Synopsys FPGA Synthesis
Attribute Reference Manual

January 2014

SYNOPSYS®

Copyright Notice and Proprietary Information

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only.

Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

"This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of _____ and its employees. This is copy number _____."

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

Registered Trademarks (®)

Trademarks (™)

Service Marks (sm)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.

ARM and AMBA are registered trademarks of ARM Limited.

Saber is a registered trademark of SabreMark Limited Partnership and is used under license.

All other product or company names may be trademarks of their respective owners.

Printed in the U.S.A
January 2014

Contents

Chapter 1: Introduction

CHAPTER 1

Introduction

This document is part of a set that includes reference and procedural information for the Synopsys® Synplify Pro® FPGA synthesis tools.

This document describes the attributes and directives available in the synthesis tools. The attributes and directives let you direct the way a design is analyzed, optimized, and mapped during synthesis. Throughout the documentation, features and procedures described apply to all tools, unless specifically stated otherwise.

This chapter includes the following introductory information:

How Attributes and Directives are Specified

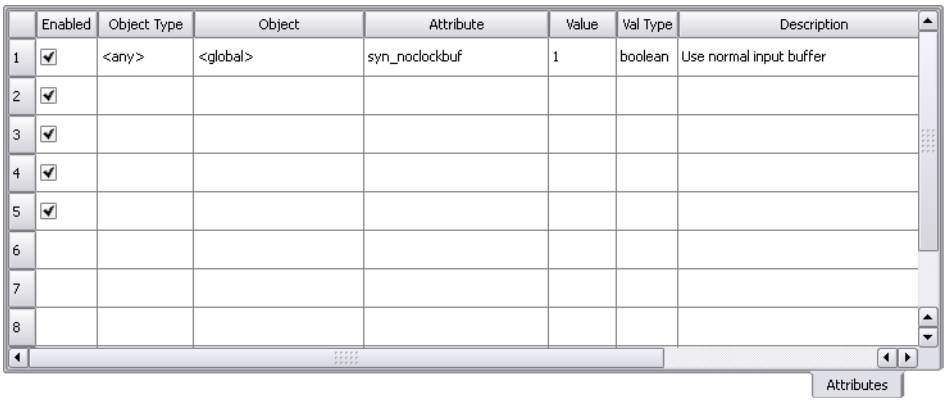
By definition, *attributes* control mapping optimizations and *directives* control compiler optimizations. Because of this difference, directives must be entered directly in the HDL source code. Attributes can be entered either in the source code, in the SCOPE Attributes tab, or manually in a constraint file. For detailed procedures on different ways to specify attributes and directives, see Specifying Attributes and Directives, on page 87 in the *User Guide*.

Verilog files are case sensitive, so attributes and directives must be entered exactly as presented in the syntax descriptions. For more information about specifying attributes and directives using C-style and Verilog 2001 syntax, see Verilog Attribute and Directive Syntax, on page 366.

The SCOPE Attributes Tab

This section describes how to enter attributes using the SCOPE Attributes tab. To use the SCOPE spreadsheet, use this procedure:

1. Start with a compiled design, then open the SCOPE window.
2. Scroll if needed and click the Attributes tab.



	Enabled	Object Type	Object	Attribute	Value	Val Type	Description
1	☑	<any>	<global>	syn_noclockbuf	1	boolean	Use normal input buffer
2	☑						
3	☑						
4	☑						
5	☑						
6							
7							
8							

Attributes

3. Click in the Attribute cell and use the pull-down menus to enter the appropriate attributes and their values.

The Attributes panel includes the following columns.

Column	Description
Enabled	(Required) Turn this on to enable the constraint.
Object Type	Specifies the type of object to which the attribute is assigned. Choose from the pull-down list, to filter the available choices in the Object field.
Object	(Required) Specifies the object to which the attribute is attached. This field is synchronized with the Attribute field, so selecting an object here filters the available choices in the Attribute field. You can also drag and drop an object from the RTL or Technology view into this column.
Attribute	(Required) Specifies the attribute name. You can choose from a pull-down list that includes all available attributes for the specified technology. This field is synchronized with the Object field. If you select an object first, the attribute list is filtered. If you select an attribute first, the synthesis tool filters the available choices in the Object field. You must select an attribute before entering a value.
Value	(Required) Specifies the attribute value. You must specify the attribute first. Clicking in the column displays the default value; a drop-down arrow lists available values where appropriate.
Val Type	Specifies the kind of value for the attribute. For example, string or boolean.
Description	Contains a one-line description of the attribute.
Comment	Contains any comments you want to add about the attributes.

For more details on how to use the Attributes panel of the SCOPE spreadsheet, see Specifying Attributes Using the SCOPE Editor, on page 90 in the *User Guide.*

When you use the SCOPE spreadsheet to create and modify a constraint file, the proper define_attribute or define_global_attribute statement is automatically generated for the constraint file. The following shows the syntax for these statements as they appear in the constraint file.

define_attribute {*object***}** *attributeName* **{***value***}**

define_global_attribute *attributeName* **{***value***}**

object	The design object, such as module, signal, input, instance, port, or wire name. The object naming syntax varies, depending on whether your source code is in Verilog or VHDL format. See syn_black_box, on page 37 for details about the syntax conventions. If you have mixed input files, use the object naming syntax appropriate for the format in which the object is defined. Global attributes, since they apply to an entire design, do not use an *object* argument.
attributeName	The name of the synthesis attribute. This must be an attribute, not a directive, as directives are not supported in constraint files.
value	String, integer, or boolean value.

See Summary of Global Attributes, on page 11 for more details on specifying global attributes in the synthesis environment.

Summary of Attributes and Directives

The following section summarizes the synthesis attributes and directives:

- Attribute and Directive Summary (Alphabetical), on page 7

For detailed descriptions of individual attributes and directives, see the individual attributes and directives, which are listed in alphabetical order.

Attribute and Directive Summary (Alphabetical)

The following table summarizes the synthesis attributes and directives. For detailed descriptions of each one, you can find them listed in alphabetical order.

Attribute/Directive	Description	Default
alsloc	Preserves relative placement in Microsemi designs.	
alspin	Assigns scalar or bus ports to I/O pin numbers in Microsemi designs.	
alspreserve	Specifies nets that must be preserved by the Microsemi place-and-route tool.	
black_box_pad_pin	Specifies that a pin on a black box is an I/O pad. It is applied to a component, architecture, or module, with a value that specifies the set of pins on the module or entity.	
black_box_tri_pins	Specifies that a pin on a black box is a tristate pin. It is applied to a component, architecture, or module, with a value that specifies the set of pins on the module or entity.	
full_case	Specifies that a Verilog case statement has covered all possible cases.	

Attribute/Directive	Description	Default
loop_limit	Specifies a loop iteration limit for for loops.	
parallel_case	Specifies a parallel multiplexed structure in a Verilog case statement, rather than a priority-encoded structure.	
pragma translate_off/pragma translate_on	Specifies sections of code to exclude from synthesis, such as simulation-specific code.	
syn_allow_retiming	Determines whether registers may be moved across combinational logic to improve performance in devices that support retiming.	
syn_black_box	Defines a black box for synthesis.	
syn_encoding	Specifies the encoding style for state machines.	Based on number of states.
syn_enum_encoding	Specifies the encoding style for enumerated types (VHDL only).	
syn_global_buffers	Determines the number of global buffers available.	
syn_hier	Determines hierarchical control across module or component boundaries.	soft
syn_insert_buffer	Inserts a clock buffer according to the specified value.	
syn_isclock	Specifies that a black-box input port is a clock, even if the name does not indicate it is one.	
syn_keep	Prevents the internal signal from being removed during synthesis.	
syn_loc	Specifies pin locations for I/O pins and cores, and forward-annotates this information to the place-and-route tool.	

Attribute/Directive	Description	Default
syn_looplimit	Specifies a loop iteration limit for while loops.	
syn_maxfan	Overrides the default fanout guide for an individual input port, net, or register output.	
syn_multstyle	Determines how multipliers are implemented.	block_mult
syn_netlist_hierarchy	Controls hierarchy generation in EDIF output files	1
syn_noarrayports	Specifies ports as individual signals or bus arrays.	1
syn_noclockbuf	Disables automatic clock buffer insertion.	0
syn_noprune	Controls the automatic removal of instances that have outputs that are not driven.	
syn_pad_type	Specifies an I/O buffer standard for certain technology families.	
syn_preserve	Preserves registers that can be optimized due to redundancy or constraint propagation.	
syn_probe	Adds probe points for testing and debugging.	
syn_radhardlevel	Specifies the radiation-resistant design technique to use.	
syn_ramstyle	Determines how RAMs are implemented.	registers
syn_reference_clock	Specifies a clock frequency other than that implied by the signal on the clock pin of the register.	
syn_replicate	Controls replication, either globally or on registers.	0
syn_resources	Specifies resources used in black boxes.	

Attribute/Directive	Description	Default
syn_sharing	Enables/disables resource sharing of operators inside a module.	
syn_state_machine	Determines if the FSM Compiler extracts a structure as a state machine.	
syn_tco<n>	Defines timing clock to output delay through a black box. The *n* indicates a value between 1 and 10.	
syn_tpd<n>	Specifies timing propagation for combinational delay through a black box. The *n* indicates a value between 1 and 10.	
syn_tristate	Specifies that a black-box pin is a tristate pin.	
syn_tsu<n>	Specifies the timing setup delay for input pins, relative to the clock. The *n* indicates a value between 1 and 10.	
translate_off/translate_on	Generates clock enable pins for registers.	1
translate_off/translate_on	Specifies sections of code to exclude from synthesis, such as simulation-specific code.	

Summary of Global Attributes

Design attributes in the synthesis environment can be defined either globally, (values are applied to all objects of the specified type in the design), or locally, values are applied only to the specified design object (module, view, port, instance, clock, and so on). When an attribute is set both globally and locally on a design object, the local specification overrides the global specification for the object.

In general, the syntax for specifying a global attribute in a constraint file is:

define_global_attribute *attribute_name* **{***value***}**

The table below contains a list of attributes that can be specified globally in the synthesis environment.

For complete descriptions of any of the attributes listed below, see Summary of Attributes and Directives, on page 7.

Global Attribute	Can Also Be Set On Design Objects
syn_allow_retiming	x
syn_hier	x
syn_multstyle	x
syn_netlist_hierarchy	
syn_noarrayports	
syn_noclockbuf	x
syn_ramstyle	x
syn_replicate	x

All attributes and directives supported for synthesis are listed in alphabetical order. Each command includes syntax, option and argument descriptions, and examples. You can apply attributes and directives globally or locally on a design object.

•

alsloc

Attribute; Microsemi . Preserves relative placements of macros and IP blocks in the Microsemi Designer place-and-route tool. The alsloc attribute has no effect on synthesis, but is passed directly to Microsemi Designer.

Constraint File Syntax and Example

define_attribute {*object***} alsloc {***location***}**

In the attribute syntax, *object* is the name of a macro or IP block and *location* is the row-column address of the macro or IP block.

