` are synthesis pragmas.

Supported Pragmas

Table 2-1 on page 83 and Table 2-2 on page 84 list the supported Verilog and VHDL synthesis pragmas. Some of the pragma names can be customized by using the specified RTL Compiler attributes.

Note: Pragma values that are lists must be surrounded by quotation marks if there is more than one item in the list:

```
//cadence sync_set_reset "sig_1 sig_2 sig_3"
```

Note: Pragmas without values must be located immediately following the HDL to which they refer:

```
reg [1:0] q /* cadence preserve_sequential*/;
```

The following sections provide more details about how to use specific synthesis pragmas.

Table 2-1 Supported Verilog Synthesis Pragmas

Pragma Name	**Pragma Value**	**Change with attribute**
async_set_reset	signal_list	input_asynchro_reset_pragma
async_set_reset_local	label_list	input_asynchro_reset_blk_pragma
black_box		
block		
dont_infer_multibit	signal_list	
full_case		input_case_cover_pragma
infer_multibit	signal_list	
inline		
inline_instance		
keep_signal_name	signal_list	
label	arch_type	
map_to_module	module_name	
map_to_mux, infer_mux	opitonal_block_list	input_map_to_mux_pragma
map_to_operator	op_name	
one_hot	signal_list	input_assert_one_hot_pragma
one_cold	signal_list	input_assert_one_cold_pragma
parallel_case		input_case_decode_pragma
preserve_sequential		
return_port_name	signal_name	
script_begin, dc_script_begin		script_begin
script_end, dc_script_end		script_end
sub_arch	arch_type	
syn_preserve		
sync_set_reset	signal_list	input_synchro_reset_pragma
sync_set_reset_local	label_list	input_synchro_reset_blk_pragma
synthesis_on, translate_on		synthesis_on_command
synthesis_off, translate_off		synthesis_off_command
template		

Table 2-2 Supported VHDL Synthesis Pragmas

Pragma Name	Pragma Value	Change with attribute
dont_infer_multibit	signal_list	
enum_encoding		
infer_multibit	signal_list	
inline		
inline_instance		
label		
keep_signal_name dont_munch		
map_to_module	module_name	
map_to_mux, infer_mux		input_map_to_mux_pragma
map_to_operator	op_name	
propagate_label_to		
resolution resolution_method	wired_and, wired_or, three_state	
return_port_name	signal_name	
script_begin, dc_script_begin		script_begin
script_end, dc_script_end		script_end
signed		
sub_arch	arch_type	
synthesis_off, translate_off		synthesis_off_command
synthesis_on, translate_on		synthesis_on_command
template		

Specifying Code Selection Pragmas

By default, the `read_hdl` command reads all HDL code in a Verilog or VHDL file. However, HDL code that is enclosed by a pair of code selection pragmas is ignored and not read. The `read_hdl` command does *not* check for syntactic correctness of the ignored code.

Unlike other pragmas, the code selection pragmas are supported by the following three commands:

- `read_hdl`
- `read_hdl -netlist`
- `read_netlist`

With the default setting, a pair of code selection pragmas can be either:

```
synthesis_off
synthesis_on
```

or

```
translate_off
translate_on
```

➤ Specify the name of code selection pragmas using the following attributes:

- `synthesis_off_command`

  Default: `translate_off synthesis_off`

- `synthesis_on_command`

  Default: `translate_on synthesis_on`

Specifying Verilog synthesis_off and synthesis_on Pragmas

As shown in Example 2-1, you can use code selection pragmas in Verilog to insert a piece of initialization code in the RTL for analysis purposes. If an initial block is surrounded by a pair of code selection pragmas, then the `read_hdl` command skips over the entire block.

Example 2-1 Specifying the translate_off and translate_on Pragmas

```
// cadence translate_off
initial begin
cond_flag = 0 ;
$display("cond_flag cleared at the beginning.") ;
end
// cadence translate_on

always @(posedge clock)
if (cond_flag)
...
```

Comparing Code Selection Pragmas with Compiler Directives

As compared with compiler directives, in Verilog, RTL Compiler treats both the following pragmas:

```
// cadence translate_off
    text
// cadence translate_on
```

and

```
//cadence synthesis_off
    text
// cadence synthesis_on
```

In the same way as the following compiler directives:

```
`if 0
    text
`endif
```

When mixing code selection pragmas and Verilog compiler directives, RTL Compiler treats them equally, depending on the order they are found in the RTL code. For example, if you use the following synthesis script and the RTL code, shown in Example 2-2:

```
rc:> set_attribute library tutorial.lbr
rc:> read_hdl test.v
rc:> elaborate
rc:> write_hdl
```

Example 2-2 Mixing Code Selection Pragmas and Compiler Directives

```
`define UNDEF
`undef UNDEF
module tst (w, x, y, a, b, c, d, e, f);
    input a, b, c, d, e, f;
    output w, x, y;
    reg w, x, y;
    always @(a or b or c or d or e or f)
    begin
        w <= a ^ b;
        `ifdef UNDEF
        w <= ~(a ^ b);
        `else
        w <= a & b;
// cadence translate_off
        w <= a | b;
        // cadence translate_on
        `endif
        x <= c ^ d;
        `ifdef UNDEF
        x <= c & d;
        // cadence translate_off
        `else
        x <= c | d;
        // cadence translate_on
        `endif
        y <= e ^ f;
        // cadence translate_off
        `ifdef UNDEF
        y <= e & f;
`else
        y <= e | f;
        `endif
        // cadence translate_on
    end
endmodule
```

Then the post-elaboration netlist is similar to that shown in Example 2-3:

Example 2-3 Post-Elaboration Netlist for Example 2-2

```
module tst (w, x, y, a, b, c, d, e, f);
    input a, b, c, d, e, f;
    output w, x, y;
    and g1 (w, a, b);
    or  g2 (x, c, d);
    xor g3 (y, e, f);
endmodule
```

The function of signal *x* is where the priority can be observed. Since the `translate_off` and `translate_on` pragmas have the same priority as the `` `ifdef `` compiler directives, the `translate_off` pragma after the `x <= c & d;` code is ignored.

Specifying VHDL synthesis_off and synthesis_on Pragmas

Example 2-4 shows how to use code selection pragmas in VHDL to insert non-synthesizable assertions in the RTL code for analysis purposes.

Example 2-4 Specifying the translate_on and translate_off Pragmas (VHDL)

```
function DIVIDE (L, R: integer) return integer
is variable RESULT: integer;
begin

    -- cadence translate_off
    assert (R /= 0)
    report "Attempt to Divide by Zero Unsupported !!!"
    severity ERROR;
    -- cadence translate_on

    RESULT:= L/R;
    return (RESULT);
end DIVIDE;
```

Specifying black_box Pragmas (Verilog)

The `elaborate` command infers a logic abstract from a Verilog module if you specify the synthesis `black_box` pragma on the module as shown in Example 2-5. This pragma is only supported for Verilog.

Example 2-5 Inferring a Logic Abstract from a Verilog Module by Specifying a black_box pragma

```
module sub (q,d,clk); // cadence black_box
    input d, clk;
    output q;
    reg q;
    always  @ (posedge clk)
        q = d;
endmodule
```

An alternative way to skip elaboration of a module is to specify `blackbox` `hdl_arch` attribute on the architecture before elaboration.

Specifying case Statement Pragmas (Verilog)

RTL Compiler automatically determines that each `case` statement is:

- `full` if all possible `case` condition values are specified
- `parallel` if none of the `case` condition values overlap

Use the `full_case` pragma or the `parallel_case` pragma to force RTL Compiler to interpret the `case` statement as full or parallel, respectively when the `case` condition values do not meet these conditions.

Specifying the full_case Pragma

Use the `full_case` pragma to specify that the condition values listed for the case statement in the RTL represent all possible values for the case condition. This can result in a more efficient design.

In Example 2-6, the `arith_opcode` case condition can have 16 different values. However, the `full_case` pragma tells RTL Compiler that the six values listed (0000, 0001, 0010, 1001, 1101, and 1010) are the only possible values that `arith_opcode` will have. RTL Compiler assumes that all other values of `arith_opcode` (0011, 0100, and so on) represent `dont care` conditions.

Example 2-6 Specifying the full_case Pragma to Suppress the Latch Inference (Verilog)

```
module case_full (arith_opcode,src1,src2,result);
    input [3:0] arith_opcode;
    input [2:0] src1,src2;
    output [2:0] result;
    reg [2:0] result;

    always @(arith_opcode or src1 or src2) begin
        case (arith_opcode) // cadence full_case
        4'b0000 : result = 0 ;
        4'b0001 : result = src1 + src2 ;
        4'b0010 : result = src1 + 1 ;
        4'b1001 : result = src1 - src2 ;
        4'b1101 : result = src2 - src1 ;
        4'b1010 : result = src1 - 1 ;
        endcase
    end
endmodule
```

A latch is not inferred for a variable assigned a value in all branches of a `full_case` statement, such as the `result` variable shown in Example 2-6). In Example 2-7, a latch is inferred for the `reg2` variable but not for the `reg1` variable.

Example 2-7 Specifying the full_case Pragma to Infer a Latch (Verilog)

```
module case_full2 (cntl,data_in,reg1,reg2);
    input [1:0] cntl;
    input       data_in;
    output      reg1,reg2;
    reg               reg1,reg2;

    always @(cntl or data_in) begin
        case (cntl) // cadence full_case
    2'b00 : begin
        reg1 = 0;
        reg2 = data_in;
    end
    2'b01 : begin
        reg1 = data_in; // latch inferred for reg2
    end
    2'b10 : begin
        reg1 = data_in;
        reg2 = 0;
    end
        endcase
    end
endmodule
```

Note: Comparing the RTL to the synthesized design using formal verification or simulation may result in mismatches. This can occur if the `case` condition values that are not specified in the RTL, such as `arith_opcode = 0011` shown in Example 2-6, are applied during verification to `case` statements marked as `full_case`.

Specifying the parallel_case Pragma

Use the `parallel_case` pragma to specify that the condition values listed for the `case` statement in the RTL are non-overlapping. This can result in a more efficient design because it prevents RTL Compiler from generating logic to prioritize the conditions.

For example, the `parallel_case` pragma shown in Example 2-8, directs RTL Compiler to implement the following logic for `cntr`.

```
cntr = (cc[0] and '0') or
(cc[1] and data_in) or
(cc[2] and data_in-1) or
(cc[3] and data_in+1)
```

rather than:

```
cntr = (cc[0] and '0') or
(!cc[0] and cc[1] and data_in) or
(!cc[0] and !cc[1] and cc[2] and data_in-1) or
(!cc[0] and !cc[1] and !cc[2] and cc[3] and data_in+1)
```

Example 2-8 Specifying the parallel_case Pragma (Verilog)

```
module case_parallel (clk,cc,data_in,cntr);
    input clk;
    input [3:0] cc;
    input [2:0] data_in;
    output [2:0] cntr;
    reg [2:0] cntr;

    always @(posedge clk) begin
        case (1'b1) // cadence parallel_case
    cc[0] : cntr = 0 ;
    cc[1] : cntr = data_in ;
    cc[2] : cntr = data_in - 1 ;
    cc[3] : cntr = data_in + 1 ;
        endcase
    end
endmodule
```

Note: Comparing the RTL to the synthesized design using formal verification or simulation may result in mismatches if overlapping case condition values, such as `cc = 1111` shown in Example 2-8, are applied during verification to `case` statements marked as `parallel_case`.

Reporting case Pragma Information

If the `hdl_report_case_info` attribute is set to `true`, the `elaborate` command reports how RTL Compiler honors the `full_case` and `parallel_case` pragmas and how it automatically infers full-case and parallel-case "multi-way decision statements" (like `case` statements).

The reporting is done on a module-by-module basis. The output is inserted into the log file.

The report indicates whether with each multi-way decision statement RTL Compiler found that the RTL description:

- Already implies a full-case.
- Already implies a parallel-case
- Is accompanied by a `full_case` pragma
- Is accompanied by a `parallel_case` pragma

There are four possible statuses that correspond to the scenarios above:

- `AUTO` — The RTL code is a full or parallel case. Pragmas are not specified in the RTL code.
- `USER` — The RTL code is not a full or parallel case but the pragma is specified in RTL code.
- `YES` — RTL code is a full or parallel case and the pragma is specified in the RTL code.
- `NO` — RTL code is not a full or parallel case and the pragma is not specified in RTL code.

Here is a matrix that summarizes the scenarios with their statuses:

	case_yes	**case_no**
pragma_yes	`YES`	`USER`
pragma_no	`AUTO`	`NO`

Example 2-9 on page 94 shows a `case` report for the example RTL.

Example 2-9 Reporting case Information

Here is some sample RTL code:

```
module test (a, b, y);
    input [1:0] a, b;
    output [3:0] y;
    reg [3:0] y;
    always @(a or b)
    begin
        case (a) // cadence parallel_case
        2'b11 : case (b)
            2'b00 : y = 1;
            2'b01 : y = 2;
            2'b10 : y = 3;
            2'b11 : y = 4;
            endcase
        2'b01 : casez (b) // cadence full_case
            2'b10 : y = 6;
            2'b00 : y = 7;
            2'b1z : y = 8;
            endcase
        2'b10 : casex (b) // cadence full_case parallel_case
            2'b10 : y = 9;
            2'b00 : y = 10;
            2'bx1 : y = 11;
            2'b01 : y = 12;
            endcase
        endcase
    end
endmodule
```

Take the above RTL code through these commands:

```
set_attribute library tutorial.lbr
set_attribute hdl_report_case_info true
read_hdl test.v
elaborate
```

RTL Compiler produces the following example report:

```
Case Statement Report for module test from file test.v

Line No   Type    Full    Parallel
----------------------------------
8         case    AUTO    AUTO
14        casez   USER    NO
19        casex   YES     USER
7         case    NO      YES
----------------------------------
```

Specifying Set and Reset Synthesis Pragmas

The `elaborate` command infers flip-flops and latches to implement signals in the Verilog or VHDL description as described in Modeling Flip-Flops and Modeling Latches, respectively. The synchronous set and reset behavior for flip-flops and the asynchronous set and reset behavior for latches can be implemented with logic that drives either the set and reset pins or the data pins. Use the set and reset synthesis pragmas to direct RTL Compiler to generate logic for the set and reset pins.

Note: The `synthesize` command may implement the set and reset behavior using data pins in the final mapped netlist. Use the `exact_match_seq_sync_ctrls` attribute to force `synthesize` to use the pins selected by the `elaborate` command.

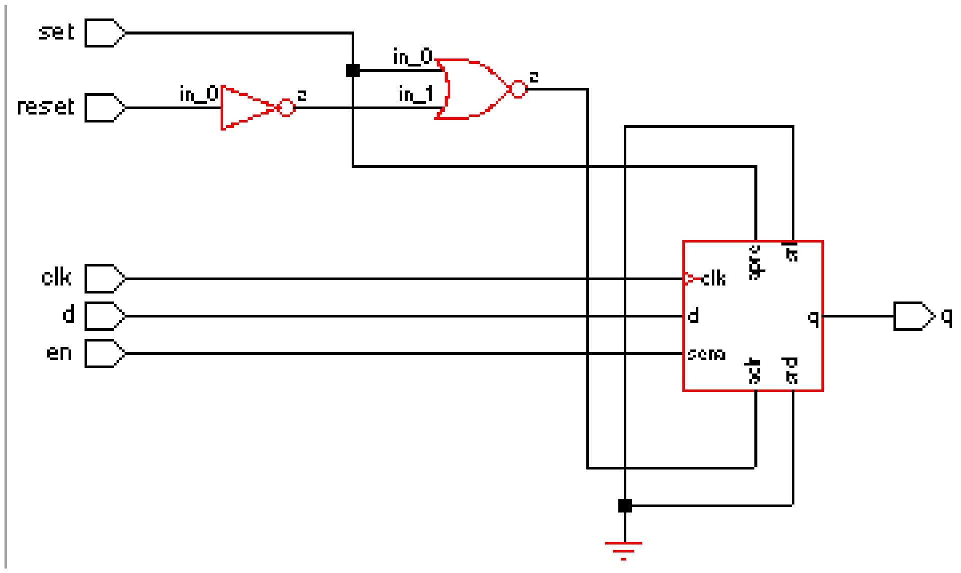
Note: Asynchronous set and reset behavior on a flip-flop is always implemented with asynchronous set and reset pins, regardless of the state of the set and reset synthesis pragmas, as shown Example 2-10.

Example 2-10 Specifying Asynchronous Set and Reset Control Logic for Flip-Flops (Verilog)

```
module dff_async_sr(clk,d,en,set,reset,q);
    input clk,d,en,set,reset;
    output q;
    reg q;

    always @ (posedge clk or posedge set or posedge reset) begin
        if (set)
         q <= 1'b1;
        else if (reset)
         q <= 1'b0;
        else if (en)
     q <= d;
end
endmodule
```

Figure 2-1 on page 96 shows the corresponding schematic for Example 2-10.

Figure 2-1 Asynchronous Set and Reset Control Logic for Flip-Flops (Verilog)



Specifying Verilog Set and Reset Synthesis Pragmas

Specify set and reset behavior in Verilog designs using the ’signal’ and ’local’ set and reset pragmas.

Specifying Set and Reset Signal Pragmas

Specify the set and reset signal pragmas as follows:

```
// cadence async_set_reset signal_name_list
// cadence sync_set_reset signal_name_list
```

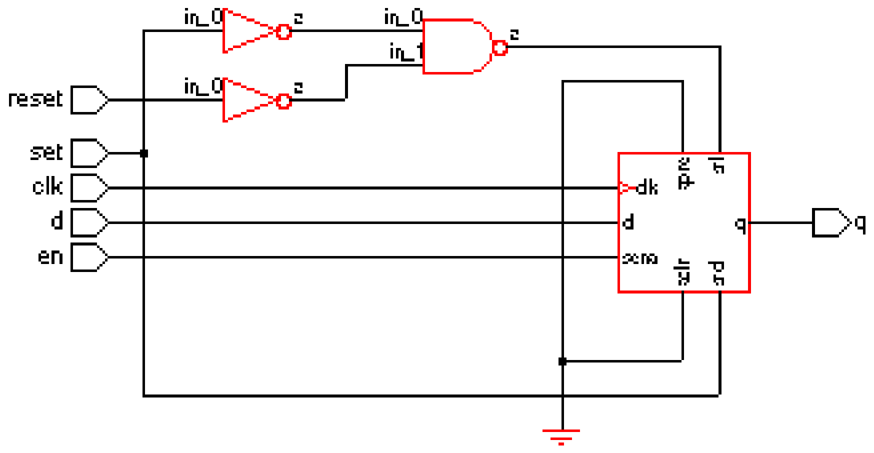
The signal pragmas must be used within a module and precede all `always` blocks. Do not list an undefined or an unused signal. The signal pragma must be in the same declarative region as the specified signal. The *`signal_name_list`* is a comma separated list of signal names in a module.

The `sync_set_reset` signal pragma is shown in Example 2-11 on page 97. Figure 2-2 on page 97 shows the corresponding schematic for Example 2-11.

Example 2-11 Specifying the Set and Reset Signal Pragma (Verilog)

```
module dff_sync_sr(clk,d,en,set,reset,q);
    input clk,d,en,set,reset;
    output q;
    reg q;

// cadence sync_set_reset "set, reset"

always @ (posedge clk) begin
        if (set)
             q <= 1'b1;
        else if (reset)
             q <= 1'b0;
        else if (en)
             q <= d;
    end
endmodule
```

Figure 2-2 Synchronous Set and Reset Control Logic (Verilog)



Specifying Local Set and Reset Pragmas

Specify the local set and reset pragmas as follows:

`//cadence async_set_reset_local` *`block_name optional_signal_name_list`*

`//cadence sync_set_reset_local` *`block_name optional_signal_name_list`*

The pragma only applies to statements within the named block. If you also specify signal names, the pragma applies to the named signals in the named block. This is shown in Example 2-12.

Example 2-12 Specifying the Local Set and Reset Pragma (Verilog)

```
module sync_block_sig_dff(out1, out2, clk, in, rst);
    output out1, out2;
    input in, clk, rst;
    reg out1, out2;

/*cadence sync_set_reset_local blk_1 */
always @(posedge clk) begin: blk_1
    if (rst)
        out1 <= 0;
    else out1 <= in;
end

always @(posedge clk) begin: blk_2
    if (rst)
        out2 <= 0;
    else out2 <= in;
    end
endmodule
```

Specifying VHDL Set and Reset Synthesis Pragmas

Specify set and reset behavior in VHDL designs using the `process`, `signal`, and `local` set and reset pragmas.

Specifying Process Pragmas

Use the `sync_set_reset_process` process (or block) pragmas to control the connection of set and reset control logic for all the registers inferred within a specific process. Specify process pragmas using Boolean-valued attributes attached directly to the process labels, as shown in Example 2-13.

Example 2-13 VHDL Process Pragma

```
attribute SYNC_SET_RESET_PROCESS: boolean;
attribute SYNC_SET_RESET_PROCESS of P1: label is TRUE;
attribute SYNC_SET_RESET_PROCESS: boolean;
attribute SYNC_SET_RESET_PROCESS of P2: label is TRUE;
```

`P1` and `P2` are the labels for the processes. These pragmas indicate that the set and reset control logic for all the registers inferred within the process is directly connected to the synchronous (for `SYNC_SET_RESET_PROCESS`) and asynchronous (for `ASYNC_SET_RESET_PROCESS`) pins of the register component. The `SYNC_SET_RESET_PROCESS` and `ASYNC_SET_RESET_PROCESS` attributes are declared in the `cadence.attributes` package.

