cadence

# HDL Modeling in Encounter® RTL Compiler

**Product Version 6.1**
**June 2006**

© 2006 Cadence Design Systems, Inc. All rights reserved.
Printed in the United States of America.

Cadence Design Systems, Inc., 555 River Oaks Parkway, San Jose, CA 95134, USA

**Trademarks:** Trademarks and service marks of Cadence Design Systems, Inc. (Cadence) contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence's trademarks, contact the corporate legal department at the address shown above or call 800.862.4522.

Open SystemC, Open SystemC Initiative, OSCI, SystemC, and SystemC Initiative are trademarks or registered trademarks of Open SystemC Initiative, Inc. in the United States and other countries and are used with permission.

All other trademarks are the property of their respective holders.

**Restricted Print Permission:** This publication is protected by copyright and any unauthorized use of this publication may violate copyright, trademark, and other laws. Except as specified in this permission statement, this publication may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without prior written permission from Cadence. This statement grants you permission to print one (1) hard copy of this publication subject to the following conditions:

1. The publication may be used solely for personal, informational, and noncommercial purposes;
2. The publication may not be modified in any way;
3. Any copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement; and
4. Cadence reserves the right to revoke this authorization at any time, and any such use shall be discontinued immediately upon written notice from Cadence.

**Disclaimer:** Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. The information contained herein is the proprietary and confidential information of Cadence or its licensors, and is supplied subject to, and may be used only by Cadence's customer in accordance with, a written agreement between Cadence and its customer. Except as may be explicitly set forth in such agreement, Cadence does not make, and expressly disclaims, any representations or warranties as to the completeness, accuracy or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from use of such information.

**Restricted Rights:** Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor.

# 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.

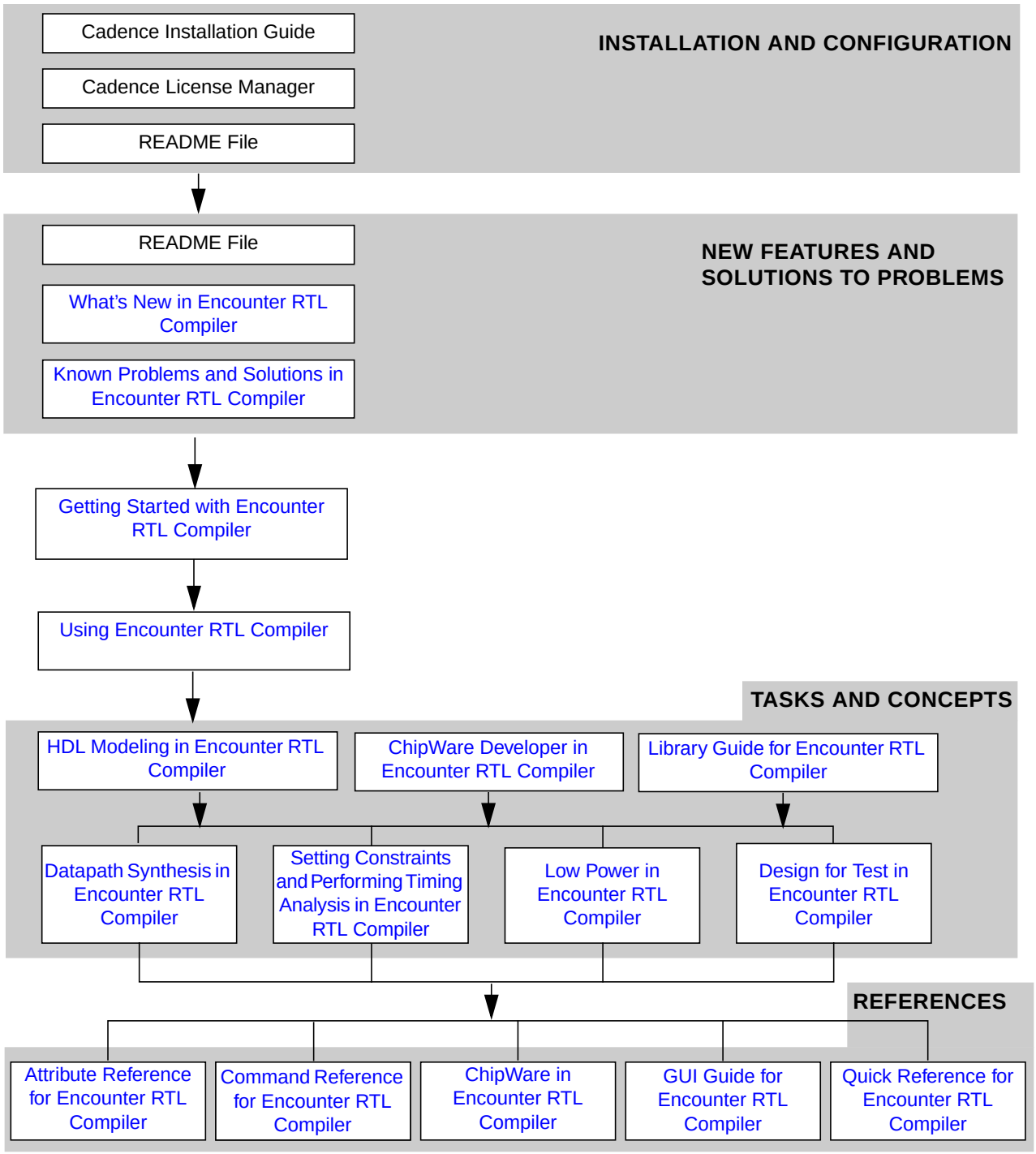
# Additional References

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



# Reporting Problems or Errors in Manuals

The Cadence Online Documentation System, CDSDoc, lets you view, search, and print Cadence product documentation. You can access CDSDoc by typing `cdsdoc` from your Cadence tools hierarchy.

Clicking the *Feedback* button lets you send e-mail directly to Cadence Technical Publications. Use it if you find:

- An error in the manual
- An omission of information in a manual
- A problem displaying documents

# Customer Support

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

## SourceLink Online Customer Support

SourceLink® online customer support 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, service request tracking, up-to-date release information, full site search capabilities, software update ordering, and much more.

For more information on SourceLink go to:

http://www.cadence.com/support/sourcelink.aspx

## 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`

# 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.

■ If you only know the first few letters of a command, then you can get a list of commands that begin with that letter.

For example, to get a list of commands that begin with “ed”, you would type the following command:

```
rc:/> ed* -h
```

■ Type a single letter and press `Tab` to get a list of all commands that contains 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 also type a sequence of letters and press `Tab` to get a list of all commands that contain those letters.

For example:

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

This returns the following commands:

```
ambiguous "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.<br>`{ 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

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 163 and Supported VHDL Constructs on page 178 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, “Synthesis Pragmas” for detailed information. See Supported Synopsys Pragmas on page 90 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 - 1995Standard* and 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)

VHDL designs have the following restrictions:

- Read an entity before any of the entity’s architectures and packages.
- Read 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

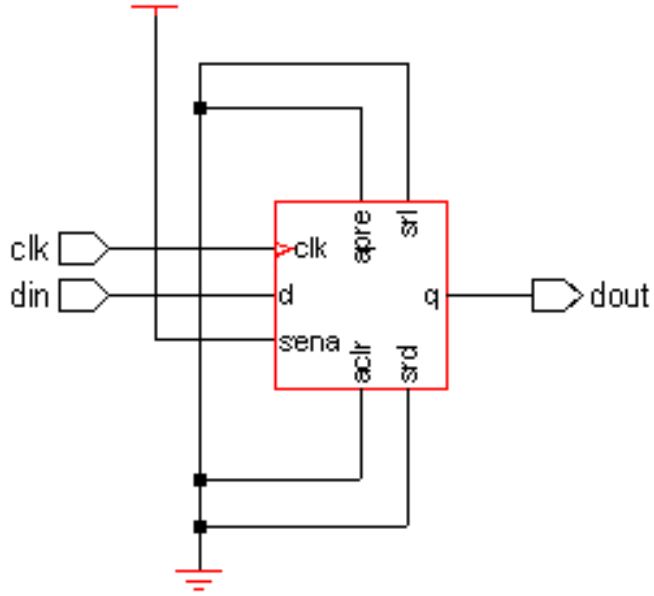
When an assignment is conditioned upon a rising or falling transition on a signal, an edge-triggered flip-flop is inferred to implement the variable on the left side of the assignment, as shown in Example 1-1.

### 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 shows the corresponding schematic for Example 1-1.

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



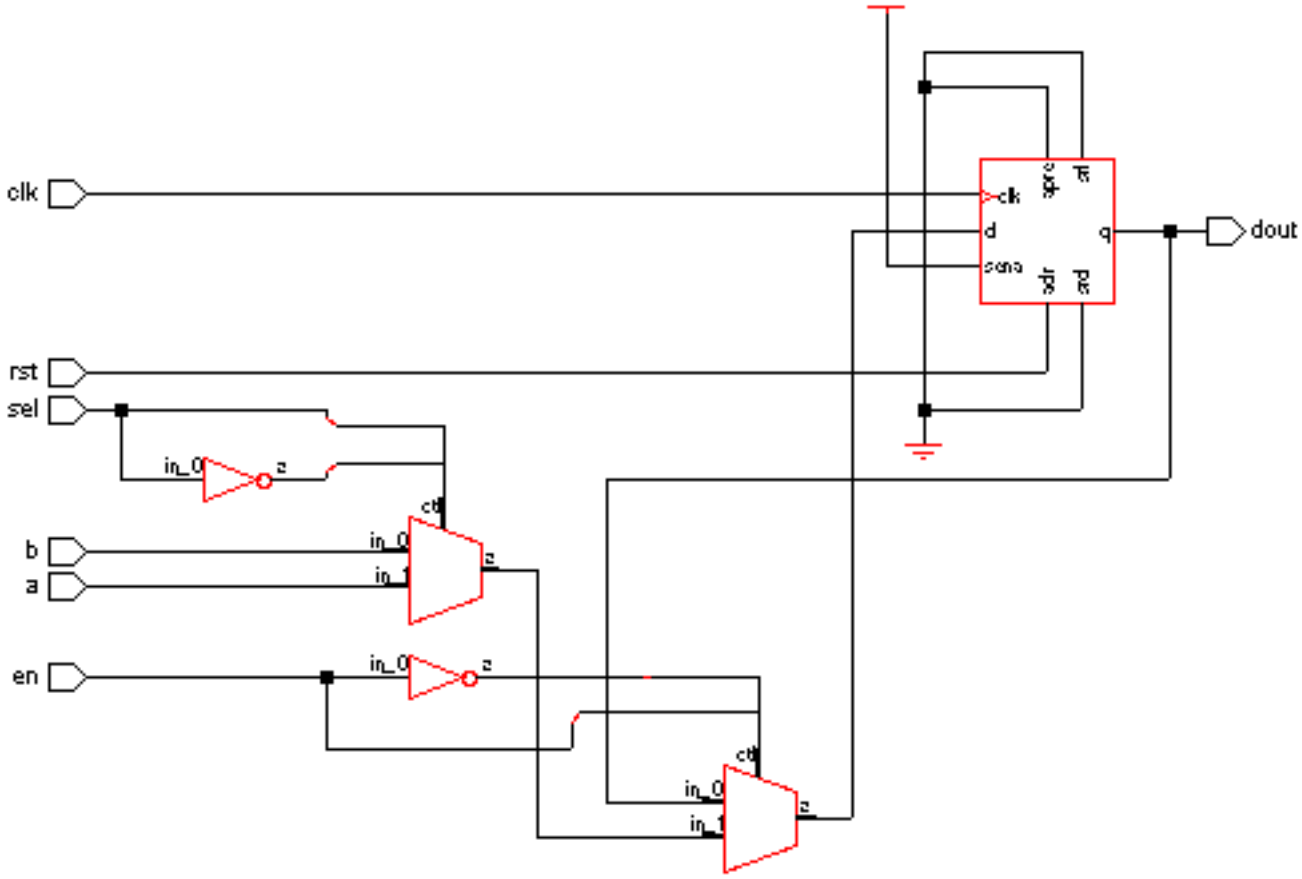

A flip-flop with an asynchronous operation is inferred, as shown in Example 1-2, when an assignment is made without being dependent on the clock edge.

**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
```

Figure 1-2 shows the corresponding schematic for Example 1-2.

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




The `always` block is triggered when a rising edge is detected on `clk` or a rising edge on `rst`. If `rst` is active low, then the event in the sensitivity list, and the condition in the `if` statement should be negated.

## Modeling Flip-Flops in VHDL

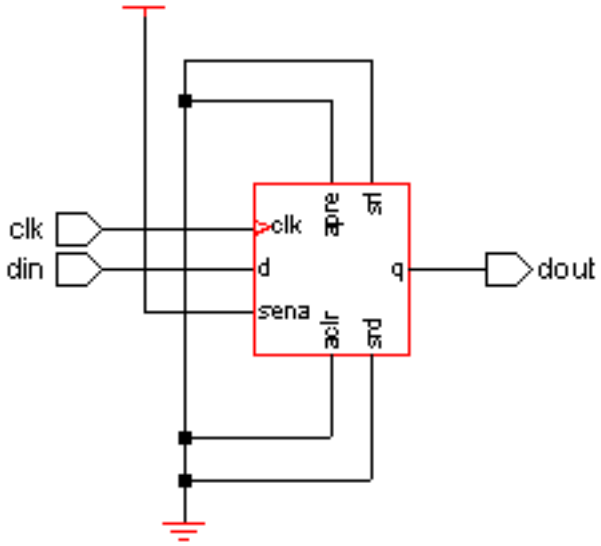
When a process is triggered by a rising edge or a falling edge transition on a signal, typically a clock signal, the variable or signal on the left side of a procedural assignment is inferred as a flip-flop, as shown in Example 1-3.

### Example 1-3 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
  process(clk) begin
    if clk'event and clk = '1' then
      dout <= din;
    end if;
  end process;
end;
```

Figure 1-3 shows the corresponding schematic.

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




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

```
dout <= din when rising_edge(clk);
```

**Note:** Example 1-3 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;
process;
```

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

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

Use the model, as shown in Example 1-4, to synthesize a flip-flop with synchronous `set` and `reset` connections.