Following is an example of setting alsloc on a macro (u1).

```
define_attribute {u1} alsloc {R15C6}
```

Verilog Syntax and Example

object **/* synthesis alsloc = "***location***" */ ;**

Where *object* is a macro or IP block and *location* is the row-column string. For example:

```
module test(in1, in2, in3, clk, q);
input in1, in2, in3, clk;
output q;
wire out1 /* synthesis syn_keep = 1 */, out2;
and2a u1 (.A (in1), .B (in2), .Y (out1))
            /* synthesis alsloc="R15C6" */;
assign out2 = out1 & in3;
df1 u2 (.D (out2), .CLK (clk), .Q (q))
            /* synthesis alsloc="R35C6" */;
endmodule

module and2a(A, B, Y); // synthesis syn_black_box
input A, B;
output Y;
endmodule

module df1(D, CLK, Q); // synthesis syn_black_box
input D, CLK;
output Q;
endmodule
```

VHDL Syntax and Example

attribute alsloc of *object* **: label is "***location***" ;**

Where *object* is a macro or IP block and *location* is the row-column string. See VHDL Attribute and Directive Syntax, on page 554 for different ways to specify VHDL attributes and directives.

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

entity test is
port (in1, in2, in3, clk : in std_logic;
      q : out std_logic);
end test;

architecture rtl of test is
signal out1, out2 : std_logic;

component AND2A
port (A, B : in std_logic;
      Y : out std_logic);
end component;

component df1
port (D, CLK : in std_logic;
      Q : out std_logic);
end component;

attribute syn_keep : boolean;
attribute syn_keep of out1 : signal is true;
attribute alsloc: string;
attribute alsloc of U1: label is "R15C6";
attribute alsloc of U2: label is "R35C6";
attribute syn_black_box : boolean;
attribute syn_black_box of AND2A, df1 : component is true;
begin
U1: AND2A port map (A => in1, B => in2, Y => out1);
out2 <= in3 and out1;
U2: df1 port map (D => out2, CLK => clk, Q => q);
end rtl;
```

alspin

Attribute; Microsemi. The alspin attribute assigns the scalar or bus ports of the design to Microsemi I/O pin numbers (pad locations). Refer to the Microsemi databook for valid pin numbers. If you want to use alspin for bus ports or for slices of bus ports, you must also use the syn_noarrayports attribute. See *Specifying Locations for Microsemi Bus Ports, on page 487* of the *User Guide* for information on assigning pin numbers to buses and slices.

Constraint File Syntax and Example

define_attribute {*port_name***} alspin {***pin_number***}**

In the attribute syntax, *port_name* is the name of the port and *pin_number* is the Microsemi I/O pin.

```
define_attribute {DATAOUT} alspin {48}
```

Verilog Syntax and Example

object **/* synthesis alspin = "***pin_number***" */ ;**

Where *object* is the port and *pin_number* is the Microsemi I/O pin. For example:

```
module comparator (datain, clk, dataout);
output dataout /* synthesis alspin="48" */;
input [7:0] datain;
input clk;

// Other code
```

VHDL Syntax and Example

See VHDL Attribute and Directive Syntax, on page 554 for different ways to specify VHDL attributes and directives.

attribute alspin of *object* **:** *objectType* **is "***pin_number***" ;**

Where *object* is the port, *objectType* is signal, and *pin_number* is the Microsemi I/O pin. For example:

```
entity comparator is
   port (datain : in bit_vector(7 downto 0);
      clk : in bit;
      dataout : out bit);
attribute alspin : string;
attribute alspin of dataout : signal is "48";

-- Other code
```

alspreserve

Attribute; Microsemi . Specifies a net that you do not want removed (optimized away) by the Microsemi Designer place-and-route tool. The alspreserve attribute has no effect on synthesis, but is passed directly to the Microsemi Designer place-and-route software. However, to prevent the net from being removed during the synthesis process, you must also use the syn_keep directive.

Constraint File Syntax and Example

define_attribute {n:*net_name***} alspreserve {1}**

In the attribute syntax, *net_name* is the name of the net to preserve.

```
define_attribute {n:and_out3} alspreserve {1};
define_attribute {n:or_out1} alspreserve {1};
```

Verilog Syntax and Example

object **/* synthesis alspreserve = 1 */ ;**

Where *object* is the name of the net to preserve. For example:

```
module complex (in1, out1);
input [6:1] in1;oh
output out1;
wire out1;
wire or_oosut1 /* synthesis syn_keep=1 alspreserve=1 */;
wire and_out1;
wire and_out2;
wire and_out3 /* synthesis syn_keep=1 alspreserve=1 */;
assign and_out1 = in1[1] & in1[2];
assign and_out2 = in1[3] & in1[4];
assign and_out3 = in1[5] & in1[6];
assign or_out1 = and_out1 | and_out2;
assign out1 = or_out1 & and_out3;
endmodule
```

VHDL Syntax and Example

See VHDL Attribute and Directive Syntax, on page 554 for different ways to specify VHDL attributes and directives.

attribute alspreserve of *object* **: signal is true ;**

Where *object* is the name of the net to preserve.

For example:

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

entity complex is
port (input : in std_logic_vector (6 downto 1);
      output : out std_logic);
end complex;

architecture RTL of complex is
signal and_out1 : std_logic;
signal and_out2 : std_logic;
signal and_out3 : std_logic;
signal or_out1 : std_logic;
attribute syn_keep of and_out3 : signal is true;
attribute syn_keep of or_out1 : signal is true;
attribute alspreserve of and_out3 : signal is true;
attribute alspreserve of or_out1 : signal is true;

begin
   and_out1 <= input(1) and input(2);
   and_out2 <= input(3) and input(4);
   and_out3 <= input(5) and input(6);
   or_out1 <= and_out1 or and_out2;
   output <= or_out1 and and_out3;
end;
```

black_box_pad_pin

Directive

Used with the syn_black_box directive and specifies that the pins on black box are I/O pads visible to the outside environment. To specify more than one port as an I/O pad, list the ports inside double-quotes ("), separated by commas, and without enclosed spaces.

To instantiate an I/O from your programmable logic vendor, you usually do not need to define a black box or this directive. The synthesis tool provides predefined black boxes for vendor I/Os. For more information, refer to your vendor section under FPGA and CPLD Support.

The black_box_pad_pin directive is one of several directives that you can use with the syn_black_box directive to define timing for a black box. See syn_black_box, on page 37 for a list of the associated directives.

Verilog Syntax and Example

object **/* synthesis syn_black_box black_box_pad_pin = "***portList***" */ ;**

where *portList* is a spaceless, comma-separated list of the names of the ports on black boxes that are I/O pads. For example:

```
module BBDLHS(D,E,GIN,GOUT,PAD,Q)
   /* synthesis syn_black_box black_box_pad_pin="GIN[2:0],Q" */;
```

VHDL Syntax and Example

attribute black_box_pad_pin of *object* **:** *objectType* **is "***portList***" ;**

where *object* is an architecture or component declaration of a black box. Data type is string; *portList* is a spaceless, comma-separated list of the black-box port names that are I/O pads.

See VHDL Attribute and Directive Syntax, on page 554 for different ways to specify VHDL attributes and directives.

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

Entity top is
generic ( width : integer := 4);
   port (in1,in2 : in std_logic_vector(width downto 0);
   clk : in std_logic;
   q : out std_logic_vector (width downto 0)
   );
end top;

architecture top1_arch of top is
component test is
   generic (width1 : integer := 2);
      port (in1,in2 : in std_logic_vector(width1 downto 0);
      clk : in std_logic;
      q : out std_logic_vector (width1 downto 0)
   );
end component;

attribute syn_black_box : boolean;
attribute black_box_pad_pin : string;
attribute syn_black_box of test : component is true;
attribute black_box_pad_pin of test : component is "in1(4:0),
in2[4:0], q(4:0)";

begin
   test123 : test generic map (width) port map (in1,in2,clk,q);
end top1_arch;
```

black_box_tri_pins

Directive. Used with the syn_black_box directive and specifies that an output port on a black box component is a tristate. This directive eliminates multiple driver errors when the output of a black box has more than one driver. To specify more than one tristate port, list the ports inside double-quotes ("), separated by commas (,), and without enclosed spaces.

The black_box_tri_pins directive is one of several directives that you can use with the syn_black_box directive to define timing for a black box. See syn_black_box, on page 37 for a list of the associated directives.

Verilog Syntax and Examples

object **/* synthesis syn_black_box black_box_tri_pins = "***portList***" */ ;**

where *portList* is a spaceless, comma-separated list of multiple pins.

Here is an example with a single port name:

```
module BBDLHS(D,E,GIN,GOUT,PAD,Q)
   /* synthesis syn_black_box black_box_tri_pins="PAD" */;
```

Here is an example with a list of multiple pins:

```
module bb1(D,E,tri1,tri2,tri3,Q)
/* synthesis syn_black_box black_box_tri_pins="tri1,tri2,tri3" */;
```

For a bus, you specify the port name followed by all the bits on the bus:

```
module bb1(D,bus1,E,GIN,GOUT,Q)
   /* synthesis syn_black_box black_box_tri_pins="bus1[7:0]" */;
```

VHDL Syntax and Examples

attribute black_box_tri_pins of *object* **:** *objectType* **is "***portList***" ;**

where *object* is a component declaration or architecture. Data type is string, and *portList* is a spaceless, comma-separated list of the tristate output port names.

See VHDL Attribute and Directive Syntax, on page 554 for different ways to specify VHDL attributes and directives.

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

package my_components is
component BBDLHS
   port (D: in std_logic;
         E: in std_logic;
         GIN : in std_logic;
         GOUT : in std_logic;
         PAD : inout std_logic;
         Q: out std_logic );
end component;

attribute syn_black_box : boolean;
attribute syn_black_box of BBDLHS : component is true;
attribute black_box_tri_pins : string;
attribute black_box_tri_pins of BBDLHS : component is "PAD";
end package my_components;
```

Multiple pins on the same component can be specified as a list:

```
attribute black_box_tri_pins of bb1 : component is
   "tri,tri2,tri3";
```

To apply this directive to a port that is a bus, specify all the bits on the bus:

```
attribute black_box_tri_pins of bb1 : component is "bus1[7:0]";
```

full_case

Directive. For Verilog designs only. When used with a case, casex, or casez statement, this directive indicates that all possible values have been given, and that no additional hardware is needed to preserve signal values.

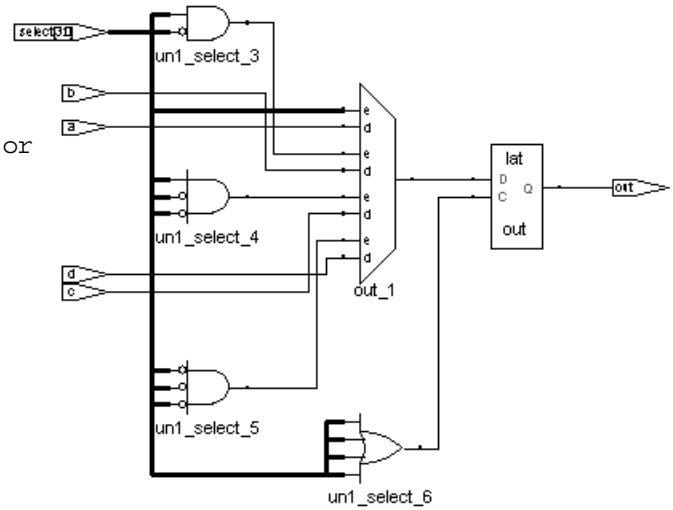
Verilog Syntax and Example

object **/* synthesis full_case */**

where *object* is case, casex, or casez statement declarations.