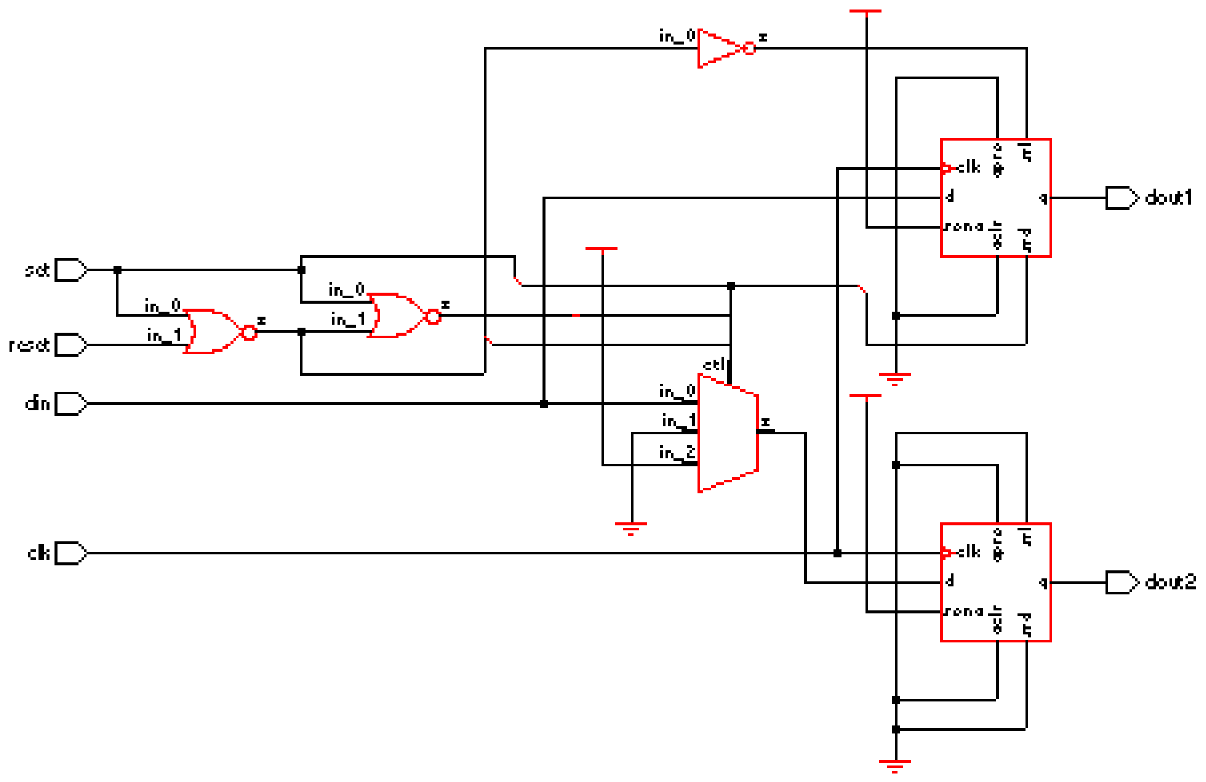
These process pragmas must be specified in the declarative region of the architecture that contains the corresponding processes. In Example 2-14, D-type flip-flops are inferred for the `dout1` and `dout2` signals. For `dout1`, the synchronous set and reset operations controlled by the set and reset signals are implemented in the elaborated netlist through the `srl` and `srd` pins on the generic `CDN_flop` component. Logic for the `dout2` signal however, is implemented entirely through the `D` pin on the `CDN_flop` component.

Example 2-14 Specifying the sync_set_reset_process Synthesis Pragma (VHDL)

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

entity sync_sr3 is
    port (
    din, clk, set, reset: in std_logic;
    dout1, dout2 : out std_logic);
end;

architecture rtl of sync_sr3 is
    attribute sync_set_reset_process of p1: label is true;
begin
    p1: process(clk) begin
        if rising_edge(clk) then
            if set = '1' then
                dout1 <= '1';
            elsif reset = '1' then
                dout1 <= '0';
            else
                dout1 <= din;
            end if;
        end if;
    end process;
    p2: process(clk) begin
        if rising_edge(clk) then
            if set = '1' then
                dout2 <= '1';
            elsif reset = '1' then
                dout2 <= '0';
            else
                dout2 <= din;
            end if;
        end if;
    end process;
end;
```

Figure 2-3 on page 101 shows the corresponding schematic for Example 2-14.

Figure 2-3 Implementing Set and Reset Synchronous Block Logic (VHDL)



VHDL Signal Pragmas

Use signal pragmas to selectively connect some of the signals directly to the `set` or `reset` pin of the component and let the other signals propagate through logic onto the data pin.

The `signal` pragma states that the specified signal should be connected directly to the `set` and `reset` pin of any inferred registers for which the signal causes a set or reset. Specify the `signal` pragma using Boolean-valued attributes attached directly to the appropriate signals, as shown in Example 2-9.

Example 2-15 VHDL Signal Pragmas

```
attribute SYNC_SET_RESET: boolean;
attribute SYNC_SET_RESET of S: signal is true;
attribute ASYNC_SET_RESET: boolean;
attribute ASYNC_SET_RESET of R: signal is true;
```

The signals are tagged `S` and `R` with the `SYNC_SET_RESET` and `ASYNC_SET_RESET` attributes respectively, indicating that they should be connected directly to the synchronous `set` and asynchronous `reset` pins of the inferred registers. The `SYNC_SET_RESET` and `ASYNC_SET_RESET` attributes are declared in the `cadence.attributes` package.

Note: Specify the `signal` pragma in the same declarative region as the signal being attributed. An error occurs if you specify these pragma for a non-existent or unused signal.

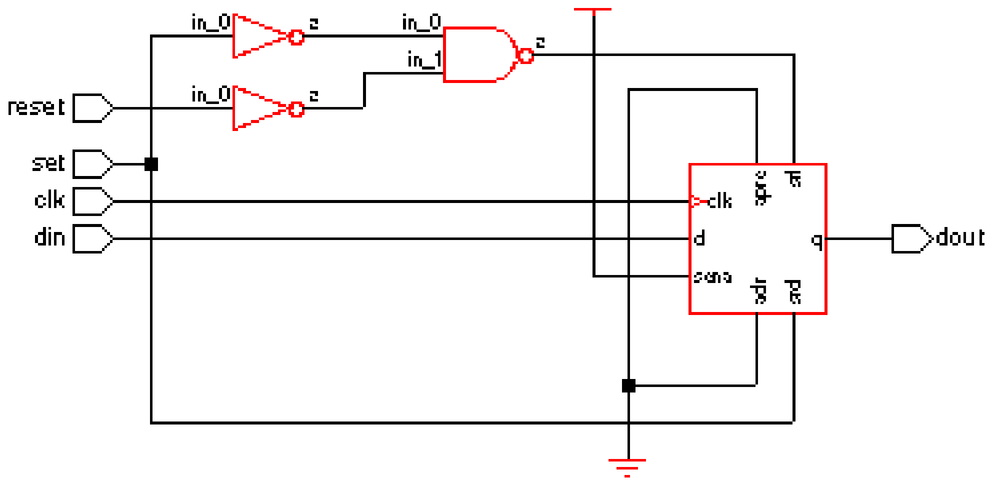
The flip-flop inferred for `out1` and `out2`, shown in Example 2-16, is connected so that the `set` signal connects to the synchronous `set` pin and the `reset` signal is connected through combinational logic feeding the `D` data port.

Example 2-16 Specifying the Signal Pragma (VHDL)

```
library ieee, cadence;
use ieee.std_logic_1164.all;
use cadence.attributes.all;
entity sync_sr4 is
    port (
        din, clk, set, reset: in std_logic;
        dout : out std_logic);
        attribute sync_set_reset of set: signal is true;
end;
architecture rtl of sync_sr4 is
begin
    process(clk) begin
        if rising_edge(clk) then
            if set = '1' then
                dout <= '1';
            elsif reset = '1' then
                dout <= '0';
            else
                dout <= din;
            end if;
        end if;
    end process;
end;
```

The generated logic is shown in Figure 2-4 on page 103.

Figure 2-4 Implementing Set and Reset Synchronous Signal Logic (VHDL)



Signals in a Process Pragma

Sometimes it is necessary to connect signals directly to the `set` and `reset` pins of certain registers and through the data input of other registers. In this situation, two synthesis pragmas that provide a combination of the synthesis pragmas, discussed in Specifying Process Pragmas on page 98, are useful. These synthesis pragma combinations let you specify both the process and the signal names.

Using the sync_set_reset_local and async_set_reset_local Attributes

The model, shown in Example 2-17, uses the `sync_set_reset_local` attribute to indicate that the `rst` signal should be connected to the synchronous `set` and `reset` pins of the flip-flops inferred in process `P1`.

Example 2-17 VHDL sync_set_reset_local and async_set_reset_local Attributes

```
signal rst, set: std_logic;
attribute sync_set_reset_local: string;
attribute sync_set_reset_local of P1: label is "rst";
attribute sync_set_reset_local: string;
attribute sync_set_reset_local of P2: label is "set";
```

The `sync_set_reset_local` attribute indicates that the `set` signal should be connected to the asynchronous `set` or `reset` pin of the latches inferred in `P2`.

The `sync_set_reset_local` and `async_set_reset_local` attributes are declared in the `cadence.attributes` package.

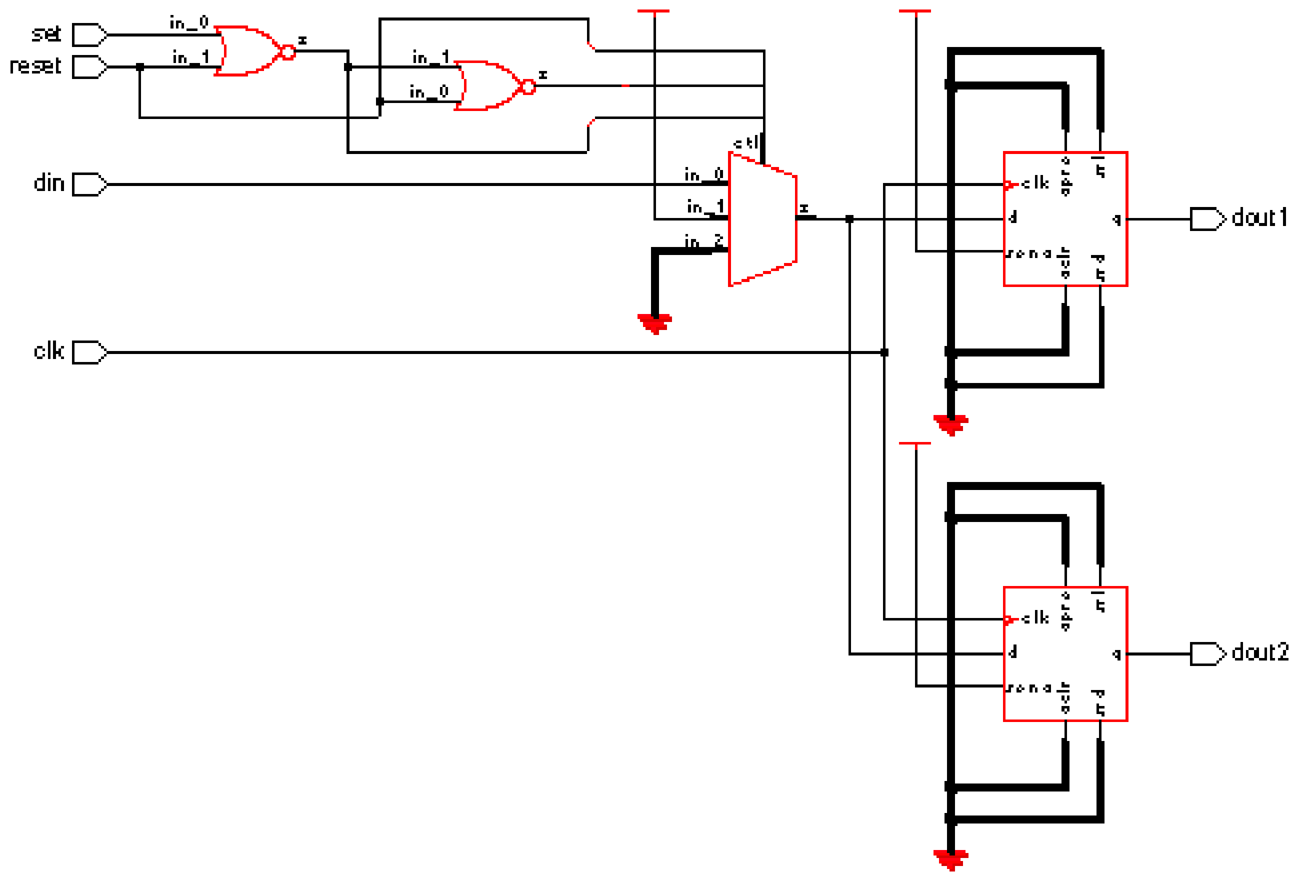
Only the listed signals in the process are inferred as synchronous or asynchronous `set` and `reset` signals and will be connected to the synchronous or asynchronous pins respectively. For registers inferred from other processes, signals can be connected to the data input as appropriate. Example 2-18 on page 104 shows how to use the `sync_set_reset_local` synthesis pragma. The value of `hdl_auto_sync_set_reset` attribute should be set to `false`.

Example 2-18 Specifying the sync_set_reset_local Synthesis Pragma (VHDL)

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

entity sync_sr5 is
    port (
        din,clk,set,reset: in std_logic;
        dout1,dout2 : out std_logic);
end;

architecture rtl of sync_sr5 is
    attribute sync_set_reset_local of p1: label is "reset";
begin
    p1: process(clk) begin
        if rising_edge(clk) then
            if reset = '1' then
                dout1 <= '0';
            elsif set = '1' then
                dout1 <= '1';
            else
                dout1 <= din;
            end if;
        end if;
    end process;

    p2: process(clk) begin
        if rising_edge(clk) then
            if reset = '1' then
                dout2 <= '0';
            elsif set = '1' then
                dout2 <= '1';
            else
                dout2 <= din;
            end if;
        end if;
    end process;
end;
```

The generated logic is shown in Figure 2-5. The reset control (`rst` signal) for the `out1` flip-flop is connected directly to the synchronous `reset` pin, whereas the `reset` control for the `out2` flip-flop is connected through logic to the input pin. This is because the `rst` signal was identified as synchronous in the pragma for `process P1` only.

Figure 2-5 Implementing Set and Reset Synchronous Signals in a Block Logic (VHDL)



Specifying Multiplexer Mapping Pragma

Use the `map_to_mux` pragma with the `case` or `if-then-else` statements, with the Verilog choice expression (`sel?a:b`) or VHDL choice expression (`a when sel=1 else b`), with Verilog named blocks, and with VHDL process statements to force RTL Compiler to implement variables assigned with the statement with multiplexer components from the technology library.

When a statement or expression is marked with the `map_to_mux` pragma, RTL Compiler implements the logic using a multiplexer with 2^n data inputs, where `n` is the width of the condition expression. For example, RTL Compiler generates a 16-to-1 multiplexer for variables assigned within a `case(sel[3:0])` statement.

There are several situations for which RTL Compiler ignores the `map_to_mux` pragma and implements the logic with AND and OR gates instead. For most of these situations, the tool issues a warning message explaining the behavior. For example, to prevent generating unreasonably large multiplexers, RTL Compiler ignores the `map_to_mux` pragma when `n` exceeds the `hdl_max_map_to_mux_control_width` attribute value. Other reasons for ignoring the `map_to_mux` pragma include:

- Absence of multiplexer components in the technology library.
- The logic can be minimized to a form that does not represent multiplexing behavior.

  For example, RTL Compiler ignores the `map_to_mux` pragma for the following "`if`" statements and implements "y = a" and "z = s":

```
if (s) // cadence map_to_mux
        y = a;
    else
        y = a;
if (s) // cadence map_to_mux
        z = 1;
    else
        z = 0;
```

Specifying Verilog Multiplexer Mapping Pragma

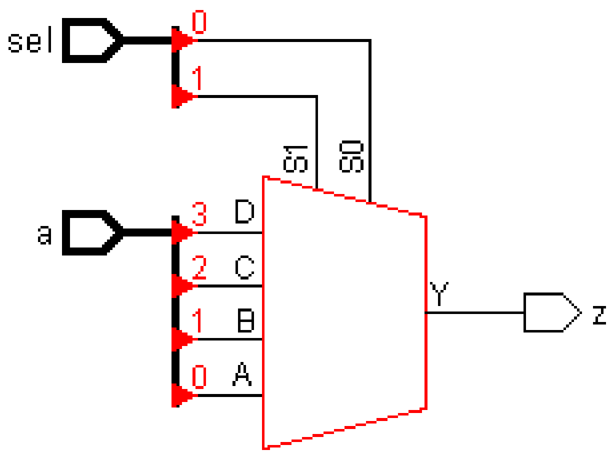
Specifying the map_to_mux Pragma With a Case Statement

Example 2-19 shows the `map_to_mux` pragma with a `case` statement. Figure 2-6 on page 107 shows the resulting schematic.

Example 2-19 Specifying map_to_mux Pragma With a Case Statement (Verilog)

```
module map2mux1 (a,sel,z);
    input [3:0] a;
    input [1:0] sel;
    output z;
    reg z;
    always @ (a or sel) begin
        case (sel)  // cadence map_to_mux
    2'b00 : z <= a[0];
    2'b01 : z <= a[1];
    2'b10 : z <= a[2];
    2'b11 : z <= a[3];
        endcase
    end
endmodule
```

Figure 2-6 map_to_mux Pragma With a case Statement Schematic (Verilog)



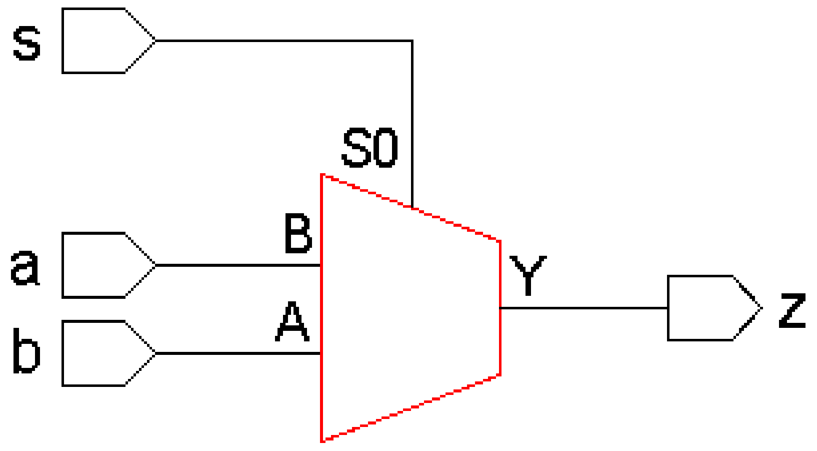
Specifying the map_to_mux Pragma With an if Statement

Example 2-20 shows the `map_to_mux` pragma with an `if` statement and Figure 2-7 shows the resulting schematic.

Example 2-20 Specifying map_to_mux Pragma With an if Statement (Verilog)

```
module map2mux2(a,b,s,z);
    input a,b,s;
    output z;
    reg z;

    always @(a or b or s) begin
            if (s) // cadence map_to_mux
        z = a;
            else
        z = b;
    end
endmodule
```

Figure 2-7 map_to_mux Pragma With an if Statement Schematic (Verilog)



Specifying the map_to_mux Pragma With a Choice Statement

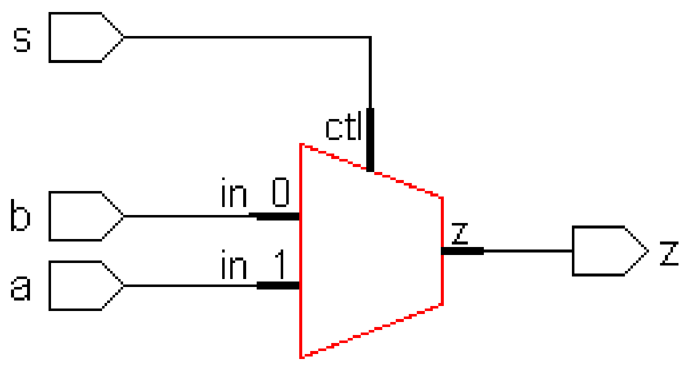
Example 2-21 shows the `map_to_mux` pragma with a `choice` statement and Figure 2-8 shows the resulting schematic.

Note: Like other operator pragmas, the `map_to_mux` pragma should be specified next to the "?" of the ternary operator in the `choice` statement.

Example 2-21 Specifying map_to_mux Pragma With a Choice Statement (Verilog)

```
module map2mux3(a,b,s,z);
    input a,b,s;
    output z;
    assign z = s ? // cadence map_to_mux
        a : b;
endmodule
```

Figure 2-8 map_to_mux Pragma With a choice Statement Schematic (Verilog)



Specifying the map_to_mux Pragma for Named Blocks

Use the `map_to_mux` pragma to force RTL Compiler to implement multiplexers for all mux possibilities, such as `if` and `case` statements, variable bit-select operations, and within named blocks, as shown in Example 2-22.

Example 2-22 Specifying map_to_mux Pragma for Named Blocks (Verilog)

```
// cadence map_to_mux "blk1, blk2”
always @ (d1 or sel)
    begin: blk1
        q1 = d1[sel];
end
always @ (d2 or x0 or x1 or x2 or x3)
    begin: blk2
        case (d2)
        2'b00: q1 = x0;
        2'b01: q1 = x1;
        2'b10: q1 = x2;
        2'b11: q1 = x3;
        endcase
end
```

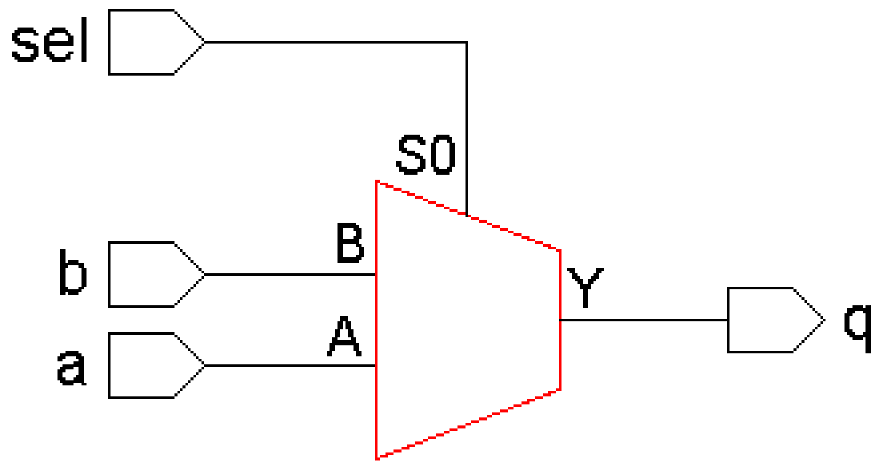
Specifying the VHDL Multiplexer Mapping Pragma

Example 2-23 shows the `map_to_mux` pragma with a `case` statement and Figure 2-9 shows the resulting schematic.