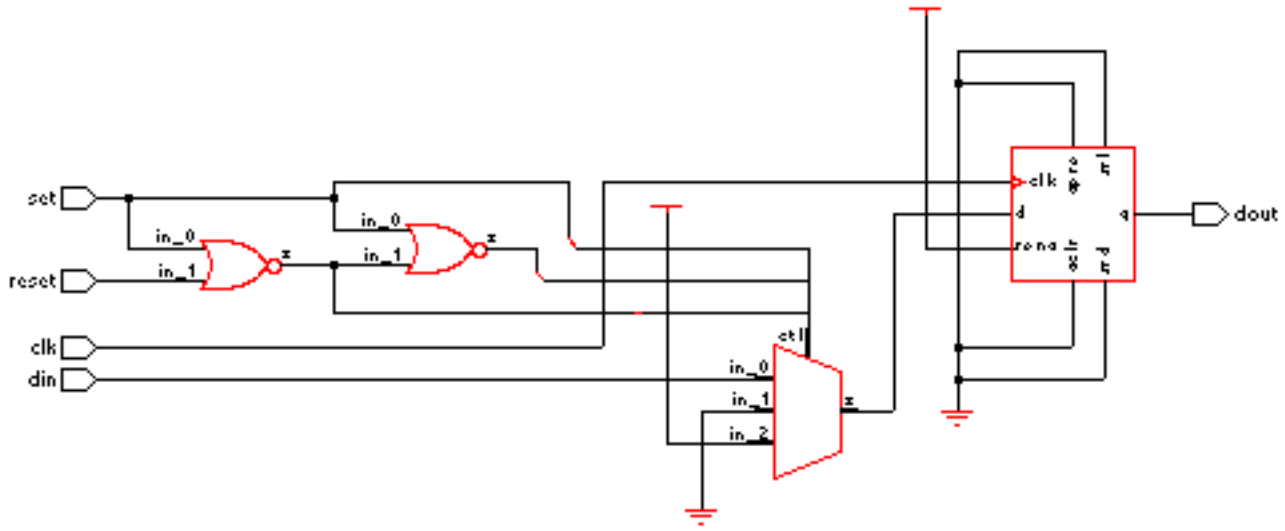
### Example 1-4 Synthesizing Synchronous Set and Reset Signals On a Flip-Flop (VHDL)

```
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 Synthesis Pragmas on page 87 for more information on controlling the `set` and `reset` connections for a flip-flop.

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

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



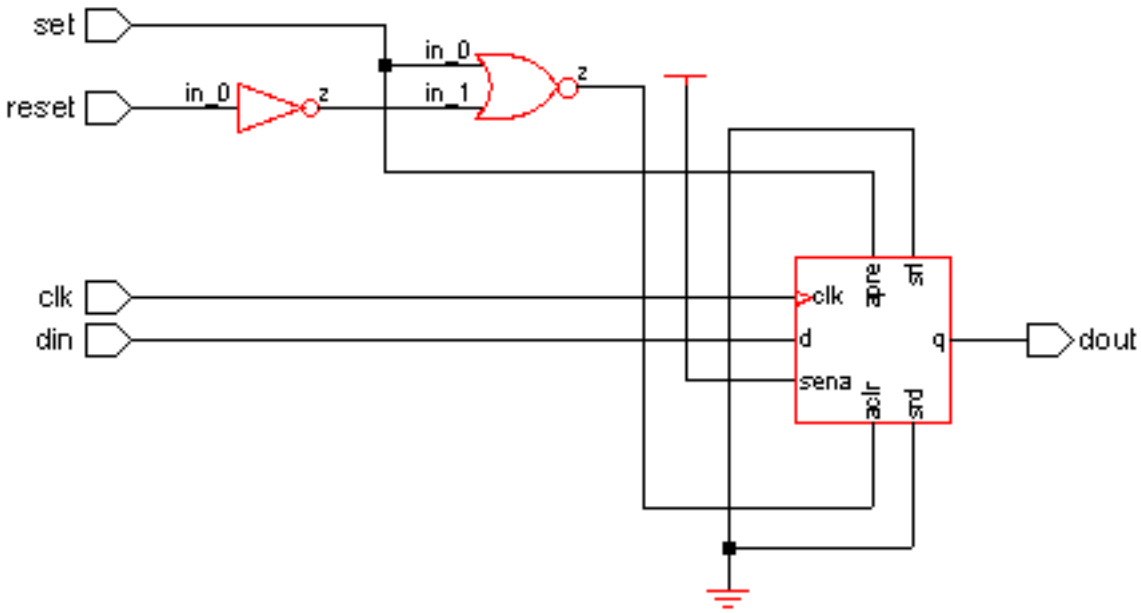

Use the model, as shown in Example 1-5, to synthesize a flip-flop with asynchronous `set` and `reset` connections.

### Example 1-5 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 1-5.

**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

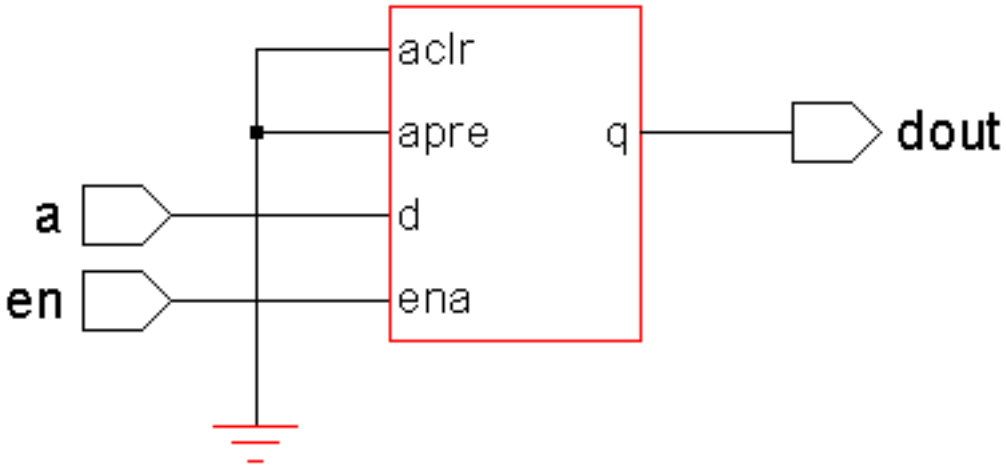
RTL Compiler infers a latch for a variable if it is updated whenever any of the variables that contribute to its value change when the enable signal is valid, as shown in Example 1-6. The `dout` signal is updated when `en` is high, otherwise signal `dout` retains its previous value. RTL Compiler infers a latch to implement the `dout` variable.

### Example 1-6 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 1-6.

**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 1-7.

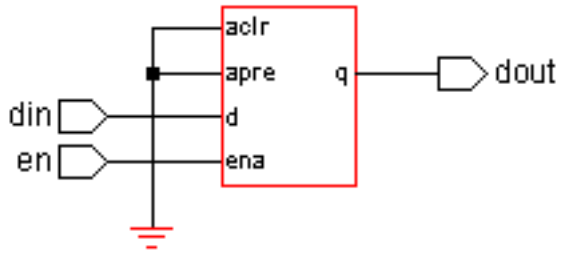
**Example 1-7 Modeling a Latch (VHDL)**

```
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 1-7 (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

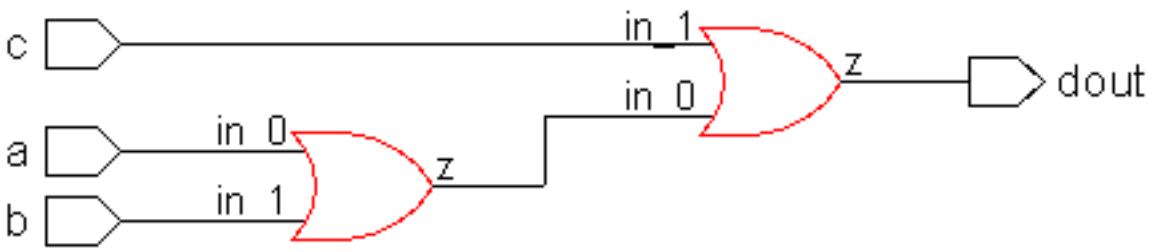
Continuous assignments are introduced by the *assign* keyword. Combinational logic is inferred for any variable assigned with continuous assignments, as shown in Example 1-8.

### Example 1-8 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 1-8.

### 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

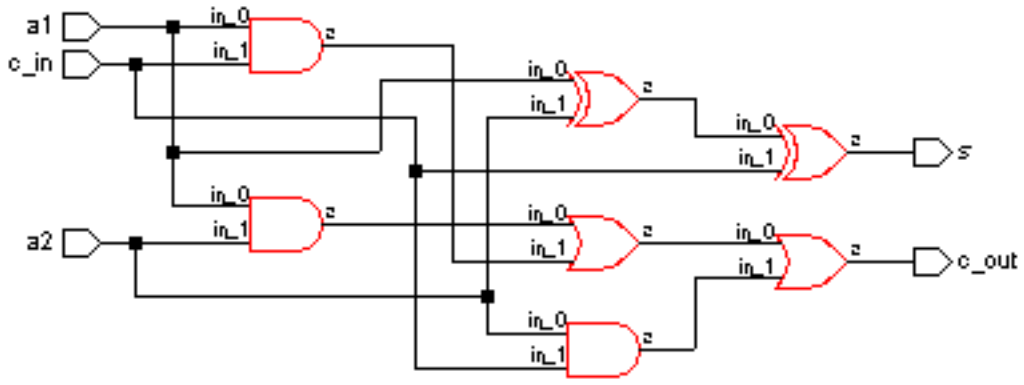
Combinational logic is synthesized to implement the `c_out` variable in Example 1-9.

### Example 1-9 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 1-9.

### 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 1-10 and Example 1-11 signal `en` is used for gating `clk`. Example 1-10 shows an incomplete conditional statement, while Example 1-11 shows a complete conditional statement. Low Power can use both conditions to insert clock-gating logic.

### Example 1-10 Modeling Incomplete Conditional Statements (Verilog)

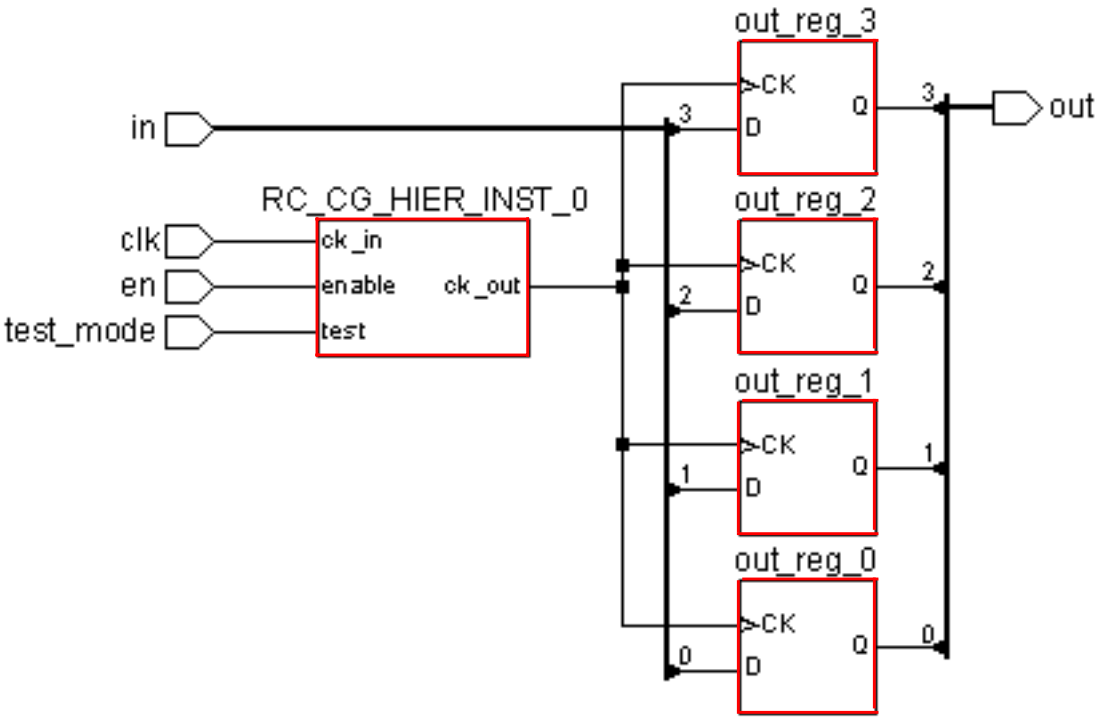
```
module ex1 (in, out, en, clk);
    input clk, en;
    input [3:0] in;
    output [3:0] out;
    reg [3:0] out;
    always @ (posedege clk) begin
      if (en)
        out <= in;
    end
endmodule
```

### Example 1-11 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;
endmodule
```

Figure 1-10 shows the mapped netlist for Example 1-10 and Example 1-11 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 1-12 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 1-13 and the corresponding schematic shown in Figure 1-12.

### Example 1-12 Modeling Combinational Logic With an Unconditional Assignment (VHDL)

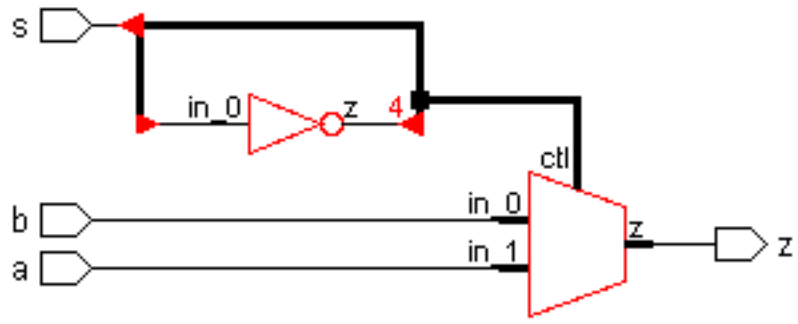
```
library ieee;
use ieee.numeric_std.all;

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 1-12 (VHDL)

### Example 1-13 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.

# Adders

## Modeling an Unsigned Adder in Verilog and VHDL

### Example 1-14 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 1-15 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 1-16 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 1-17 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 1-18 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 1-19 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;
```

# Subtractors

## Modeling an Unsigned Subtractor in Verilog and VHDL

### Example 1-20 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 1-21 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 1-22 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 1-23 Modeling an 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 1-24 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 1-25 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 1-26 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 1-27 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 1-28 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 1-29 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 1-30 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;
```

# Multipliers

## Modeling an Unsigned Multiplier in Verilog and VHDL

### Example 1-31 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 1-32 Modeling an Unsigned Multiplier in VHDL

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

### Example 1-33 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 1-34 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 1-35 Modeling a Signed Multiplier in VHDL

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

### Example 1-36 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;
```

# Dividers

## Modeling an Unsigned Divider in Verilog and VHDL

### Example 1-37 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 1-38 Modeling an Unsigned Divider in VHDL

```
library ieee;
use ieee.numeric_std.all;
entity unsigned_divide is
    generic (wA : integer := 16);
            (wB : interger := 6);
    port (y : out unsigned (wA-1 downto 0);
          a : in unsigned (wA-1 downto 0)  );
          b : in unsigned (wB-l 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 1-39 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 1-40 Modeling a Signed Divider in VHDL

```
library ieee;
use ieee.numeric_std.all;
entity signed_divide is
    generic (wA : integer := 16);
            (wB : interger := 6);
    port (y : out signed (wA-1 downto 0);
          a : in signed (wA-1 downto 0)  );
          b : in signed (wB-l 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 1-41 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 1-42 Modeling an Unsigned Modulus in VHDL

```
library ieee;
use ieee.numeric_std.all;
entity unsigned_modulus is
    generic (wA : integer := 16);
            (wB : interger := 6);
    port (y : out unsigned (wB-1 downto 0);
          a : in unsigned (wA-1 downto 0)  );
          b : in unsigned (wB-l 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 1-43 Modeling an 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 1-44 Modeling an 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-l 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 1-45 Modeling an Unsigned Remainder in VHDL

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

### Example 1-46 Modeling a Signed Remainder in VHDL

```
library ieee;
use ieee.numeric_std.all;
entity signed_remainder is
    generic (wA : integer := 16);
            (wB : interger := 6);
    port (y : out signed (wB-1 downto 0);
          a : in signed (wA-1 downto 0)  );
          b : in signed (wB-l 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 full_case Pragma on page 96, and parallel_case Pragma on page 97).

## 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

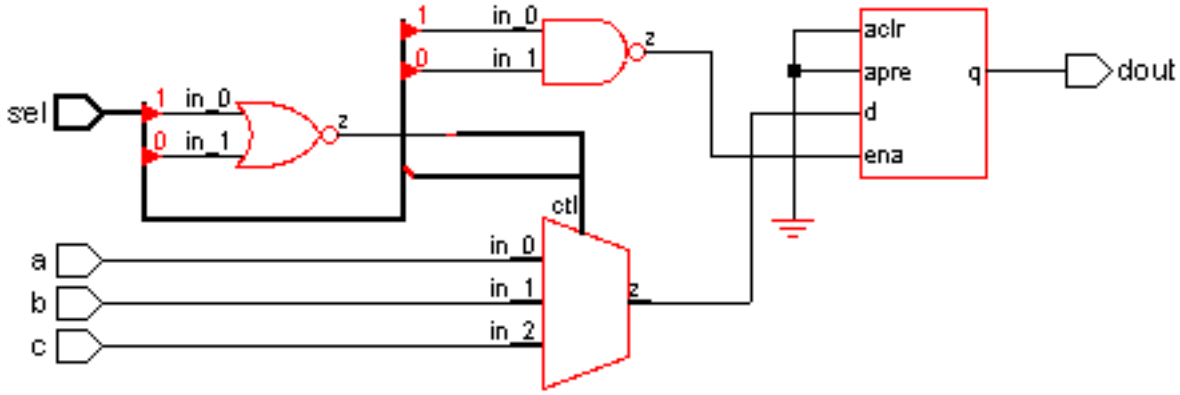
When a `case` statement does not specify all possible `case` condition values, a latch is inferred. If RTL Compiler determines that the case is not `full`, it uses a latch to implement a state transition table, as shown in Example 1-47.

**Example 1-47 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 1-47.

**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 1-48.

**Example 1-48 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
```