The following casez statement creates a 4-input multiplexer with a pre-decoded select bus (a decoded select bus has exactly one bit enabled at a time):

```
module muxnew1 (out, a, b, c, d, select);
output out;
input a, b, c, d;
input [3:0] select;
reg out;

always @(select or a or b or
c or d)

begin
   casez (select)
      4'b???1: out = a;
      4'b??1?: out = b;
      4'b?1??: out = c;
      4'b1???: out = d;
   endcase
end
endmodule
```



This code does not specify what to do if the select bus has all zeros. If the select bus is being driven from outside the current module, the current module has no information about the legal values of select, and the synthesis tool must preserve the value of the output out when all bits of select are zero. Preserving the value of out requires the tool to add extraneous level-sensitive latches if out is not assigned elsewhere through every path of the always block. A warning message like the following is issued:

```
"Latch generated from always block for signal out, probably missing
assignment in branch of if or case."
```

If you add the full_case directive, it instructs the synthesis tool not to preserve the value of out when all bits of select are zero.

```
module muxnew3 (out, a, b, c, d, select);
output out;
input a, b, c, d;
input [3:0] select;
reg out;

always @(select or a or b or c or d)

begin
   casez (select) /* synthesis full_case */
      4'b???1: out = a;
      4'b??1?: out = b;
      4'b?1??: out = c;
      4'b1???: out = d;
   endcase
end
endmodule
```

If the select bus is decoded in the same module as the case statement, the synthesis tool automatically determines that all possible values are specified, so the full_case directive is unnecessary.

Assigned Default and full_case

As an alternative to full_case, you can assign a default in the case statement. The default is assigned a value of 'bx (a 'bx in an assignment is treated as a “don't care”). The software assigns the default at each pass through the casez statement in which the select bus does not match one of the explicitly given values; this ensures that the value of out is not preserved and no extraneous level-sensitive latches are generated.

The following code shows a default assignment in Verilog:

```
module muxnew2 (out, a, b, c, d, select);
output out;
input a, b, c, d;
input [3:0] select;
reg out;

always @(select or a or b or c or d)
begin
   casez (select)
      4'b???1: out = a;
      4'b??1?: out = b;
      4'b?1??: out = c;
      4'b1???: out = d;
      default: out = 'bx;
   endcase
end
endmodule
```

Both techniques help keep the code concise because you do not need to declare all the conditions of the statement. The following table compares them:

Default Assignment	full_case
Stays within Verilog to get the desired hardware	Must use a synthesis directive to get the desired hardware
Helps simulation debugging because you can easily find that the invalid select is assigned a 'bx	Can cause mismatches between pre- and post-synthesis simulation because the simulator does not use full_case

loop_limit

Directive

For Verilog designs only. Specifies a loop iteration limit for for loops in the design when the loop index is a variable, not a constant. The compiler uses the default iteration limit of 1999 when the exit or terminating condition does not compute a constant value, or to avoid infinite loops. The default limit ensures the effective use of runtime and memory resources.

If your design requires a variable loop index or if the number of loops is greater than the default limit, use the loop_limit directive to specify a new limit for the compiler. If you do not, you get a compiler error. You must hard code the limit at the beginning of the loop statement. The limit cannot be an expression. The higher the value you set, the longer the runtime. To override the default limit of 2000 in the RTL, use the Loop Limit option on the Verilog tab of the Implementation Options panel. See Verilog Panel, on page 195 in the *Command Reference*.

Note: VHDL applications use the syn_looplimit directive (see syn_looplimit, on page 88).

Verilog Syntax and Example

beginning_of_loop_statement **/* synthesis loop_limit** *integer* ***/**

The following is an example where the loop limit is set to 2000:

```
module test(din,dout,clk);
input[1999 : 0] din;
input clk;
output[1999 : 0] dout;
reg[1999 : 0] dout;
integer i;
```

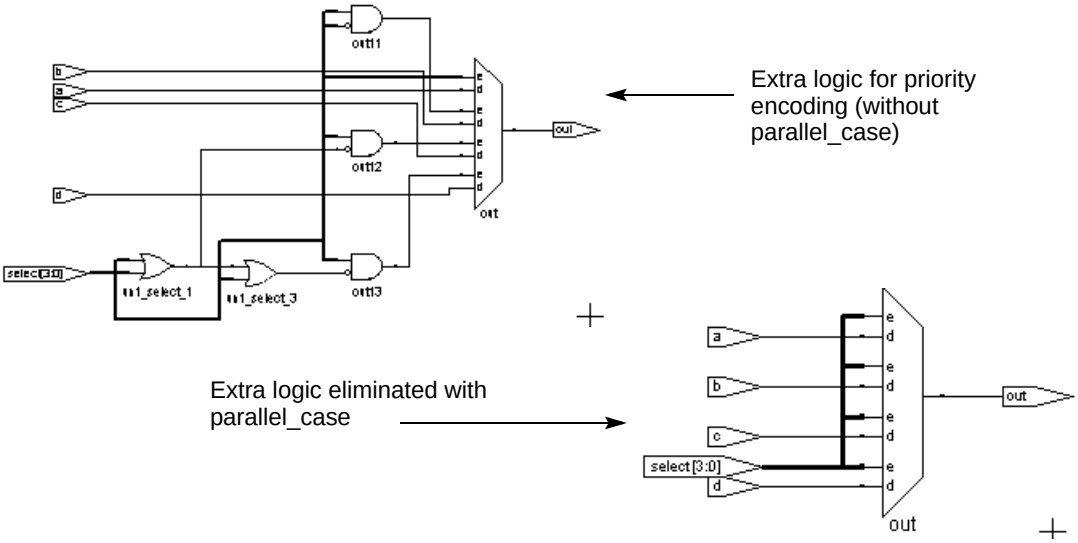
```
always @(posedge clk)
begin
   /* synthesis loop_limit 2000 */
   for(i=0;i<=1999;i=i+1)
   begin
      dout[i] <= din[i];
   end
end

endmodule
```

parallel_case

Directive

For Verilog designs only. Forces a parallel-multiplexed structure rather than a priority-encoded structure. This is useful because case statements are defined to work in priority order, executing (only) the first statement with a tag that matches the select value.

If the select bus is driven from outside the current module, the current module has no information about the legal values of select, and the software must create a chain of disabling logic so that a match on a statement tag disables all following statements. However, if you know the legal values of select, you can eliminate extra priority-encoding logic with the parallel_case directive. In the following example, the only legal values of select are 4'b1000, 4'b0100, 4'b0010, and 4'b0001, and only one of the tags can be matched at a time. Specify the parallel_case directive so that tag-matching logic can be parallel and independent, instead of chained.



Verilog Syntax and Example

You specify the directive as a comment immediately following the select value of the case statement.

object **/* synthesis parallel_case */**

where *object* is a case, casex or casez statement declaration.

```
module muxnew4 (out, a, b, c, d, select);
output out;
input a, b, c, d;
input [3:0] select;
reg out;

always @(select or a or b or c or d)

begin
   casez (select) /* synthesis parallel_case */
      4'b???1: out = a;
      4'b??1?: out = b;
      4'b?1??: out = c;
      4'b1???: out = d;
      default: out = 'bx;
   endcase
end
endmodule
```

If the select bus is decoded within the same module as the case statement, the parallelism of the tag matching is determined automatically, and the parallel_case directive is unnecessary.

pragma translate_off/pragma translate_on

Directive

Allows you to synthesize designs originally written for use with other synthesis tools without needing to modify source code. All source code that is between these two directives is ignored during synthesis.

Another use of these directives is to prevent the synthesis of stimulus source code that only has meaning for logic simulation. You can use pragma translate_off/translate_on to skip over simulation-specific lines of code that are not synthesizable.

When you use pragma translate_off in a module, synthesis of all source code that follows is halted until pragma translate_on is encountered. Every pragma translate_off must have a corresponding pragma translate_on. These directives cannot be nested, therefore, the pragma translate_off directive can only be followed by a pragma translate_on directive.

Note: See also, translate_off/translate_on, on page 180. These directives are implemented the same in the source code.

Verilog Syntax and Example

The Verilog syntax for these directives is as follows:

/* pragma translate_off */

/* pragma translate_on */

For example:

```
module real_time (ina, inb, out);
input ina, inb;
output out;

/* pragma translate_off */
```

```
realtime cur_time;
/* pragma translate_on */

assign out = ina & inb;
endmodule
```

VHDL Syntax and Example

The following is the VHDL syntax for these directives:

pragma translate_off

pragma translate_on

For example:

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

entity adder is
   port (a, b, cin:in std_logic;
         sum, cout:out std_logic );
end adder;

architecture behave of adder is
signal a1:std_logic;

--pragma translate_off

constant a1:std_logic:='0';

--pragma translate_on

begin
   sum <= (a xor b xor cin);
   cout <= (a and b) or (a and cin) or (b and cin); end behave;
```

syn_allow_retiming

Attribute

Determines if registers can be moved across combinational logic to improve performance.Return to Summary of Attributes and Directives.**syn_allow_retiming values**

1 | true	Allows registers to be moved during retiming.
0 | false	Does not allow retimed registers to be moved.

DescriptionThe syn_allow_retiming attribute determines if registers can be moved across combinational logic to improve performance.

The attribute can be applied either globally or to specific registers. Typically, you enable the global Retiming option in the UI (or the set_option -retiming 1 switch in Tcl) and use the syn_allow_retiming attribute to disable retiming for specific objects that you do not want moved.**syn_allow_retiming Syntax**

Global	Object
Yes	Register

You can specify the attribute in the following files: **FDC Example**define_attribute {*register*} syn_allow_retiming {1|0}
define_global_attribute syn_allow_retiming {1|0}

Enable	Object Type	Object	Attribute	Value	Value Type	Description
☑	<any>	<Global>	syn_allow_retiming	1	boolean	Controls retiming of reg...

Verilog Example

object **/* synthesis syn_allow_retiming = 0 | 1 */ ;**

Here is an example of applying it to a register:

```
module parity_check (clk,data,count_one);
input clk;
input [20:0]data ;
output reg [3:0]count_one /* synthesis syn_allow_retiming=1*/;

integer i;
reg parity= 1'b1;

always @(posedge clk)
begin
   for (i=0; i<21; i=i+1)
      if (data[i] == parity)
         count_one<=count_one+1;


end
endmodule
```

VHDL Example

attribute syn_allow_retiming of *object* **:** *objectType* **is true | false ;**

The data type is Boolean. Here is an example of applying it to a register:

```
LIBRARY IEEE;
USE     IEEE.STD_LOGIC_1164.ALL;
USE     IEEE.std_logic_unsigned.ALL;

ENTITY ones_cnt IS
   PORT ( vin  : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
      vout : OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
      clk  : IN STD_LOGIC );
END ones_cnt;

ARCHITECTURE lan OF ones_cnt IS
signal vout_reg : STD_LOGIC_VECTOR (3 DOWNTO 0);
attribute syn_allow_retiming : boolean;
attribute syn_allow_retiming of vout_reg : signal is true;

;

BEGIN
   gen_vout: PROCESS(clk,vin)
      VARIABLE count : STD_LOGIC_VECTOR(vout'RANGE);
   BEGIN
      if rising_edge(clk) then
      count := (OTHERS => '0');
      FOR I IN vin'RANGE LOOP
```