Example 2-23 Specifying map_to_mux Pragma With a case Statement (VHDL)

```
entity map2mux1 is
    port (
        sel : in integer range 0 to 1;
        a, b : in bit;
        q : out bit);
end;

architecture rtl of map2mux1 is
begin
    process(sel, a, b) begin
        case sel is-- cadence map_to_mux
    when 0 => q <= a;
    when 1 => q <= b;
        end case;
    end process;
end;
```

Figure 2-9 map_to_mux Pragma With a Case Statement Schematic (VHDL)

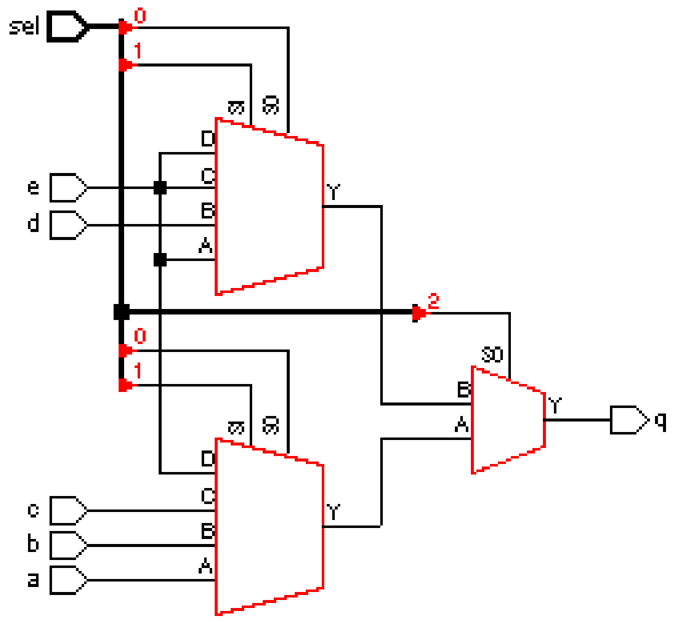


Example 2-24 shows the `map_to_mux` pragma with an `if` statement and Figure 2-10 shows the resulting schematic.

Example 2-24 Specifying the map_to_mux Pragma With an if Statement (VHDL)

```
entity map2mux2 is
    port (
        sel : in integer range 0 to 7;
        a, b, c, d, e : in bit;
        q : out bit);
end;

architecture rtl of map2mux2 is
begin
    process (sel, a, b, c, d, e) begin
        if sel = 0 then -- cadence map_to_mux
            q <= a;
        elsif sel = 1 then
            q <= b;
        elsif sel = 2 then
            q <= c;
        elsif sel = 5 then
            q <= d;
        else
            q <= e;
        end if;
    end process;
end;
```

Figure 2-10 map_to_mux Pragma With an if Statement Schematic (VHDL)

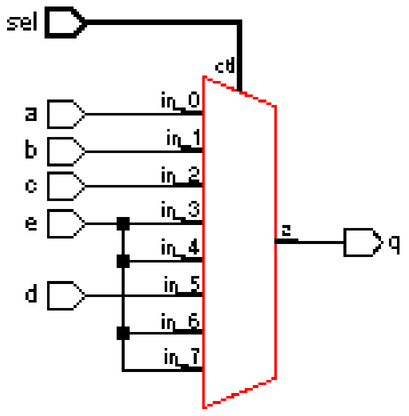


Example 2-25 shows the `map_to_mux` pragma with a `choice` statement RTL and Figure 2-11 shows the resulting schematic.

Example 2-25 Specifying map_to_mux Pragma With a Choice Statement (VHDL)

```
entity map2mux3 is
    port (
        sel : in integer range 0 to 7;
        a, b, c, d, e : in bit;
        q : out bit);
end;

architecture rtl of map2mux3 is
begin
    q <= a when sel = 0 -- cadence map_to_mux
        else
        b when sel = 1 else
        c when sel = 2 else
        d when sel = 5 else
        e;
end;
```

Figure 2-11 map_to_mux Pragma With a Choice Statement Schematic (VHDL)



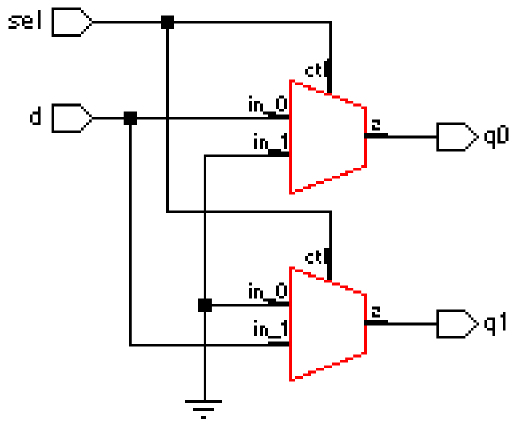
Specifying the map_to_mux Pragma for Labeled Process Statements

Specify the `map_to_mux` pragma as a VHDL process label attribute to force RTL Compiler to implement multiplexers for all mux possibilities, such as `if` and `case` statements, and within process statements, as shown in Example 2-26. Figure 2-12 shows the resulting schematic.

Example 2-26 Specifying map_to_mux Pragma as a Process Label (VHDL)

```
library ieee;
use ieee.std_logic_1164.all;
entity map2mux4 is
    port (q0,q1 : out std_logic;
        sel0,sel1 : integer range 0 to 1;
        d0,d1,d2,d3 : in std_logic);
end;

architecture rtl of map2mux4 is
    attribute map_to_mux : string;
    attribute map_to_mux of p1: label is "true";
begin
    p1: process(sel0,sel1,d0,d1,d2,d3) begin
        case sel0 is
        when 0 =>
            q0 <= d0;
        when 1 =>
            q0 <= d1;
        end case;
        case sel1 is
        when 0 =>
            q1 <= d2;
        when 1 =>
            q1 <= d3;
        end case;
    end process;
end;
```

Figure 2-12 Schematic map_to_mux Pragma as a Process Label (VHDL)



Specifying Pragmas to Preserve Internal Nets and Registers (Verilog)

Specifying the keep_signal_name Pragma

Use the `keep_signal_name` pragma to prevent unused logic driving the named signal from being deleted.

In Example 2-27, the logic driving wire `w1` is preserved during synthesis whereas the logic driving wire `w2` is deleted.

Example 2-27 Specifying the keep_signal_name Pragma (1)

```
module top(input in1,in2, output out1);
    wire w1, w2; // cadence keep_signal_name w1
    assign w1 = in1 & in2;
    assign w2 = in1 | in2;
    assign out1 = in1;
endmodule
```

In Example 2-28, the logic driving wires `w1` and `w2` is preserved during synthesis. Wire `w1` is deleted.

Example 2-28 Specifying the keep_signal_name Pragma (2)

```
module top(input in1,in2, output out1);
    wire w1, w2, w3; // cadence keep_signal_name "w2, w3"
    assign w1 = in1 & in2;
    assign w2 = in1 | in2;
    assign w3 = in1 ^ in2;
    assign out1 = in1;
endmodule
```

Specifying the preserve_sequential Pragma

Use the `preserve_sequential` signal pragma to prevent unused sequential cells from being deleted. Specify the pragma in the declaration of a signal of any type (wire, register, logic and so on) for which RTL Compiler infers a flip-flop or latch.

In Example 2-29 the flip-flop inferred for `q1` is preserved during synthesis whereas the flip-flop inferred for `q0` is deleted.

Example 2-29 Specifying the preserve_sequential Pragma (1)

```
module top(input clk, input in1, in2, output out1);
    reg q0;
    reg q1 /*cadence preserve_sequential */;
    always@(posedge clk)
    begin
        q0 <= in1;
        q1 <= in2;
    end
    assign out1 = in1;
endmodule
```

In Example 2-30, only the flip-flop inferred for `r2` is preserved after synthesis.

Example 2-30 Specifying the preserve_sequential Pragma (2)

```
module top(input clk, input in1, in2,output reg out1);
struct packed { reg  r1;
                reg r2/*cadence preserve_sequential */;
}S1;
always@(posedge clk)
begin
    S1[0] <= in1;
    S1[1] <= S1[0];
end
assign out1 = in1;
endmodule
```

In Example 2-31, both `r1[1]` and `r1[0]` are preserved because the pragma applies to bus `r1`.

Example 2-31 Specifying the preserve_sequential Pragma (3)

```
module top(input clk,input in1,in2,in3,output reg out1);

reg  r1[1:0] /* cadence preserve_sequential */;
function reg func1(input in1,in2);
    func1 = in1 & in2;
endfunction

always@(posedge clk)
begin
    r1[1] <= func1(in1, in2);
    r1[0] <= 1;
    out1 <= in1;
end
endmodule
```

The behavior of pragma `syn_preserve` is similar to the behavior of `preserve_sequential`.

Specifying Pragmas to Control Mapping to Multi-Bit Registers

Use the `infer_multibit` pragma to tell RTL Compiler to consider mapping the specified signals to multiple-bit flip-flop or latch library cells in the technology library.

In the following example, signals `outA` and `outB` will each be mapped to a four-bit flip-flop library cell component, whereas signals `A2` and `outC` will each be mapped to four individual flip-flop library cells.

Example 2-32 Specifying the infer_multibit Pragma

```
module debug (
          input wire [3:0] inA,
          input wire [3:0] inB,
          input wire [3:0] inC,
          input wire clk,
          output reg [3:0] outA,
          output reg [3:0] outB,
          output reg [3:0] outC
          );

reg [3:0]      A2;

// cadence infer_multibit "outB outA"
always @(posedge clk) begin
    A2 <= inA;
    outA <= A2;
    outB <= inB;
    outC <= inC;
end // always @ (posedge clk)
endmodule // debug
```

Example 2-33 Specifying the infer_multibit Pragma (VHDL)

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

entity debug is
port (
      CK            :  in std_logic;
      inA, inB, inC :  in std_logic_vector( 3 downto 0 );
      outA, outB, outC :  out std_logic_vector( 3 downto 0 ));
end entity;

architecture arch of debug is
   begin
        -- cadence infer_multibit "outA outB"
       process
       begin
           wait until ( CK'EVENT and ( CK = '1' )) ;
           outA <= inA;
       end process;
       process
       begin
           wait until ( CK'EVENT and ( CK = '1' )) ;
           outB <= inB;
       end process;
       process
       begin
           wait until ( CK'EVENT and ( CK = '1' )) ;
           outC <= inC;
       end process;
end;
```

Use the `dont_infer_multibit` pragma to prevent RTL Compiler from mapping the specified signals to multiple-bit flip-flop or latch cells in the technology library.

In the following example, signal `q1` will be mapped to four individual flip-flop library cells.

Example 2-34 Specifying the dont_infer_multibit Pragma

```
always @ (posedge clk) //cadence dont_infer_multibit "q1"
    q1 = d [3:0];
```

Note: Independent of the setting of the `use_multibit_cells` root attribute, RTL Compiler uses the pragmas to set the respective `infer_multibit` or `dont_infer_multibit` instance attributes on the flip-flop or latch instances generated by the elaborate command.

Specifying Function and Task Mapping Pragmas (Verilog and VHDL)

Use the `map_to_module` pragma in functions and tasks, and use the `return_port_name` pragma only in functions. These pragmas should appear within the declaration of a task or function. For example:

```
// cadence map_to_module module_name
```

The `map_to_module` pragma specifies that any call to the given function or task is to be internally mapped to an instantiation of the specified module. The statements in the function or task body are therefore ignored. Arguments to the function or task are mapped positionally onto ports in the module as follows:

```
// cadence return_port_name port_name
```

The `return_port_name` pragma applies only to a function to which the `map_to_module` pragma is in effect, and specifies that the return value for the function call is given by the output port of the mapped-to module.

Example 2-35 maps a function to the `BUF` entity with a `z` output.

Example 2-35 Specifying the Function and Task Mapping Pragmas

```
function f(d : in std_logic) return std_logic is

-- cadence map_to_module my_buf

-- cadence return_port_name z

begin
return d;
end;
```

The following function call:

```
q <= f(d);
```

is equivalent to the following entity instantiation:

```
i1 : entity work.my_buf port map(z, d);
```

Specifying the Signed Type Pragma (VHDL)

Use this pragma to specify that the annotated vector type is to be treated like a signed type for all arithmetic, logical, and relational operations. The `SIGNED_TYPE` attribute is a Boolean-valued attribute declared in the `cadence.attributes` package.

Example 2-36 shows the `ieee.numeric_std.signed` type.

Example 2-36 Specifying the Signed Type Pragma (VHDL)

```
use cadence.attributes.all;

....

type SIGNED is array (NATURAL range <>) of STD_LOGIC;

-- Attribute the type ’SIGNED’ for synthesis

attribute SIGNED_TYPE of SIGNED : type is TRUE;
```

Specifying the Template Pragma (Verilog and VHDL)

The `elaborate` command runs faster by designating Verilog modules or VHDL entities as templates, which eliminates synthesizing the template modules or entities that are not actually used in the hierarchical design as stand-alone modules or entities. The `TEMPLATE` attribute is declared in the `cadence.attributes` package.

When a module or entity is written with generic declarations for use as a template, only the instantiated, parameterized design is synthesized. Use the `TEMPLATE` pragma on a module or entity to indicate that the template module or entity is *not to be synthesized* except in the context of an instantiation from a higher level module or entity, never as a top-level module or entity. Specify the `TEMPLATE` pragma as `TRUE` in the module or in the entity declaration, as shown in Example 2-37.

Example 2-37 Specifying the Entity Template Pragma

```
use cadence.attributes.all;
entity FOO is
    generic (Width : integer := 64);
    port (DIN : bit_vector (Width - 1 downto 0);
        DOUT : bit_vector (Width - 1 downto 0));
    attribute TEMPLATE of FOO: entity is TRUE;
end FOO;
```

Example 2-38 Specifying the Module Template Pragma (Verilog)

```
module dff(clk,d,q);
// cadence template
parameter size=1;
input clk;
input [size-1:0]  d;
output reg [size-1:0] q;
always @ (posedge clk) q <= d;
endmodule
```

Specifying the Enumeration Encoding Pragma (VHDL)

Use this pragma to override the default encoding of enumeration literals. In Example 2-39, the literals `RED` and `YELLOW` would normally be encoded as `00` and `11,` respectively, corresponding to their position in the `COLOR` type, starting from `0`. Because of the `ENUM_ENCODING` attribute, `RED` and `YELLOW` are encoded as `10` and `01,`respectively. The `ENUM_ENCODING` attribute is declared in the `cadence.attributes` package.

The `ENUM_ENCODING` value string must contain as many encodings as there are literals in the corresponding enumeration type. All encodings contain only `0`’s or `1`’s and should have an identical number of bits.

Example 2-39 Specifying the Enumeration Encoding Pragma (VHDL)

```
type COLOR is (RED, BLUE, GREEN, YELLOW);
attribute ENUM_ENCODING: string;
attribute ENUM_ENCODING of COLOR: type is "10 00 11 01”;
```

Specifying Resolution Function Pragmas (VHDL)

Use the `RESOLUTON` function pragmas to identify and define the intended behavior of a resolution function in the design.

Define the resolution by specifying the string-valued `RESOLUTION` attribute to control how a signal with multiple drivers and resolved by the attributed function is synthesized.

The following pragmas will cause a `WIRED_AND`, `WIRED_OR`, or `WIRED_TRI` (three-state) behavior to be synthesized for any signal that is resolved by the `MYRES` function.

Example 2-40 Resolution Function Pragmas (VHDL)

```
attribute RESOLUTION: string;
attribute RESOLUTION of MYRES: function is "WIRED_AND";
attribute RESOLUTION of MYRES: function is "WIRED_OR";
attribute RESOLUTION of MYRES: function is "WIRED_TRI";
```

In Example 2-41, the `MYRES` function has been tagged as having `WIRED_OR` behavior using the `RESOLUTION` attribute. `signal X` with the `MYRES` resolution function is synthesized to exhibit a `WIRED_OR` behavior.

Example 2-41 Specifying the Resolution Function Pragma (VHDL)

```
function MYRES(bv: bit_vector) return ulogic_4 is
variable tmp: bit:= `0';
begin
    for I in vtbr'range loop
        tmp:= tmp or bv(I);
    end loop;
    return tmp;
end;

attribute RESOLUTION of MYRES: function is "WIRED_OR";
signal X: MYRES bit;
```

The `RESOLUTION` attribute is declared in the `cadence.attributes` package.

Specifying Pragmas for Embedded Script Capability

RTL Compiler supports the embedded script capability through the script_begin pragma (for dc_script_begin) and script_end pragma (for dc_script_end).

By default, RTL Compiler automatically executes the script between the pragmas at the end of elaboration. To prevent automatic execution of the scripts, set the hdl_auto_exec_sdc_scripts root attribute to false.

If the scripts are not executed during elaboration, you can use the exec_embedded_script command to execute the scripts embedded in the RTL. The embedded scripts can be accessed or changed using the embedded_script attribute at either the design or subdesign level.

More pragma names can be defined by using the root level attributes script_begin and script_end.

In the following example, an RTL Compiler script is embedded in the top level module IP and the module clock_gate_latch.

```
module IP ( CLK, TMode, a_in, InstStall_R, c_out, SCAN_EN );
    input CLK, SCAN_EN;
    input TMode;
    input a_in;
    input InstStall_R;
    output c_out;
    wire  stallB_s1R;
    wire  G2SCLK;
    reg   out;
    reg   b;
    reg   c;

    clock_gate_latch G2SCLK_nlatch (stallB_s1R, ~(InstStall_R) || TMode, CLK);
    clock_gate_and   G2SCLK_and    (G2SCLK, CLK, stallB_s1R);

// misc other logic
    always @( posedge G2SCLK )
        begin
            out <= a_in;
            b <= out;
        c <= b;
        end // always @ ( posedge G2SCLK )

    //cadence script_begin
    //set_attr dft_dont_scan true out_reg
    //cadence script_end

    assign c_out = c;

endmodule // IP
```

```
module clock_gate_latch(out,in,en);
    output out;
    input  in;
    input  en;
    reg   out;

    //cadence script_begin
    //set_attr dft_dont_scan true out_reg
    //cadence script_end

    always @(en or in) begin
        if (~en) begin
        out <= #1 in;
        end
    end // always @ (en or in)

endmodule // clock_gate_latch

module clock_gate_and(out,clk,en);
    output out;
    input  clk;
    input  en;
    assign out = (en & clk);

endmodule // clock_gate_and
```

After reading and elaborating the design, the embedded scripts are stored in the design and subdesign level attribute `embedded_script`:

```
rc:/> get_attribute embedded_script [find / -design IP]

dc::push_file embedded.v 26
set_attribute dft_dont_scan true out_reg
dc::pop_file

rc:/> get_attribute embedded_script [find / -subdesign clock_gate_latch]

dc::push_file embedded.v 40
set_attribute dft_dont_scan true out_reg
dc::pop_file
```

If you did set the root attribute `hdl_auto_exec_sdc_scripts` to `false`, you must execute these embedded scripts using the `exec_embedded_script` command:

```
rc:/> exec_embedded_script

  Setting attribute of instance 'out_reg': 'dft_dont_scan' = true

  Setting attribute of instance 'out_reg': 'dft_dont_scan' = true
```

The search for any design object while executing an embedded script is hierarchical. When executing the script embedded in a particular module, design, or subdesign, RTL Compiler searches for design objects relative to the RTL Compiler path of that particular design or subdesign.

The example below has an instance, `out_reg,` at the top-level of design `IP` as well as in the hierarchical instance `clock_gate_latch`.