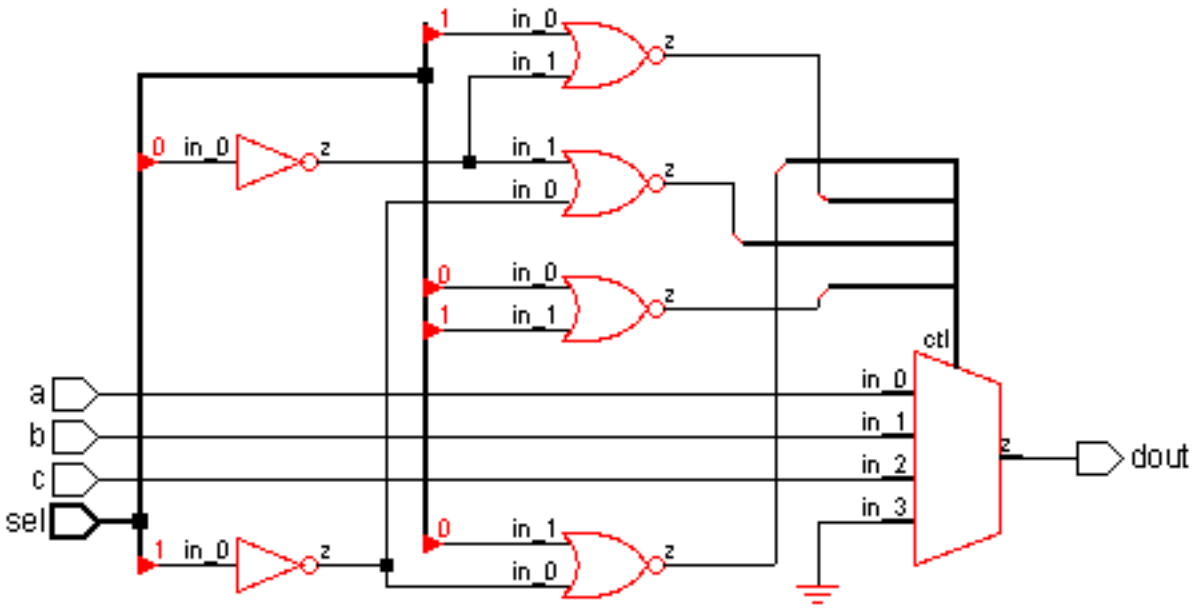
- Use the default case in the `case` statement to capture all the remaining cases where the next state variable is assigned a value, as shown in Example 1-49.

**Example 1-49 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 1-48 and Example 1-49.

**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 1-50 shows a `casez` statement using don't care conditions to mask three of the four bits in the decoding select line (`input sel`).

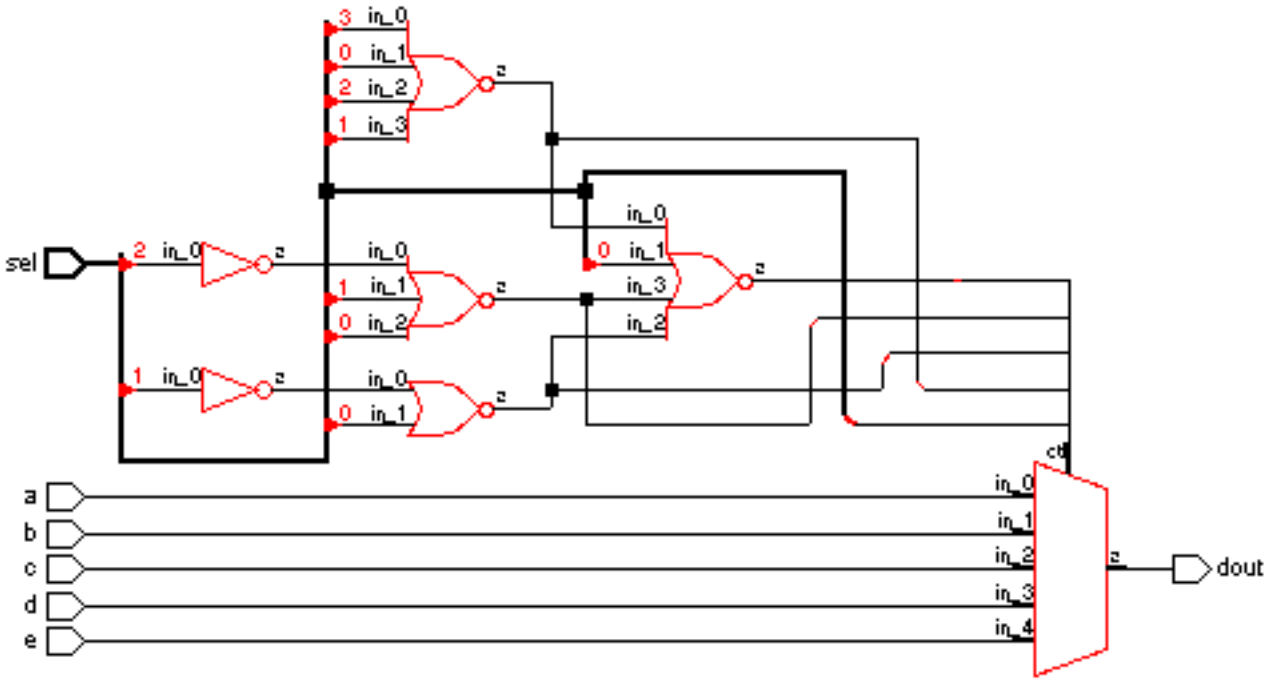
**Example 1-50 Modeling Don't Care Conditions in a Casez Statement (Verilog)**

```
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 1-50.

**Figure 1-15 Don't Care Conditions in a Casez Statement Schematic (Verilog)**



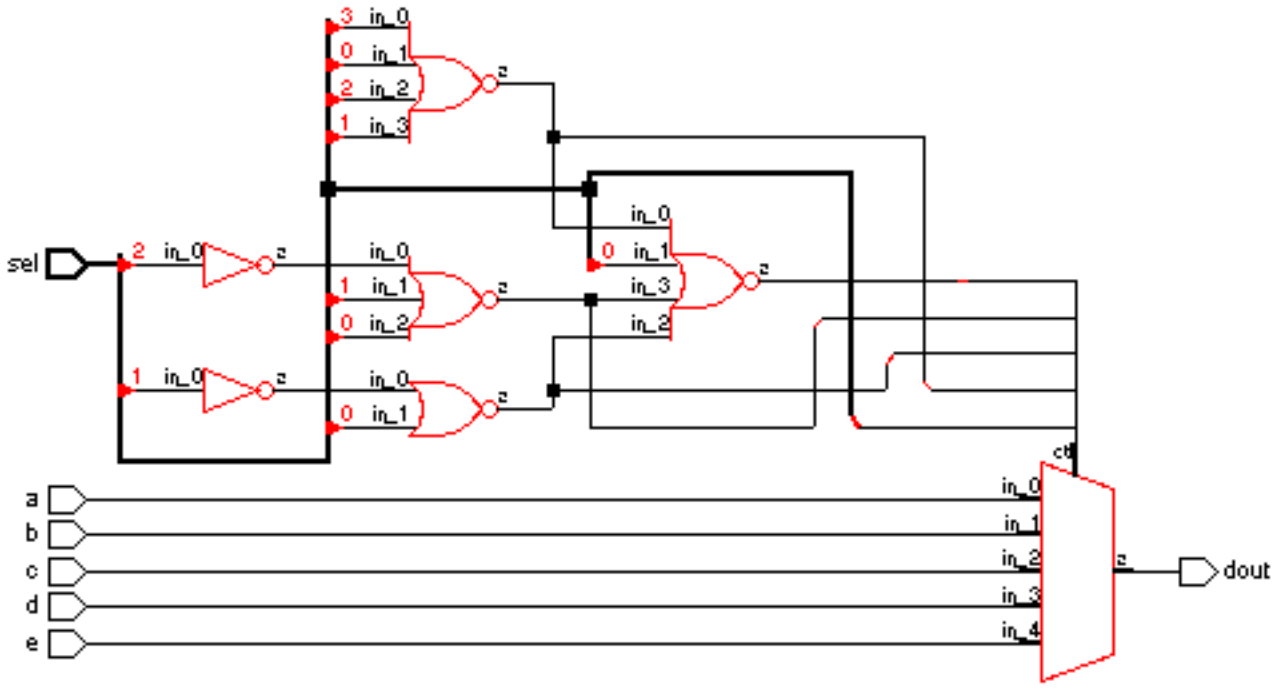

Example 1-51 shows a `casex` statement using don't care conditions in the same manner as the `casez` statement. The difference between the two models is that the `casex` statement masks three bits of the select line that would match `x`, `z`, or `?`, but the `casez` statement will not mask `x` in the select line.

### Example 1-51 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 1-51.

### 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

When a `case` statement specifies only some of the values that the `case` expression can possibly have, a latch is inferred, as shown in Example 1-52.

### Example 1-52 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 1-53.

### Example 1-53 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 1-54.

### Example 1-54 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 1-55 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 1-56.

### Example 1-55 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 1-56. 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 1-56 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 := 0;
end case;
```

# Using a for Statement to Describe Repetitive Operations

Use the `for` statement to describe repetitive operations.

## Using a for Statement in Verilog

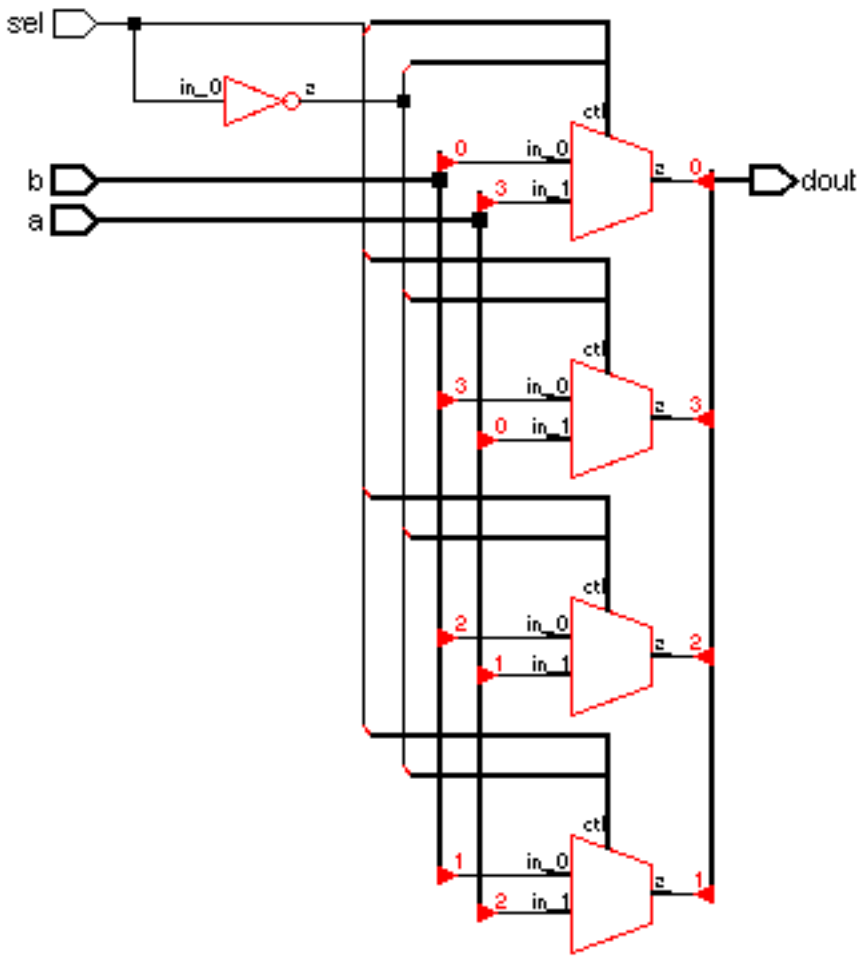
Example 1-57 uses the `for` statement where `i` is declared as an integer and `dout` is a 4-bit register. The `for` statement is expanded to repeat the operations over the range of the index.

### Example 1-57 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 1-57.

**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 1-58.

**Example 1-58 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 1-59.

**Example 1-59 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 1-59 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 1-60.

### Example 1-60 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 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 1-61 infers a logic abstract from the `my_sub_empty` module:

**Example 1-61 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 1-62, infers a logic abstract for the `my_sub_gray` module.

**Example 1-62 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 1-63, infers a logic abstract for the `my_sub_empty` component.

### Example 1-63 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 1-64, infers a logic abstract from the `my_sub_empty` component.

### Example 1-64 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 1-65 infers a logic abstract for the `my_sub_gray` component.

### Example 1-65 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`, RTL Compiler picks up the user module. If this attribute is set to `true`, 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 1-62, then it goes through the library look-up process. If a library cell of the same name is found, it becomes an instance of that library cell. If not, 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 1-61, or in VHDL, if a logic abstract is inferred from either an entity without an architecture, as shown in Example 1-63, or an entity whose architecture is empty, as shown in Example 1-64, 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`, 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.

## Writing Out a Logic Abstract in Verilog

If a logic abstract is internally treated as an unresolved reference, 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 1-67.

For example, for the RTL code shown in Example 1-67, 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 1-66, then Example 1-67 shows the resulting Verilog netlist.

### Example 1-66 Writing an Unresolved Reference as an Empty Module

```
set_attribute library tutorial.lbr
set_attribute write_vlog_empty_module_for_logic_abstract true
read_hdl test.v
elaborate
write_hdl
```

### Example 1-67 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 1-69.

If you set the `write_vlog_empty_module_for_logic_abstract` attribute to `false`, as shown in Example 1-68, then Example 1-69 shows the resulting Verilog netlist.

**Example 1-68 Writing an Unresolved Reference That Remains Unresolved in Netlist**

```
set_attr library tutorial.lbr
set_attr write_vlog_empty_module_for_logic_abstract false
read_hdl tst.v
elaborate
write_hdl
```

**Example 1-69 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 Black Box 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, 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, 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

# Synthesis Pragmas

# Overview

Synthesis pragmas are specially-formatted comments. Do not confuse these comments with Verilog HDL compiler directives that begin with `. Synthesis pragmas perform code selection or specify how the `set` and `reset` pins of a register are wired.

- RTL Compiler supports the following two forms of Verilog synthesis pragmas:
  - Short comments that terminate at the end of the line:

    `// cadence pragma_name`
  - Long comments that extend beyond one line:

    `/* cadence pragma_name */`
- RTL Compiler supports the following two forms of VHDL synthesis pragmas:
  - Attributes—Defines VHDL attributes attached to appropriate objects in the source VHDL.
  - Meta-comment—Defines the VHDL comments embedded in the VHDL source code. These pragmas begin with the `cadence synthesis` keyword.

**Note:** When using a comment for specifying a synthesis pragma, that comment should not contain any extra characters other than what is necessary for the synthesis pragma.

# Supported Synopsys Pragmas

## Verilog Supported Synopsys Pragmas

Table 2-1 lists the supported Verilog Synopsys pragmas. The pragma keyword in RTL Encounter is `cadence`. RTL Encounter also supports the `synopsys` pragma keyword.