```
            count := count + vin(i);
         END LOOP;
         vout_reg <= count;
      end if;
   vout <= vout_reg;
   END PROCESS gen_vout;
   END lan;
```

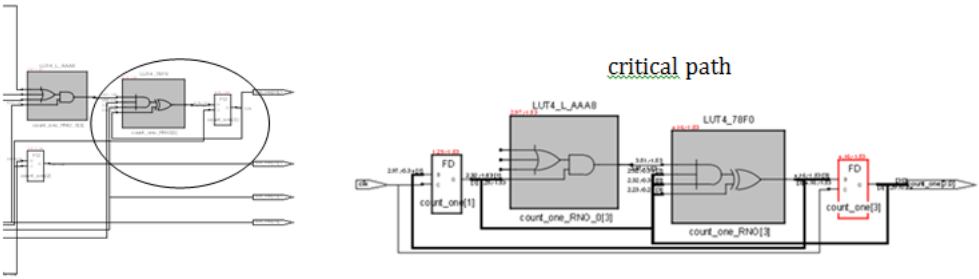
See VHDL Attribute and Directive Syntax, on page 554 for different ways to specify VHDL attributes and directives.

Effect of using syn_allow_retiming

Before applying syn_allow_retiming.

Verilog	output reg [3:0]count_one /* synthesis syn_allow_retiming=0*/;
VHDL	attribute syn_allow_retiming of vout_reg : signal is false;

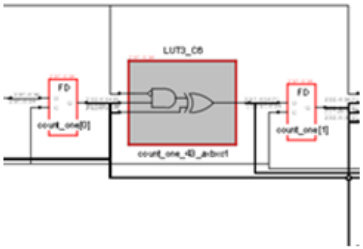
The critical path and the worst slack for this scenario are given below along with the original count_one [3] register (before being retimed) as found in the design.

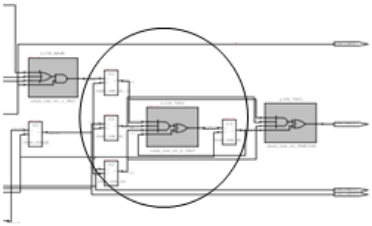


After applying syn_allow_retiming.

Verilog	output reg [3:0]count_one /* synthesis syn_allow_retiming=1*/;
VHDL	attribute syn_allow_retiming of vout_reg : signal is true;

The critical path and the worst slack for this scenario are shown along with the four '*_ret' retimed registers.





syn_black_box

Directive

Defines a module or component as a black box.

syn_black_box Value

Value	Default	Description
moduleName	N/A	Defines an object as a black box.

Description

Specifies that a module or component is a black box for synthesis. A black box module has only its interface defined for synthesis; its contents are not accessible and cannot be optimized during synthesis. A module can be a black box whether or not it is empty.

Typically, you set syn_black_box on objects like the ones listed below. You do not need to define a black box for such an object if the synthesis tool includes a predefined black box for it.

- Vendor primitives and macros (including I/Os).
- User-designed macros whose functionality is defined in a schematic editor, IP, or another input source where the place-and-route tool merges design netlists from different sources.

In certain cases, the tool does not honor a syn_black_box directive:

- In mixed language designs where a black box is defined in one language at the top level but where there is an existing description for it in another language, the tool can replace the -declared black box with the description from the other language.
- If your project includes black box descriptions in `srs`, `ngc`, or `edf` formats, the tool uses these black box descriptions even if you have specified syn_black_box at the top level.

To override this and ensure that the attribute is honored, use these methods:

- Set a syn_black_box directive on the module or entity in the HDL file that contains the description, not at the top level. The contents will be black-boxed.

- If you want to define a black box when you have an `srs`, `ngc`, or `edf` description for it, remove the description from the project.

Once you define a black box with syn_black_box, you use other -source code directives to define timing for the black box. You must add the directives to the source code because the timing models are specific to individual instances. There are no corresponding Tcl directives you can add to a constraint file.

Black-box Source Code Directives

Use the following directives with syn_black_box to characterize black-box timing:

syn_isclock	Specifies a clock port on a black box.
syn_tco<n>	Sets timing propagation for combinational delay through the black box.
syn_tsu<n>	Defines timing setup delay required for input pins relative to the clock.
syn_tco<n>	Defines the timing clock to output delay through the black box.

Black Box Pin Definitions

You define the pins on a black box with these directives in the source code:

black_box_pad_pin	Indicates that a black box is an I/O pad for the rest of the design.
black_box_tri_pins	Indicates tristates on black boxes.

For more information on black boxes, see Instantiating Black Boxes in Verilog, on page 357, and Instantiating Black Boxes in VHDL, on page 552.

syn_black_box Syntax Specification

Verilog	*object* /* synthesis syn_black_box */ ;	Verilog Example
VHDL	attribute syn_black_box of *object* : *objectType* is true ;	VHDL Example

Verilog Example

```
module top(clk, in1, in2, out1, out2);

input clk;
input [1:0]in1;
input [1:0]in2;

output [1:0]out1;
output [1:0]out2;

add           U1 (clk, in1, in2, out1);
black_box_add U2 (in1, in2, out2);

endmodule

module add (clk, in1, in2, out1);

input clk;
input  [1:0]in1;
input  [1:0]in2;

output [1:0]out1;
reg [1:0]out1;

always@(posedge clk)
   begin
      out1 <= in1 + in2;
   end
endmodule

module black_box_add(A, B, C)/* synthesis syn_black_box */;

input [1:0]A;
input [1:0]B;

output [1:0]C;

assign C = A + B;

endmodule
```

VHDL Example

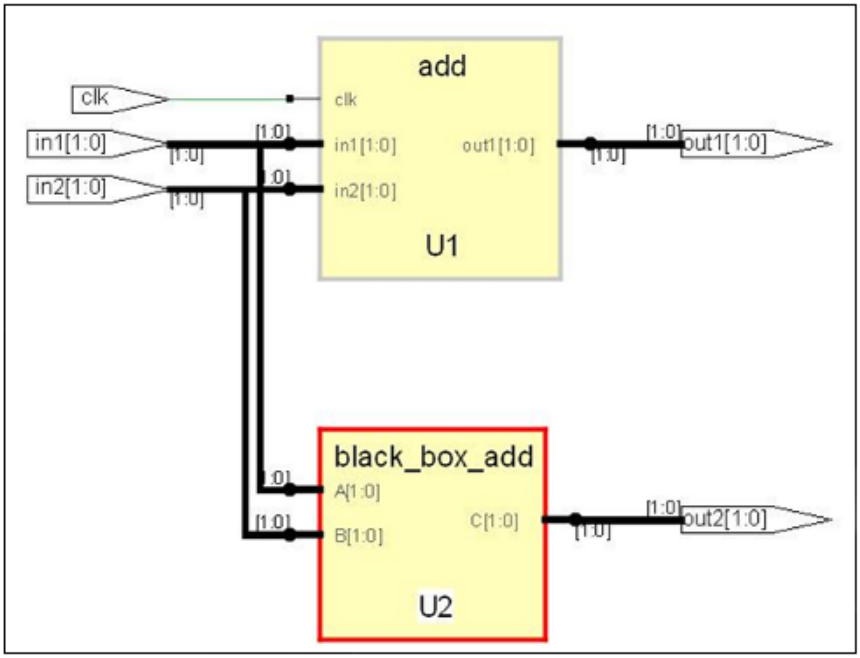
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity add is
   port(
      in1 : in std_logic_vector(1 downto 0);
      in2 : in std_logic_vector(1 downto 0);
      clk : in std_logic;
      out1 : out std_logic_vector(1 downto 0));
end;

architecture rtl of add is
begin

process(clk)
begin
   if(clk'event and clk='1') then
      out1 <= (in1 + in2);
   end if;
end process;
end;

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

entity black_box_add is
   port(
      A : in std_logic_vector(1 downto 0);
      B : in std_logic_vector(1 downto 0);
      C : out std_logic_vector(1 downto 0));
end;

architecture rtl of black_box_add is

attribute syn_black_box : boolean;
attribute syn_black_box of rtl: architecture is true;
begin

C <= A + B;
end;

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

```
entity top is
   port(
      in1 : in std_logic_vector(1 downto 0);
      in2 : in std_logic_vector(1 downto 0);
      clk : in std_logic;
      out1 : out std_logic_vector(1 downto 0);
      out2 : out std_logic_vector(1 downto 0));
end;

architecture rtl of top is

component add is
   port(
      in1 : in std_logic_vector(1 downto 0);
      in2 : in std_logic_vector(1 downto 0);
      clk : in std_logic;
      out1 : out std_logic_vector(1 downto 0));
end component;

component black_box_add
   port(
      A : in std_logic_vector(1 downto 0);
      B : in std_logic_vector(1 downto 0);
      C : out std_logic_vector(1 downto 0));
end component;

begin
U1: add port map(in1, in2, clk, out1);
U2: black_box_add port map(in1, in2, out2);
end;
```

Effect of Using syn_black_box

When the syn_black_box attribute is not set on the black_box_add module, its content are accessible, as shown in the example below:

```
module black_box_add( input [1:0]A, [1:0]B, output [1:0]C);
```



After applying syn_black_box:, the contents of the black box are no longer visible:

```
module black_box_add( input [1:0]A, [1:0]B, output [1:0]C)/* synthesis
syn_black_box */;
```

syn_encoding

Attribute

Overrides the default FSM Compiler encoding for a state machine and applies the specified encoding.

Vendor	Devices
Microsemi	ProASIC3, Fusion, SmartFusion2, IGLOO2, older devices

syn_encoding Values

The default is that the tool automatically picks an encoding style that results in the best performance. To ensure that a particular encoding style is used, explicitly specify that style, using the values below:

Value	Description
onehot	Only two bits of the state register change (one goes to 0, one goes to 1) and only one of the state registers is hot (driven by 1) at a time. For example: 0001, 0010, 0100, 1000 Because onehot is not a simple encoding (more than one bit can be set), the value must be decoded to determine the state. This encoding style can be slower than a gray style if you have a large output decoder following a state machine.
gray	More than one of the state registers can be hot. The synthesis tool *attempts* to have only one bit of the state registers change at a time, but it can allow more than one bit to change, depending upon certain conditions for optimization. For example: 000, 001, 011, 010, 110 Because gray is not a simple encoding (more than one bit can be set), the value must be decoded to determine the state. This encoding style can be faster than a onehot style if you have a large output decoder following a state machine.

Value	Description
sequential	More than one bit of the state register can be hot. The synthesis tool makes no attempt at limiting the number of bits that can change at a time. For example: 000, 001, 010, 011, 100 This is one of the smallest encoding styles, so it is often used when area is a concern. Because more than one bit can be set (1), the value must be decoded to determine the state. This encoding style can be faster than a onehot style if you have a large output decoder following a state machine.
safe	This implements the state machine in the default encoding and adds reset logic to force the state machine to a known state if it reaches an invalid state. This value can be used in combination with any of the other encoding styles described above. You specify safe before the encoding style. The safe value is only valid for a state register, in conjunction with an encoding style specification. For example, if the default encoding is onehot and the state machine reaches a state where all the bits are 0, which is an invalid state, the safe value ensures that the state machine is reset to a valid state. If recovery from an invalid state is a concern, it may be appropriate to use this encoding style, in conjunction with onehot, sequential or gray, in order to force the state machine to reset. When you specify safe, the state machine can be reset from an unknown state to its reset state.
original	This respects the encoding you set, but the software still does state machine and reachability analysis.

You can specify multiple values. This snippet uses safe, gray. The encoding style for register OUT is set to gray, but if the state machine reaches an invalid state the synthesis tool will reset the values to a valid state.