```
rc:/> find / -inst out_reg
/designs/IP/instances_hier/G2SCLK_nlatch/instances_seq/out_reg \
    /designs/IP/instances_seq/out_reg
```

When executing a script assigned to `clock_gate_latch`, RTL Compiler will only consider the instance in the hierarchical instance `clock_gate_latch:`

```
rc:/> exec_embedded_script -subdesign [find / -subdesign clock_gate_latch]
  Setting attribute of instance 'out_reg': 'dft_dont_scan' = true
rc:/> filter dft_dont_scan true [find / -inst *]
/designs/IP/instances_hier/G2SCLK_nlatch/instances_seq/out_reg
```

Note: Currently, this functionality is supported only in the pure RTL flow. It is not supported in the structural or in the mixed-RTL-structural flow.

When the embedded script is not included in a module or entity description, the following applies:

- For Verilog, RTL Compiler sets the `embedded_script` attribute on the module or subdesign that precedes the `embedded_script` except.

  Note: If no module precedes the embedded script, RTL Compiler sets the `embedded_script` attribute on the module that immediately follows the script.

- For VHDL, RTL Compiler sets the `embedded_script` attribute on the `entity` that follows the embedded script.

  Note: If the embedded script is not followed by a single entity, RTL Compiler considers the script as a comment.

When executing `embedded_script`, RTL Compiler will look for design objects in that particular design or subdesign hierarchy scope. Therefore the following searches are different:

```
[find ./des* -instance u0]
```

and

```
[find /des* -instance u0]
```

Be careful when using relative paths.

3

Using HDL Commands and Attributes

HDL-Related Commands

Table 3-1 HDL-Related Commands

Command	Description
`elaborate`	Creates a design from a Verilog module or from a VHDL entity and architecture. Undefined modules and VHDL entities are labeled “unresolved” and treated as blackboxes.
`read_hdl`	Loads one or more HDL files in the order given into memory.
`read_netlist`	Reads and elaborates a Verilog 1995 structural netlist.
`write_hdl`	Generates one of the following design descriptions in Verilog format: ■ A structural netlist using generic logic ■ A structural netlist using mapped logic
`write_sv_wrapper`	Writes out a wrapper SystemVerilog module for a design or subdesign. Such a wrapper module can help in a comparative simulation of the output netlist from RTL Compiler and the input RTL design, especially when the design description in the input RTL has complex ports.

HDL-Related Attributes

The following attributes are commonly used for Verilog and VHDL designs.

- `hdl_all_filelist` {*hdl_library language_standard* {*define_value*...} {*hdl_file*...} {*search_path*}}...

  Stores a list of all files including the ‘include files, if any, read into RTL Compiler for the specified design to keep track of files. The library, language, and list of files specified with each `read_hdl` command (including their include and package files) are appended to this design attribute. The `language_standard` information is derived from the `-v1995`, `-v2001`(default), `-sv` and `-vhdl` options of the `read_hdl` command. For vhdl, the value of the `hdl_vhdl_read_version` attribute is appended to the -`vhdl` option .

- `hdl_allow_inout_const_port_connect` {true | false}

  Default: `false`

  If this attribute is set to `false`, then an error message is issued if an output or inout pin of an instantiated submodule is connected to a constant value.

- `hdl_array_naming_style` *string*

  Chooses a scheme to name individual bits of array ports and registers. The string argument must include `%s` to indicate the record name of the bus signal, and `%d` to indicate the array index. Set this attribute before using the `elaborate` command.

  Default: `%s\[%d\]`

- `hdl_async_set_reset`

  Specifies that RTL Compiler implement the listed signals using asynchronous set and reset pins on a latch if that logic controls an asynchronous assignment.

  Default: “ “

  The following command implements the reset signal for the RTL, shown in Example 3-1:

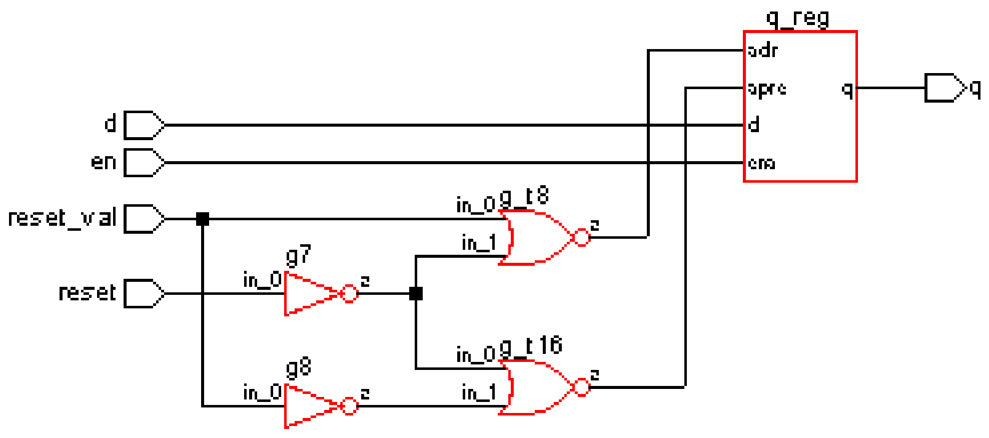
  ```
  rc:/> set_attr hdl_async_set_reset "reset"
  ```

Example 3-1 RTL Asynchronous Set and Reset

```
module asynch1(clk,d,en,reset,q);
    input clk,d,en,reset;
    output q;
    reg   q;
    always @ (clk or reset or en or d) begin
        if (reset)
            q <= 1'b0;
        else if (en)
            q <= d;
    end
endmodule
```

The corresponding schematic is shown in Figure 3-1 on page 132:

Figure 3-1 Schematic hdl_async_set_reset



- `hdl_auto_async_set_reset` {false | true}

  When set to `true`, specifies that RTL Compiler implement logic using asynchronous set and reset pins on a latch if that logic controls an asynchronous assignment of a constant 0 or constant 1.

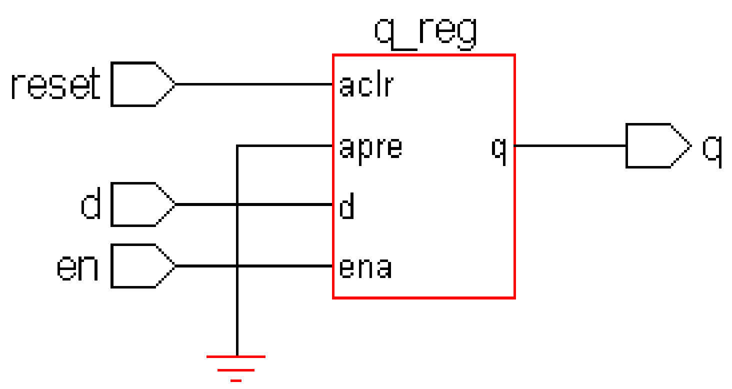
  Default: `false`

  The following command implements the `reset` signal in the RTL (shown in Example 3-1) using a latch asynchronous reset pin:

  ```
  rc:/> set_attribute hdl_auto_async_set_reset true
  ```

  The corresponding schematic is shown in Figure 3-2.

Figure 3-2 Schematic hdl_auto_async_set_reset



- `hdl_auto_exec_sdc_scripts` {true| false}

  Default: `true`

  Automatically runs SDC scripts found in the RTL input during elaboration.

- `hdl_auto_sync_set_reset` `{true| false}`

  Default: `true`

  When set to `true`, specifies that RTL Compiler implement logic using synchronous set and reset pins on a flip-flop if that logic controls a synchronous assignment of a constant 0 or constant 1.

  The following command implements the `reset` signal shown in the RTL, as shown in Example 3-2 using a flip-flop synchronous reset pin:

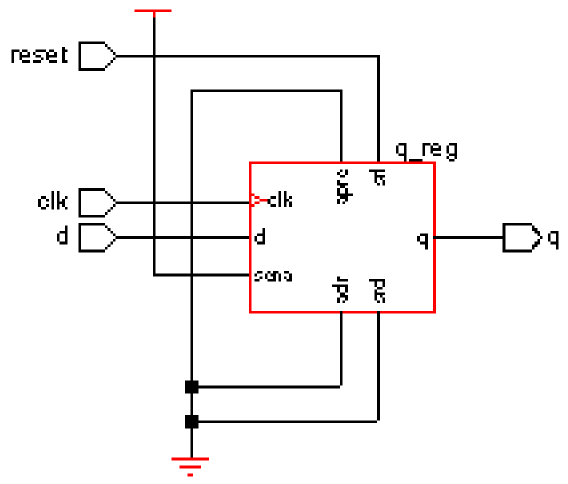
  ```
  rc:/> set_attribute hdl_auto_sync_set_reset true
  ```

Example 3-2 RTL Synchronous Reset

```
module synch1(clk,d,reset,q);
    input clk,d,reset;
    output q;
    reg q;
    always @ (posedge clk) begin
        if (reset)
            q <= 1'b0;
        else
            q <= d;
    end
endmodule
```

The corresponding schematic is shown in Figure 3-3.

Figure 3-3 Schematic hdl_auto_sync_set_reset



- `hdl_delete_transparent_latch` `{true | false}`

  Controls whether transparent latches are preserved or deleted during elaboration. When set to `true`, deletes latches that are always enabled.

- `hdl_enable_proc_name` `{true | false}`

  When set to `true`, updates the value of the `hdl_proc_name` instance attribute for sequential elements during elaboration.

- `hdl_error_on_blackbox` `{true | false}`

  Default: `false`

  When set to `true`, an error message is issued if there is an unresolved reference (blackbox) during elaboration.

- `hdl_error_on_latch` `{true | false}`

  Default: `false`

  When set to `true`, issues an error message if a latch is inferred for a design.

- `hdl_error_on_negedge` `{true | false}`

  Default: `false`

  When set to `true`, issues an error message if a design infers a flip-flop that is triggered by a falling clock edge.

- `hdl_ff_keep_feedback {true | false}`

  Default: `false`

  Controls how flip-flop stable states are implemented. When set to `true`, implements a feedback path from the `Q` output to the `D` input. When set to `false`, implements a synchronous enable signal.

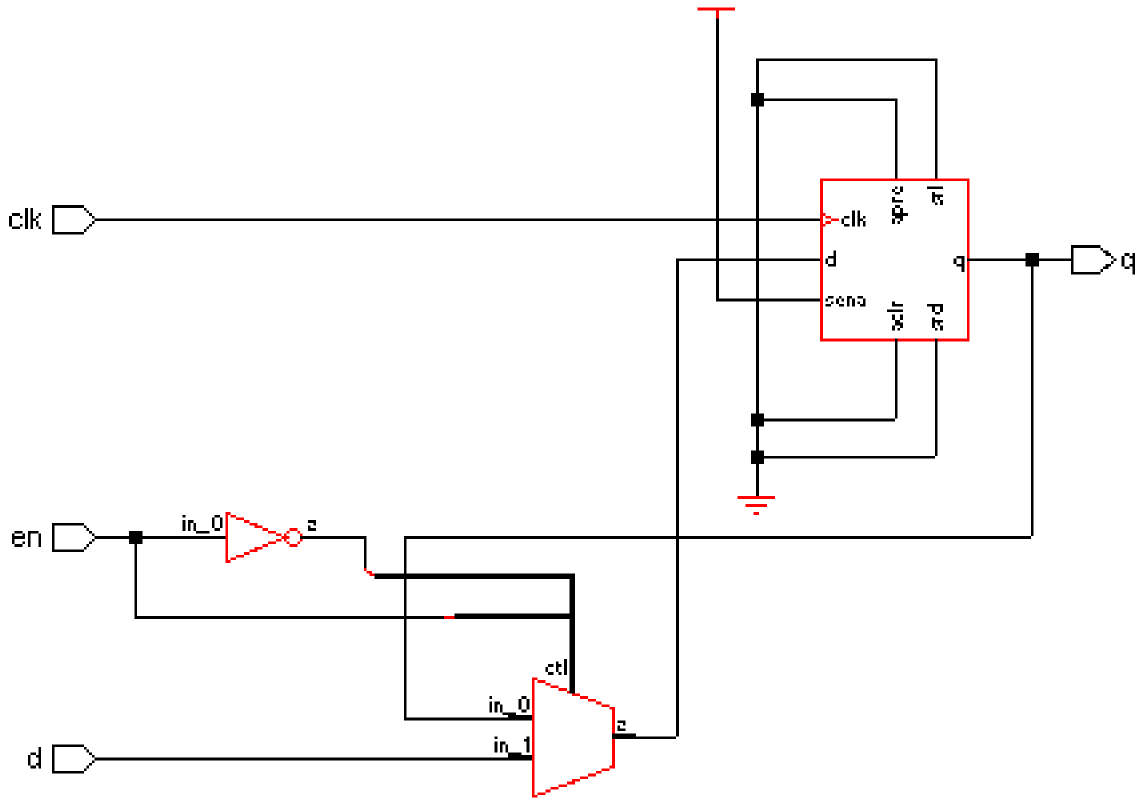
  - ❑ The following command implements a feedback path from the `Q` output to the `D` input for the RTL, as shown in Example 3-3 and in the schematic in Figure 3-4.

    ```
    rc:/> set_attribute hdl_ff_keep_feedback true
    ```

Example 3-3 RTL hdl_ff_keep_feedback true

```
module dff1(clk,d,en,q);
    input clk, d,en;
    output q;
    reg q;

    always @ ( posedge clk) begin
        if (en)
            q <= d;
    end
endmodule
```

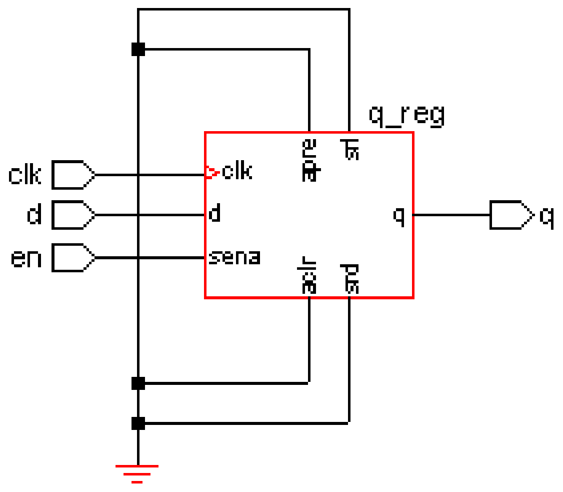
Figure 3-4 Schematic hdl_ff_keep_feedback true



- ❑ The following command implements a synchronous enable signal from the `Q` output to the `D` input for the RTL (shown in Example 3-3) and the corresponding schematic shown in Figure 3-5:

  ```
  rc:/> set_attribute hdl_ff_keep_feedback false
  ```

Figure 3-5 Schematic hdl_ff_keep_feedback false



- `hdl_ff_keep_explicit_feedback`

  Controls how flip-flop stable states are implemented for feedback assignments that are explicitly specified in the RTL.

  - The following command implements flip-flop stable states for feedback assignments that are explicitly specified in the RTL, as shown in Example 3-4:

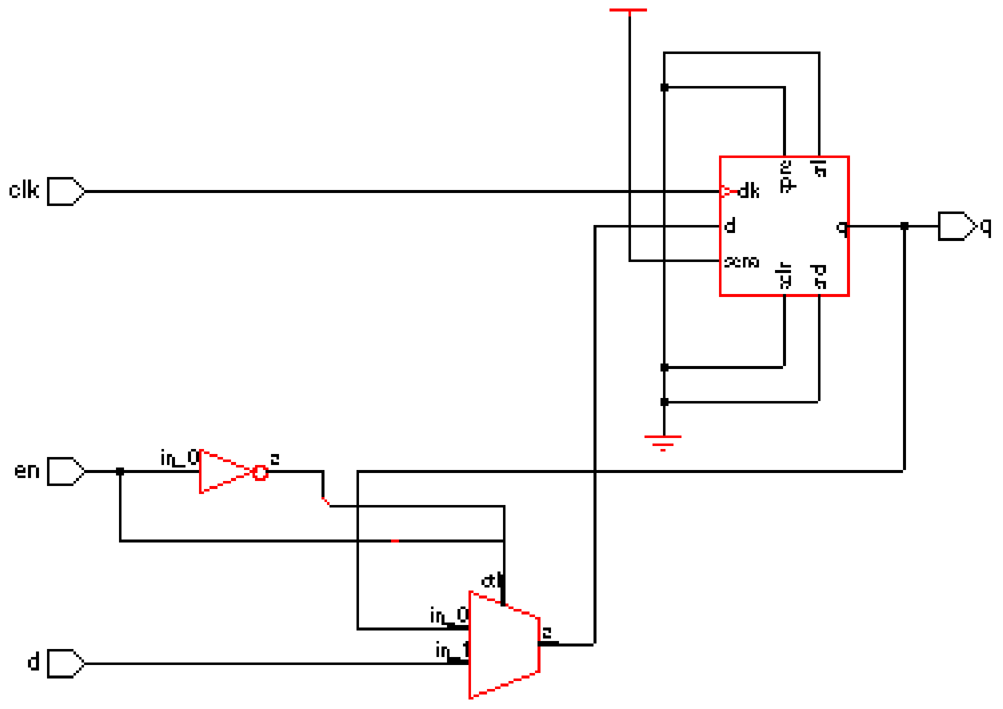
    ```
    rc:/> set_attribute hdl_ff_keep_explicit_feedback true
    ```

Example 3-4 hdl_ff_keep_explicit_feedback true

```
module dff3(clk,d,en,q);
    input clk,d,en;
    output q;
    reg q;
    always @ (posedge clk) begin
        if (en)
                q <= d;
        else
                q <= q;
    end
endmodule
```

The corresponding schematic is shown in Figure 3-6.

Figure 3-6 Schematic hdl_ff_keep_explicit_feedback true



- ❑ The following command implements a synchronous enable signal from the `Q` output to the `D` input for the RTL, shown in Example 3-5:

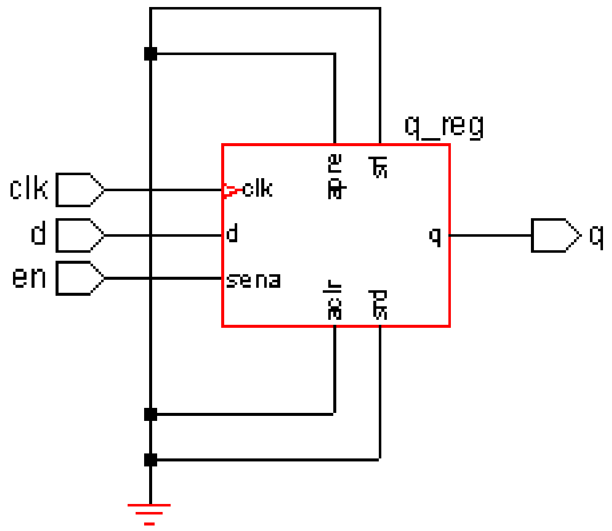
  ```
  rc:/> set_attribute hdl_ff_keep_feedback false
  rc:/> set_attribute hdl_ff_keep_explicit_feedback false
  ```

Example 3-5 RTL hdl_ff_keep_explicit_feedback false

```
module dff4(clk,d,en,q);
    input clk,d,en;
    output q;
    reg q;

    always @ (posedge clk) begin
        if (en)
                q <= d;
        else
                q <= q;
    end
endmodule
```

The corresponding schematic is shown in Figure 3-7.

Figure 3-7 Schematic hdl_ff_keep_explicit_feedback false



- `hdl_filelist` `{`*`hdl_library language_standard`*` {`*`define_value`*` ...}`
  `{`*`hdl_file`*` ...}{`*`search_path`*`} }...`

  Automatically set by the `read_hdl` command to keep track of which files are being read into RTL Compiler. The library, language, and list of files specified with each `read_hdl` command are appended to this root attribute. The `language_standard` information is derived from the `-v1995`, `-v2001`(default), and `-vhdl` options of the `read_hdl` command. For vhdl, the value of the `hdl_vhdl_read_version` attribute is appended to the -`vhdl` option

- `hdl_infer_unresolved_from_logic_abstract true | false`

  Default: `true`

  See Modeling Logic Abstracts on page 71 for detailed information.

- `hdl_language {v1995 | v2001 | vhdl | sv}`

  Default: `v2001`

  Specifies the default HDL language mode assumed when you use the `read_hdl` command without specifying the language mode.

- `hdl_latch_keep_feedback` `{false | true}`

  Controls how explicitly-specified latch stable states (for example, q <= q) are implemented. When set to `true`, implements a feedback path from the Q output to the D input, resulting in a combinational loop. When set to `false`, implements a latch with an enable signal.

  - The following command implements a feedback path from the `Q` output to the `D` input for the RTL, shown in Example 3-6. This results in a combinational loop, as shown in Figure 3-8:

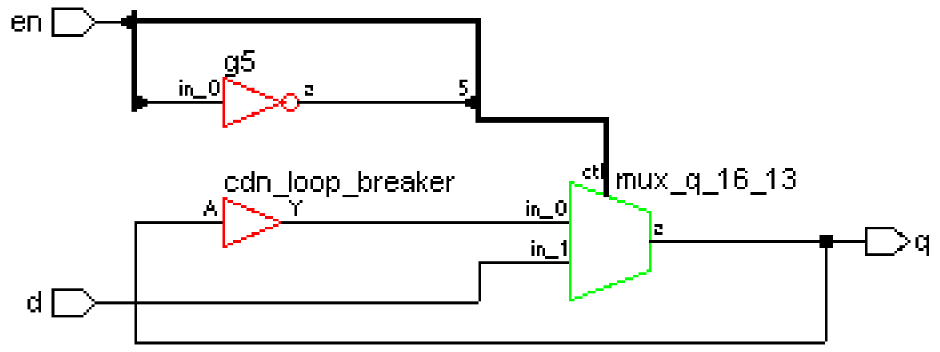
    ```
    rc:/> set_attribute hdl_latch_keep_feedback true
    ```

Example 3-6 RTL for hdl_latch_keep_feedback

```
module latch1(d,en,q);
    input d,en;
    output q;
    reg q;

    always @ (en or d) begin
        if (en)
                q <= d;
        else
                q <= q;
    end
endmodule
```

The corresponding schematic is shown in Figure 3-8.

Figure 3-8 Schematic of hdl_latch_keep_feedback true



- ❑ For the following command, RTL Compiler implements a latch with an enable signal specified in the RTL shown in Example 3-7:

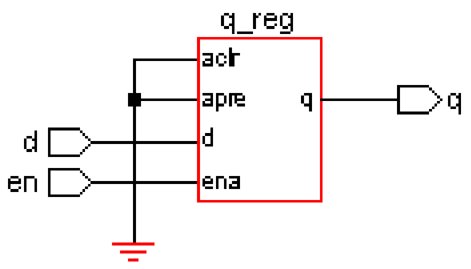
  ```
  rc:/ set_attribute hdl_latch_keep_feedback false
  ```

Example 3-7 RTL hdl_latch_keep_feedback false

```
module latch2(d,en,q);
    input d,en;
    output q;
    reg q;

    always @ (en or d) begin
        if (en)
                 q <= d;
        else
                 q <= q;
    end
endmodule
```

The corresponding schematic is shown in Figure 3-9.

Figure 3-9 Schematic hdl_latch_keep_feedback false



- `hdl_link_from_any_lib {true | false}`

  Default: `true`

  When set to `true`, links to a unique definition from any library. This is useful when a module is instantiated and there is no specification in the instantiation as to which definition it should be linked to. If there are multiple definitions, then the instantiated module is not linked.

- `hdl_max_loop_limit` *`integer`*

  Default: `1024`

  Determines the maximum number of iterations for unfolding a loop construct of any type. RTL Compiler stops and produces an error message when it needs to unroll a loop that has more iterations than the specified threshold.

- `hdl_max_recursion_limit` *`integer`*

  Default: `1024`

  Sets the maximum number of elaborations for recursive instantiations to prevent possible infinite recursions.

- `hdl_nc_compatible_module_linking {true | false}`

  Default: `true`

  When set to `true`, implements Native Compiler (NC) compatible binding rules for linking module or entity instantiations with corresponding definitions.

  To link a module instantiation and choose a definition that should be linked, see the rules that describe the order in which a search is applied to find a module to link in the *Attribute Reference for Encounter RTL Compiler*.

- `hdl_parameter {true | false}`

  Returns whether the specified parameter is visible within the ChipWare component. If the specified parameter was created with the `hdl_create parameter` command without its `-hdl_invisible` option, then the default value of this attribute will be `false`. If the specified parameter was created with the `-hdl_invisible` option, this attribute value becomes `true`. This attribute is valid on all parameters (`hdl_param` objects), not just those created by the `hdl_create parameter` command.

- `hdl_parameter_naming_style` *`string`*

  Default: `_%s%d`

  Specifies the format of the suffix added to the original module name for each parameter overwrite. For more information, see Naming Individual Bits of Array and Record Ports and Registers in *Using Encounter RTL Compiler*.

- `hdl_parameters` *`string`*

  Keeps track, in a Tcl list, both parameters explicitly set by the instantiating module and unset parameters, which use their default values while reading the top-level design. Also tracks attributes set through the `elaborate -parameters` command.

- `hdl_preserve_dangling_output_nets` `{true | false}`

  Default: `false`

  When set to `true`, RTL Compiler preserves the names of dangling output nets in designs that are read using the `read_netlist` command or the `read_hdl -netlist` command.

- `hdl_preserve_unused_registers` `{true | false}`

  Default: `false`

  When set to `true`, RTL Compiler does not remove registers (latches and flip-flops) that do not, directly or indirectly, affect any outputs. This can be used, for example, to keep registers that are only used to observe internal nets through scan chains in test mode.

- `hdl_proc_name` *`string`*

  If the `hdl_enable_proc_name` attribute is set to `true`, specifies for sequential elements either

  - The Verilog block identifier of the named always block that infers this sequential element
  - The VHDL label of the process that infers this sequential element

  If no name was given to the Verilog block or VHDL process, a tool-generated name is given.

  Note: This attribute is created during elaboration. After elaboration, it has no value for hierarchical instances, or for instances that are not sequential elements.

- `hdl_record_naming_style` *`string`*

  Default: `%s\[%s\]`

  Chooses a scheme to name individual bits of record ports and registers. The string argument must include `%s` to indicate the record name of the bus signal and a second `%s` to indicate the field name. Set this attribute before using the `elaborate` command.

  See Naming Individual Bits of Array and Record Ports and Registers in *Using Encounter RTL Compiler* for detailed examples.

- `hdl_reg_naming_style` *`string`*

  Default: %s`_reg%s`

  Specifies the format in which flops of vectored variables and latches of scalar variables are printed out. For more information, see Naming Individual Bits of Array and Record Ports and Register in *Using Encounter RTL Compiler.*

- .`hdl_report_case_info` `{true | false}`

  Default: `false`

  When set to `true`, RTL Compiler reports the `case` pragma information about how each `case` statement is synthesized for a module in the log file during elaboration, as shown in Example 3-8.

Example 3-8 hdl_report_case_info true