**Table 2-1 Supported Verilog Synopsys Pragmas**

| Synopsys | Cadence RTL Compiler |
|---|---|
| `// synopsys label` | `// cadence label` |
| `// synopsys async_set_reset` | `// cadence async_set_reset` |
| `// synopsys async_set_reset_local` | `// cadence async_set_reset_local` |
| `// synopsys dc_script_begin` | `// cadence dc_script_begin` |
| `// synopsys dc_script_end` | `// cadence dc_script_end` |
| `// synopsys full_case` | `// cadence full_case` |
| `// synopsys map_to_module` | `// cadence map_to_module` |
| `// synopsys infer_mux` | `// cadence map_to_mux` |
| `// synopsys map_to_operator` | `// cadence map_to_operator` |
| `// synopsys parallel_case` | `// cadence parallel_case` |
| `// synopsys return_port_name` | `// cadence return_port_name` |
| `// synopsys sync_set_reset` | `// cadence sync_set_reset` |
| `// synopsys sync_set_reset_local` | `// cadence sync_set_reset_local` |
| `// synopsys template` | `// cadence template` |
| `// synopsys translate_off` | `// cadence translate_off` |
| `// synopsys translate_on` | `// cadence translate_on` |

## VHDL Supported Synopsys Pragmas

Table 2-2 lists the supported VHDL Synopsys pragmas.

**Table 2-2 Supported VHDL Synopsys Pragmas**

| Synopsys | RTL Compiler |
|---|---|
| `--synopsys label` | `--cadence label` |
| `--synopsys label_applies_to` | `--cadence propagate_label_to` |
| `--synopsys map_to_module` | `--cadence map_to_module` |
| `--synopsys infer_mux` | `--cadence map_to_mux` |
| `--synopsys map_to_operator` | `--synopsys map_to_operator` |
| `--synopsys return_port_name` | `--cadence return_port_name` |
| `--synopsys synthesis_off` | `--cadence synthesis off` |
| `--synopsys synthesis_on` | `--cadence synthesis on` |
| `--synopsys template` | `--cadence template` |
| `--synopsys translate_off` | `--cadence translate off` |
| `--synopsys translate_on` | `--cadence translate_on` |

# Specifying Synthesis Pragma Keywords

Normally, comments are meant to be ignored by RTL Compiler. However, setting a synthesis pragma keyword enables RTL Compiler to process a comment that begins with the specified keyword.

You can specify a pragma keyword and the name of individual pragmas. If the pragmas in the RTL code do not use the same keyword, then you can define a set of pragma keywords using the following two attributes. To define multiple keywords, put them in a TCL list.

- Set a pragma keyword using the following attribute.

  ```
  rc:/> set_attribute input_pragma_keyword cadence
  ```

  *Default*: `get2chip g2c ambit synopsys cadence pragma`

  Setting this pragma keyword tells RTL Compiler that every comment beginning with `cadence` is a synthesis pragma and should *not* be ignored.

  For example, if the RTL code has the `-- pragma translate_off` code selection pragma, then tell RTL Compiler to use the `pragma` keyword by setting the `input_pragma_keyword` attribute.

In the RTL code, a pragma has the following form:

```
// pragma_keyword pragma_name [pragma_value]
```

Some pragmas have a `pragma_value`. For popular pragmas, you can customize the `pragma_name`. If the RTL code uses multiple names for one pragma, then you can define a set of names for that pragma.

**Note:** Set this attribute before using the `read_hdl` command.

Use the attributes listed in Table 2-3 to define the names of individual pragmas. To define multiple names for one pragma, put them in a TCL list.

**Table 2-3 Synthesis Pragma Keyword Names**

| Attribute | Default |
|---|---|
| synthesis_off_command | {translate_off synthesis_off} |
| synthesis_on_command | {translate_on synthesis_on} |
| input_case_cover_pragma | full_case |
| input_case_decode_pragma | parallel_case |
| input_asynchro_reset_pragma | {async_set_reset asynchro_reset} |
| input_asynchro_reset_blk_pragma | {async_set_reset_local asynchro_reset_blk} |
| input_map_to_mux_pragma | {map_to_mux infer_mux} |
| input_synchro_reset_pragma | {sync_set_reset synchro_reset} |
| input_synchro_reset_blk_pragma | {sync_set_reset_local synchro_reset_blk} |
| input_synchro_enable_pragma | {synchro_enable} |
| input_synchro_enable_blk_pragma | {synchro_enable_blk} |
| delayed_pragma_commands_interpreter | dc |
| script_begin | {dc_script_begin script_begin} |
| script_end | {dc_script_end script_end} |

# Code Selection Pragmas

By default, RTL Compiler compiles all HDL code from a file. Use the code selection synthesis pragmas in pairs around HDL code that should not be compiled for synthesis. However, the code between the two pragmas will be checked for syntactic correctness.

## Verilog translate_on and translate_off Pragmas

In Verilog, all the code following the `// cadence translate_off` synthesis pragma up to and including the `// cadence translate_on` synthesis pragma is ignored by RTL Compiler.

For example, initialization code can be added for analysis purposes, as shown in Example 2-1. This code is not synthesized. If the `initial` block is surrounded by these synthesis pragmas, RTL Compiler will skip over the entire block.

### Example 2-1 Modeling 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)
...
```

## VHDL translate_on and translate_off Pragmas

In VHDL, all the code following the -- `cadence translate_off` synthesis pragma up to and including the `-- cadence translate_on` synthesis pragma is ignored by RTL Compiler.

You can add assertions in your model that are not synthesized for analysis purposes. If the assertions are surrounded by the `translate_on` and `translate_off` pragmas, RTL Compiler ignores them for synthesis, but verifies the syntax between the pragmas.

Use the `translate_on` and `translate_off` code selection pragmas, shown in Example 2-2, around VHDL code that should be completely ignored by the VHDL parser and that should not be synthesized by RTL Compiler. All the code following the synthesis pragma `cadence translate_off` up to and including the `cadence translate_on` synthesis pragma is ignored by RTL Compiler even if it contains syntax errors.

### Example 2-2 Modeling 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;
```

# case Statement Pragmas (Verilog)

A `case` statement can be interpreted in many ways. The default interpretation decodes the `case` labels in the order listed in the model. That is, the `case` statement is interpreted as a nested `if-else` statement.

The `full_case` and the `parallel_case` synthesis pragmas provide a mechanism to modify the default interpretation.

If the `case` statement has sufficient information, these synthesis pragmas are automatically inferred, even if they are not included in the code.

## full_case Pragma

If the synthesis pragma is `full_case`, then the `case` expression evaluates to only those values specified by the `case` labels in the `case` statement, as shown in Example 2-3. This implies that all other possible values of the `case` expression are treated as `don't care` conditions.

**Note:** This further implies that there is no need for a default clause in the `case` statement and a latch is not inferred.

### Example 2-3 Modeling the full_case Pragma to Suppress the Latch Inference (Verilog)

```
case (arith_opcode) // cadence full_case
    4`b0000 : result = 32'h0 ;// clear
    4`b0001 : result = src1 + src2 ;// add
    4`b0010 : result = src1 + 1`b1 ;// inc
    4`b1001 : result = src1 - src2 ;// sub1
    4`b1101 : result = src2 - src1 ;// sub2
    4`b1010 : result = src1 - 1`b1 ;// dec
endcase
```

Use the `full_case` synthesis pragma to suppress the latch inference only if the procedural assignments in each `case` item are made to all the variables modified in the `case` statement.

In the `case` statement, shown in Example 2-4, the second `case` item does not modify `reg2`, so it is inferred as a latch to retain the last value.

**Example 2-4 Modeling the full_case Pragma to Infer a Latch (Verilog)**

```
case (cntr_sig) // cadence full_case
   2`b00 : begin reg1 = 0 ; reg2 = v_field ; end
   2`b01 : reg1 = v_field ; // latch inferred for reg2
   2`b10 : begin reg1 = v_field ; reg2 = 0 ; end
endcase
```

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.

## parallel_case Pragma

If the synthesis pragma is `parallel_case`, then all the `case` labels have equal priority of matching the `case` expression. The optimizer uses this information to avoid building a decoder to decode for $2^n$ alternatives, where n is the size in bits of the `case` expression. The optimizer builds a parallel decoding logic instead of priority encoder logic to drive the select lines for the multiplexer. Example 2-5 shows how to model the `parallel_case` pragma.

**Example 2-5 Modeling the parallel_case Pragma (Verilog)**

```
case (1`b1) // cadence parallel_case
   cc[0] : cntr = 0 ;
   cc[1] : cntr = data_in ;
   cc[2] : cntr = cntr - 1 ;
   cc[3] : cntr = cntr + 1 ;
endcase
```

During simulation, if the `case` expression matches more than one `case` label, the logic corresponds to each `case` label. This causes the results to differ between RTL simulation and netlist simulation. This occurs if you use `casex` or `casez` statements to mask certain combinations. The RTL simulation performs the procedural assignment corresponding to the first `case` label match, whereas the gate-level simulation enables the logic for all the matching `case` labels. Therefore, ensure that only one `case` label is matched in the `case` statement before using the `parallel_case` synthesis pragma. Example 2-5 shows the logic which guarantees that only one of the four bits of `cc` is high at any given time.

# Set and Reset Synthesis Pragmas

Using the Verilog and VHDL set and reset synthesis pragmas only convey user preferences. They *do not force* RTL Compiler to honor the pragmas or change the behavior of the design. Therefore, in some scenarios the pragmas may be ignored to provide a better quality netlist. If the design is written with synchronous control on a flip-flop and the synthesis pragma specifies asynchronous selection, the resulting implementation will still be synchronous. A warning is displayed if the synthesis pragma conflicts with the model.

## Verilog Set and Reset Synthesis Pragmas

Use the set and reset synthesis pragmas to guide RTL Compiler to use set and reset pins to implement synchronous set and reset behavior on a flip-flop or asynchronous set and reset behavior on a latch. The default behavior is to implement these operations with the data input pins. The set and reset pragmas are honored in the elaborated netlist only if constant `0` or `1` assignments are made under the control of the specified set and reset signal.

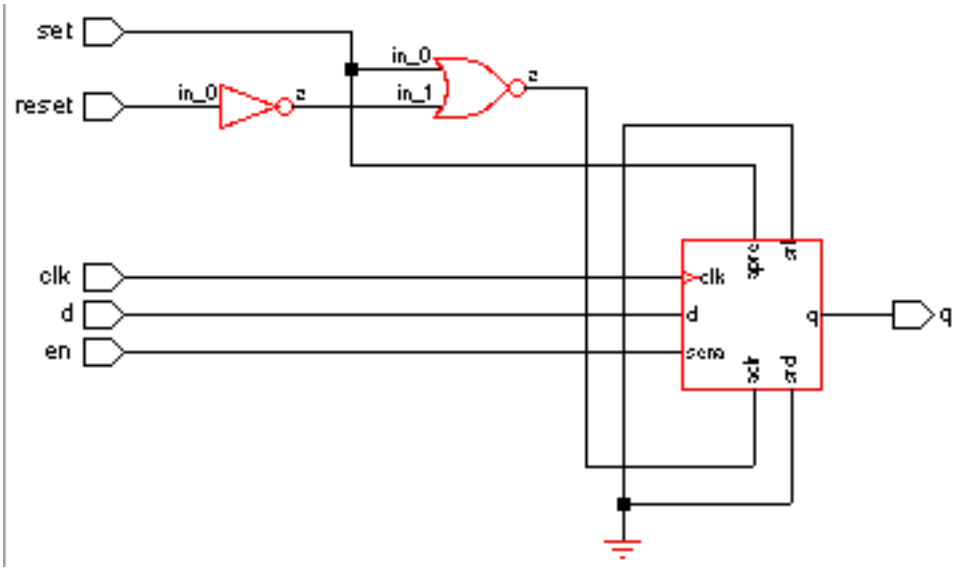
**Note:** Asynchronous set and reset behavior on a flip-flop is always implemented with asynchronous set and reset pins, regardless of the state of the set and reset synthesis pragmas, as shown Example 2-6.

### Example 2-6 Modeling 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 shows the corresponding schematic for Example 2-6.

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




## 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 `sync_set_reset` signal pragma is shown in Example 2-7 and Figure 2-2.

The set and reset signal pragmas are honored in the elaborated netlist only if constant `0` or `1` assignments are made under the control of the specified set and reset signal.

The *`signal_name_list`* is a comma separated list of signal names in a module, as shown in Example 2-7.

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.

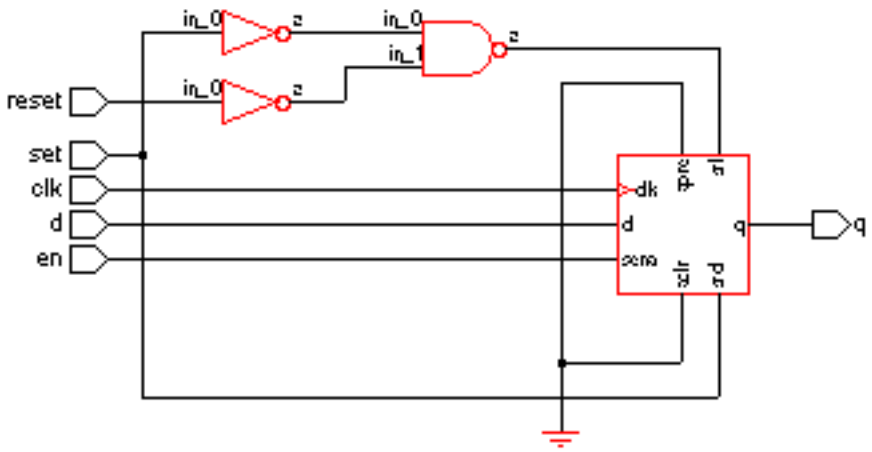
The flip-flop inferred for `q` is connected so that the set and reset signals connect to the synchronous `sr1` and `srd` pins. The `d` and `en` signals are connected through combinational logic feeding the `D` flip-flop and sena pins.

### Example 2-7 Modeling the synchronous_set_reset 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 shows the corresponding schematic for Example 2-7.

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



### Signals in a Block Pragma

For Verilog, specify the signal name for the `set` and `reset` operation by using the following pragmas in the named block, as shown in Example 2-8:

```
//cadence async_set_reset_local signal_name_list
//cadence sync_set_reset_local signal_name_list
```

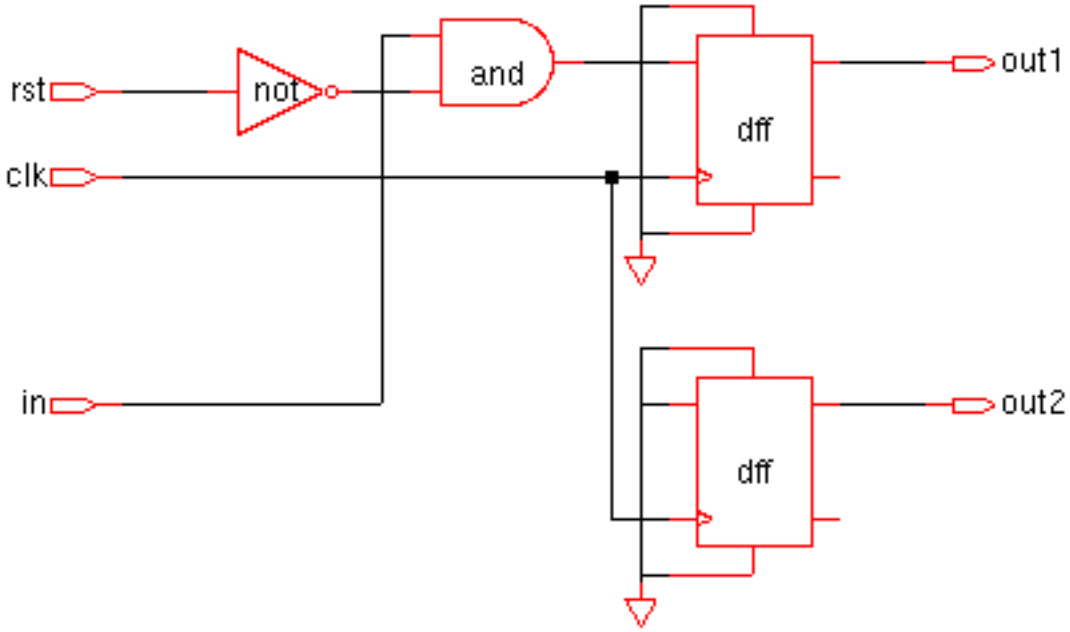
Only the signals listed in the named block that perform synchronous or asynchronous `set` and `reset` operations are connected to the synchronous or asynchronous pins respectively. For registers inferred from other blocks, these signals are connected to the data input.