```
module prep3 (CLK, RST, IN, OUT);
input CLK, RST;
input [7:0] IN;
output [7:0] OUT;
reg [7:0] OUT;
reg [7:0] current_state /* synthesis syn_encoding="safe,gray" */;

// Other code
```

Description

This attribute takes effect only when FSM Compiler is enabled. It overrides the default FSM Compiler encoding for a state machine. For the specified encoding to take effect, the design must contain state machines that have been inferred by the FSM Compiler. Setting this attribute when syn_state_machine is set to 0 will not have any effect.

The default encoding style automatically assigns encoding based on the number of states in the state machine. Use the syn_encoding attribute when you want to override these defaults. You can also use syn_encoding when you want to disable the FSM Compiler globally but there are a select number of state registers in your design that you want extracted. In this case, use this attribute with the syn_state_machine directive on for just those specific registers.

The encoding specified by this attribute applies to the final mapped netlist. For other kinds of enumerated encoding, use syn_enum_encoding. See syn_enum_encoding, on page 52 and syn_encoding Compared to syn_enum_encoding, on page 56 for more information.

Encoding Style Implementation

The encoding style is implemented during the mapping phase. A message appears when the synthesis tool extracts a state machine, for example:

```
@N: CL201 : "c:\design\..."|Trying to extract state machine for
register current_state
```

The log file reports the encoding styles used for the state machines in your design. This information is also available in the FSM Viewer (see FSM Viewer Window, on page 67).

See also the following:

- For information on enabling state machine optimization for individual modules, see syn_state_machine, on page 164.
- For VHDL designs, see syn_encoding Compared to syn_enum_encoding, on page 56 for comparative usage information.

Syntax Specification

Global	Object
No	Instance, register

This table shows how to specify the attribute in different files:

FDC	define_attribute {*object*} syn_encoding {*value*}	SCOPE Example
Verilog	*Object* /* synthesis syn_encoding = "*value*" */;	Verilog Example
VHDL	attribute syn_encoding of *object*: objectType is "*value*";	VHDL Example

If you specify the syn_encoding attribute in Verilog or VHDL, all instances of that FSM use the same syn_encoding value. To have unique syn_encoding values for each FSM instance, use different entities or modules, or specify the syn_encoding attribute in a constraint file.

SCOPE Example

	Enabled	Object Type	Object	Attribute	Value	Val Type	Description
1	☑	fsm	i:state[3:0]	syn_encoding	gray	string	FSM encoding (onehot, sequential, gray, original, safe)

The *object* must be an instance prefixed with **i:**, as in **i:***instance*. The instance must be a sequential instance with a view name of statemachine.

Although you cannot set this attribute globally, you can define a SCOPE collection and then apply the attribute to the collection. For example:

```
define_scope_collection sm {find -hier -inst * -filter
   @inst_of==statemachine}
define_attribute {$sm} {syn_encoding} {safe}
```

Verilog Example

The object can be a register definition signals that hold the state values of state machines.

```
module fsm (clk, reset, x1, outp);
input        clk, reset, x1;
output       outp;
reg          outp;
reg    [1:0] state /* synthesis syn_encoding = "onehot" */;
parameter s1 = 2'b00; parameter s2 = 2'b01;
parameter s3 = 2'b10; parameter s4 = 2'b11;

always @(posedge clk or posedge reset)
begin
   if (reset)
      state <= s1;
   else begin
      case (state)
      s1: if (x1 == 1'b1)
         state <= s2;
      else
         state <= s3; s2: state <= s4;
   s3: state <= s4;
   s4: state <= s1;
   endcase
end
end

always @(state) begin
   case (state)
      s1: outp = 1'b1;
      s2: outp = 1'b1;
      s3: outp = 1'b0;
      s4: outp = 1'b0;
   endcase
   end
endmodule
```

VHDL Example

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity fsm is
   port ( x1  : in std_logic;
   reset : in std_logic;
   clk : in std_logic;
   outp : out std_logic);
end fsm;
```

```
architecture rtl of fsm is
signal state : std_logic_vector(1 downto 0);
constant s1 : std_logic_vector := "00";
constant s2 : std_logic_vector := "01";
constant s3 : std_logic_vector := "10";
constant s4 : std_logic_vector := "11";
attribute syn_encoding : string;
attribute syn_encoding of state : signal is "onehot";

begin
process (clk,reset)
   begin
   if (clk'event and clk = '1') then
      if (reset = '1') then
         state <= s1 ;
      else
         case state is
            when s1 =>
            if x1 = '1' then
               state <= s2;
            else
               state <= s3;
            end if;
            when s2 =>
               state <= s4;
            when s3 =>
               state <= s4;
            when s4 =>
               state <= s1;
         end case;
      end if;
   end if;
end process;

process (state)
begin
   case state is
      when s1 =>
         outp <= '1';
      when s2 =>
         outp <= '1';
      when s3 =>
         outp <= '0';
```

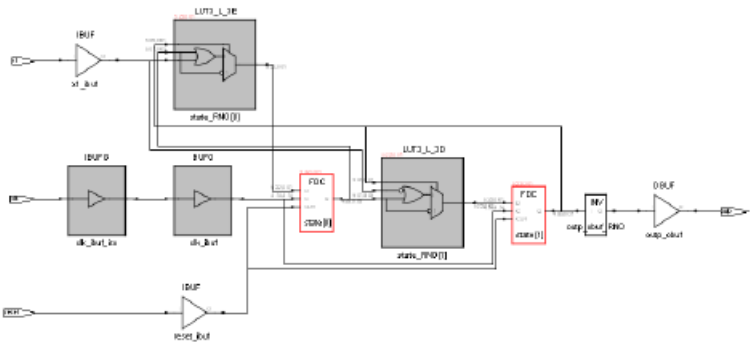
```
            when s4 =>
               outp <= '0';
            end case;
      end process;
      end rtl;
```

See VHDL Attribute and Directive Syntax, on page 554 for different ways to specify VHDL attributes and directives.

Effect of Using syn_encoding

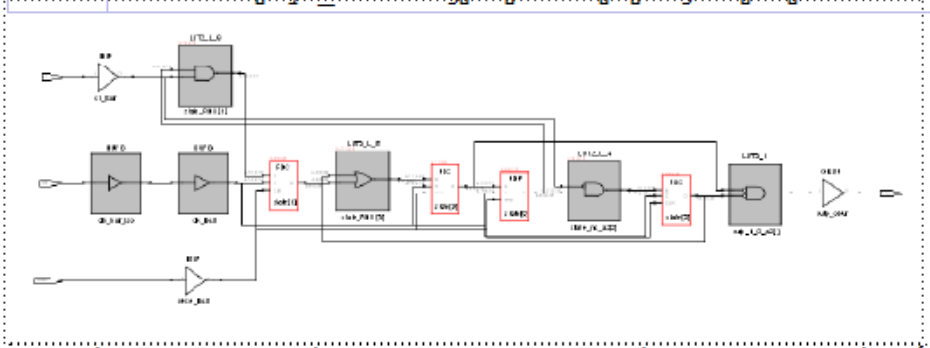
The following figure shows the default implementation of a state machine, with these encoding details reported:

```
Encoding state machine state [3:0] (netlist: statemachine)
original code -> new code
00 -> 00
01 -> 01
10 -> 10
11 -> 11
```



The next figure shows the state machine when the syn_encoding attribute is set to onehot, and the accompanying changes in the code:

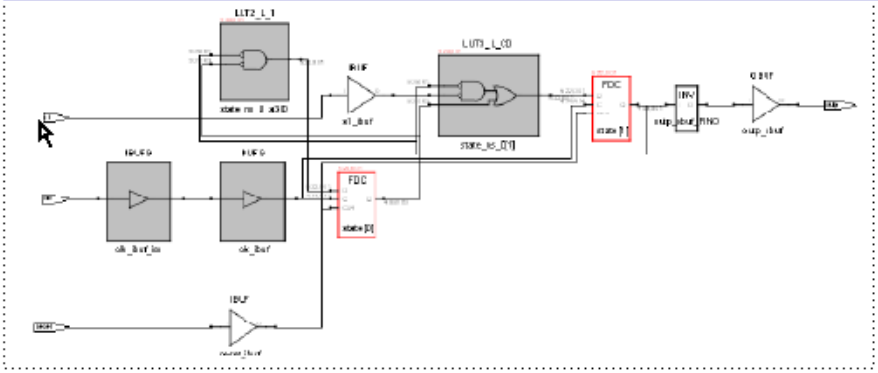
Verilog	reg [1:0] state /* synthesis syn_encoding = "onehot"*/;
VHDL	attribute syn_encoding of state : signal is "onehot";



```
Encoding state machine state [3:0] (netlist: statemachine)
original code -> new code
   00 -> 0001
   01 -> 0010
   10 -> 0100
   11 -> 1000
```

The next figure shows the state machine when the syn_encoding attribute is set to gray:

Verilog	`reg [1:0] state /* synthesis syn_encoding = "gray"*/;`
VHDL	`attribute syn_encoding of state : signal is "gray";`




```
Encoding state machine state [3:0] (netlist: statemachine)
original code -> new code
   00 -> 00
   00 -> 01
   10 -> 11
   11 -> 10
```

syn_enum_encoding

Directive

For VHDL designs. Defines how enumerated data types are implemented. The type of implementation affects the performance and device utilization.

If FSM Compiler is enabled, this directive has no effect on the encoding styles of extracted state machines; the tool uses the values specified in the syn_encoding attribute instead. However, if you have enumerated data types and you turn off the FSM Compiler so that no state machines are extracted, the syn_enum_encoding style is implemented in the final circuit. See syn_encoding Compared to syn_enum_encoding, on page 56 for more information. For step-by-step details about setting coding styles with this attribute see Defining State Machines in VHDL, on page 304 of the *User Guide*.