```
set_attribute hdl_report_case_info true /
Line No  Type      Full    Parallel
-------------------------------------
    11   case      NO        YES
    15   casez     AUTO      YES
    21   casex     AUTO      YES
    27   case      AUTO      YES
    10   case      AUTO      YES
-------------------------------------
```

- `hdl_search_path` *`Tcl_list`*

  Default: { . }

  Specifies a list of UNIX directories that RTL Compiler should search for files associated with the `read_hdl` command. The behavior is similar to the search path in UNIX.

  In Verilog, this attribute directs the search of Verilog files specified with the `read_hdl` command and `` `include`` files specified in Verilog code.

  In VHDL, this attribute directs the search of VHDL files specified with the `read_hdl` command.

- hdl_sync_set_reset *"Tcl list"*

  Default: `null`

  Specifies that RTL Compiler implement the listed signals using synchronous set and reset pins on a flip-flop if that logic controls a synchronous assignment.

  - For the following RTL, shown in Example 3-9, the `"reset"` signal is implemented using synchronous set and reset pins on a flip-flop, as shown in Figure 3-10:

    ```
    rc:/> set_attr hdl_sync_set_reset "reset"
    ```

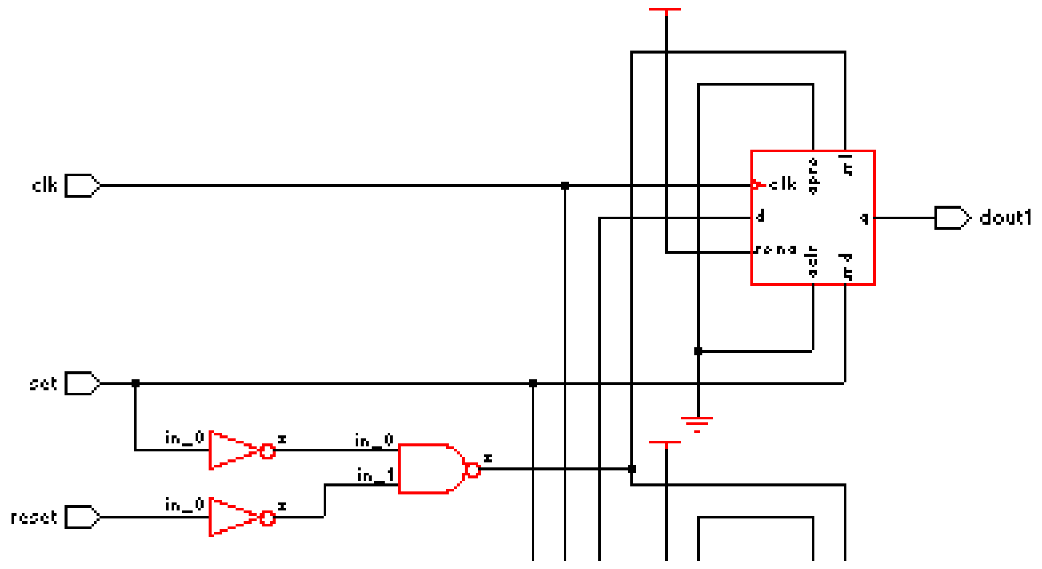
Example 3-9 RTL hdl_sync_set_reset "reset"

```
module sync(d,reset,clk,q);
    input reset,clk,d;
    output q;
    reg q;
    always @ (posedge clk)
        begin
            if (reset)
                q <= 1'b0;
            else
                q <= d;
    end
endmodule
```

Using this attribute has the same effect as using the `sync_set_reset` pragma in the RTL:

```
... //cadence sync_set_reset "Tcl list"
```

Figure 3-10 Schematic of Reset Signal

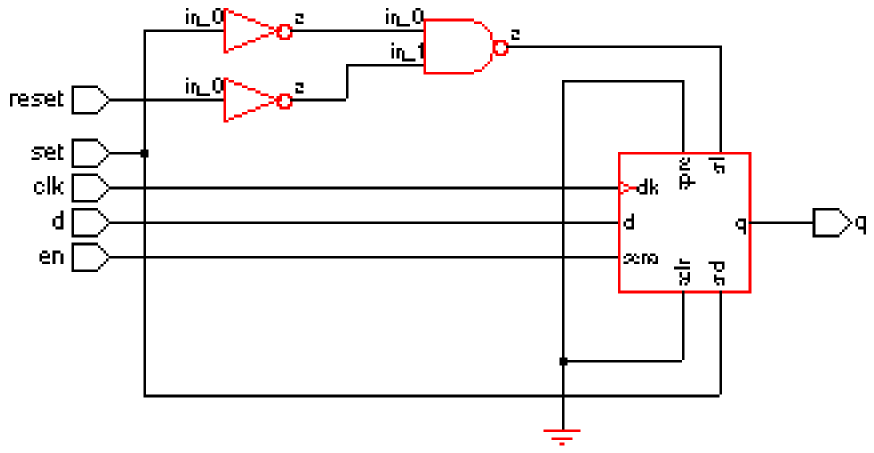


- ❑ In the RTL, shown in Example 3-10, RTL Compiler implements the set and reset operations using flip-flop synchronous set and reset pins, as shown in Figure 3-11.

Example 3-10 Implementing Flip-Flop Synchronous set and reset Pins in the RTL

```
module syncff(d,en,set,reset,clk,q);
    input d,en,set,reset,clk;
    output q;
    reg   q;
    always @ (posedge clk)
        begin //cadence sync_set_reset "set, reset"
            if (set)
                q <= 1'b1;
            else if (reset)
                q <= 1'b0;
            else if (en)
                q <= d;

        end
endmodule
```

Figure 3-11 Schematic of Set and Reset Operations



- `hdl_track_filename_row_col` `{true | false}`

  Default: `false`

  Enables or disables file/row/col information tracking. When you set this attribute to `false`, all the file, row, and column information is deleted.

  Set this attribute to `true` to enable file, row, column information before using the `elaborate` command.

- `hdl_unconnected_input_port_value` `{0 | 1 | X | Z | none}`

  Default: `none`

  Connects each undriven input pin in a module or cell instantiation to the specified value unless the `none` value is specified. If the `none` value is specified, undriven pins remain undriven.

- `hdl_undriven_output_port_value` `{0 | 1 | X | Z | none}`

  Default: `none`

  Connects each undriven output port in a module to the specified value unless the `none` value is specified. If the `none` value is specified, undriven ports remain unconnected.

- `hdl_undriven_signal_value` `{0 | 1 | X | none}`

  Default: `none`

  Connects each undriven signal, including undriven bits of a bus, to the specified value. If the `none` value is specified, undriven signals remain undriven.

- `hdl_use_default_parameter_values_in_design_name` `{true | false}`

  Default: `false`

  When set to `false` shortens the name of the parameterized module by using only the parameter values specified at instantiation, while setting the attribute to `true` uses all the available parameters in the module name.

- `hdl_use_parameterized_module_by_name` `{true | false}`

  Default: `false`

  When set to `true`, RTL Compiler tries to bind, for an `u1` instance of a parameterized `M` design with a parameter overwrite, a module or architecture named with the generated parameterized name, including parameter names and values.

  For example, for a Verilog instance `M #(1,5) u0();`, RTL Compiler tries to bind `M_width_1_depth_5`, rather than using the definition of module `M` with the parameter overwrite (width, 1) and (depth, 5).

- `hdl_use_port_default_value` `{true | false}`

  Default: `false`

  When set to `true`, RTL Compiler honors default initial values of input ports in a VHDL component declaration or entity declaration.

- `hdl_user_name` *`string`*

  Represents the name of the Verilog module or the VHDL entity from which the given design was derived. The design's name may differ from the `hdl_user_name` value and from the addition of suffixes for the module name uniquification.

- `hdl_use_techelt_first` `{true | false}`

  Default: `false`

  When set to `true`, RTL Compiler tries to bind, for an `u1` instance of design `M`, a gate from a technology library named `M`, rather than a module or architecture named `M`.

- `input_pragma_keyword` *`string`*

  Default: `get2chip g2c ambit synopsys pragma cadence`

  Specifies a keyword that RTL Compiler must consider as an input pragma when it encounters it as the first word in a Verilog or VHDL source comment.

Verilog-Specific Attributes

Table 3-2 Verilog-Specific Attributes

Attribute	Description

`hdl_language`
`{v1995 | v2001 | vhdl | sv}`
	Specifies the default HDL language mode assumed when you use the `read_hdl` command without specifying the language mode.
	Default: `v2001`

`hdl_v2001`	Sets or retrieves Verilog 2001 attributes of the following format:
	`(* attribute[=value]`
	`[, attribute[=value]]... *)...`
	These attributes apply to the design, instance, port, subdesign and subport objects.

VHDL-Specific Attributes

Table 3-3 VHDL-Specific Attributes

Attribute	Description

`hdl_vhdl_assign_width_mismatch`
`{false | true}`
	Controls whether to allow an assignment when lhs and rhs have the same array type but mismatching width.
	Default: `false`

`hdl_vhdl_case`
`{original | lower | upper}`
	Stores VHDL identifiers and operators in lower case, upper case, or the case given in the source file.
	Default: `original`

`hdl_vhdl_environment`
`{common | synopsys}`
	Specifies the selection of the predefined arithmetic libraries.
	Default: `common`

Attribute	Description

`hdl_vhdl_lrm_compliance`
 `{false | true}`
	When set to `true`, the `read_hdl` command enforces a more strict interpretation of the VHDL LRM.
	 Default: `false`

`hdl_vhdl_preferred_architecture`
 `string`
	Specifies the name of the preferred architecture to use with an entity when there are multiple architectures.
	 Default: `""`

`hdl_vhdl_read_version`
 `{ 1993 | 1987 }`
	Specifies the VHDL version when files are analyzed using `read_hdl`.
	 Default: `1993`

Global Versus Local User Control

This chapter discusses the arbitration between global and local control of HDL modeling settings. It also demonstrates use models and offers usage examples.

Global and User Control of Elaboration

This section only applies to all root attributes that have a local `hdl_arch` attribute of the same name. Before we discuss the attributes, it is useful to examine how the commands interact with the attributes:

- `read_hdl` creates `hdl_arch` objects, one for each `module` or `entity`.
- `elaborate` operates on hdl_arch objects. It creates design or subdesign objects, one for each `hdl_arch`.
- `synthesize` operates on design or subdesign objects.

You can affect how elaboration behaves by setting certain attributes before elaboration.

A global control mechanism operates conveniently at a global scale. A local control mechanism operates precisely at a specific local spot. The difference is in the granularity of user control over elaboration:

- Changing a global `root` attribute affects all modules (that is, all `hdl_arch` objects) in the design.
- Changing a local `hdl_arch` attribute only affects the particular module represented by that `hdl_arch` object.

The follow example accesses the `hdl_arch` objects called `test`:

```
rc> find /hdl_libraries -hdl_arch test*
/hdl_libraries/default/architectures/test
/hdl_libraries/default/architectures/test1
```

The following example accesses `hdl_arch` object attributes

```
rc> get_attribute start_source_line [find /hdl_libraries -hdl_arch test1]
```

The following `hdl_arch` attributes enhance the granularity of user control over elaboration. Each of these `hdl_arch` attributes comes from a `root` attribute of the same meaning:

- `hdl_error_on_blackbox`
- `hdl_error_on_latch`

- hdl_error_on_logic_abstract
- hdl_error_on_negedge
- hdl_ff_keep_explicit_feedback
- hdl_ff_keep_feedback
- hdl_latch_keep_feedback

Global control can be exercised by setting the value of a `root` attribute while local control can be exercised by setting the value of the `hdl_arch` attribute of a certain `module` or `entity`.

If an elaboration transformation has global but not local control, the following rules apply:

- It can be controlled by a `root` attribute.
- There is not an `hdl_arch` attribute to control this transformation.
- Setting of the `root` attribute is applied to all `hdl_arch` objects.
- The `elaborate` command consults with the `root` attribute.

If an elaboration transformation has local but not global control, the following rules apply:

- It can be controlled by an `hdl_arch` attribute.
- There is not a `root` attribute to control this transformation.
- Setting of an hdl_arch attribute is applied to that specific `hdl_arch` object.
- The `elaborate` command consults with the `hdl_arch` attribute of individual `hdl_arch` objects.

If an elaboration transformation has both global and local control, the following rules apply:

- It can be controlled by a `root` and an `hdl_arch` attribute.
- The `root` and `hdl_arch` attribute share the same name, the same initial or default value, and the same interpretation.
- Setting the `root` attribute also applies the setting to all `hdl_arch` objects.
- Setting the `hdl_arch` attribute applies the setting only to the specific `hdl_arch` object.
- The `elaborate` command consults with the `hdl_arch` attribute of individual `hdl_arch` objects.
- The `elaborate` command does not consult with the `root` attribute.

Global Versus Local Arbitration

This section only applies to all root attributes that have a local `hdl_arch` attribute of the same name. Each pair of `root` and `hdl_arch` attributes control whether a specific optimization or transformation should be applied during elaboration. Elaboration looks at the local `hdl_arch` attributes on a module-by-module basis to decide whether that transformation should be exercised for the particular module. Elaboration does not consult with the global attribute at all.

Local control allows you to set the attribute of an `hdl_arch` to a value that may or may not be the same as the value it currently has.

When applying global control, three sub-tasks take place.

1. The global `root` attribute is set to the given value, which may or may not be the same as the value it currently has.
2. The corresponding local attribute of all existing `hdl_arch` is set to that given value. This is equivalent to doing:

   ```
   rc:/> set_attribute $attr_name $new_value [find / -hdl_arch *]
   ```

   - ❑ Doing `set_attribute` on a local attribute is effective only if there is not any `set_attribute` on the global attribute later in the synthesis script.
   - ❑ Doing `set_attribute` on a local attribute is useless if the synthesis script later does `set_attribute` on the global attribute before elaboration.
3. The given value becomes the initial or default value of the corresponding local attribute for `hdl_arch` that will be created by subsequent `read_hdl` commands, if any. This is equivalent to doing:

   ```
   rc:/> set_attribute default_value $new_value \
       /object_types/hdl_arch/attributes/$attr_name
   ```

Note: Sub-task 3 is the only occasion in RTL Compiler where the default or initial value of an attribute (the corresponding local attribute) is changed by the synthesis script.

With each pair of global and local attributes:

- If the synthesis script only changes the global attribute, the tool behavior is the same as before.
- If the synthesis script changes the local attribute, the new scripting has no concerns about backward compatibility.
- If the synthesis script changes both global and local attributes, be aware of the tool behavior described above.

Use Model

For an attribute to affect elaboration, it needs to be set before elaboration. However, it can be set either before or after `read_hdl` or even between `read_hdl` commands.

The examples of use model in this section assumes:

- There is some transformation `xyz` that can be pursued during elaboration.
- There is a global root attribute `hdl_apply_xyz` for this transformation.
- There is a local `hdl_arch` attribute `hdl_apply_xyz` for this transform.

Globally Changed Setting

If you want to globally control how or whether elaboration applies some transformation `xyz` on all modules, set the `root` attribute anywhere before the `elaborate` command.

Here are the possible scenarios:

```
set_attribute hdl_apply_xyz $desired_value /
read_hdl
read_hdl
elaborate
```

or

```
read_hdl
set_attribute hdl_apply_xyz $desired_value /
read_hdl
elaborate
```

or

```
read_hdl
read_hdl
set_attribute hdl_apply_xyz $desired_value [find / -hdl_arch *]
elaborate
```

Locally Changed Setting

If you need to locally control how or whether elaboration applies some transformation `xyz` on specific modules, set the local attribute *after* the last `set_attribute` to the global attribute and before elaboration.

Here are the possible scenarios:

```
read_hdl
read_hdl
set_attribute hdl_apply_xyz $desired_value $some_hdl_arch'es
elaborate
```

or

```
read_hdl
set_attribute hdl_apply_xyz $desired_value $some_hdl_arch'es
read_hdl
elaborate
```

or

```
set_attr hdl_apply_xyz $other_value /
read_hdl
set_attribute hdl_apply_xyz $desired_value $some_hdl_arch'es
read_hdl
elaborate
```

or

```
set_attribute hdl_apply_xyz $other_value /
read_hdl
read_hdl
set_attribute hdl_apply_xyz $desired_value $some_hdl_arch'es
elaborate
```

or

```
read_hdl
set_attribute hdl_apply_xyz $other_value /
read_hdl
set_attr hdl_apply_xyz $desired_value $some_hdl_arch'es
elaborate
```

or

```
read_hdl
set_attribute hdl_apply_xyz $other_value /
set_attribute hdl_apply_xyz $desired_value $some_hdl_arch'es
read_hdl
elaborate
```

or

```
read_hdl
read_hdl
set_attribute hdl_apply_xyz $other_value /
set_attribute hdl_apply_xyz $desired_value $some_hdl_arch'es
elaborate
```

These are *invalid* scenarios:

```
read_hdl
read_hdl
set_attribute hdl_apply_xyz $desired_value $some_hdl_arch'es
set_attribute hdl_apply_xyz $other_value /
elaborate
```

or

```
read_hdl
set_attribute hdl_apply_xyz $desired_value $some_hdl_arch'es
read_hdl
set_attribute hdl_apply_xyz $other_value /
elaborate
```

or

```
read_hdl
set_attribute hdl_apply_xyz $desired_value $some_hdl_arch'es
set_attribute hdl_apply_xyz $other_value /
read_hdl
elaborate
```

Global Versus Local Examples

Getting values through `get_attribute`, the following rules apply.

- With a `root` attribute, its value is always what it is given by the latest `set_attribute` on itself. If no `set_attribute` has been used on this `root` attribute, its value is always the default value hard-coded in RTL Compiler.
- With an `hdl_arch` attribute, as indicated by `get_attribute`, its value can be changed by a `set_attribute` on the corresponding `root` attribute, as well as a `set_attribute` upon itself, whichever comes last. If no `set_attribute` has been exercised upon either this `hdl_arch` attribute or its corresponding `root` attribute, the value of the `hdl_arch` attribute is the default value hard-coded in RTL Compiler.

With a `root` attribute whose local `hdl_arch` control exists, whether the transformation is applied onto an `hdl_arch` depends solely on the attribute value at that `hdl_arch`. Elaboration does not consult with the `root` attribute at all.

Assume a design has three RTL files:

Example 3-11 Example RTL Files

```
top.v
    module tst (x, y, a, b, en, clk);
        input en, clk;
        input [2:0] a;    input [3:0] b;
        output [2:0] x;   output [3:0] y;
        fw3 u1 (x, a, en, clk);
        fw4 u2 (y, b, en, clk);
    endmodule

fw3.v
    module fw3 (y, a, en, clk);
        input en, clk;
        input [2:0] a;
        output [2:0] y;
        reg [2:0] y;
        always @(posedge clk)  if (en)  y <= a;
    endmodule

fw4.v
    module fw4 (y, a, en, clk);
        input en, clk;
        input [3:0] a;
        output [3:0] y;
        reg [3:0] y;
        always @(posedge clk)  if (en)  y <= a;
    endmodule
```

Example 3-12 Example Tcl Procedure

Assume the following examples all use this Tcl procedure:

```
proc rpt_then_elab_then_write {} {
    set value_at_root [get_attr hdl_ff_keep_feedback /]
    set value_at_fw3 [get_attr hdl_ff_keep_feedback \
                     [find / -hdl_arch fw3]]
    set value_at_fw4 [get_attr hdl_ff_keep_feedback \
                     [find / -hdl_arch fw4]]
    puts "RPT: root: $value_at_root"
    puts "RPT: fw3: $value_at_fw3"
    puts "RPT: fw4: $value_at_fw4"
    elaborate
    write_hdl
}
```

Example 3-13

If the synthesis script is any of the following:

```
set_attribute hdl_ff_keep_feedback true /
read_hdl top.v fw3.v fw4.v
rpt_then_elab_then_write
```

or

```
read_hdl top.v fw3.v fw4.v
set_attribute hdl_ff_keep_feedback true /
rpt_then_elab_then_write
```

or

```
read_hdl top.v
read_hdl fw3.v
set_attribute hdl_ff_keep_feedback true /
read_hdl fw4.v
rpt_then_elab_then_write
```

The logfile says:

```
RPT: root: true
RPT: fw3: true
RPT: fw4: true
```

The post-elaboration netlist is like:

```
        module fw3 (y, a, en, clk);
            input [2:0] a;
            input en, clk;
            output [2:0] y;
            wire [2:0] d;
            assign d[2:0] = en ? y[2:0] : a[2:0]; // by CDN_mux gates
            CDN_flop \y_reg[0] (... .d(d[0]), .sena(1'b1), .q(y[0]));
            CDN_flop \y_reg[1] (... .d(d[1]), .sena(1'b1), .q(y[1]));
            CDN_flop \y_reg[2] (... .d(d[2]), .sena(1'b1), .q(y[2]));
        endmodule
        module fw4 (y, a, en, clk);
            input [3:0] a;
            input en, clk;
            output [3:0] y;
            wire [3:0] d;
            assign d[3:0] = en ? y[3:0] : a[3:0]; // by CDN_mux gates
            CDN_flop \y_reg[0] (... .d(d[0]), .sena(1'b1), .q(y[0]));
            CDN_flop \y_reg[1] (... .d(d[1]), .sena(1'b1), .q(y[1]));
            CDN_flop \y_reg[2] (... .d(d[2]), .sena(1'b1), .q(y[2]));
            CDN_flop \y_reg[3] (... .d(d[3]), .sena(1'b1), .q(y[3]));
        endmodule
        module tst (x, y, a, b, en, clk);
            input en, clk;
            input [2:0] a;  output [2:0] x;
            input [3:0] b;  output [3:0] y;
            fw3 u1 (x, a, en, clk);
            fw4 u2 (y, b, en, clk);
        endmodule
```

Example 3-14

If the synthesis script is any of the following:

```
set_attribute hdl_ff_keep_feedback false /
read_hdl top.v fw3.v fw4.v
rpt_then_elab_then_write
```

or

```
read_hdl top.v fw3.v fw4.v
set_attr hdl_ff_keep_feedback false /
rpt_then_elab_then_write
```

or

```
read_hdl top.v
read_hdl fw3.v
set_attribute hdl_ff_keep_feedback false /
read_hdl fw4.v
rpt_then_elab_then_write
```

The logfile says:

```
RPT: root: false
RPT: fw3: false
RPT: fw4: false
```

The post-elaboration netlist is like:

```
        module fw3 (y, a, en, clk);
            input [2:0] a;
            input en, clk;
            output [2:0] y;
            CDN_flop \y_reg[0] (... .d(a[0]), .sena(en), .q(y[0]));
            CDN_flop \y_reg[1] (... .d(a[1]), .sena(en), .q(y[1]));
            CDN_flop \y_reg[2] (... .d(a[2]), .sena(en), .q(y[2]));
        endmodule
        module fw4 (y, a, en, clk);
            input [3:0] a;
            input en, clk;
            output [3:0] y;
            CDN_flop \y_reg[0] (... .d(a[0]), .sena(en), .q(y[0]));
            CDN_flop \y_reg[1] (... .d(a[1]), .sena(en), .q(y[1]));
            CDN_flop \y_reg[2] (... .d(a[2]), .sena(en), .q(y[2]));
            CDN_flop \y_reg[3] (... .d(a[3]), .sena(en), .q(y[3]));
        endmodule
        module tst (x, y, a, b, en, clk);
            input en, clk;
            input [2:0] a;  output [2:0] x;
            input [3:0] b;  output [3:0] y;
            fw3 u1 (x, a, en, clk);
            fw4 u2 (y, b, en, clk);
        endmodule
```

Example 3-15

If the synthesis script is any of the following:

```
set_attribute hdl_ff_keep_feedback true /
read_hdl top.v
read_hdl fw3.v
read_hdl fw4.v
set_attribute hdl_ff_keep_feedback false [find / -hdl_arch fw4]
rpt_then_elab_then_write
```

or

```
read_hdl top.v
read_hdl fw3.v
read_hdl fw4.v
set_attribute hdl_ff_keep_feedback true /
set_attribute hdl_ff_keep_feedback false [find / -hdl_arch fw4]
rpt_then_elab_then_write
```

or

```
read_hdl top.v
read_hdl fw3.v
set_attribute hdl_ff_keep_feedback true /
read_hdl fw4.v
set_attribute hdl_ff_keep_feedback false [find / -hdl_arch fw4]
rpt_then_elab_then_write
```

or

```
read_hdl top.v
set_attr hdl_ff_keep_feedback true /
read_hdl fw4.v
set_attr hdl_ff_keep_feedback false [find / -hdl_arch fw4]
read_hdl fw3.v
rpt_then_elab_then_write
```

The logfile says:

```
RPT: root: true
RPT: fw3: true
RPT: fw4: false
```

The post-elaboration netlist is like:

```
module fw3 (y, a, en, clk);
    input [2:0] a;
    input en, clk;
    output [2:0] y;
    wire [2:0] d;
    assign d[2:0] = en ? y[2:0] : a[2:0]; // by CDN_mux gates
    CDN_flop \y_reg[0] (... .d(d[0]), .sena(1'b1), .q(y[0]));
    CDN_flop \y_reg[1] (... .d(d[1]), .sena(1'b1), .q(y[1]));
    CDN_flop \y_reg[2] (... .d(d[2]), .sena(1'b1), .q(y[2]));
endmodule
module fw4 (y, a, en, clk);
    input [3:0] a;
    input en, clk;
    output [3:0] y;
    CDN_flop \y_reg[0] (... .d(a[0]), .sena(en), .q(y[0]));
    CDN_flop \y_reg[1] (... .d(a[1]), .sena(en), .q(y[1]));
    CDN_flop \y_reg[2] (... .d(a[2]), .sena(en), .q(y[2]));
    CDN_flop \y_reg[3] (... .d(a[3]), .sena(en), .q(y[3]));
endmodule
module tst (x, y, a, b, en, clk);
    input en, clk;
    input [2:0] a;  output [2:0] x;
    input [3:0] b;  output [3:0] y;
    fw3 u1 (x, a, en, clk);
    fw4 u2 (y, b, en, clk);
endmodule
```

Example 3-16

If the synthesis script is any of the following:

```
set_attribute hdl_ff_keep_feedback false /
read_hdl top.v
read_hdl fw3.v
read_hdl fw4.v
set_attribute hdl_ff_keep_feedback true [find / -hdl_arch fw4]
rpt_then_elab_then_write
```

or

```
read_hdl top.v
read_hdl fw3.v
read_hdl fw4.v
set_attr hdl_ff_keep_feedback false /
set_attr hdl_ff_keep_feedback true [find / -hdl_arch fw4]
rpt_then_elab_then_write
```

or

```
read_hdl top.v
read_hdl fw3.v
set_attr hdl_ff_keep_feedback false /
read_hdl fw4.v
set_attr hdl_ff_keep_feedback true [find / -hdl_arch fw4]
rpt_then_elab_then_write
```

or

```
read_hdl top.v
set_attr hdl_ff_keep_feedback false /
read_hdl fw4.v
set_attr hdl_ff_keep_feedback true [find / -hdl_arch fw4]
read_hdl fw3.v
rpt_then_elab_then_write
```

The logfile says:

```
RPT: root: false
RPT: fw3: false
RPT: fw4: true
```

The post-elaboration netlist is like:

```
        module fw3 (y, a, en, clk);
            input [2:0] a;
            input en, clk;
            output [2:0] y;
            CDN_flop \y_reg[0] (... .d(a[0]), .sena(en), .q(y[0]));
            CDN_flop \y_reg[1] (... .d(a[1]), .sena(en), .q(y[1]));
            CDN_flop \y_reg[2] (... .d(a[2]), .sena(en), .q(y[2]));
        endmodule
        module fw4 (y, a, en, clk);
            input [3:0] a;
            input en, clk;
            output [3:0] y;
            wire [3:0] d;
            assign d[3:0] = en ? y[3:0] : a[3:0]; // by CDN_mux gates
            CDN_flop \y_reg[0] (... .d(d[0]), .sena(1'b1), .q(y[0]));
            CDN_flop \y_reg[1] (... .d(d[1]), .sena(1'b1), .q(y[1]));
            CDN_flop \y_reg[2] (... .d(d[2]), .sena(1'b1), .q(y[2]));
            CDN_flop \y_reg[3] (... .d(d[3]), .sena(1'b1), .q(y[3]));
        endmodule
        module tst (x, y, a, b, en, clk);
            input en, clk;
            input [2:0] a;  output [2:0] x;
            input [3:0] b;  output [3:0] y;
            fw3 u1 (x, a, en, clk);
            fw4 u2 (y, b, en, clk);
        endmodule
```

4

Synthesizing Verilog Designs

Overview of Verilog Modeling

This chapter is organized for synthesizing Verilog RTL designs and provides links to the corresponding Verilog sections throughout the manual.

For information on using mixed Verilog and VHDL, Chapter 1, "Modeling HDL Designs" provides modeling guidelines in both languages in one convenient location.

Modeling Verilog Designs

Specifying Synthesis Pragmas

Using HDL Commands and Attributes

- HDL-Related Commands on page 130
- HDL-Related Attributes on page 131

Verilog-2001 Hardware Description Language Extensions

Verilog-2001 is the latest version of the IEEE 1364 Verilog HDL standard. The Verilog-2001 extensions are a superset of the existing Verilog-1995 language. These extensions increase design productivity and enhance synthesis capability. Prior knowledge and experience with Verilog-1995 is assumed.

The new Verilog-2001 language features supported in this release are explained in detail in the *IEEE 1364-2001 Verilog HDL standard Language Reference Manual* (LRM). For information on purchasing IEEE specifications go to
http://shop.ieee.org/store/ and click on *Standards.*

This section describes how to handle incompatibilities between the various Verilog versions and explains the new Verilog-2001 synthesis-specific features relevant to RTL synthesis. The features supported in this release include a reference to the corresponding chapter number of the Verilog-2001 LRM.

- Specifying Verilog-1995 and Verilog-2001 Modes of Parsing on page 169
- Using Generate Statements on page 169 (LRM 12.1.3)
- Specifying Multidimensional Arrays on page 174 (LRM 3.10)
- Specifying Automatic Functions and Tasks on page 175 (LRM 10)
- Passing Parameters by Name on page 175 (LRM 12.2.2.2)
- Using a Comma-Separated Sensitivity List on page 176 (LRM 9.7.4)
- Using ANSI-Style Input and Output Declarations on page 176 (LRM 12.3.4)
- Using Variable Part Selects on page 177 (LRM 4.2.1)
- Using Constant Functions on page 177 (LRM 10.3.5)
- Using New Preprocessor Directives on page 178 (LRM 19)

In addition, the following HDL extensions are supported, but are not described:

- Signed arithmetic extensions
- Combinational logic sensitivity list
- Automatic width extension beyond 32 bits for `'bz`, `'bx`
- Sized and typed parameters
- Localparams

- Combined port and data type declarations
- Enhanced conditional compilation
- `` `line `` compiler directive
- Attributes

Specifying Verilog-1995 and Verilog-2001 Modes of Parsing

➤ To handle potential incompatibilities, RTL Compiler supports separate Verilog-2001 and Verilog-1995 modes of parsing using the following attribute:

```
set_attribute hdl_language {v1995 | v2001| vhdl |sv}
```

In addition to enabling Verilog parsing for Verilog-1995 and Verilog-2001, the `hdl_language` attribute also turns on language-specific error checks.

In most cases, a Verilog-2001 design behaves like a Verilog-1995 design. Verilog-2001 adds several new keywords to the Verilog language. Older models, which happen to use one of these new reserved words, will not work with a Verilog-2001 simulator or other software tools. For example, `generate` is a new keyword in Verilog-2001. Therefore, a Verilog-1995 design that has a `generate` wire name will not compile under Verilog-2001 rules.

Using Generate Statements

Use Verilog `generate` statements to conditionally compile concurrent constructs. The Verilog-2001 `generate` statements are modeled on VHDL `generate` statements.

Using Concurrent Begin and End Blocks

Use the `begin` and `end` keywords to group concurrent statements within a `generate` statement. A `begin` and `end` block must be labeled if declarations are included within it. There are three types of `generate` statements:

- Specifying the if generate Statement – Performs a set of concurrent statements if a specified condition is met.
- Specifying the case generate Statement – Behaves like a nested `if` statement, and selects from a set of concurrent statements.
- Using the for generate Statement – Replicates a set of concurrent statements.

The `if`, `case`, and `for generate` statements provide different ways of conditionally compiling a declaration, a concurrent statement, or a block of declarations and concurrent statements.

Note: The condition must not depend on dynamic values, such as the values of wires or registers. The `if generate` condition, the `case generate` expression and choices, and the `for generate` loop bounds must be constant expressions.

For both while and for statements, the loop termination condition must be known at elaboration time.

Specifying the if generate Statement

Use the `if generate` statement to conditionally generate a concurrent statement, as shown in Example 4-1.

Example 4-1 Specifying the if generate Statement

```
module top  (d,q);
    input d;
    output q;
    parameter p1=1,p2=2;
    generate if (p1 == p2)
        assign q = d;
    else
        assign q = ~d;
    endgenerate
endmodule
```

In this example, one of two possible assignment statements is generated depending on the values of the parameters. If the condition `p1 == p2` evaluates to `true`, taking into account any parameter overrides or defparams, then the result of the `if generate` statement is that the first assignment statement will be processed and the second will be ignored. Otherwise, only the second assignment will be processed.

The determination of which concurrent statement to process is made after the design has been linked together and the module instantiations and defparams have been processed.

Generate statements let you choose concurrent models (a particular instance) based on the selection criteria, as shown in Example 4-2.

Example 4-2 Specifying the if generate Statement

```
module crc_gen (a,b,crc_out);

parameter a_width = 8,b_width = 15;
parameter crc_en = 1, crc8 = 1;
input [a_width-1:0] a;
input [b_width-1:0] b;
output crc_out;
generate
    if ((crc_en == 1`b1) & (crc8 == 1`b1))
        CRC8 #(a_width) U1 (a, crc_en, crc_out);
            //Instantiate an 8 bit crc generator
    else
        CRC16 #(b_width) U1 (b, crc_en, crc_out);
            // Instantiate a 16 bit crc generator
endgenerate// The generated instance is U1
endmodule
```

Specifying the case generate Statement

Use a `case generate` statement for multi-way branching in a functional description, as shown in Example 4-3.

Example 4-3 Specifying the case generate Statement for Multi-Way Branching

```
module top (d,q);
    input d;
    output q;
    parameter p=2;
    generate case (p)
            1: assign q = d;
            2: assign q = ~d;
            3: assign q = 1'b1;
      default: assign q = 1'b1;
        endcase
    endgenerate
endmodule
```

The value of `p` determines which one of the assignment statements is processed. The `p case` expression is evaluated after the design has been linked together.

A `case generate` statement permits modules, lets you define primitives, and lets `initial` and `always` blocks be conditionally instantiated into another module based on a `case` construct, as shown in Example 4-4.

Example 4-4 Using the case generate Statement to Define Primitives

```
module top (en, reset, preset, datain, dataout);
parameter width=2;
input en, reset, preset;
input [width:0] datain;
output [width:0] dataout;

generate
    case (width)
            1: counter_2bitx1 counter_2(en, reset, preset, datain, dataout);
            // 2 bit counter implementation
            2: counter_3bitx1 counter_3(en, reset, preset, datain, dataout);
            // 3 bit counter implementation
      default: counter_4bit counter_4 (en, reset, preset, datin, dataout);
        // others - 4 bit counter implementation
    endcase
endgenerate // generated instance is counter_3
endmodule
```

Using the for generate Statement

Use a `for generate` statement to replicate a concurrent block. The `for generate` statement uses a `genvar`.

Specifying Genvar Declarations

A `genvar` is a new declaration that resembles an integer declaration, except that it is used only within a `for generate` statement. A `genvar` is a 32-bit integer that is treated as a constant when referenced. Assign a `genvar` value only in a `for generate` statement between the parentheses following the keyword `for`, as shown in Example 4-5.

Example 4-5 Specifying the for generate Statement

```
module top(b,c,a);
    input [7:0] b,
    input c;
    output [7:0] a;
    genvar i;
    generate
    for (i = 0; i <= 7; i = i + 1)
        begin : blah
            assign a[i] = b[i] & c;
        end
    endgenerate
endmodule
```

Nest a `for generate` statement to generate multi-dimensional arrays of component instances or other concurrent statements. In Example 4-6, eight copies of the assignment statement are created. In each copy, any reference to the genvar `'i'` is replaced by its value during iteration. Therefore, the generate statement shown in Example 4-5 is equivalent to the following:

```
module
    assign a[0] = b[0] & c;
    assign a[1] = b[1] & c;
    assign a[2] = b[2] & c;
    assign a[3] = b[3] & c;
    assign a[4] = b[4] & c;
    assign a[5] = b[5] & c;
    assign a[6] = b[6] & c;
    assign a[7] = b[7] & c;
endmodule
```

The `for generate` statement, like the procedural `for` statement, is restricted to the following form:

```
for (i = <expr>; i <relop> <expr>; i = i <addop> <expr>)
```

Example 4-6 Specifying the for generate Statement

```
module top(a,b,c,sum);
    input [3:0] a,b,c;
    output [3:0] sum;
    wire [3:0] t;
    parameter size = 4;
    genvar i;
    generate
        for (i = 0; i < size; i = i + 1) begin:bit
             xor g1 ( t[i], a[i], b[i], c[i]);
             and g2 ( sum [i], t[i], c[i] );
        end
    endgenerate
endmodule
// Generated instance name are:
// xor gates : bit[0].g1, bit[1].g1, bit[2].g1 bit[3].g1
// and gates: bit[0].g2, bit[1].g2, bit[2].g2, bit[3].g2
```

Specifying Multidimensional Arrays

In Verilog-1995, only one dimensional arrays of `reg` are allowed. In contrast, Verilog-2001 allows multi-dimensional arrays of `wire` and `reg` (See Example 4-7). Verilog-2001 allows reading and writing array words and bits within array words, but does not allow reading or writing of array slices or whole arrays.

Example 4-7 Multi-Dimensional Arrays of wire and reg

```
reg [7:0] tmp;

-- one-dimensional array of reg

reg [7:0] ml[3:0];        //legal in Verilog-1995 and 2001
reg [7:0] m2[3:0];        // legal in Verilog-1995 and 2001

-- one- and two-dimensional arrays of wire

wire [7:0] w1[3:0];       // illegal in Verilog-1995 legal in 2001
wire [7:0] w2[3:0] [2:0];// illegal in Verilog-1995, legal in 2001

-- two-dimensional arrays of reg

reg [7:0] al[3:0] [2:0];  // illegal in Verilog-1995, legal in 2001
reg [7:0] a2[3:0] [2:0];  // illegal in Verilog-1995, legal in 2001

-- reading and writing within an array

m1[1] = tmp;              // legal in Verilog-1995, 2001
tmp = m1[1];              // legal in Verilog-1995, 2001
```

Specifying Automatic Functions and Tasks

Verilog-1995 functions or tasks use static memory for arguments and local variables, which is why a task enable is not permitted in a concurrent context. If two tasks start at the same time, then they will write over each other's data.

Verilog-2001 includes re-entrant procedures that are implemented so that more than one process can perform it at the same time without conflict. By using the `automatic` keyword to mark a task or function that performs in a per-call context, just as C or VHDL functions or procedures do, Verilog compilers treat the variables inside of the task as unique stacked variables. The parameters and local variables for these procedures are allocated immediately when they are called then they are discarded when the procedures exit.

RTL Compiler treats Verilog functions and tasks as automatic procedures, whether the keyword `automatic` is specified or not. For this reason, synthesis of a non-automatic function or task, which relies on static allocation of local variables, will produce a simulation mismatch.

Passing Parameters by Name

Verilog-1995 defines two ways to change parameters for instantiated modules: parameter re-definition and `defparam` statements.

Verilog-2001 lets you specify module instance parameters, such as module instance ports by name, as shown in Example 4-8.

Example 4-8 Specifying Module Instance Parameters by Name

```
mod #(.width(1), .length(2)) ul(q,d);
```

Passing parameters by name is similar to `defparam` statements, except only the parameters that change are referenced in named port instantiations.

Example 4-9 Using the defparam Keyword

```
defparam ul.width = 1;
defparam u1.length = 2;
mod ul (q,d);
```

Using a Comma-Separated Sensitivity List

Verilog-1995 uses the keyword `or` as a separator between signals in the sensitivity list. Verilog-2001 lets a comma take the place of the `or` keyword in an event list, as shown in Example 4-10.

Example 4-10 Using a Comma-Separated Sensitivity List

```
module top(clk,reset,d,q);
    input clk,reset,d;
    output q;
    reg q;
    always @ (posedge clk, negedge reset) //signals separated with a comma
        begin
            if (!reset)
                q <= 0;
            else
                q <= d;
        end
endmodule
```

Using ANSI-Style Input and Output Declarations

The Verilog-1995 mode uses the older Kernighan and Ritchie C language syntax to declare module ports, as shown in Example 4-11, which requires that module header ports be declared up to three times: in the module header port list, in an output port declaration, and in a `reg` data-type declaration. Verilog-2001 updates the syntax for declaring ports and parameters in a more ANSI C fashion, as shown in Example 4-12, that combines the header port list, port direction, and data-type declarations into a single declaration.

Example 4-11 Verilog-1995 Style Declaration

```
module m(q, d);
parameter p = 2;
output [p-1:0] q;
reg [p-1:0]q;
input [p-1:0] d;
wire [p-1:0] d;
always @(d)
    q = d;
endmodule
```

Example 4-12 Verilog-2001 ANSI C-like Declaration

```
module m #(parameter p = 2) (output reg [p-1:0] q, input wire [p-1:0] d);
always @(d)
    q = d;
endmodule
```

Use this enhancement in functions and tasks to make port declarations more compact.

Using Variable Part Selects

Verilog-1995 permits variable bit selects of a vector, but the part selects must be constant; thus, you cannot use a variable to select a specific byte out of a word.

Verilog-2001 lets a slice have a variable base offset and a constant width. This means that the starting point of the part select can vary during simulation run time, but the width of the part select remains constant, as shown in Example 4-13.

Example 4-13 Variable Part Select