### Example 2-8 Modeling sync_set_reset Signals in a Block Pragma (Verilog)

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

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

  always @(posedge clk) begin: blk_2
    if (rst)
      out2 <= 0;
    else out2 <= in;
      out2 <= 1'b0;
  end
endmodule
```

Figure 2-3 shows the corresponding schematic for Example 2-8.

**Figure 2-3 sync_set_reset Signals in a Block Synthesis Pragma (Verilog)**



## VHDL Set and Reset Synthesis Pragmas

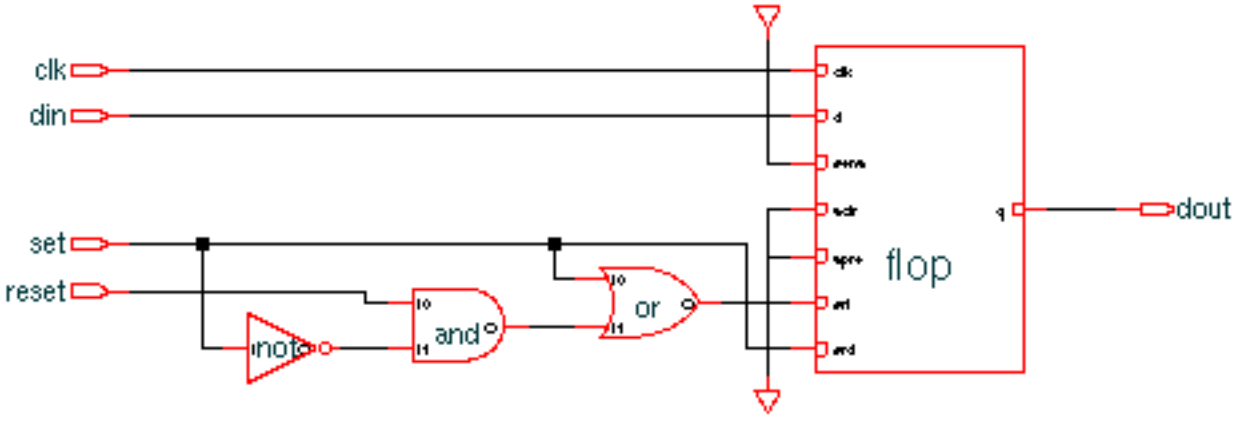
When the `elaborate` command infers a register from a VHDL description, the command also infers `set` and `reset` control of the register and defines whether these controls are synchronous or asynchronous. For examples showing flip-flops and latches with set and reset operations, see Modeling Latches on page 38 and Rising Edge Triggered Flip-Flop Schematic (Verilog) on page 29 in Chapter 1, "Modeling HDL Designs."

There are two ways to implement the synchronous `set` and `reset` logic for these inferred registers.

- Control the input to the data pin – Controls the input to the data pin of a register component using `set` and `reset` logic so that the data value is `1` when `set` is active, `0` when `reset` is active, and driven by the data source when both `set` and `reset` are inactive. This is the default approach.
- Implement `set` and `reset` control – Implements `set` and `reset` control of a register by selecting the appropriate register component (cell) from the technology library and connecting the output of set and reset logic directly to the `set` and `reset` pins of the component. The data pin of the component is driven directly by the data source.

Figure 2-4 shows the default implementation for the `set` and `reset` control logic.

**Figure 2-4 Default Implementation of Set and Reset Control Logic (VHDL)**




There are synthesis pragmas to support set and reset logic at the process level, signal level, or a mix of the process and signal levels for each register inferred. These synthesis pragmas are advisory pragmas only. They do not force the tool to implement set and reset logic with one approach; rather, they drive the selection of the component from the technology library to provide additional options.

## 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 below.

### Example 2-9 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.

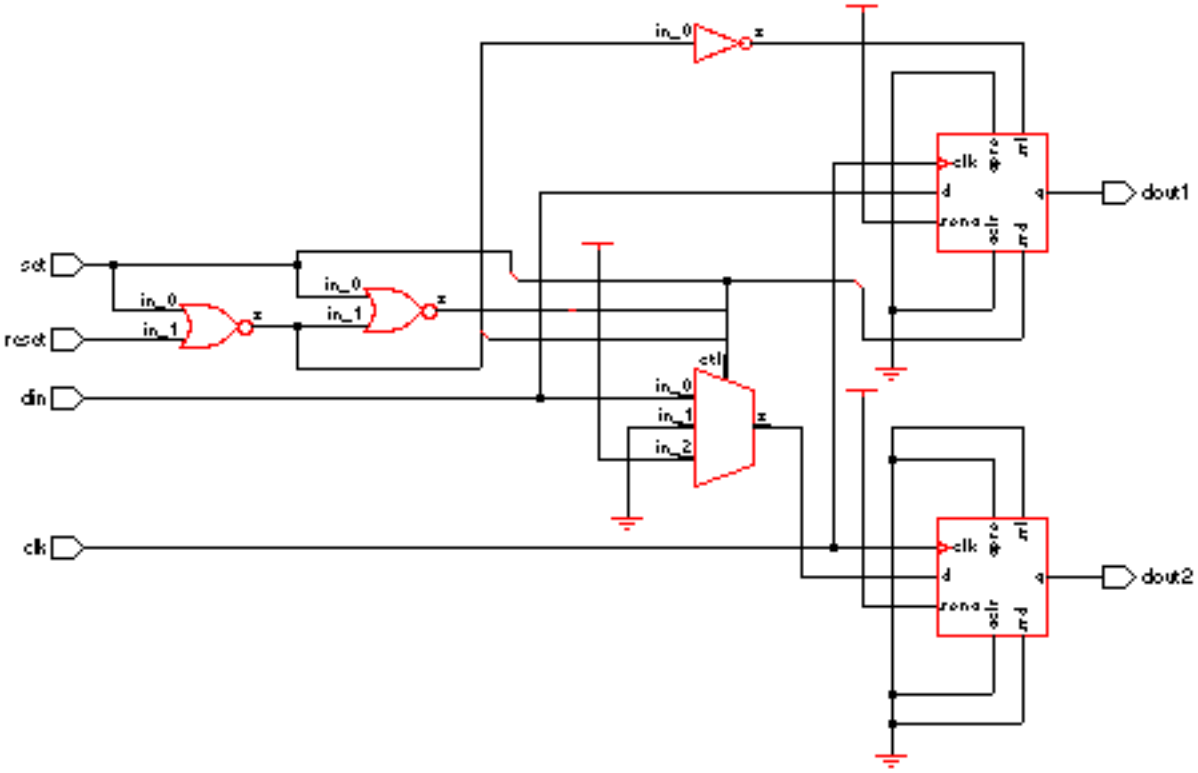
These process pragmas must be specified in the declarative region of the architecture that contains the corresponding processes. In Example 2-10, D-type flip-flops are inferred for the `dout1` and `dout2` signals. For `dout1`, the synchronous set and reset operations, controlled by the set and reset 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 2-10 Modeling 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 2-5 shows the corresponding schematic for Example 2-10.

**Figure 2-5 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 2-11.

### Example 2-11 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.

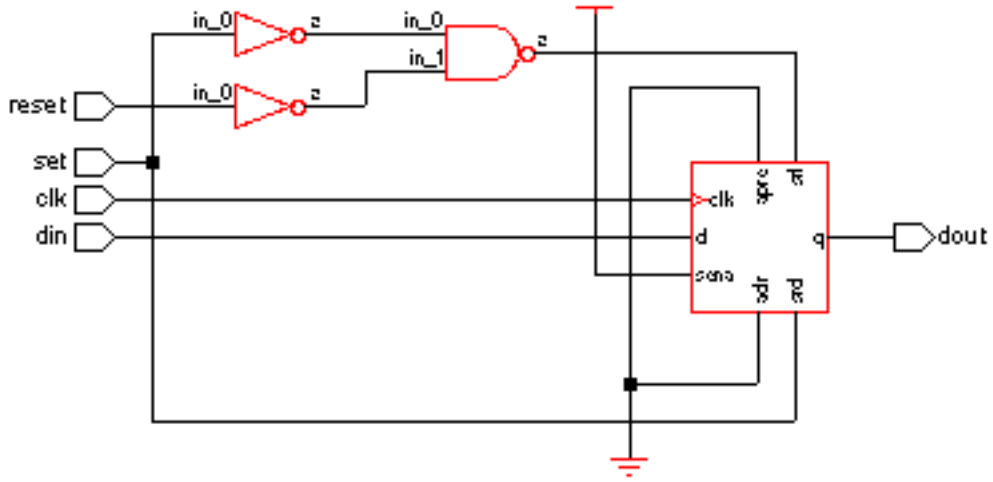
The flip-flop inferred for out1 and out2, shown in Example 2-12, is connected so that the set signal connects to the synchronous set pin and the reset signal is connected through combinational logic feeding the D data port.

### Example 2-12 Modeling 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 2-6.

**Figure 2-6 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 Process Pragmas on page 104, 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 2-13, 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 2-13 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 signal `set` 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.

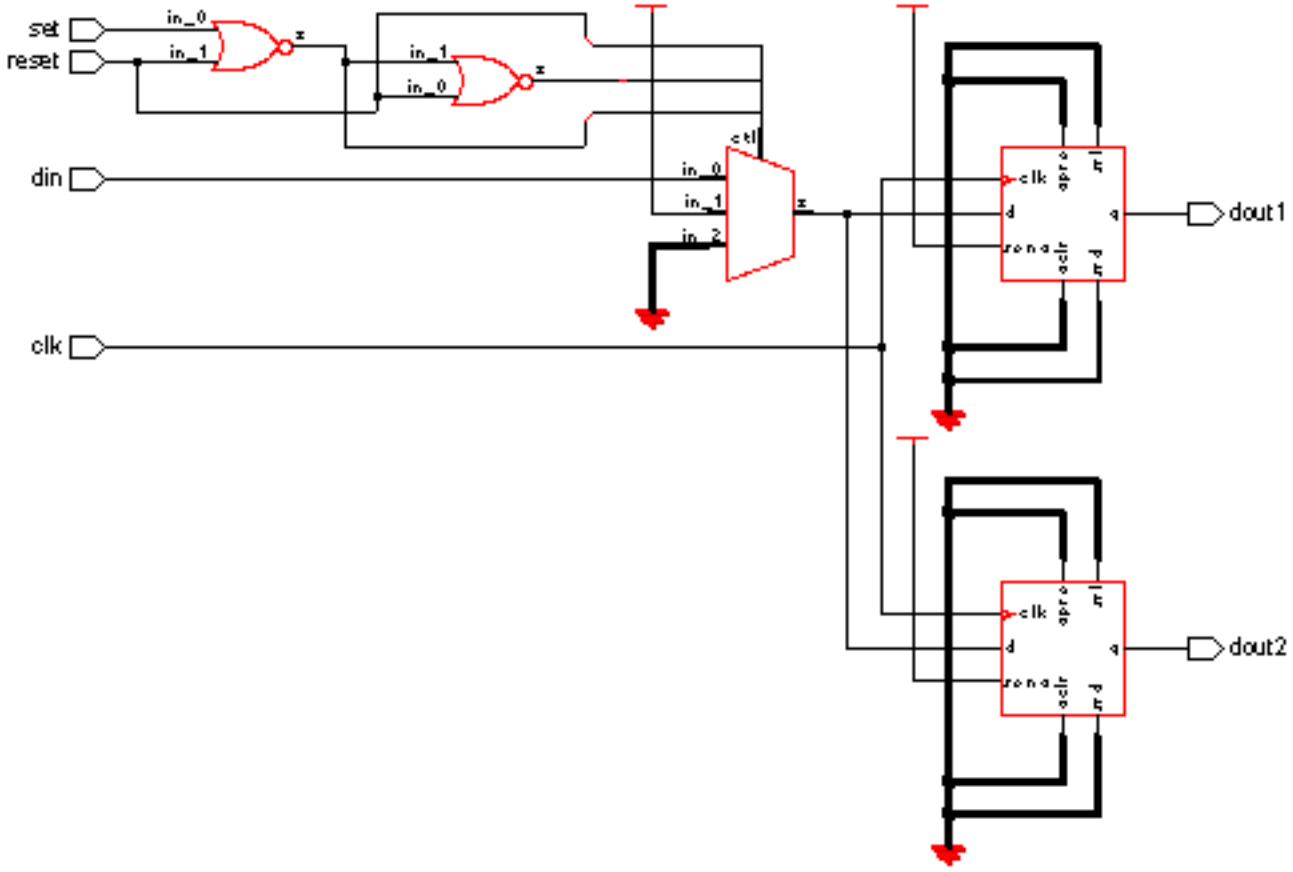
Only the listed signals in the process are inferred as synchronous or asynchronous `set` and `reset` signals and will be connected to the synchronous or asynchronous pins respectively. For registers inferred from other processes, signals can be connected to the data input as appropriate. Example 2-14 shows how to use the `sync_set_reset_local` synthesis pragma.

### Example 2-14 Modeling 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 2-7. 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 2-7 Implementing Set and Reset Synchronous Signals in a Block Logic (VHDL)**



# Multiplexer Mapping Pragma

Use the `map_to_mux` pragma (also called `infer_mux`) with the `case`, `if-then-else`, and Verilog or VHDL choice, such as `y = sel? a: b;` statements, and with Verilog named blocks to force RTL Compiler to implement the statement with multiplexer components from the technology library.

**Note:** The resulting netlist may have worse area, delay, or power than if RTL Compiler were not forced to map to multiplexers.

## Verilog Multiplexer Mapping Pragma

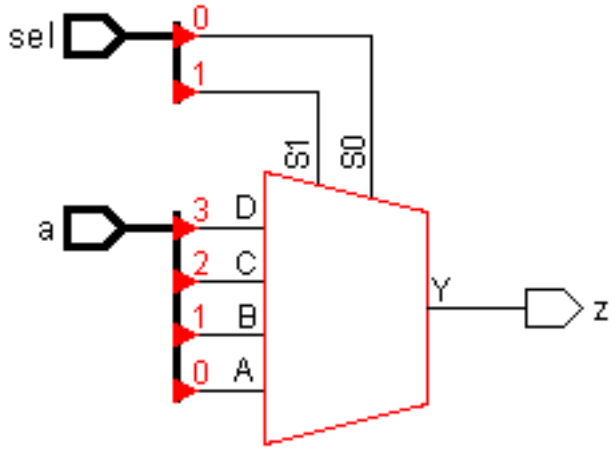
### Modeling the map_to_mux Pragma With a Case Statement

Example 2-15 shows the `map_to_mux` pragma with a `case` statement and Figure 2-8 shows the resulting schematic.

**Example 2-15 Modeling 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 2-8 map_to_mux (infer_mux) Pragma With a case Statement (Verilog)**



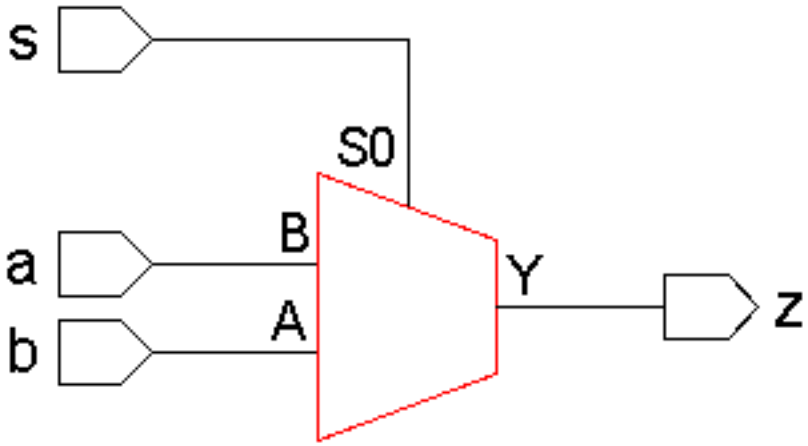

### Modeling the map_to_mux Pragma With an if Statement

Example 2-16 shows the `map_to_mux` pragma with an `if` statement and Figure 2-9 shows the resulting schematic.

**Example 2-16 Modelling map_to_mux (infer_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 2-9 map_to_mux Pragma (infer_mux) With an if Statement (Verilog)**



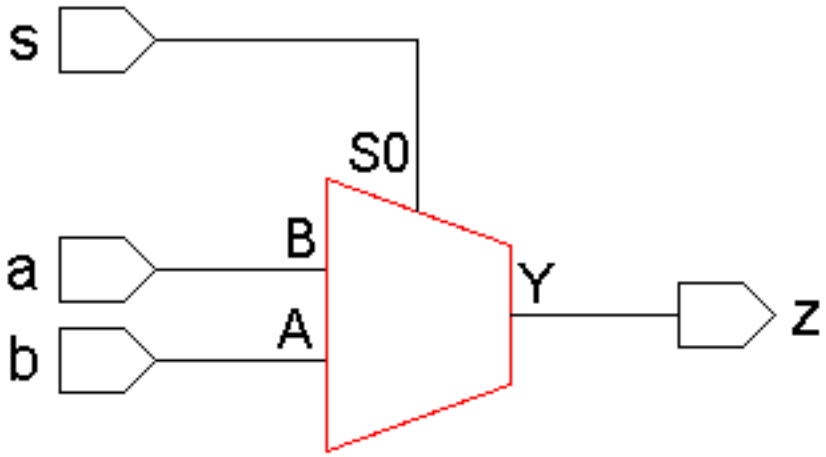

### Modeling the map_to_mux Pragma With a Choice Statement

Example 2-17 shows the `map_to_mux` pragma with a `choice` statement and Figure 2-10 shows the resulting schematic.

**Example 2-17 Modeling map_to_mux (infer_mux) Pragma With a Choice Statement**

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

**Figure 2-10 map_to_mux Pragma With a choice Statement**




## Modeling the map_to_mux Pragma for Named Blocks

Use the `map_to_mux` pragma for named blocks such that all mux possibilities within the block (`if`, `case`, variable bit-selects) are mapped to muxes. As shown in Example 2-18, the syntax generates muxes for case statements and indexed names within the named always blocks given in the pragma.

**Example 2-18 Modeling map_to_mux Pragma for Named Blocks**