Values for syn_enum_encoding

Values for syn_enum_encoding are as follows:

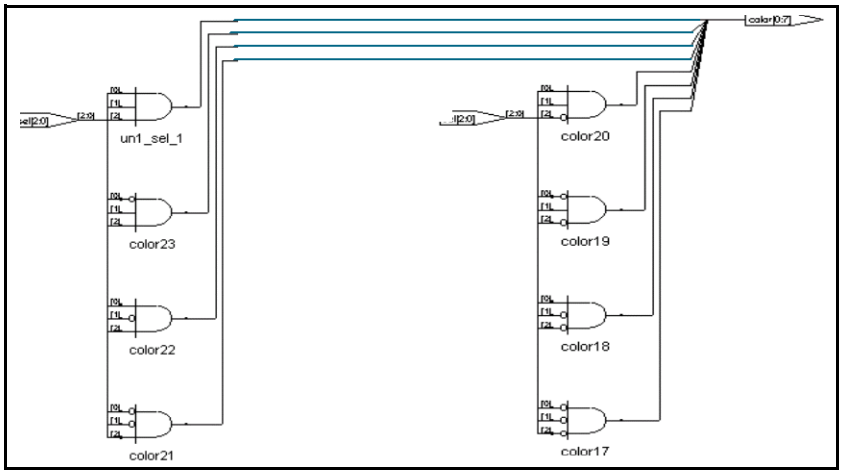
- default – Automatically assigns an encoding style that results in the best performance.
- sequential – More than one bit of the state register can change at a time, but because more than one bit can be hot, the value must be decoded to determine the state. For example: 000, 001, 010, 011, 100
- onehot – Only two bits of the state register change (one goes to 0; one goes to 1) and only one of the state registers is hot (driven by a 1) at a time. For example: 0000, 0001, 0010, 0100, 1000
- gray – Only one bit of the state register changes at a time, but because more than one bit can be hot, the value must be decoded to determine the state. For example: 000, 001, 011, 010, 110
- *string* – This can be any value you define. For example: 001, 010, 101. See Example of syn_enum_encoding for User-Defined Encoding, on page 56.

A message appears in the log file when you use the syn_enum_encoding directive; for example:

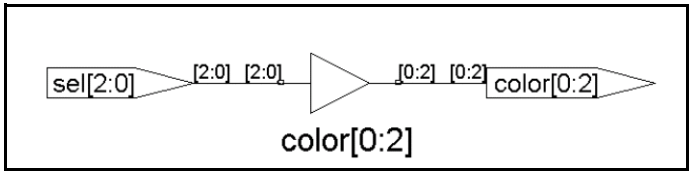
```
CD231: Using onehot encoding for type mytype (red="10000000")
```

Effect of Encoding Styles

The following figure provides an example of two versions of a design: one with the default encoding style, the other with the syn_enum_encoding directive overriding the default enumerated data types that define a set of eight colors.



syn_enum_encoding = “default” Based on 8 states, onehot assigned



syn_enum_encoding = “sequential”

In this example, using the default value for syn_enum_encoding, onehot is assigned because there are eight states in this design. The onehot style implements the output color as 8 bits wide and creates decode logic to convert the input sel to the output. Using sequential for syn_enum_encoding, the logic is reduced to a buffer. The size of output color is 3 bits.

See the following section for the source code used to generate the schematics above.

VHDL Syntax and Examples

attribute syn_enum_encoding of *object* **:** *objectType* **is "***value***" ;**

Where *object* is an enumerated type and *value* is one of the following: default, sequential, onehot or gray. See VHDL Attribute and Directive Syntax, on page 554 for different ways to specify VHDL attributes and directives.

Here is the code used to generate the second schematic in the previous figure. (The first schematic will be generated instead, if "sequential" is replaced by "onehot" as the syn_enum_encoding value.)

```
package testpkg is
type mytype is (red, yellow, blue, green, white,
    violet, indigo, orange);
attribute syn_enum_encoding : string;
attribute syn_enum_encoding of mytype : type is "sequential";
end package testpkg;

library IEEE;
use IEEE.std_logic_1164.all;
use work.testpkg.all;

entity decoder is
   port (sel : in std_logic_vector(2 downto 0);
   color : out mytype );
end decoder;
architecture rtl of decoder is
begin
   process(sel)
   begin
      case sel is
         when "000" => color <= red;
         when "001" => color <= yellow;
         when "010" => color <= blue;
         when "011" => color <= green;
         when "100" => color <= white;
         when "101" => color <= violet;
         when "110" => color <= indigo;
         when others => color <= orange;
      end case;
   end process;
end rtl;
```

syn_enum_encoding, enum_encoding, and syn_encoding

Custom attributes are attributes that are not defined in the IEEE specifications, but which you or a tool vendor define for your own use. They provide a convenient back door in VHDL, and are used to better control the synthesis and simulation process. enum_encoding is one of these custom attributes that is widely used to allow specific binary encodings to be attached to objects of enumerated types.

The enum_encoding attribute is declared as follows:

```
attribute enum_encoding: string;
```

This can be either written directly in your VHDL design description, or provided to you by the tool vendor in a package. Once the attribute has been declared and given a name, it can be referenced as needed in the design description:

```
type statevalue is (INIT, IDLE, READ, WRITE, ERROR);
attribute enum_encoding of statevalue: type is
   "000 001 011 010 110";
```

When this is processed by a tool that supports the enum_encoding attribute, it uses the information about the statevalue encoding. Tools that do not recognize the enum_encoding attribute ignore the encoding.

Although it is recommended that you use syn_enum_encoding, the Synopsys FPGA tools recognize enum_encoding and treat it just like syn_enum_encoding. The tool uses the specified encoding when the FSM compiler is disabled, and ignores the value when the FSM Compiler is enabled.

If enum_encoding and syn_encoding are both defined and the FSM compiler is enabled, the tool uses the value of syn_encoding. If you have both syn_enum_encoding and enum_encoding defined, the value of syn_enum_encoding prevails.

syn_encoding Compared to syn_enum_encoding

To implement a state machine with a particular encoding style when the FSM Compiler is enabled, use the syn_encoding attribute. The syn_encoding attribute affects how the technology mapper implements state machines in the final netlist. The syn_enum_encoding directive only affects how the compiler interprets the associated enumerated data types. Therefore, the encoding defined by syn_enum_encoding is *not propagated* to the implementation of the state machine. However, when FSM Compiler is disabled, the value of syn_enum_encoding is implemented in the final circuit.

Example of syn_enum_encoding for User-Defined Encoding

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

entity shift_enum is
   port (clk, rst : bit;
         O : out std_logic_vector(2 downto 0) );
end shift_enum;

architecture behave of shift_enum is
type state_type is (S0, S1, S2);
attribute syn_enum_encoding: string;
attribute syn_enum_encoding of state_type : type is "001 010 101";
signal machine : state_type;
begin
   process (clk, rst)
   begin
      if rst = '1' then
         machine <= S0;
      elsif clk = '1' and clk'event then
         case machine is
            when S0 => machine <= S1;
            when S1 => machine <= S2;
            when S2 => machine <= S0;
         end case;
      end if;
   end process;

with machine select
O <= "001" when S0,
"010" when S1,
"101" when S2;
end behave;
```

syn_global_buffers

Attribute

Microsemi IGLOO/IGLOOe, ProASIC3/3E

Specifies the number of global buffers to be used in a design. The synthesis tool automatically adds global buffers for clock nets with high fanout; use this attribute to specify a maximum number of buffers and restrict the amount of global buffer resources used. Also, if there is a black box in the design that has global buffers, you can use syn_global_buffers to prevent the synthesis tool from inferring clock buffers or exceeding the number of global resources.

You specify the attribute globally on the top-level module/entity or view. For Microsemi designs, it can be any integer between 6 and 18. If you specify an integer less than 6, the software infers six global buffers.

Constraint File Syntax and Example

define_attribute {*view***} syn_global_buffers {***maximum***}**

define_global_attribute syn_global_buffers {*maximum***}**

For example:

```
define_global_attribute syn_global_buffers {10}
```

Verilog Syntax and Example

object **/* synthesis syn_global_buffers =** *maximum* ***/;**

For example:

```
module top (clk1, clk2, clk3, clk4, clk5, clk6, clk7,clk8,clk9,
   clk10, clk11, clk12, clk13, clk14, clk15, clk16, clk17, clk18,
   clk19, clk20, d1, d2, d3, d4, d5, d6, d7, d8, d9, d10, d11,
   d12, d13, d14, d15, d16, d17, d18, d19, d20, q1, q2, q3, q4, q5,
   q6, q7, q8, q9, q10, q11, q12, q13, q14, q15, q16, q17, q18,
   q19, q20, reset) /* synthesis syn_global_buffers = 10 */;
input clk1, clk2, clk3, clk4, clk5, clk6, clk7,clk8,clk9, clk10,
   clk11, clk12, clk13, clk14, clk15, clk16, clk17, clk18,
   clk19, clk20;
input d1, d2, d3, d4, d5, d6, d7, d8, d9, d10, d11, d12, d13, d14,
   d15, d16, d17, d18, d19, d20;output q1, q2, q3, q4, q5, q6, q7,
q8, q9, q10, q11, q12, q13, q14,
   q15, q16, q17, q18, q19, q20;
input reset;
reg q1, q2, q3, q4, q5, q6, q7, q8, q9, q10,
    q11, q12, q13, q14, q15, q16, q17, q18, q19, q20;

always @(posedge clk1 or posedge reset)
   if (reset)
      q1 <= 1'b0;
   else
      q1 <= d1;

always @(posedge clk2 or posedge reset)
   if (reset)
      q2 <= 1'b0;
   else
      q2 <= d2;

always @(posedge clk3 or posedge reset)
   if (reset)
      q3 <= 1'b0;
   else
      q3 <= d3;

always @(posedge clk4 or posedge reset)
   if (reset)
      q4 <= 1'b0;
   else
      q4 <= d4;

always @(posedge clk5 or posedge reset)
   if (reset)
      q5 <= 1'b0;
   else
      q5 <= d5;
```

```
always @(posedge clk6 or posedge reset)
   if (reset)
      q6 <= 1'b0;
   else
      q6 <= d6;

always @(posedge clk7 or posedge reset)
   if (reset)
      q7 <= 1'b0;
   else
      q7 <= d7;

always @(posedge clk8 or posedge reset)
   if (reset)
      q8 <= 1'b0;
   else
      q8 <= d8;

always @(posedge clk9 or posedge reset)
   if (reset)
      q9 <= 1'b0;
   else
      q9 <= d9;

always @(posedge clk10 or posedge reset)
   if (reset)
      q10 <= 1'b0;
   else
      q10 <= d10;

always @(posedge clk11 or posedge reset)
   if (reset)
      q11 <= 1'b0;
   else
      q11 <= d11;

always @(posedge clk12 or posedge reset)
   if (reset)
      q12 <= 1'b0;
   else
      q12 <= d12

always @(posedge clk13 or posedge reset)
   if (reset)
      q13 <= 1'b0;
   else
      q13 <= d13;
```

```
always @(posedge clk14 or posedge reset)
   if (reset)
      q14 <= 1'b0;
   else
      q14 <= d14;

always @(posedge clk15 or posedge reset)
   if (reset)
      q15 <= 1'b0;
   else
      q15 <= d15;

always @(posedge clk16 or posedge reset)
   if (reset)
      q16 <= 1'b0;
   else
      q16 <= d16;

always @(posedge clk17 or posedge reset)
   if (reset)
      q17 <= 1'b0;
   else
      q17 <= d17;

always @(posedge clk18 or posedge reset)
   if (reset)
      q18 <= 1'b0;
   else
      q18 <= d18;

always @(posedge clk19 or posedge reset)
   if (reset)
      q19 <= 1'b0;
   else
      q19 <= d19;

always @(posedge clk20 or posedge reset)
   if (reset)
      q20 <= 1'b0;
   else
      q20 <= d20;

endmodule
```

VHDL Syntax and Example

attribute syn_global_buffers of *object* **:** *objectType* **is** *maximum***;**

See VHDL Attribute and Directive Syntax, on page 554 for different ways to specify VHDL attributes and directives.

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

entity top is
   port (clk : in std_logic_vector(19 downto 0);
         d : in std_logic_vector(19 downto 0);
         q : out std_logic_vector(19 downto 0);
         reset : in std_logic );
end top;

architecture behave of top is
attribute syn_global_buffers : integer;
attribute syn_global_buffers of behave : architecture is 10;
begin
   process (clk, reset)
   begin
      for i in 0 to 19 loop
         if (reset = '1') then
            q(i) <= '0';
         elsif clk(i) = '1' and clk(i)' event then
            q(i) <= d(i);
         end if;
      end loop;
   end process;
end behave;
```

syn_hier

Attribute

Lets you control the amount of hierarchical transformation across boundaries on module or component instances during optimization.