```
wire [18:0] d;
wire [3:0] x;
wire [3:0] q;
assign q = d[x+:4]; //is equivalent to the following:
assign q = {d[x+3], d[x+2], d[x+1], d[x]};
```

Using Constant Functions

A constant expression is required in certain contexts, for example, when specifying a range in a declaration or a part select. In Verilog-1995, a constant expression is either a literal, a parameter, or some arithmetic expression whose operands are constant expressions. Verilog-2001 allows a function call to appear in a constant expression in certain circumstances. Mainly, the arguments to the function must be constant expressions, and the function must compute its result entirely on the basis of its arguments.

In Example 4-14, the `min` and `max` functions are used to size the declaration of `wire x`. Because these functions are called with constant arguments and return a result based only on their arguments, their calls are considered constant expressions. In Verilog-1995, it is illegal to use a function call in sizing a declaration.

Example 4-14 Specifying a Function Call in a Constant Expression

```
module m;

parameter pl = 1, p2 = 2;
wire [max(pl,p2):min(pl,p2)] x;

    function min;
        input x, y;
        integer x, y;
            min = x < y ? x : y;
    endfunction

    function max;
        input x, y;
        integer x, y;
            max = x > y ? x : y;
    endfunction
endmodule
```

Using New Preprocessor Directives

Preprocessor directives let you define and use macro definitions, file inclusion, and conditional compilation.

Verilog-1995 supports conditional compilation using only a few compiler directives, such as `` `ifdef``, `` `else``, and `` `endif``.

Verilog-2001 adds the following C-like preprocessor directives:

- Using the ifndef Directive (comparable to `#ifndef`)
- Using the line Directive (comparable to `#line`).
- Using the elsif Directive (comparable to `#elif`)

Using the ifndef Directive

Use an `` `ifndef`` directive, as shown in Example 4-15, to discard code in a program if an identifier is defined as a macro. If the `ifndef` text macro identifier is defined, then the `ifndef` group of lines is ignored.

Example 4-15 Using the `ifndef Directive

```
`define first_block
`ifdef first_block
    `ifndef second_nest
        initial $display ( first block is defined );
    `else
        initial $display ("first block and second_nest defined");
    `endif
...
```

Using the line Directive

The `` `line `` directive is mainly used by a source preprocessor to relate the processed output back to the original source file. Tracking the filenames of Verilog source files and the line numbers in the files is useful for error messages and source code debugging.

Use the `` `line `` directive to change the source file and the line number. For example, if your Verilog file is called `foo.v`:

```
foo.v:
    module m;
        some_syntax_error
```

then you will see a message when using the `read_hdl` command pointing to a syntax error on line 2 of `foo.v`. However, if you use the `` `line `` directive, then the compiler thinks it is looking at a different file or line. For example:

```
foo.v:
    module m;
`line 1 “bar.v” 25
        some_syntax_error
```

The `read_hdl` command message reports that the syntax error occurred on line 25 of `bar.v` (`bar.v` is an example file name). Even if there are no syntax errors, the line number and file name given in the `` `line `` directive can affect other reports, such as messages from `elaborate`, or the line number and file name on netlist objects.

Using the elsif Directive

The `` `elsif `` directive must appear after an `` `ifdef `` or `` `ifndef `` directive. The `` `elsif `` directive is short hand for `` `else...`ifdef...`endif ``. For example:

```
`ifdef x
    ...
`elsif y
    ...
    `endif
```

is equivalent to:

```
`ifdef x
    ...
`else
    `ifdef y
        ...
    `endif
`endif
```

Supported Verilog Compiler Directives

The `read_hdl` command supports and interprets the following Verilog HDL compiler directives:

- `` `define ``
- `` `ifdef ``
- `` `ifndef ``
- `` `else ``
- `` `elsif ``
- `` `endif ``
- `` `include ``
- `` `undef ``
- `` `default_nettype ``
- `` `line ``

Supported Macros

The Verilog parser in RTL Compiler supports the following predefined macros:

- `CDS_TOOL_DEFINE`
- `SYNTHESIS`

The `CDS_TOOL_DEFINE` macro is supported by all Cadence tools that read Verilog.

The `SYNTHESIS` macro is required by Section 6.2 “Compiler directives and implicit synthesis defined macros” of the IEEE Std 1364.1-2002 LRM.

RTL Compiler supports these built-in macros in both RTL mode and structural mode.

Supported Verilog Modeling Constructs

- Verilog and Verilog-2001 Constructs and Level of Support on page 181
- Notes on Verilog Constructs on page 188

Verilog and Verilog-2001 Constructs and Level of Support

Table 4-1 lists the Verilog constructs supported by RTL Compiler and classifies the level of support by one of the following categories:

- Synthesized fully (Full)
- Synthesized partially or in specific contexts (Partial)
- Construct is ignored and a warning is generated (Ignored)
- Construct is unsupported and an error message is generated (No)

Wherever possible, restrictions are listed to describe the partially supported language constructs. The extension column specifies whether the construct is a Verilog-2001 extension, otherwise the construct is Verilog.

Table 4-1 Verilog Constructs and Level of Support

Group	**Construct**	**Suppo rt**	**Extension**
Basic	Identifiers	Full	
	Escaped identifiers	Full	
	Sized constants (`b, o, d, h`)	Full	
	Unsized constants `2'b11, 3'07, 32'd123, 8'hff`	Full	
	Signed constants (s) `3'bs101`	Full	Verilog-2001
	String constants	Full	
	Real constants	No	
	Use of `z`, `?` in constants	Full	
	Use of `x` in constants	Full	

Table 4-1 Verilog Constructs and Level of Support, *continued*

Group	Construct	Suppo rt	Extension
	`module`, `endmodule`	Full	
	`macromodule`	Full	
	Hierarchical references	No	
	`//`comment	Full	
	`/*`comment`*/`	Full	
	System tasks `$display`	Ignored	
	System functions Only `$signed` and `$unsigned`	Partial	
	ANSI-style module, task, and function port lists See ANSI-Style Declarations for more information.	Full	Verilog-2001
	Attributes	Ignored	Verilog-2001
Data types	`wire`, `wand`, `wor`, `tri`, `triand`, `trior`	Full	
	`tri0`, `tri1`	No	
	`supply0`, `supply1`	Full	
	`trireg` treated as `wire`	Partial	
	`reg`, `integer`	Full	
	`real`	No	
	`time`	No	
	`event`	No	
	`parameter`	Full	
	Range and type in parameter declaration	Full	Verilog-2001
	`scalared`, `vectored`	Ignored	
	`input`, `output`, `inout`	Full	

Table 4-1 Verilog Constructs and Level of Support, *continued*

Group	Construct	Suppo rt	Extension
	Memory For example, `reg [7:0] x [3:0];`	Full	
	N-dimensional arrays	Full	Verilog-2001
Drive strengths		Ignored	
Module instances	Connect port by name, order	Full	
	Override parameter by order	Full	
	Override parameter by name	Full	Verilog-2001
	`defparam`	Partial	
	Constants connected to ports	Full	
	Unconnected ports	Full	
	Expressions connected to ports	Full	
	Delay on built-in gates	Ignored	
Generate statements	`if` generate	Full	Verilog-2001
	`case` generate	Full	Verilog-2001
	`for` generate	Full	Verilog-2001
	`concurrent begin end` blocks	Full	Verilog-2001
	`genvar`	Full	Verilog-2001
Built-in primitives	`and`, `or`, `nand`, `nor`, `xor`, `xnor`	Full	
	`not`, `notif0`, `notif1`	Full	
	`buf`, `bufif0`, `bufif1`	Full	
	`tran`	Full	
	`tranif0`, `tranif1`, `rtran`, `rtranif0`, `rtranif1`	No	
	`pmos`, `nmos`, `cmos`, `rpmos`, `rnmos`, `rcmos`	No	
	`pullup`, `pulldown`	No	

Table 4-1 Verilog Constructs and Level of Support, *continued*

Group	**Construct**	**Support**	**Extension**
User defined primitives (UDPs)	`primitive`	No	
	`table`	No	
Operators and expressions	`+, -` (binary and unary)	Full	
Report operators and expressions	`/, %` See Notes on Verilog Constructs on page 188	Full	
	`*`	Full	
	`~`	Full	
Bitwise operations	`&, |, ^, ~^, ^~`	Full	
Reduction operations	`&, |, ^, ~&, ~|, ~^, ^~`	Full	Verilog-2001
	`!, &&, ||`	Full	Verilog-2001
	`==, !=, <, <=, >, >=`	Full	Verilog-2001
	`<<, >>`	Full	Verilog-2001
	`<<< >>>`	Full	Verilog-2001
	`{}, {n{}}`	Full	Verilog-2001
	`?:`	Full	Verilog-2001
	function call	Full	Verilog-2001
	`===, !==`	No	
	`**` *Supported only when both the operands are constants.	Partial	Verilog-2001
Event control	`event or`	Full	
	`@`	Partial	
	`delay` and `wait` (#)	Ignored	
	`event or` using comma syntax	Full	Verilog-2001

Table 4-1 Verilog Constructs and Level of Support, *continued*

Group	Construct	Suppo rt	Extension
	`posedge`, `negedge`	Partial	
	`wait`	Ignored	
	Intra-assignment event control	Ignored	
	Event trigger (`->`)	No	
Bit and part selects	Bit select	Full	
	Bit select of array element	Full	Verilog-2001
	Constant part select **Note:** The bounds of a part select may be elaboration-time constants.	Full	
	Variable part select (`+:`, `-:`) **Note:** Part select of higher dimensions of multidimensional array is not supported.	Partial	Verilog-2001
	Variable bit-select on left side of an assignment	Full	Verilog-2001
Continuous assignments	`net` and `wire` declaration	Full	
	Using assign	Full	
	Using delay	Ignored	
Procedural blocks	`always` (exactly one @ required)	Partial	
	`initial`	Ignored	
Procedural statements	;	Full	
	`begin-end`	Full	
	`if`, `else`	Full	
	`repeat`* The `repeat` statement must have a constant repeat count.	Full	
	`case`, `casex`, `casez`, `default`	Full	
	Task enable	Full	

Table 4-1 Verilog Constructs and Level of Support, *continued*

Group	Construct	Suppo rt	Extension
	`for` (constant bounds, only + and - operation on index)* The `for` statement must have constant bounds. **Note:** The loop termination condition must be known when elaborating the design.	Partial	
	`while`* The `while` statement must have constant bounds. **Note:** The loop termination condition must be known when elaborating the design.	Partial	
	`forever`* The `forever` statement must contain a disable statement.	Partial	
	*A loop is unrolled to a maximum count specified in `hdl_max_loop limit`		
	`disable` The `disabl`e statement must be applied to an enclosing task or named block.	Partial	
	`fork`, `join`	No	
Procedural assignments	Blocking (=) assignments	Full	
	Non-blocking (<=) assignments	Full	
	Procedural continuous assignments (`assign`)	No	
	`deassign`	No	
	`force`, `release`	No	
Functions and tasks	`function`	Full	

Table 4-1 Verilog Constructs and Level of Support, *continued*

Group	Construct	Suppo rt	Extension
	`task`	Full	
	Automatic tasks and functions	Full	Verilog-2001
Named blocks	Named block creation	Full	
	Local variable declaration	Full	
Specify block	`specify`	Ignored	
	`specparam`	Ignored	
	Module path delays	Ignored	
Compiler directives	`` `define ``	Full	
	`` `undef ``	Full	
	`` `resetall ``	Full	
	`` `ifndef ``, `` `elsif ``, `` `line ``	Full	Verilog-2001
	`` `ifdef ``, `` `else ``, `` `endif ``	Full	
	`` `include ``	Full	
Assertions	`assert`	Ignored	
	`assume`	Ignored	
	`cover`	Ignored	

Notes on Verilog Constructs

- For Verilog module instances, there is limited support for `defparam` using hierarchical names. The `defparam` must refer to a module instance in the current module.
- A `for` and `while` statement is unrolled to a maximum count specified in the `hdl_max_loop_limit` attribute. The loop termination condition must be known when elaborating the design.
- The Verilog-2001 `$signed` and `$unsigned` system functions are also supported in the Verilog 1995 mode.
- A single variable cannot have both blocking and non-blocking assignments in an `always` block as shown in Example 4-16.

Example 4-16 Bitwise Assignment Restriction

```
module TOP(a,b,o);
    input a, b;
    output o;
    reg o;
    always @(a or b) begin:comb
        o = a;
        o <= b;
    end
endmodule

//Results in the following error:

Error   : Variables written with both blocking and nonblocking assignments are not
supported.[ELAB-VLOG-1400]

   : Variable `o' in block `comb' in file `top.v' at line 8, column 5
     Always block `comb' contains unsynthesizable constructs
     Module `TOP' contains errors and cannot be elaborated
```

- All Verilog conditional assignments must be either blocking or non-blocking or an error message displays.
- If a signal has multiple, non-blocking assignments, RTL Compiler uses the order of the assignments to determine the priority as specified in the Verilog Language Standard IEEE Std.1364-1995. For example, RTL Compiler generates the same logic for the two modules shown in Example 4-17 on page 189.

Example 4-17 Multiple Non-Blocking Assignments versus Single Assignment

```
module multiple_assign(clk,rst,set,q);
    input   clk,rst,set;
    output  q;
    reg     q;
    always @ (posedge clk) begin
        if (set)
            q <= 1'b1;
        if (rst)
            q <= 1'b0;
    end
endmodule
```

```
module single_assign(clk,rst,set,q);
    input clk,rst,set;
    output q;
    reg       q;
    always @ (posedge clk) begin
        if (rst)
            q <= 1'b0;
        else if (set)
            q <= 1'b1;
    end
endmodule
```

Supported SystemVerilog Hardware Description Language Constructs

RTL Compiler supports the synthesizable subset of SystemVerilog 1800-2009. This standard represents a merger of two previous standards: IEEE Std1364™-2005 Verilog hardware description language (HDL) and IEEE Std 1800-2005.

Table 4-2 lists the level of support for the SystemVerilog 1800-2009 constructs and indicates the level as fully supported (Full), partially supported (Partial), and ignored (Ignored). The chapter and section numbers follow the IEEE1800-2009 standard. For more information on some of the restrictions, refer to Notes on SystemVerilog Constructs on page 202.

Table 4-2 Supported SystemVerilog Constructs

Construct		Chapter & Section	Support
Design and verification building blocks		**3**	
Compilation and elaboration		3.12	
	compilation units	3.12.1	Full
Simulation time units and precision		3.14	
	timeunit and timeprecision keywords	3.14.2.2	Ignored
Lexical Conventions		**5**	
Numbers		5.7	
	Integer literal constants	5.7.1	Full
	Signed integer	5.7.1	Full
	Unsigned integer	5.7.1	Full
	Unsigned number	5.7.1	Full
	Real literal constants	5.7.2	Full
Time literals		5.8	Ignored
String literals		5.9	Full
Structure literals		5.10	Full
Array literals		5.11	Full
Attributes		5.12	Ignored

Table 4-2 Supported SystemVerilog Constructs , *continued*

Data Types		**6**	
Singular and aggregate types		6.4	Full
Nets and variables		6.5	
	Variable	6.5	Full
Variable declarations		6.8	Full
Vector declarations		6.9	Full
Implicit declarations		6.10	Full
Integer data types		6.11	
	shortint, int, longint, byte, bit, logic, time	6.11	Full
	Integral types	6.11.1	Full
	2-state and 4-state data types	6.11.2	Full
	Signed and unsigned integer types	6.11.3	Full
Void data type		6.13	Full
User-defined types		6.18	
	Simple typedef	6.18	Full
	Forward typedef	6.18	Full
	Interface-based typedef	6.18	Not supported
Enumerations		6.19	
	With data type declaration	6.19	Full
	Without data type declaration	6.19	Full
	Defining new data types as enumerated types	6.19.1	Full
	Element range (name)	6.19.2	Full
	Element range (name = C)	6.19.2	Full
	Element range (name[N])	6.19.2	Full
	Element range (name[N]= C)	6.19.2	Full
	Element range (name[N: M])	6.19.2	Full

Table 4-2 Supported SystemVerilog Constructs , *continued*

	Element range (name[N: M]= C)	6.19.2	Full
	Type checking	6.19.3	Partial
	Enumerated types in numerical expressions	6.19.4	Full
	Enumerated type methods	6.19.5	
	First	6.19.5.1	Full
	Last	6.19.5.2	Full
	Next	6.19.5.3	Full
	Prev	6.19.5.4	Full
	Num	6.19.5.5	Full
	Name	6.19.5.6	Full
Constants		6.20	
	Parameter	6.20.1	Full
	Aggregate Parameter types **Note:** Packed and unpacked arrays as parameters are supported. Other complex parameter types are not supported.	6.20.2	Partial
	Type parameters	6.20.3	Full
	Local parameters (localparam)	6.20.4	Full
	Specify parameters (specparam)	6.20.5	Ignored
	Const constants (parsed and ignored)	6.20.6	Full
Scope and lifetime		6.21	Full
	static/ auto task/ function/ block data	6.21	Partial
Type compatibility		6.22	
	Matching types	6.22.1	Partial
	Equivalent types	6.22.2	Partial

Table 4-2 Supported SystemVerilog Constructs , *continued*

	Assignment compatible	6.22.3	Partial
	Cast compatible	6.22.4	Partial
	Type incompatible	6.22.5	Partial
Type operator		6.23	Full
Casting		6.24	
	Integer type	6.24.1	Full
	Non-integer type	6.24.1	Full
	Type identifier	6.24.1	Full
	Parameter identifier	6.24.1	Full
	Signed and unsigned integer types	6.24.1	Full
	String	6.24.1	Not supported
	Const	6.24.1	Not supported
	$cast dynamic casting	6.24.2	Full
	Bit stream casting	6.24.3	Full
Aggregate data types		**7**	
Structures		7.2	
	Unpacked structures	7.2	Full
	Packed structures	7.2.1	Full
	Assigning to structures	7.2.2	Full
Unions		7.3	
	Unpacked unions	7.3	Partial
	Packed unions	7.3.1	Full
	Tagged unions	7.3.2	Not supported
Packed and unpacked arrays		7.4	
	Packed arrays	7.4.1	Full
	Unpacked arrays	7.4.2	Full
	Operation on arrays	7.4.3	Full

Table 4-2 Supported SystemVerilog Constructs , *continued*

	Memories	7.4.4	Full
	Multidimensional arrays	7.4.5	Full
	Indexing and slicing of arrays	7.4.6	Full
Array assignment		7.6	Full
Arrays as arguments to subroutines		7.7	Full
Array query functions		7.11	
	$left	7.11 (20.7)	Full
	$right	7.11 (20.7)	Full
	$low	7.11 (20.7)	Full
	$high	7.11 (20.7)	Full
	$size	7.11 (20.7)	Full
	$dimensions	7.11 (20.7)	Full
	$unpacked_dimensions	7.11 (20.7)	Not supported
Array manipulation methods		7.12	
	Array locator methods	7.12.1	Not supported
	Array ordering methods	7.12.2	Not supported
	Array reduction methods	7.12.3	Not supported
	Iterator index querying	7.12.4	Not supported
Processes		**9**	
Initial procedures		9.2.1	Ignored
Always procedures		9.2.2	
	General purpose always procedure	9.2.2.1	Full
	Combinational logic always_comb procedure	9.2.2.2	Full
	Latched logic always_latch procedure	9.2.2.3	Full

Table 4-2 Supported SystemVerilog Constructs , *continued*

	Sequential logic always_ff procedure	9.2.2.4	Full
Final procedures		9.2.3	Ignored
Sequential blocks		9.3.1	
	Named	9.3.1	Full
	Un-named	9.3.1	Full
Block names		9.3.4	Full
Statement labels		9.3.5	Full
Procedural timing controls		9.4	
	Repeat statement with constant expression	9.4	Full
	Delay control	9.4.1	Ignored
	Conditional event control with level sensitive	9.4.2.3	Full
	Conditional event control with edge sensitive	9.4.2.3	Full
Disable statement		9.6.2	Partial
Assignment statements		**10**	
Continuous assignments		10.3	Full
	to variables	10.3.2	Full
Assignment extension and truncation		10.7	Full
Assignment patterns		10.9	
	Array assignment patterns	10.9.1	Full
	Structure assignment patterns	10.9.2	Full
Unpacked array concatenation		10.10	Not supported
Net aliasing		10.11	Not supported

Table 4-2 Supported SystemVerilog Constructs , *continued*

Operators and Expressions		**11**	
Constant expressions		11.2.1	Full
Aggregate expressions		11.2.2	Full
Operators		11.3	
	Unary operator	11.3	Full
	Binary operator (+ - * / % == != ==? !=? &&)	11.3	Full
	Binary operator (|| ** < <= > >= & | ^ ^~ ~^)	11.3	Full
	Binary operator (>> << >>> <<<)	11.3	Full
	Binary operator (-> <->)	11.3	Not supported
	Assignment within an expression	11.3.6	Full
Operator descriptions		11.4	
	Assignment operators	11.4.1	Full
	Increment and decrement operators	11.4.2	Full
	Conditional operators	11.4.11	Full
	Concatenation operators	11.4.12	Full
	Part select on concatenation operator	11.4.12	Not supported
	Replication operator	11.4.12.1	Full
	Set membership operator	11.4.13	Full
	Streaming operators	11.4.14	Full
Operands		11.5	
	Vector bit-select and part-select addressing	11.5.1	Full
	Array and memory addressing	11.5.2	Full
Operator overloading		11.11	Not supported

Table 4-2 Supported SystemVerilog Constructs , *continued*

Let construct		11.13	Not supported
Procedural programming statements		**12**	
Conditional if-else statement		12.4	
	simple	12.4	Full
	`unique if`	12.4.2	Full
	`unique0 if`	12.4.2	Not supported
	`priority if`	12.4.2	Full
Case statements		12.5	
	simple case	12.5	Full
	Case statement with do-not-cares	12.5.1	Full
	`unique case`	12.5.3	Full
	`unique0 case`	12.5.3	Full
	`priority case`	12.5.3	Full
	Set membership case statement	12.5.4	Full
Pattern matching conditional statements		12.6	
	Pattern matching in case statements	12.6.1	Not supported
	Pattern matching in if statements	12.6.2	Not supported
	Pattern matching in conditional expressions	12.6.3	Not supported
Loop statements		12.7	
	for loop	12.7.1	Full
	repeat loop	12.7.2	Full
	foreach loop	12.7.3	Full
	while loop	12.7.4	Full
	do while loop	12.7.5	Full
	forever loop	12.7.6	Not supported

Table 4-2 Supported SystemVerilog Constructs , *continued*

Jump statements		12.8	
	break	12.8	Full
	continue	12.8	Full
	return	12.8	Full
	return expression	12.8	Full
Tasks and Functions		**13**	
Tasks		13.3	
	Static task (treated as automatic)	13.3.1	Full
	Automatic task	13.3.1	Full
Functions		13.4	
	Return values and void functions	13.4.1	Full
	Static function (treated as automatic)	13.4.2	Full
	Automatic function	13.4.2	Full
	Constant function	13.4.3	Full
Subroutine calls and argument passing		13.5	
	Pass by value	13.5.1	Full
	Default argument values	13.5.3	Full
	Argument binding by name	13.5.4	Full
	Optional argument list	13.5.5	Full
Utility system tasks and system functions		**20**	
Conversion functions (with constant value only)		20.5	
	`$rtoi`		Full
	`$itor`		Full
	`$realtobits`		Not supported
	`$bitstoreal`		Not supported
	`$shortrealtobits`		Not supported
	`$bitstoshortreal`		Not supported

Table 4-2 Supported SystemVerilog Constructs , *continued*

Expression size system function		20.6.2	
	`$bits`	20.6.2	Full
Math functions (with constant expressions only)		20.8	
	Integer math functions	20.8.1	Full
	Real math functions	20.8.2	Full
Compiler Directives		**22**	
`‘resetall`		22.3	Full
`‘include`		22.4	Full
`‘define, ‘undef, ‘undefineall`		22.5	Full
`‘ifdef, ‘else, ‘elseif, ‘endif, ‘ifndef`		22.6	Full
`‘timescale`		22.7	Ignored
`‘default_nettype`		22.8	Full
`‘celldefine` and `‘endcelldefine`		22.10	Full
`‘begin_ keywords` and `‘end_ keywords`		22.14	Full
Modules and hierarchy		**23**	
Module definition		23.2	Full
Module header definition		23.2.1	
	Parameter port list with optional value	23.2.1	Not supported
	Parameter port list without parameter keyword	23.2.1	Full
	Parameter port list with localparam declaration	23.2.1	Not supported
	Parameter port list with parameter type declaration	23.2.1	Full

Table 4-2 Supported SystemVerilog Constructs , *continued*

	Package import declaration	23.2.1	Full
Port declarations		23.2.2	
	Port expressions	23.2.2.1	Full
	Generic interface port	23.2.2.2	Full
	Interface type port	23.2.2.2	Full
	bit, logic, int, shortint, longint, byte type ports	23.2.2.2	Full
	packed array ports	23.2.2.2	Full
	unpacked array ports	23.2.2.2	Full
	packed structure ports	23.2.2.2	Full
	unpacked structure ports	23.2.2.2	Full
	packed union ports	23.2.2.2	Full
	unpacked union ports	23.2.2.2	Partial
	ref port	23.2.2.2	Full
	variable port type	23.2.2.2	Full
	default port value	23.2.2.4	Not supported
Module instances		23.3	
	Top- level modules and $root	23.3.1	Full
	Array of instances (only one dimensional)	23.3.2	Full
	Connecting module instance ports by name	23.3.2.2	Full
	Connecting module instance ports using implicit named port connections	23.3.2.3	Full
	Connecting module instance ports using wildcard named port connections	23.3.2.4	Full
Nested modules		23.4	Not supported
Extern modules		23.5	Full

Table 4-2 Supported SystemVerilog Constructs , *continued*

Hierarchical names		23.6	Partial
Names with package or class scope resolution operator prefixes		23.7.1	Full
Interfaces		**25**	
Interface syntax		25.3	
	Interface with ansi header	25.3	Full
	Interface with non ansi header	25.3	Full
	Array of interface type port	25.3	Full
	Extern interface declaration	25.3	Full
	Attributes on interfaces	25.3	Ignored
	Nested Interface Definition	25.3	Not supported
	Nested Interface Instances	25.3	Supported
	Interface using a named bundle	25.3.2	Full
	Interface using a generic bundle	25.3.3	Full
Ports in interfaces		25.4	Full
Modports		25.5	
	Within interface declaration	25.5	Full
	Within generate block inside interface declaration	25.5	Not supported
	Named port bundle	25.5.1	Full
	Connecting port bundle	25.5.2	Full
	Connecting port bundle to generic interface	25.5.3	Full
	Modport expressions	25.5.4	Full
Tasks and functions in interfaces		25.7	Full
	Importing tasks and functions	25.7.2	Full
	Exporting tasks and functions	25.7.3	Not supported
Parameterized interfaces		25.8	Full

Table 4-2 Supported SystemVerilog Constructs , *continued*

Access to interface objects		25.10	Full
Packages		**26**	
Package declarations		26.2	
	package_export_declaration	26.2	Not supported
	data_declaration	26.2	Full
	task_declaration	26.2	Full
	function_declaration	26.2	Full
	local_parameter_declaration	26.2	Full
	parameter_declaration	26.2	Full
Referencing data in packages		26.3	Full
Using packages in module headers		26.4	Full
Using packages in interface headers		26.4	Full
Exporting imported names from packages		26.6	Not supported

Notes on SystemVerilog Constructs

- The unpacked union data type is supported but a verification mismatch is possible in case of unpacked unions with different element widths. The tool will issue a warning (CDFG-185) for such cases.
- Hierarchical references across modules are not supported.

Reading Designs with Mixed Verilog-2001 and SystemVerilog Files

RTL Compiler can read an HDL file that contains a mix of Verilog-2001 and SystemVerilog commands. However, SystemVerilog defines some new keywords. If these keywords are used as identifiers in a `-v2001` design, then RTL Compiler will report syntax errors if the design is read in the `-sv` mode. Keywords that may have been used as identifiers include `bit`, `int`, `char`, `break`, and so on.

To workaround this problem use the `` `begin_keywords `` compiler directive as follows:

```
interface intf;
        ... sv code ...
    endinterface
    `begin_keywords "1364-2001"
    module interface(output bit, input logic);
    ... other v2001 code which uses sv

keywords ...
    endmodule
    `end_keywords
```

The `` `begin_keywords `` directive tells the parser to recognize only those keywords defined by the specified language dialect. This lets you parse legacy code even in the `-sv` mode.

You can use the following options with the `` `begin_keyword ``s compiler directive:

- `1364_1995`
- `1364_2001`
- `1364_2001-noconfig`

    Disables `config`, `library`, and other configuration-related keywords
- `1364_2005`
- `1800_2005`

In the `-sv` mode, the default is `1800-2005`. In the `-v2001` mode, the default is 1364-2001. In the -v1995 mode, the default is `364-1995`.

5

Synthesizing VHDL Designs

Overview of VHDL Modeling

This chapter is organized for synthesizing VHDL RTL designs and provides links to the corresponding VHDL sections throughout the manual.

For mixed Verilog and VHDL usage, Chapter 1, "Modeling HDL Designs" provides modeling guidelines in both languages in one convenient location.

Modeling VHDL Designs

- Modeling Arithmetic Components (Verilog and VHDL) on page 48
- Modeling Combinational Logic in VHDL on page 45
- Modeling Latches in VHDL on page 41
- Modeling Latches in VHDL on page 41
- Modeling Flip-Flops in VHDL on page 35
- Using Case Statements in VHDL on page 64
- Using a for Statement in VHDL on page 69
- Inferring a Logic Abstract From the RTL in VHDL on page 72
- Interpreting a Logic Abstract in Verilog or VHDL on page 74

Specifying Synthesis Pragmas

- Supported Pragmas on page 82
- Specifying VHDL synthesis_off and synthesis_on Pragmas on page 88
- Specifying VHDL Set and Reset Synthesis Pragmas on page 98
- VHDL Signal Pragmas on page 101
- Specifying the VHDL Multiplexer Mapping Pragma on page 111
- Specifying Function and Task Mapping Pragmas (Verilog and VHDL) on page 120
- Specifying the Template Pragma (Verilog and VHDL) on page 122
- Specifying the Enumeration Encoding Pragma (VHDL) on page 123
- Specifying Resolution Function Pragmas (VHDL) on page 124

Using HDL Commands and Attributes

- HDL-Related Commands on page 130
- HDL-Related Attributes on page 131
- VHDL-Specific Attributes on page 150

VHDL Constructs

Supported VHDL Constructs

Table 5-1 lists the VHDL constructs supported by RTL Compiler. See Notes on Supported Constructs on page 214 for more information and license requirements. Both VHDL87 and VHDL93 style descriptions are supported. The constructs are classified by one of the following four categories:

■ Synthesized fully (Full)

■ Synthesized partially or in specific contexts (Partial)

■ Construct is ignored and a warning is generated (Ignored)

■ Construct is unsupported and an error message is generated (No)

Table 5-1 VHDL Constructs Supported in RTL Compiler

Construct			**Support**
Design Entity and Configuration	Entity Declaration	Entity header	Full
		Entity declarative part	Full
		Entity statement part	Ignored
	Architecture Body	Architecture declarative part	Full
		Architecture statement part	Full
	Configuration Declaration	Configuration declarative part	Partial
		Block configuration	Full
		Component configuration	Full
Subprogram and Packages	Subprogram Declaration		Full
	Subprogram Body	Subprogram declarative part	Full
		Subprogram statement part	Full
	Subprogram Overloading		Full
	Resolution Function		Partial
	Package Declaration	Package declarative part	Full
		Deferred constants	Full
	Package Body		Full
Types	Scalar Type Definition	Enumeration type	Full
		Integer	Full
		Physical	Ignored
		Floating	Ignored
	Composite Type Definition	Array	Full
		Record	Full
	Access Type Definition		Ignored
	File Type Definition		Ignored

Construct			**Support**
Declarations	Subprogram Declaration		Full
	Subprogram Body		Full
	Type Declaration		Full
	Subtype Declaration		Full
	Object Declaration	Constant	Full
		Signal	Full
		Variable	Full
		Shared variable	No
		File	No
	Alias Declaration		Full
	Attribute Declaration		Full
	Component Declaration		Full
	Group Template Declaration		No
	Group Declaration		No
Specifications	Attribute Specification		Full
	Configuration Specification		Full
	Disconnection Specification		No

Construct			Support
Expressions	Binary Logical Operators	and, or, nand, nor, xor, xnor	Full
	Relational Operators	=, /=, >, <, >=, <=,	Full
	Shift Operators	sll (shift left logical) srl (shift right logical) sra (shift right arithmetic) sla (shift left arithmetic)	Full
		ror, rol	Full
	Arithmetic Operators	`+, -,` &	Full
	Sign Operators	+, -	Full
	Multiplying Operators	* mod /, rem	Full Full Full
	Miscellaneous Operators	* * abs `not`	Partial Full Full
	Operands	Integer literal	Full
		Real literal	Ignore
		Physical literal	Ignore
		Enumeration literal	Full
		String literal	Full
		Bit string literal	Full
	Aggregates	Record aggregates	Full
		Array aggregates	Full
	Function calls	Qualified expression	Full
		Type conversion	Full
		Allocators	No

Construct			**Support**
Sequential Statements	Wait	Sensitivity clause	Partial
		Condition clause	Partial
		Timeout clause	Ignored
	Assertion		Ignored
	Report		Ignored
	Signal Assignment		Full
	Variable Assignment		Full
	Procedure Call		Full
	If		Full
	Case		Full
	Loop	Unconditional loop	No
		while loop	Partial
		for loop	Full
	Next		Full
	Exit		Full
	Return		Full
	Null		Full

Construct			**Support**
Concurrent Statements	Block	Guard	No
		Block header	No
		Block declarative part	Full
		Block statement part	Full
		Timeout clause	Ignored
	Process		Full
	Concurrent Procedure Call		Full
	Concurrent Assertion		Ignored
	Concurrent Signal Assignment	Conditional signal assignment	Full
		Selected signal assignment	Full
	Component Instantiation		Full
	Generate Statement	if generate	Full
		for generate	Full

Notes on Supported Constructs

Design Entities

- Generics and ports in an entity header can be of any allowable synthesizable type in an interface object, such as `bit`, `boolean`, `bit_vector`, and `integer`. See Types on page 215 for more information.
- Generics must have a default value specified, unless the entity has a `TEMPLATE` attribute set to `true`. See Chapter 2, "Specifying Synthesis Pragmas" for more information.
- Declarations in an entity or architecture declarative part must be supported declarations. See Declarations on page 216 for more information.

VHDL Configurations

- Configuration declarations and configuration specifications are supported with the restriction that only one unique architecture is bound to an entity throughout the design.
- Nested VHDL configurations are supported.
- `elaborate` command also accepts a configuration name as value of design name argument. In such a case, this configuration is used to elaborate that VHDL entity (as top module), whose configuration was used as elaborate argument.
- Elaboration would not use a configuration, unless it is explicitly mentioned in `elaborate` command as a design name. In other words, if you `elaborate` (without any design name argument), RTL Compiler would elaborate top-level entities or modules, and will ignore their configurations, even if they were read in during `read_hdl`. Similarly, if you elaborate a module; for example, `elab mytop`, where `mytop` is the name of an entity and not of a configuration, then `elaborate` would generate its netlist, without using the configuration for the design.

Subprograms and Packages

- Impure functions are unsupported.
- Recursive subprograms are supported.
- Formal parameters in a subprogram declaration can be of any synthesizable type allowed for an interface object (for example, `bit`, `boolean`, `bit_vector`, `integer`). See Types below for more information.

- Declarations in a subprogram declarative part, package declarative part, or package body declarative part must be a supported declaration. See Declarations on page 216 for more information.
- The `resolved` function defined in package `IEEE.STD_LOGIC_1164` is the only supported resolution function. Annotate user-defined resolution functions with the `RESOLUTION` attribute to force a `WIRED_AND`, `WIRED_OR`, or `WIRED_TRI` behavior. See to Chapter 2, “Specifying Synthesis Pragmas” for more information.

Types

- Objects, such as constants, signals, and variables declared with a subtype that is an ignored type or derived from an ignored type are unsupported. For example, floating type definitions are ignored but a signal of that floating type is flagged as an error, as shown in Example 5-1.

Example 5-1 Declaring an Object with an Unsupported Subtype Results in Error

```
type GET_REAL is 2.4 to 3.9; --Ignored type definition
signal S: GET_REAL; <--Error!
```

- Use the `_ENCODING` attribute to override the default mapping between an enumerated type and the corresponding encoding value. See Chapter 2, “Specifying Synthesis Pragmas” for more information.
- Array type definitions are supported, as shown in Example 5-2.

Example 5-2 Supported Array Type Definitions

```
subtype BYTE is bit_vector(7 downto 0);
type COLORS is (SAFFRON, WHITE, GREEN, BLUE);
type BIT_2D is array (0 to 255, 0 to 7) of bit;
type ANOTHER_BIT_2D is array (0 to 10) of BYTE;
type BITVECTOR_1D is array (0 to 255) of BYTE;
type INTEGER_1D is array (0 to 255) of integer;
type _1D is array (0 to 255) of COLOR;
type BOOL_1D is array (COLORS) of boolean;
-- a three dimensional bit
type BIT_3D is array (0 to 10) of BIT_2D;
-- a two dimensional integer
type INTEGER_2D is array (0 TO 10, 0 TO 10) of integer;
```

- Interface objects (formal ports of an entity or a component, formal parameters of a subprogram) can be of any supported type.

Declarations

- Initial values are supported for variables in a subprogram body.
- Deferred constants are supported.
- User-defined attribute declarations and specifications are supported.
- All type declarations can be read in, but only objects of supported types described in the types section are declared.
- Signal kinds (bus and register) are unsupported.
- Mode `linkage` in interface objects is unsupported.

Names

- Selected names that refer to elements of a record are supported.
- Selected names used as expanded names are supported. An expanded name is used to denote a declaration from a library, package, or other named construct.
- The following predefined attributes are supported: `'base`, `'left`, `'right`, `'high`, `'low`, `'range`, `'reverse_range`, `'length`, `'Succ`, `'Pred`, `'Leftof`, `'Rightof`
- The `event` and `stable` predefined attributes are only supported in the context of clock edge specifications.
- User defined attribute names are supported.
- Indexing and slicing of function return values are supported.
- Expressions in attribute names are unsupported.

Expressions

- Mixing array and scalar arguments for Binary Logical Operators is allowed.
- Signed arithmetic is supported.
- The following operators are only supported in the VHDL IEEE 1076-1993 standard mode: `'xnor`, `'sll`, `'srl`, `'sla`,`'sra`, `'rol`, `'ror`
- The `**` operator is only supported when both the operands are constants or when the left operand is a power of 2.
- Real and physical literals may only exist in after clauses, where they are ignored.
- The `TYPE_CONVERSION` pragmas may be used to tag user-defined functions as having a type conversion behavior. Refer to Chapter 2, "Specifying Synthesis Pragmas" for further information.
- Slices of array objects are supported. Similarly, direct indexing of a bit within an array is supported, as shown in Example 5-3.
- Null Slices as arguments to concatenation operator are supported.
- Null Ranges inside aggregates is also supported.

Example 5-3 Direct Indexing of a Bit Within an Array

```
subtype BYTE is bit_vector(3 downto 0);
type MEMTYPE is array (255 downto 0) of BYTE;
variable MEM: MEMTYPE;
variable B1: bit;
…
MEM(3 downto 0):= X; -- supported multi-word slice
B1:= MEM(3)(0);      -- supported reference to bit
```

- Slices whose ranges cannot be determined statically are not supported.
- `ror` and `rol` operators are available with *Datapath Synthesis in Encounter RTL Compiler*.

Sequential Statements

- When an explicit `wait` statement is used, it must be the first statement of a process. The condition clause must represent the clock edge specification. The sensitivity clause, if any, must only contain the clock signal specified in the condition clause.
- Multiple wait statements in a process (implicit state machines) are unsupported.
- Assignments that involve multiple “words” of two-dimensional or higher objects are supported.
- The range in a `for` loop must be globally static.
- Delay mechanisms in signal assignments are ignored.
- Multiple waveforms in signal assignments are unsupported.
- `while` loops are supported with the restriction that looping behavior is statically determined.

Concurrent Statements

- Postponed processes including postponed concurrent procedure calls and postponed concurrent signal assignments are unsupported.
- Signal assignments that involve multiple “words” of 2-dimensional (or higher) objects are supported.
- Delay mechanisms in signal assignments are ignored.
- Multiple waveforms in signal assignments are unsupported.
- Guarded signal assignments are unsupported.
- The range in a `for-generate` statement must be globally static.
- Declarations in a generate statement are only supported in the VHDL IEEE 1076-1993 standard mode.

Supported VHDL 2008 Enhancements

Table 5-2 lists the level of support for the IEEE 1076-2008 VHDL standard constructs and indicates the level as fully supported (Full) or partially supported (Partial) or not supported (No) or ignored (Ignored). To use the following enhancements, set the attribute `hdl_vhdl_read_version` to 2008.

```
set_attr hdl_vhdl_read_version 2008
```

Table 5-2 Supported VHDL 2008 Enhancements in RTL Compiler

Construct			Support
Types	Unconstrained Elements	Arrays	Full
		Records	No
Declarations	PSL Declarations		Ignored
	Reading of Output Ports		Full
	Non-static Expressions in Port Map		Full
Expressions	Unary Logical Operators	and, or, nand, nor, xor, xnor	Full
	Matching Relational Operators	?=, ?/= , ?<,?<=,?>,?>=	Full
	Conditional Operators	??	Full
	External Names	Relative Paths	Partial
		Absolute Paths	No
		Package Paths	No
Concurrent Statements	PSL Directives		Ignored
	Process All		Full
Lexical	Comments	C Style Comments	Full
	Literals	Enhanced Bit String Literals	Full

Notes on VHDL-2008 Enhancements

Declarations

- Reading of output ports is supported
- Non-static expressions in port maps are supported.

Expressions

- Only signal type external names are supported.

Sequential Statements

- Non-static expressions are allowed in case statements.

VHDL Predefined IEEE Packages

Table 5-3 and Table 5-4 list the VHDL packages that are predefined in the IEEE library and how they are supported for the VHDL versions specified by the attribute `hdl_vhdl_read_version`.

Table 5-3 IEEE Standard VHDL Packages

Package Name	**VHDL 1987 | 1993**	**VHDL 2008**
`std_logic_1164`	Supported	Supported
`numeric_std` `numeric_bit`	Supported	Supported
`numeric_std_unsigned` `numeric_bit_unsigned`	Not Applicable	Not Supported
`math_real` `math_complex`	Supported	Supported using the 1993 version of the package
`fixed_generic_pkg` `float_generic_pkg`	Not Applicable	Not Supported

Note: Packages in Table 5-3 are IEEE standards. Except as noted, RTL Compiler supports the package published by the IEEE for each version of VHDL. The packages in Table 5-4 are non-standard, copyrighted by Synopsys in 1992, and are the same for each version of VHDL.

Table 5-4 Non-Standard Synopsys VHDL Packages

Package Name	**VHDL 1987 | 1993**	**VHDL 2008**
`std_logic_arith`	Supported	Supported
`std_logic_unsigned`	Supported	Supported
`std_logic_signed`	Supported	Supported
`std_logic_misc`	Supported	Supported

VHDL Predefined Attributes

Table 5-5 VHDL Predefined Attributes

Pre-defined Attribute	**Support**
`'base`	Partial
`'left`	Full
`'right`	Full
`'high`	Full
`'low`	Full
`'ascending`	Partial
`'image`	No
`'value`	No
`'pos`	Partial
`'val`	Partial
`'succ`	Full
`'pred`	Full
`'leftof`	Full
`'rightof`	Full
`'range`	Full
`'reverse_range`	Full
`'length`	Full
`'delayed`	No
`'stable`	Partial
`'quiet`	No
`'transaction`	No
`'event`	Partial
`'active`	No
`'last_event`	No

Pre-defined Attribute	**Support**
`'last_active`	No
`'last_value`	No
`'driving`	No
`'driving_value`	No
`'simple_name`	No
`'instance_name`	No
`'path_name`	No

Notes on Pre-defined Attributes

- The following pre-defined attributes are supported only when the prefix is a static type mark: `'base`, `'ascending`, `'pos`, `'val`, `'succ`, `'pred`, `'leftof`, `'rightof`
- The following pre-defined attributes are supported only in the context of clock edge specifications: `Event`, `Stable`
- Expressions in attribute names are not supported.

Index