```
// 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
```

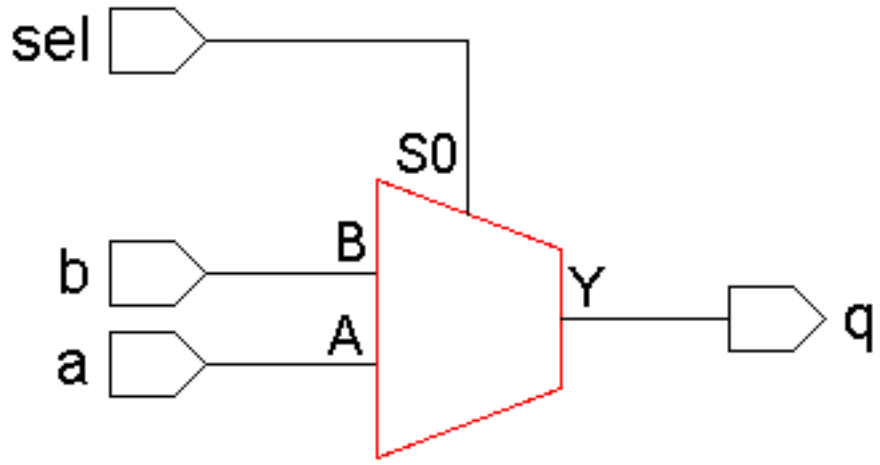
## VHDL Multiplexer Mapping Pragma

Example 2-19 shows the `map_to_mux` pragma with a `case` statement and Figure 2-11 shows the resulting schematic.

**Example 2-19 Modeling map_to_mux (infer_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 2-11 map_to_mux (infer_mux)Pragma With a Case Statement Schematic (VHDL)**

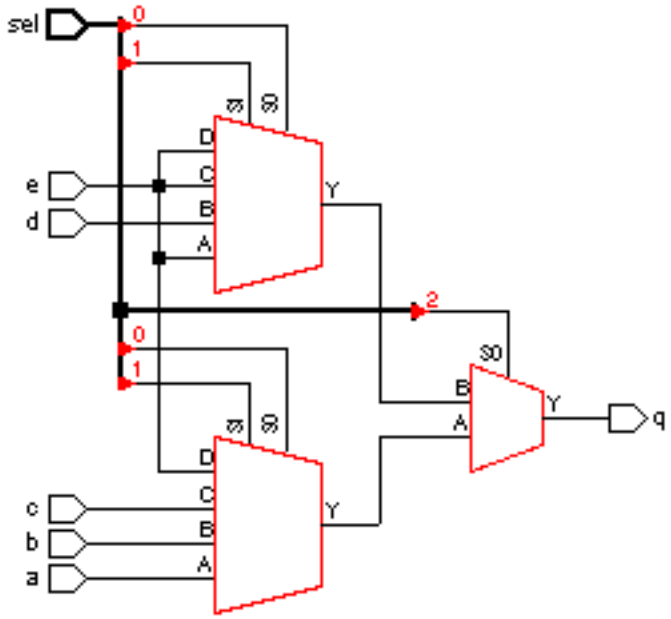


Example 2-20 shows the `map_to_mux` pragma with a choice statement and Figure 2-12 shows the resulting schematic.

**Example 2-20 Modeling the map_to_mux (infer_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 2-12 map_to_mux (infer_mux)Pragma With an if Statement Schematic (VHDL)**

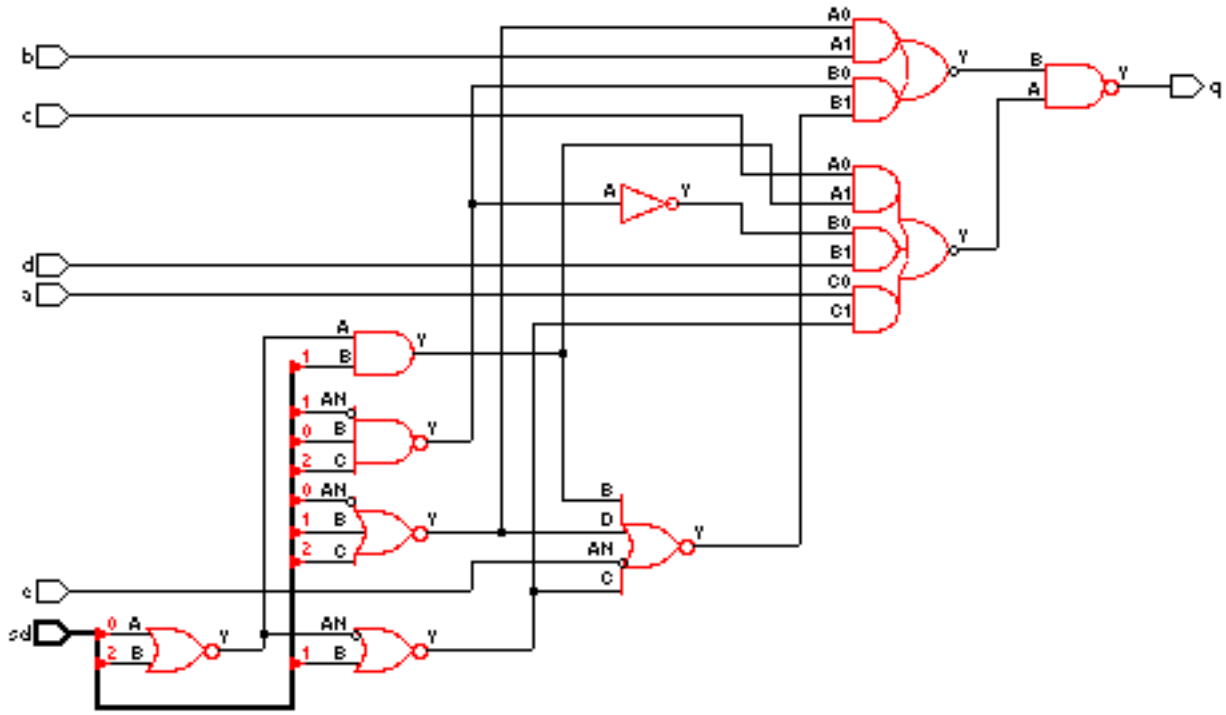



Example 2-21 shows the `map_to_mux` pragma with a choice statement RTL and Figure 2-13 shows the resulting schematic.

**Example 2-21 Modeling map_to_mux (infer_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 2-13 map_to_mux (infer_mux)Pragma With a Choice Statement Schematic (VHDL)**



# 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 2-22 maps a function to the BUF entity with a z output.

### Example 2-22 Modeling 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 entity instantiation:

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

is equivalent to the following function call:

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

# 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 2-23 shows the `ieee.numeric_std.signed` type.

### Example 2-23 Modeling 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;
```

# 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 2-24.

### Example 2-24 Modeling 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;
```

# Enumeration Encoding Pragma (VHDL)

Use this pragma to override the default encoding of enumeration literals. In Example 2-25, 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 2-25 Modeling 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”;
```

# 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 2-26 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 2-27, 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 2-27 Modeling 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.

# 3

# Using HDL Commands and Attributes

# HDL-Related Commands

**Table 3-1 HDL-Related Commands**

| Variable | 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. |
| `write_hdl` | Generates one of the following design descriptions in Verilog format:<br>■ A structural netlist using generic logic<br>■ A structural netlist using mapped logic |
| `read_netlist` | Reads and elaborates a Verilog 1995 structural netlist. |

# HDL-Related Attributes

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

- `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 port 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:

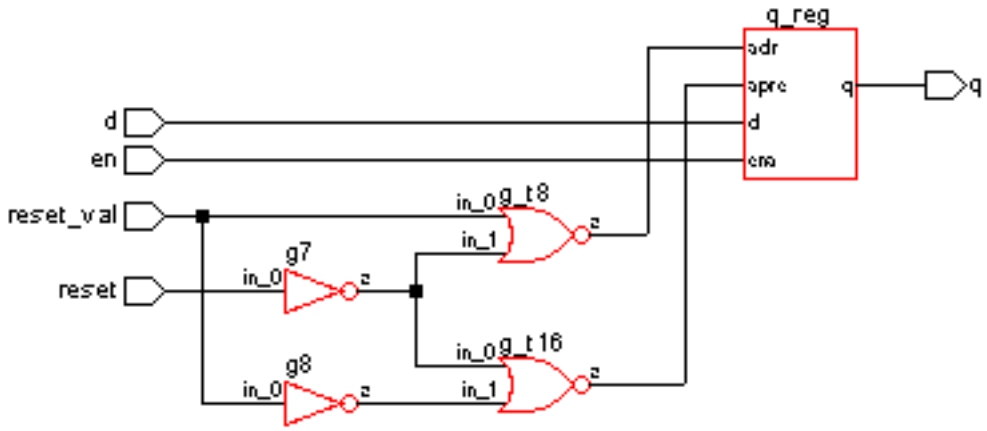
  ```
  rc:/> set_attr hdl_async_set_reset "reset"
  ```

**Example 3-1 RTL Asynchronous Set and Reset**

```
module asynch1(clk, d,en,q);
always @ (reset or en or d or reset_val) begin
    if (reset)
        q <= reset_val;
    else if (en)
        q <= d;
end
endmodule
```

The corresponding schematic is shown in Figure 3-1:

**Figure 3-1 Schematic hdl_async_set_reset**




- `hdl_auto_async_set_reset {true | false}`

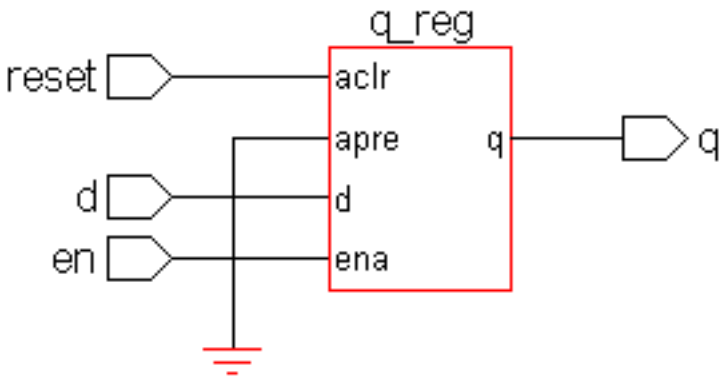
  Specifies that RTL Compiler implement logic using asynchronous set and reset pins on a latch if that logic controls an asynchronous assignment of a constant 0 or constant 1.
  *Default*: `false`

  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_sync_set_reset {true| false}`

  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.
*Default*: `false`

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

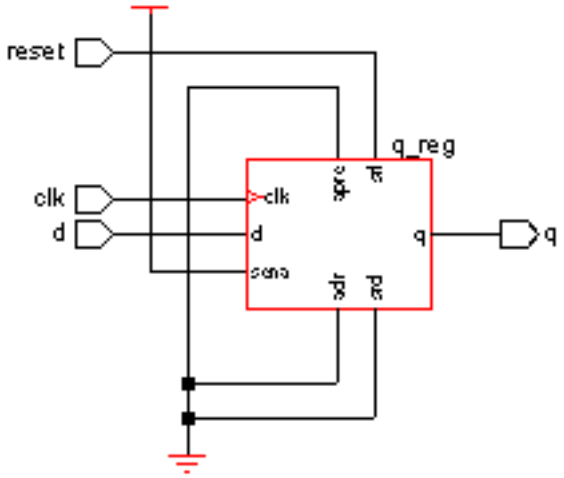
```
rc:/> set_attribute hdl_auto_sync_set_reset true
```