During synthesis, the synthesis tool dissolves as much hierarchy as possible to allow efficient logic optimization across hierarchical boundaries while maintaining optimal run times. The tool then rebuilds the hierarchy as close as possible to the original source to preserve the topology of the design. Use the syn_hier attribute to address specific needs to maintain the original design hierarchy during optimization. This attribute gives you manual control over flattening/preserving instances, modules, or architectures in the design.

Constraint File Syntax and Example

define_attribute {*object*} syn_hier {*value*}

where *object* is a view, and *value* can be any of the values described in syn_hier Values, on page 63. Note however, if you are defining syn_hier globally, it is recommended that you use the SCOPE collection to apply syn_hier on all views instead. For example:

```
define_scope_collection all_views {find -hier -view {*}}
define_attribute {$all_views} {syn_hier} {fixed}
```

Check the attribute values to determine where to attach the attribute. Here is an example:

```
define_attribute {v:fifo} syn_hier {hard}
```

Make sure to specify the attribute on the view (**v:** object type). See syn_hier in the SCOPE Window, on page 64 for details.

Verilog Syntax and Examples

object **/* synthesis syn_hier = "*value*" */ ;**

where *object* can be a module declaration and *value* can be any of the values described in syn_hier Values, on page 63. Check the attribute values to determine where to attach the attribute.

This is the Verilog syntax:

```
module fifo(out, in) /* synthesis syn_hier = "hard" */;

// Other code
```

VHDL Syntax and Examples

attribute syn_hier of *object* **: architecture is "***value***" ;**

where *object* is an architecture name and *value* can be any of the values described in syn_hier Values, on page 63. Check the attribute values to determine the level at which to attach the attribute.

See VHDL Attribute and Directive Syntax, on page 554 for different ways to specify VHDL attributes and directives. This is the VHDL syntax:

```
architecture struct of cpu is

attribute syn_hier : string;
attribute syn_hier of struct: architecture is "hard";

-- Other code
```

syn_hier Values

The following table shows the values you can use for syn_hier. For additional information about using this attribute in HDL Analyst, see Controlling Hierarchy Flattening, on page 339 and Preserving Hierarchy, on page 339 in the *User Guide*.

soft (default)	The synthesis tool determines the best optimization across hierarchical boundaries. This attribute affects only the design unit in which it is specified.
firm	Preserves the interface of the design unit. However, when there is cell packing across the boundary, it changes the interface and does not guarantee the exact RTL interface. This attribute affects only the design unit in which it is specified.
hard	Preserves the interface of the design unit and prevents most optimizations across the hierarchy. However, the boundary optimization for constant propagation is performed. This attribute affects only the specified design units.

fixed	Preserves the interface of the design unit with no exceptions. Fixed prevents all optimizations performed across hierarchical boundaries and retains the port interfaces as well. For more information, see Using syn_hier fixed, on page 65.
remove	Removes the level of hierarchy for the design unit in which it is specified. The hierarchy at lower levels is unaffected. This only affects synthesis optimization. The hierarchy is reconstructed in the netlist and Technology view schematics.
macro	Preserves the interface and contents of the design with no exceptions. This value can only be set on structural netlists. (In the constraint file, or using the SCOPE editor, set syn_hier to macro on the view (the **v:** object type).
flatten	Flattens the hierarchy of all levels below, but not the one where it is specified. This only affects synthesis optimization. The hierarchy is reconstructed in the netlist and Technology view schematics. To create a completely flattened netlist, use the syn_netlist_hierarchy attribute (syn_netlist_hierarchy, on page 101), set to false. You can use flatten in combination with other syn_hier values; the effects are described in Using syn_hier flatten with Other Values, on page 67. If you apply syn_hier to a compile point, flatten is the only valid attribute value. All other values only apply to the current level of hierarchy. The compile point hierarchy is determined by the type of compile point specified, so a syn_hier value other than flatten is redundant and is ignored.

syn_hier in the SCOPE Window

If you use the SCOPE window to specify the syn_hier attribute, do not drag and drop the object into the SCOPE spreadsheet. Instead, first select syn_hier in the Attribute column, and then use the pull-down menu in the Object column to select the object. This is because you must set the attribute on a view (v:). If you drag and drop an object, you might not get a view object. Selecting the attribute first ensures that only the appropriate objects are listed in the Object column.

Using syn_hier fixed

When you use the fixed value with syn_hier, hierarchical boundaries are preserved with no exceptions. For example, optimizations such as constant propagation are not performed across these boundaries.

Note: It is recommended that you do not use syn_hier with the fixed value on modules that have ports driven by tri-state gates. For details, see When Using Tri-states, on page 65.

When Using Tri-states

It is advised that you avoid using syn_hier="fixed" with tri-states. However, if you do, here is how the software handles the following conditions:

- Tri-states driving output ports

  If a module with syn_hier="fixed" includes tri-state gates that drive a primary output port, then the synthesis software retains a tri-state buffer so that the P&R tool can pack the tri-state into an output port.

- Tri-states driving internal logic

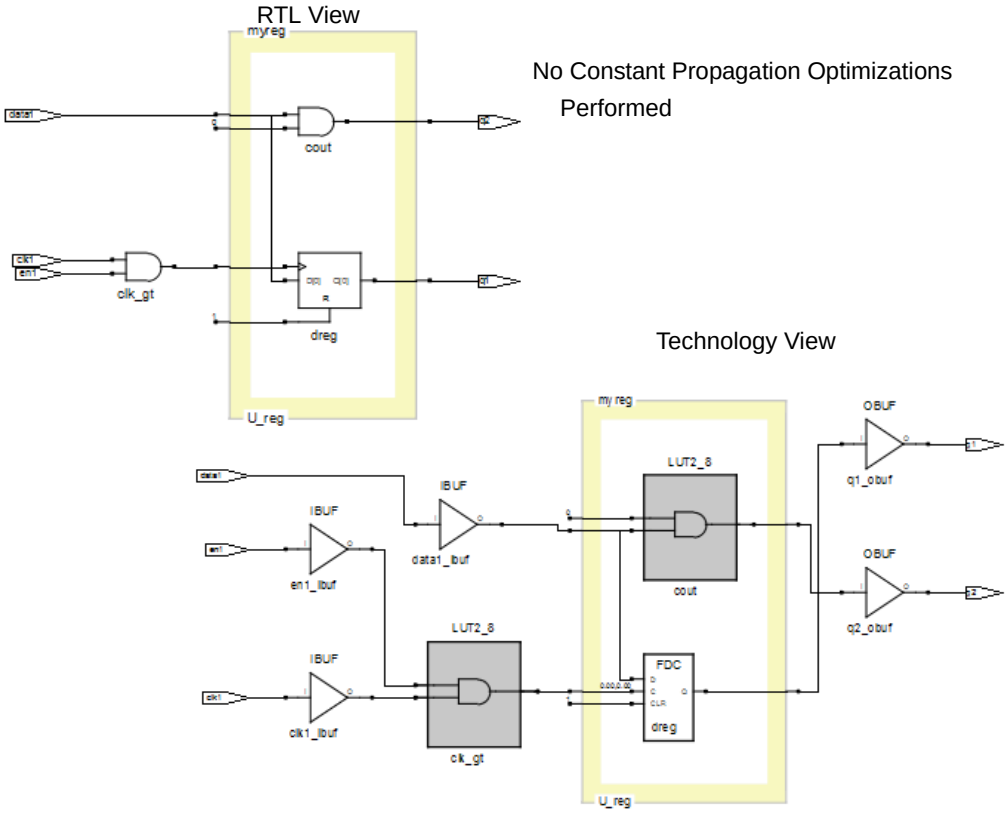
  If a module with syn_hier="fixed" includes tri-state gates that drive internal logic, then the synthesis software converts the tri-state gate to a MUX and optimizes within the module accordingly.

In the following code example, myreg has syn_hier set to fixed.

```
module top(
   clk1,en1, data1,
   q1, q2
   );

input clk1, en1;
input data1;
output q1, q2;

wire cwire, rwire;
wire clk_gt;

assign clk_gt = en1 & clk1;

// Register module
```

```
myreg U_reg (
   .datain(data1),
   .rst(1'b1),
   .clk(clk_gt),
   .en(1'b0),
   .dout(rwire),
   .cout(cwire)
   );

assign q1 = rwire;
assign q2 = cwire;

endmodule


module myreg (
   datain,
   rst,
   clk,
   en,
   dout,
   cout
   ) /* synthesis syn_hier = "fixed" */;

input clk, rst, datain, en;
output dout;
output cout;

   reg dreg;

   assign cout = en & datain;

   always @(posedge clk or posedge rst)
      begin
         if (rst)
            dreg <= 'b0;
         else
            dreg <= datain;
      end

assign dout = dreg;

endmodule
```

The HDL Analyst views show that myreg preserves its hierarchical boundaries without exceptions and prevents constant propagation optimizations.



Using syn_hier flatten with Other Values

You can combine flatten with other syn_hier values as shown below:

flatten,soft	Same as flatten.
flatten,firm	Flattens all lower levels of the design but preserves the interface of the design unit in which it is specified. This option also allows optimization of cell packing across the boundary.
flatten,remove	Flattens all lower levels of the design, including the one on which it is specified.

If you use flatten in combination with another option, the tool flattens as directed until encountering another syn_hier attribute at a lower level. The lower level syn_hier attribute then takes precedence over the higher level one.

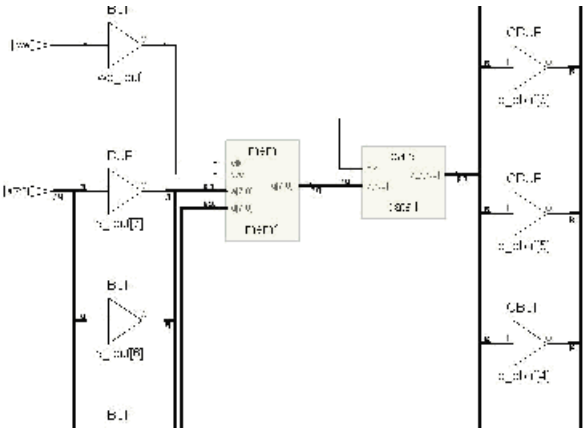
These example demonstrate the use of the flatten and remove values to flatten the current level of the hierarchy and all levels below it (unless you have defined another syn_hier attribute at a lower level).