### Example 3-2 RTL Synchronous Reset

```
module synch1(clk, d,en,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_bit_blast_threshold` {`true | false`}

  When the value of this attribute is greater than 0, vector variables whose width is the value of the `hdl_bit_blast_threshold` attribute or greater are bit-blasted during elaboration.This results in a faster runtime and less memory usage during elaboration where there are many constant bit selects of large vector variables.
  *Default*: `0`

- `hdl_delete_transparent_latches` `{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`, allows to update the value of the `hdl_proc_name` instance attribute for sequential elements during elaboration.

- `hdl_error_on_blackbox` `{true | false}`

  When set to `true`, an error message is issued if there is an unresolved reference (black box) during elaboration.
  *Default*: `false`

- `hdl_error_on_latch` `{true | false}`

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

- `hdl_ff_keep_feedback` `{true | 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.
  *Default*: `true`

  - The following command implements a feedback path from the `Q` output to the `D` input for the RTL, as shown in Example 3-3:

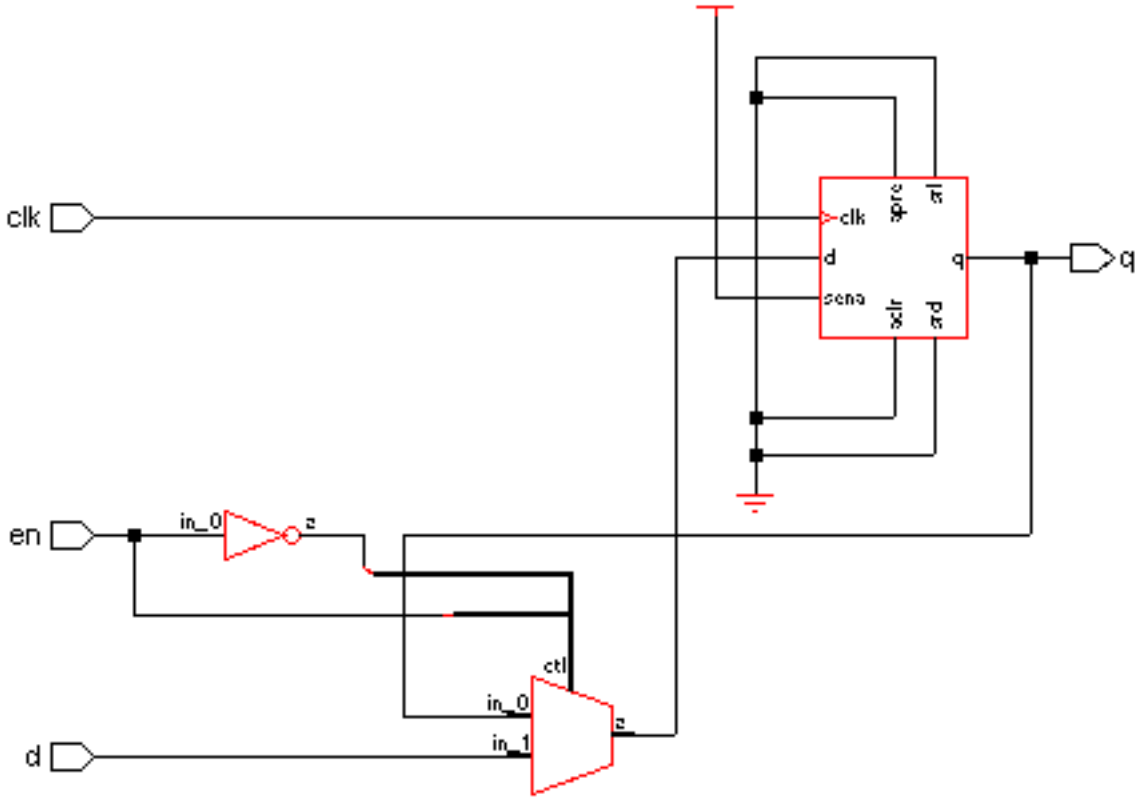
    ```
    rc:/> set_attribute hdl_ff_keep_feedback true
    ```

**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
```

The corresponding schematic is shown in Figure 3-4.

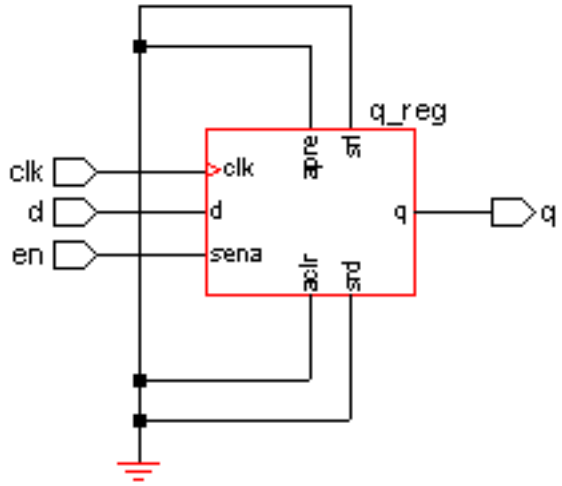
**Figure 3-4 Schematic hdl_ff_keep_feedback true**




- 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:

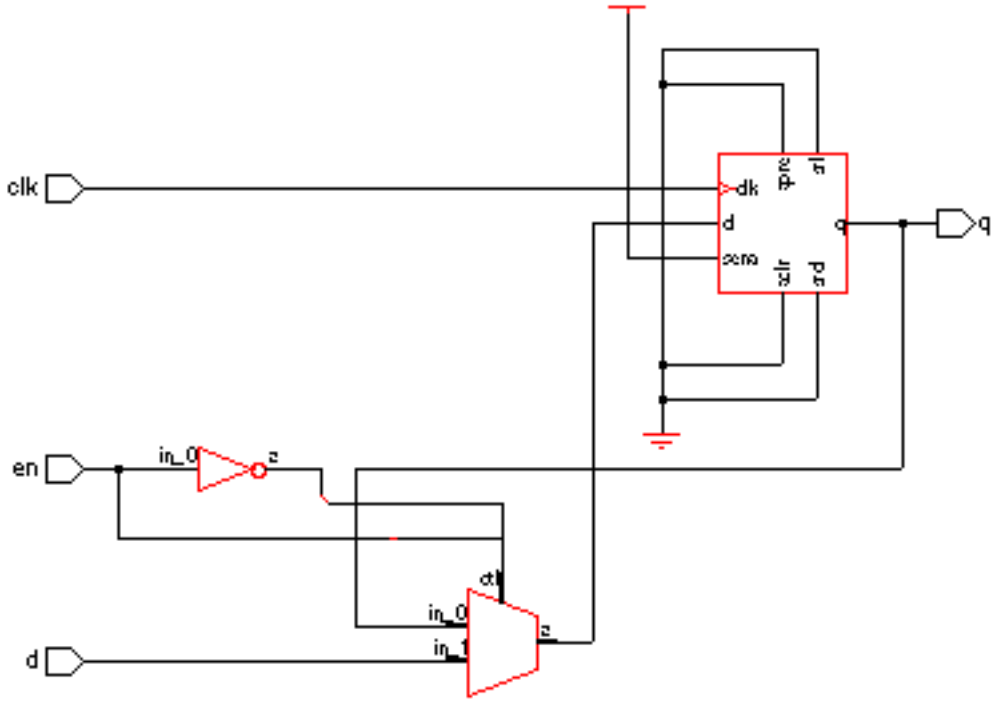
  ```
  rc:/> set_attribute hdl_ff_keep_explicit_feedback true
  ```

**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:

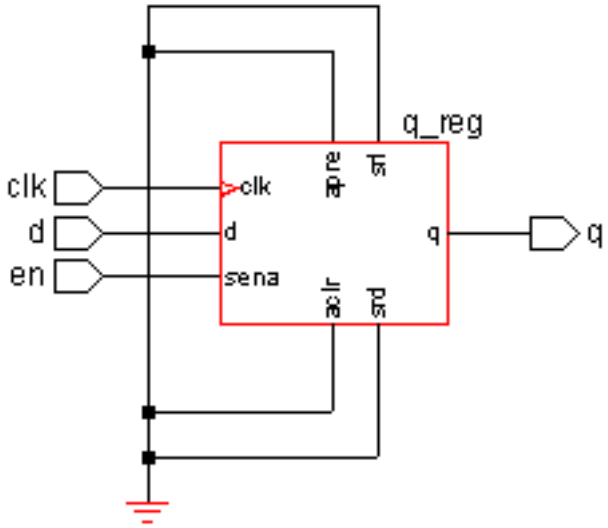
```
rc:/> set_attribute hdl_ff_keep_feedback false
rc:/> set_attribute hdl_ff_keep_explicit_feedback false
```

**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 {hdl_file ...} }...}`

  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 standards are in the `-v1995`, `-v2001`, and `-vhdl` HDL option forms.

- `hdl_infer_unresolved_from_logic_abstract true | false`

  *Default*: `true`

  See Modeling Logic Abstracts on page 77 for detailed information.

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

  *Default*: `v1995`

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

- `hdl_latch_keep_feedback` `{true | false}`

  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.

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

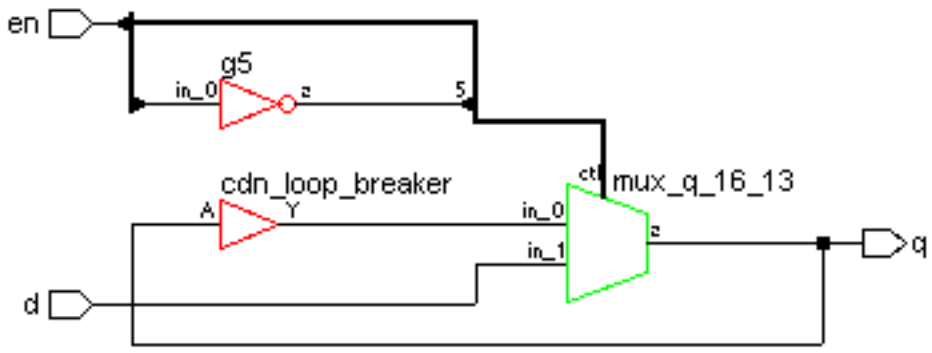
```
rc:/> set_attribute hdl_latch_keep_feedback true
```

**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:

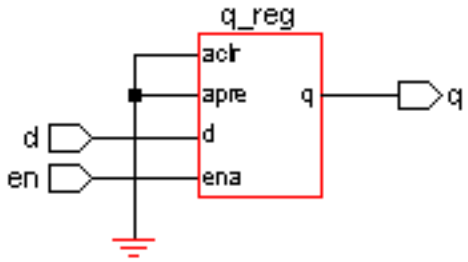
  ```
  rc:/ set_attribute hdl_latch_keep_feedback false
  ```

### 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_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:* `1000`

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

- `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 the *Using Encounter RTL Compiler* manual.

- `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 the *Using Encounter RTL Compiler* manual 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_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` *`"comma_separated_list_of_signals"`*

  *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 3-8, 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 3-8 RTL hdl_sync_set_reset “reset”

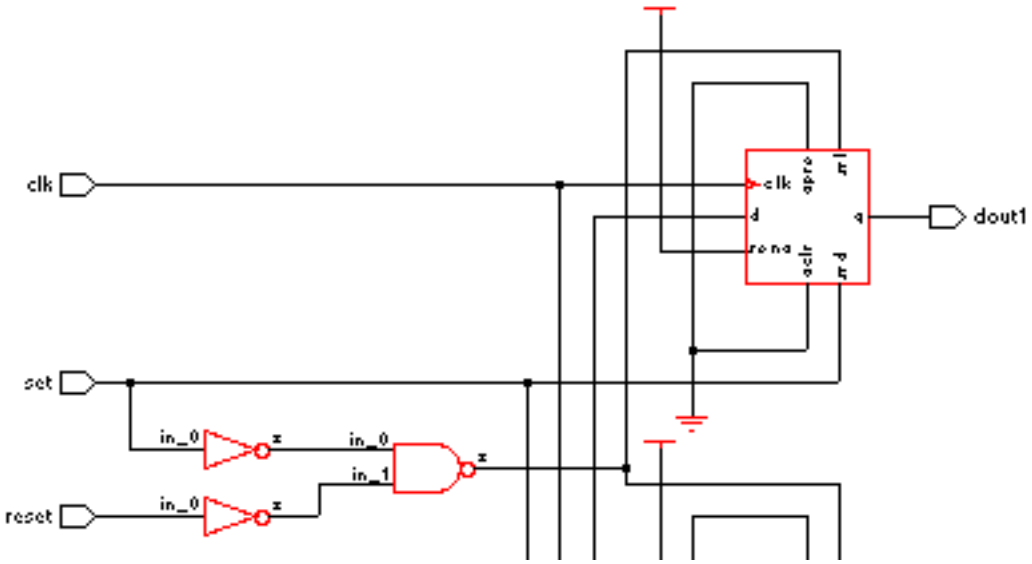
```
module sync(d,en,q);
always @ (posedge clk)
begin
    if (reset)
        q <= reset_val;
    else
        q <= d;
end
endmodule
```

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

```
... //cadence sync_set_reset "comma_separated_list_of_signals"
```

The corresponding schematic is shown in Figure 3-10.

### Figure 3-10 Schematic of Reset Signal

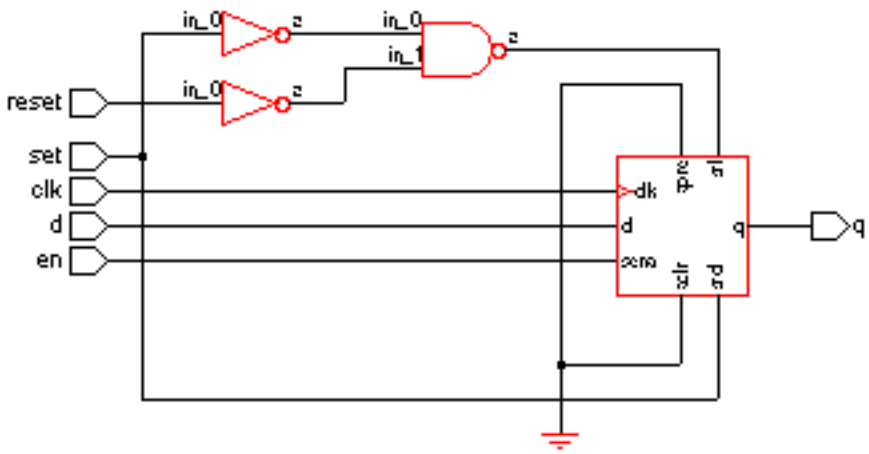



- In the RTL, shown in Example 3-9, RTL Compiler implements the set and reset operations using flip-flop synchronous set and reset pins, as shown in Figure 3-11.

### Example 3-9 Implementing Flip-Flop Synchronous set and reset Pins in the RTL

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

### Figure 3-11 Schematic of Set and Reset Operations




- <u>hdl_track_filename_row_col</u> {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.

  **Note:** Currently, only the flow down to synthesize -to_generic is supported.

- `hdl_trim_target_index` {`true` | `false`}

  *Default:* `true`

  Affects how logic is generated to implement the index of an array assignment when the index has more bits than necessary to address the array. When set to `true`, trims the index bits using the least number of bits required to address the array. This results in the most efficient implementation, but may result in a simulation mismatch between the original and synthesized design.

- `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` or `Z` values are specified, undriven signals remain undriven.

- `hdl_use_default_parameter_values_in_name` {`true` | `false`}

  *Default:* `false`

  When set to `false` shortens the name of the parmeterized module by using only the parameter values specified at instantiation, while the default 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_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`.

- `hdl_vector_naming_style` *`string`*

  *Default:* `%s_%d`

  Specifies the format in which flatten elements of array variables are printed out.

- `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**

| Command | Description (Default) |
|---|---|
| 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.<br>*Default*: v1995 |

# VHDL-Specific Attributes

**Table 3-3 VHDL-Specific Attributes**

| Command | Description (Default) |
|---|---|
| hdl_vhdl_case<br>{ lower \| upper \| original } | Stores VHDL identifiers and operators in lower case, upper case, or the case given in the source file.<br>*Default*: original |
| hdl_vhdl_environment<br>{ common \| synopsys} | Specifies the selection of the predefined arithmetic libraries.<br>*Default*: common |
| hdl_vhdl_lrm_compliance<br>{ true \| false } | When set to true, the read_hdl command enforces a more strict interpretation of the VHDL LRM.<br>*Default*: false |
| hdl_vhdl_preferred_architecture<br>*string* | Specifies the name of the preferred architecture to use with an entity when there are multiple architectures.<br>*Default*: *""* |
| hdl_vhdl_read_version<br>{ 1987 \| 1993 } | Specifies the VHDL version when files are analyzed using read_hdl.<br>*Default*: 1993 |

# 4

# Synthesizing Verilog Designs

# Overview

This chapter is organized for synthesizing Verilog RTL designs and provides links to the corresponding Verilog 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 Verilog Designs



# Synthesis Pragmas

# Using HDL Commands and Attributes

# 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.



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

## 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.

## Generate Statements

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

## 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:

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

### if generate Statement

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

**Example 4-1 Modeling the if generate Statement**

```
module
parameter pl=1,p2=2;

    generate if (pl == 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 Modeling 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;
input crc_en, crc8;
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
```

### 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 Modeling the case generate Statement for Multi-Way Branching

```
module
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 `case` expression p 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 Modeling the case generate Statement to Define Primitives**

```
module
generate
    case (width)
        1: counter_2bitx1 (en, reset, preset, datain, dataout);
                                    // 2 bit counter implementation
        2: counter_3bitx1 (en, reset, preset, datain, dataout);
                                    // 3 bit counter implementation
        default: counter_4bit #(width) x1 (en, reset, preset, datin, dataout);
                                    // others - 4 bit counter implementation
    endcase
endgenerate // generated instance is x1
endmodule
```

### for generate Statement

Use a `for generate` statement to replicate a concurrent block. The `for generate` statement uses a `genvar`.

## Genvar

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 Modeling the for generate Statement

```
module
genvar i;
generate for (i = 0; i <= 7; i = i + 1)
    begin : blah
        assign a[i] = b[i] + c[i];
    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[0];
    assign a[1] = b[1] + c[1];
    assign a[2] = b[2] + c[2];
    assign a[3] = b[3] + c[3];
    assign a[4] = b[4] + c[4];
    assign a[5] = b[5] + c[5];
    assign a[6] = b[6] + c[6];
    assign a[7] = b[7] + c[7];
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 Modeling the for generate Statement

```
module
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
```

## 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, 200l
tmp = m1[1];             // legal in Verilog-1995
```

## 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, they will write over each other's data.

Verilog-2001 includes reentrant 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.

## Parameter Passing by Name

Verilog-1995 defines two ways to change parameters for instantiated modules: parameter redefinition 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);
```

## 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
always @ (posedge clk, negedge reset)
begin
    if (!reset)
      q = 0;
    else
      q = d;
 end
endmodule
```

## 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 = 1;
output q;
reg q;
input d;
wire d;

    always @(d)
        q = d;
endmodule
```

### Example 4-12 Verilog-2001 ANSI C-like Declaration

```
module m #(parameter p = 1)
        (output reg q, input wire d);
    always @(d)
        q = d;
endmodule
```

Use this enhancement in functions and tasks to make port declarations more compact.

## 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 [3:0] 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]};
```

## 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 Modeling 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
```

## 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:

- ifndef Directive (comparable to `#ifndef`)
- line Directive (comparable to `#line`).
- elsif Directive (comparable to `#elif`)

### 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, 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”0;
    `else
        initial $display (“first block and second_nest defined”);
    `endif
...
```

### line Directive

The `line directive is mainly used by a source preprocessor to relate the processed output back to the original source file. 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.

### 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
    ...
`elsify y
    ...
`endif
```

is equivalent to:

```
`ifdef x
    ...
`else
    `ifdef y
        ...
    `endif
`endif
```

# 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 Verilog Modeling Constructs



## Verilog and Verilog-2001 Constructs and Level of Support

Table 4-1 lists the level of support for all Verilog HDL constructs and indicates the level as fully supported (Full), partially supported (Partial), ignored (Ignored), and not supported (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 | Support | Extension |
|---|---|---|---|
| Basic | Identifiers | Full | |
| | Escaped identifiers | Full | |
| | Sized constants (`b, o, d, h`) | Full | |
| | Unsized constants<br>`2'b11`, `3'07`, `32'd123`, `8'hff` | Full | |
| | Signed constants (s)<br>`3'bs101` | Full | Verilog-2001 |
| | String constants | Full | |
| | Real constants | No | |
| | Use of `z`, `?` in constants | Full | |
| | Use of `x` in constants | Full | |
| | `module`, `endmodule` | Full | |
| | `macromodule` | Full | |
| | Hierarchical references | No | |
| | `//`comment | Full | |
| | `/*`comment`*/` | Full | |

**Table 4-1 Verilog Constructs and Level of Support,** *continued*

| Group | Construct | Support | Extension |
|---|---|---|---|
| Basic, *Continued* | System tasks<br>`$display` | Ignored | |
| | System functions<br>Only `$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`, `small`, `medium`, `large` | No | |
| | `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 | |
| | Memory<br>For example, `reg [7:0] x [3:0];` | Full | |
| | N-dimensional arrays | Full | Verilog-2001 |
| | | | |
| input [ ] d; | | | |
| Drive strengths | | Ignored | |

**Table 4-1 Verilog Constructs and Level of Support,** *continued*

| Group | Construct | Support | Extension |
|---|---|---|---|
| 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 | |
| User defined primitives (UDPs) | `primitive` | No | |
| | `table` | No | |
| Operators and expressions | `+`, `-` (binary and unary) | Full | |

**Table 4-1 Verilog Constructs and Level of Support,** *continued*

| Group | Construct | Support | Extension |
|---|---|---|---|
| Report operators and expressions | /, %<br>See Notes on Verilog Constructs on page 169 | Full | |
| | * | Full | |
| | ~ | Full | |
| Bitwise operations | &, \|, ^, ~^, ^~ | Full | |
| Reduction operations | &, \|, ^, ~&, ~\|, ~^, ^~<br>!, &&, \|\|<br>==, !=, <, <=, >, >=<br><<, >><br><<< >>><br>{}, {n{}}<br>?:<br>function call | Full | 2001 |
| | ===, !== | No | |
| | **<br>*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 |
| | posedge, negedge | Partial | |
| | wait | Ignored | |
| | Intra-assignment event control | Ignored | |
| | Event trigger (->) | No | |
| Bit and part selects | Bit select | Full | |

**Table 4-1 Verilog Constructs and Level of Support,** *continued*

| Group | Construct | Support | Extension |
|---|---|---|---|
| | Bit select of array element | Full | Verilog-2001 |
| | Constant part select<br>**Note:** The bounds of a part select may be elaboration-time constants. | Full | |
| | Variable part select ( `+:`, `-:` ) | Full | 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*`<br>The `repeat` statement must have a constant repeat count. | Full | |
| | `case`, `casex`, `casez`, `default` | Full | |
| | Task enable | Full | |
| | `for` (constant bounds, only + and - operation on index)*<br>The `for` statement must have constant bounds. | Partial | |
| | `while*`<br>The `while` statement must have constant bounds. | Partial | |

**Table 4-1 Verilog Constructs and Level of Support,** *continued*

| Group | Construct | Support | Extension |
|---|---|---|---|
| | `forever*`<br>The `forever` statement must contain a disable statement. | Partial | |
| | ***A loop is unrolled to a maximum count specified in `hdl_max_loop limit` | | |
| | `disable`<br>The `disable` 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 | |
| | `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 |

**Table 4-1 Verilog Constructs and Level of Support,** *continued*

| Group | Construct | Support | Extension |
|---|---|---|---|
| | `ifdef, `else, `endif | Full | |
| | `include | Full | |

## 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 Verilog-2001 `$signed` and `$unsigned` system functions are also supported in the Verilog 1995 mode.
- The Verilog 2001 `$signed` keyword is 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.

# Supported SystemVerilog Hardware Description Language Constructs

RTL Compiler supports the synthesizable subset of SystemVerilog 1800-2005. SystemVerilog is built on top of Verilog 2001 and improves the usability of Verilog code.

Table 4-2 lists the level of support for the SystemVerilog 1800-2005 constructs and indicates the level as fully supported (Full), partially supported (Partial), and ignored (Ignored). The chapter and section numbers follow the latest draft of the 1800-2005 standard.

**Table 4-2 Supported SystemVerilog Constructs**

| Construct | Chapter & Section Number | Support |
|---|---|---|
| unsized literals | 3.3 | Full |
| time units in literals | 3.5 | Partial/Ignored |
| string literals | 3.6 | Full |
| array literals | 3.7 | Full |
| structure literals | 3.8 | Full |
| logic (4-state) data types | 4.3 | Full |
| integer and bit (2-state) data types | 4.3 | Full |
| byte, shortint, longint | 4.2 | Full |
| shortreal data type | 4.4 | Partial |
| user-defined types | 4.9 | Full |
| enumeration data type | 4.10 | Full |
| typedef enum | 4.10.1 | Full |
| enum type ranges | 4.10.2 | Full |
| packed structure data type (4-state) | 4.11 | Full |
| packed structure data type (2-state) | 4.11 | Full |
| structure data type (unpacked) | 4.11 | Full |
| union data type (packed) | 4.11 | Full |
| union data type (unpacked) | 4.11 | Partial |
| casting | 4.14 | partial |

**Table 4-2 Supported SystemVerilog Constructs,** *continued*

| | | |
|---|---|---|
| packed arrays | 5.2 | Full |
| unpacked arrays | 5.2 | Full |
| indexing and slicing of arrays | 5.4 | Full |
| array query functions | 5.5 | Full |
| array assignment | 5.7 | Full |
| arrays as arguments | 5.8 | Full |
| constants | 6.3 | Full |
| scope/lifetime (unnamed blocks) | 6.6 | Full |
| default attribute type | 6.2 | Partial/Ignored |
| assignment operators as statements | 8.3 | Full |
| assignment operators as expressions | 8.3 | Full |
| postincrement/decrement statements | 8.3 | Full |
| preincrement/decrement statements | 8.3 | Full |
| ++ and -- as expressions | 8.3 | Full |
| unpacked array expressions | 8.15 | Full |
| structure expressions | 8.13 | Full |
| aggregate expressions | 8.15 | Full |
| do while loop | 10.5 | Full |
| enhanced for loop | 10.5 | Full |
| jump statements (return, break, continue) | 10.6 | Full |
| final blocks | 10.7 | Ignored |
| named blocks (matching end block name) | 10.8 | Full |
| iff event control | 10.10 | Full |
| always_comb | 11.2 | Full |
| always_latch | 11.3 | Full |
| always_ff | 11.4 | Full |
| continuous assignments to variables | 11.5 | Full |
| void functions | 12.3.1 | Full |

**Table 4-2 Supported SystemVerilog Constructs,** *continued*

| | | |
|---|---|---|
| discarding func return | 12.3.2 | Full |
| pass by reference | 12.4.2 | Full |
| immediate assertions | 17.2 | Partial/Ignored |
| extern modules | 19.7 | Full |
| interface ports | 19.8 | Full |
| variable ports | 19.8 | Full |
| array ports | 19.8 | Full |
| structure/union ports | 19.8 | Full |
| timeunit and timeprecision | 19.10 | Partial/Ignored |
| implicit .name port connections | 19.11.3 | Full |
| implicit .* port connections | 19.11.4 | Full |
| attributes on interfaces | 20.2 | Partial/Ignored |
| named bundles | 20.2.2 | Full |
| generic bundles | 20.2.3 | Full |
| ports in interfaces | 20.3 | Full |
| tasks and functions in interfaces | 20.6 | Full |
| modports | 20.4 | Full |
| parameterized Interfaces | 20.7 | Full |
| typed parameters (V2001 feature) | 6.3 | Full |
| parameterized types | 6.3 | Full |
| expression size $bits | 22.3 | Full |
| define macros | 23.2 | Full |

# Troubleshooting

## 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_keywords` 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 is1364-2001. In the -v1995 mode, which is the default for the `read_hdl` command, the default is `364-1995`.

# 5

# Synthesizing VHDL Designs

# Overview

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 47
- Modeling Combinational Logic in VHDL on page 44
- Modeling Latches in VHDL on page 39
- Modeling Latches in VHDL on page 39
- Modeling Flip-Flops in VHDL on page 31
- Using Case Statements in VHDL on page 70
- Using a for Statement in VHDL on page 75
- Inferring a Logic Abstract From the RTL in VHDL on page 78
- Interpreting a Logic Abstract in Verilog or VHDL on page 82


# Synthesis Pragmas


- VHDL Supported Synopsys Pragmas on page 91
- Specifying Synthesis Pragma Keywords on page 92
- VHDL translate_on and translate_off Pragmas on page 95
- VHDL Set and Reset Synthesis Pragmas on page 103
- VHDL Signal Pragmas on page 106
- VHDL Multiplexer Mapping Pragma on page 116
- Function and Task Mapping Pragmas (Verilog and VHDL) on page 120
- Template Pragma (Verilog and VHDL) on page 122
- Enumeration Encoding Pragma (VHDL) on page 123

- Resolution Function Pragmas (VHDL) on page 124
- Resolution Function Pragmas (VHDL) on page 124

# Using HDL Commands and Attributes

- HDL-Related Commands on page 126
- HDL-Related Attributes on page 127
- VHDL-Specific Attributes on page 144

Supported VHDL Constructs on page 178

# Supported VHDL Constructs



Table 5-1 lists the VHDL constructs supported by RTL Compiler. See Notes on Supported Constructs on page 183 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 |

**Table 5-1 VHDL Constructs Supported in RTL Compiler ,** *continued*

| **Construct** | | | **Support** |
|---|---|---|---|
| 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 |

**Table 5-1 VHDL Constructs Supported in RTL Compiler ,** *continued*

| **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 |

**Table 5-1 VHDL Constructs Supported in RTL Compiler ,** *continued*

| **Construct** | | | **Support** |
|---|---|---|---|
| Expressions | Logical Operators | and, or, nand, nor, xor, xnor | 1993 |
| | Relational Operators | =, /=, >, <, >=, <= | Full |
| | Shift Operators | sll (shift left logical)<br>srl (shift right logical)<br>sra (shift right arithmetic)<br>sla (shift left arithmetic) | Full |
| | | ror, rol | Full |
| | Arithmetic Operators | `+, -, &` | Full |
| | Sign Operators | +, - | Full |
| | Multiplying Operators | * | Full |
| | | mod | Full |
| | | /, rem | Full |
| | Miscellaneous Operators | * * | Partial |
| | | abs | Full |
| | | `not` | Full |
| | Operands | Integer literal | Full |
| | | Real literal | Ignore |
| | | Physical literal | Ignore |
| | | Enumeration literal | Full |
| | | String literal | Full |
| | | Bit string literal | Full |
| | | Null literal | No |
| | Aggregates | Record aggregates | Full |
| | | Array aggregates | Full |
| | Function calls | Qualified expression | Full |
| | | Type conversion | Full |

**Table 5-1 VHDL Constructs Supported in RTL Compiler ,** *continued*

| **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 |
| Concurrent Statements | | | |
| | Block | Guard | No |
| | | Block header | No |
| | | Block declarative part | Full |
| | | Block statement part | Full |
| | | Timeout clause | Ignored |

**Table 5-1 VHDL Constructs Supported in RTL Compiler ,** *continued*

| **Construct** | | | **Support** |
|---|---|---|---|
| Concurrent Statements, cont. | 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 and Configurations

- 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 184 for more information.
- Generics must have a default value specified, unless the entity has a `TEMPLATE` attribute set to `true`. See Chapter 2, “Synthesis Pragmas” for more information.
- Declarations in an entity or architecture declarative part must be supported declarations. See Declarations on page 185 for more information.
- 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.

### 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 185 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. Refer to Chapter 2, “Synthesis Pragmas” for further 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 `ENUM_ENCODING` attribute to override the default mapping between an enumerated type and the corresponding encoding value. See Chapter 2, “Synthesis Pragmas” for further 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 ENUM_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.
- Null ranges are not supported.

### 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 is supported.
- Expressions in attribute names are unsupported.

### Expressions

- 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, “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.

### 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 statically computable.
- 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 statically computable.
- Declarations in a generate statement are only supported in the VHDL IEEE 1076-1993 standard mode.

## VHDL Predefined Attributes

**Table 5-2 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 |

**Table 5-2 VHDL Predefined Attributes**

| 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

## A

# B



# C



# D



# E



# F



# G

# H



# I



# K



# L



# M

# P



# R



# S



# T



# V

# W