Verilog
```
module top1 (Q, CLK, RST, LD, CE, D)
/* synthesis syn_hier = "flatten,remove" */;

// Other code
```

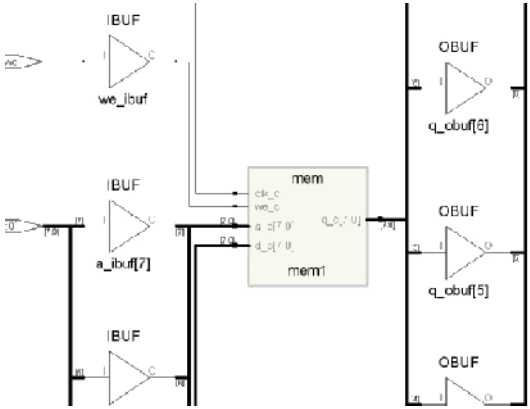
VHDL
```
architecture struct of cpu is

attribute syn_hier : string;
attribute syn_hier of struct: architecture is "flatten,remove";

-- Other code
```

Example of syn_hier hard

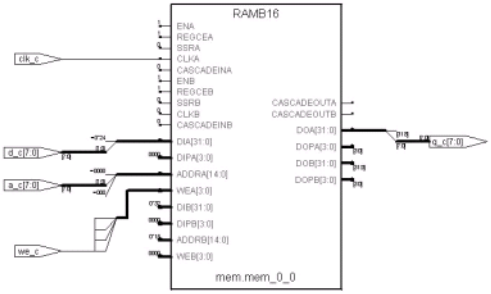
Here is an example of two versions of a design: one with syn_hier set on modules mem and data; the other shows what happens to those modules with the automatic flattening that occurs during synthesis.



With syn_hier="hard"



Without syn_hier="hard"
This is the default.



syn_insert_buffer

Attribute

Inserts a technology-specific clock buffer.

Vendor	Technologies
Microsemi	IGLOO/IGLOOe/IGLOO+/IGLOO2 ProASICPLUS, ProASIC3/3E/3L, SmartFusion2

syn_insert_buffer Values

Vendor	Value	Description
Microsemi	CLKBUF, HCLKBUF CLKINT, HCLKINT	The Microsemi IGLOO/IGLOOe/IGLOO+, and ProASICPLUS families supports these attribute values: • Pads: CLKBUF HCLKBUF (for nets that drive the clock pins of sequential primitives) • Nets: CLKINT HCLKINT (for nets that drive the clock pins of sequential primitives) Microsemi SmartFusion2 and IGLOO2 support only CLKINT.

Description

Use this attribute to insert a clock buffer. You can also use it on a non-clock high fanout net, such as reset or common enable that needs global routing, to insert a global buffer for that port. The synthesis tool inserts a technology-specific clock buffer. The object you attach the attribute to also varies with the vendor.

Vendor	Object	Description
Microsemi	Instance	Inserts the specified clock buffer.

syn_insert_buffer Syntax Specification

You cannot specify this attribute as a global value.

FDC	define_attribute object syn_insert_buffer *value*	FDC Example
Verilog	*object* /* synthesis syn_insert_buffer = "*value*" */;	Verilog Examples
VHDL	attribute syn_insert_buffer of *object* : *objectType* is "*value*";	endmoduleVHDL Example

FDC Example

Enable	Object Type	Object	Attribute	**Value**	Value Type	Description
☑	<any>	<Global>	syn_insert_buffer	CLKBUF	string	Applies an global clock...

Verilog Examples

This section provides technology-specific examples.

```
module test
   (CLK, din1, din2, din3, din4,Q1, Q2, reset, gt1, gt2);

input gt1, gt2;
input CLK;
input reset /* synthesis syn_insert_buffer = "SB_GB_IO" */;
input din1;
input din2 /* synthesis syn_insert_buffer = "SB_GB_IO" */;
input din3 /* synthesis syn_insert_buffer = "SB_GB_IO" */;
input din4;
output reg Q1, Q2;

wire gt11 /* synthesis syn_insert_buffer = "SB_GB" syn_keep = 1 */;

assign gt11 = gt1;

wire int_clk_glob;
wire int_clk_core;

wire int_clk_glob_gt;
wire int_clk_core_gt;

reg  reg_1, reg_2, reg_3, reg_4;
```

```
assign int_clk_glob_gt = CLK & gt11;
assign int_clk_core_gt = CLK & gt2;

always @(posedge int_clk_core_gt or negedge reset)
begin
   if (!reset)
   reg_1 <= 0;
   else
   begin
   reg_1 <= din1;
   reg_2 <= din2;
   Q1 <= reg_1 + reg_2;
   end
end

always @(posedge int_clk_glob_gt)
begin
   reg_3 <= din3;
   reg_4 <= din4;
   Q2 <= reg_3 + reg_4;
end

endmodule
```

This code specifies the syn_insert_buffer attribute, so the tool inserts SB_GB_IO buffers for the reset, din2, and din3 ports. Without the attribute, these ports would use the SB_IO buffer and infer an SB_GB buffer on the gt11 net.

Microsemi syn_insert_buffer Verilog Example

In the following example, the attribute is attached to LDPRE, SEL, RST, LDCOMP, and CLK.

```
module prep2_2 (DATA0, DATA1, DATA2, LDPRE, SEL, RST, CLK, LDCOMP);
output [7:0] DATA0;
input [7:0] DATA1, DATA2;
input LDPRE, SEL, RST, CLK
   /* synthesis syn_insert_buffer = "GL25" */, LDCOMP;
wire [7:0] DATA0_internal;
prep2_1 inst1 (CLK, RST, SEL, LDCOMP, LDPRE, DATA1, DATA2,
   DATA0_internal);
prep2_1 inst2 (CLK, RST, SEL, LDCOMP, LDPRE, DATA0_internal,
   DATA2, DATA0);
endmodule
```

```
module prep2_1 (CLK, RST, SEL, LDCOMP, LDPRE, DATA1, DATA2, DATA0);
input CLK, RST, SEL, LDCOMP, LDPRE ;
input [7:0] DATA1, DATA2 ;
output [7:0] DATA0 ;
reg [7:0] DATA0;
reg [7:0] highreg_output, lowreg_output; // internal registers
wire compare_output = (DATA0 == lowreg_output);  // comparator
wire [7:0]  mux_output = SEL ? DATA1 : highreg_output;

// mux registers
always @ (posedge CLK  or posedge RST)
begin
   if (RST) begin
      highreg_output = 0;
      lowreg_output = 0;
   end else begin
      if (LDPRE)
         highreg_output = DATA2;
      if (LDCOMP)
         lowreg_output  = DATA2;
   end
end


// counter
always @(posedge CLK or posedge RST)
begin
   if (RST)
      DATA0 = 0;
   else if (compare_output)  // load
      DATA0 = mux_output;
   else
      DATA0 = DATA0 + 1;
end
```

endmodule**VHDL Example**

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


entity prep2_1 is
   port (clk : in bit;
         rst : in bit;
         sel : in bit;
```

```
            ldcomp : in bit;
            ldpre : in bit;
            data1,data2 : in std_logic_vector(7 downto 0);
            data0 : out std_logic_vector(7 downto 0) );
end prep2_1;

architecture behave of prep2_1 is
signal equal: bit;
signal mux_output: std_logic_vector(7 downto 0);
signal lowreg_output: std_logic_vector(7 downto 0);
signal highreg_output: std_logic_vector(7 downto 0);
signal data0_i: std_logic_vector(7 downto 0);
begin
   compare: process(data0_i, lowreg_output)
   begin
      if data0_i = lowreg_output then
         equal <= '1';
      else
         equal <= '0';
      end if;
   end process compare;

   mux: process(sel, data1, highreg_output)
   begin
      case sel is
         when '0' =>
            mux_output <= highreg_output;
         when '1' =>
            mux_output <= data1;
      end case;
   end process mux;

   registers: process (rst,clk)
   begin
      if ( rst = '1') then
         highreg_output <= "00000000";
         lowreg_output <= "00000000";
      elsif clk = '1' and clk'event then
         if ldpre = '1' then
            highreg_output <= data2;
         end if;
         if ldcomp = '1' then
            lowreg_output <= data2;
         end if;
      end if;
   end process registers;
```

```
   counter: process (rst,clk)
   begin
      if  rst = '1' then
         data0_i <= "00000000";
      elsif clk = '1' and clk'event then
         if equal = '1' then
            data0_i <= mux_output;
         elsif equal = '0' then
            data0_i  <= data0_i + "00000001";
         end if;
      end if;
   end process counter;
data0 <= data0_i;
end behave;

library ieee;
use ieee.std_logic_1164.all;

entity prep2_2 is
   port (CLK : in bit;
         RST : in bit;
         SEL : in bit;
         LDCOMP : in bit;
         LDPRE : in bit;
         DATA1,DATA2 : in std_logic_vector(7 downto 0);
         DATA0 : out std_logic_vector(7 downto 0) );
attribute syn_insert_buffer : string;
attribute syn_insert_buffer of clk : signal is "GL25";
end prep2_2;

architecture behave of prep2_2 is
component prep2_1
   port (clk : in bit;
         rst : in bit;
         sel : in bit;
         ldcomp : in bit;
         ldpre : in bit;
         data1,data2 : in std_logic_vector(7 downto 0);
         data0 : out std_logic_vector(7 downto 0) );
end component;

signal data0_internal : std_logic_vector (7 downto 0);
```

```
begin
inst1: prep2_1 port map(clk => CLK, rst => RST, sel => SEL,
   ldcomp => LDCOMP, ldpre => LDPRE, data1 => DATA1,
   data2 => DATA2, data0 => data0_internal );
inst2: prep2_1 port map(clk => CLK, rst => RST, sel => SEL,
   ldcomp => LDCOMP, ldpre => LDPRE, data1 => data0_internal,
   data2 => DATA2, data0 => DATA0 );
end behave;
```

syn_isclock

Directive

Used with the syn_black_box directive and specifies an input port on a black box as a clock. Use the syn_isclock directive to specify that an input port on a black box is a clock, even though its name does not correspond to one of the recognized names. Using this directive connects it to a clock buffer if appropriate. The data type is Boolean.

The syn_isclock directive is one of several directives that you can use with the syn_black_box directive to define timing for a black box. See syn_black_box, on page 37 for a list of the associated directives.

Verilog Syntax and Examples

object **/* synthesis syn_isclock = 1 */ ;**

where *object* is an input port on a black box.

```
module ram4 (myclk,out,opcode,a,b) /* synthesis syn_black_box */;
output [7:0] out;
input myclk /* synthesis syn_isclock = 1 */;
input [2:0] opcode;
input [7:0] a, b;

//Other code
```

VHDL Syntax and Examples

attribute syn_isclock of *object*: *objectType* **is true ;**

where *object* is a black-box input port.

See VHDL Attribute and Directive Syntax, on page 554 for different ways to specify VHDL attributes and directives.

```
library synplify;

entity ram4 is
   port (myclk : in bit;
         opcode : in bit_vector(2 downto 0);
         a, b : in bit_vector(7 downto 0);
         rambus : out bit_vector(7 downto 0) );
attribute syn_isclock : boolean;
attribute syn_isclock of myclk: signal is true;

-- Other code
```

syn_keep

Directive

Preserves the specified net intact during optimization and synthesis.

Technology	Default Value	Global	Object
All	-	No	Net

Description

With this directive, the tool preserves the net without optimizing it away by placing a temporary keep buffer primitive on the net as a placeholder. You can view this buffer in the schematic views (see Effect of Using syn_keep, on page 83 for an example). The buffer is not part of the final netlist, so no extra logic is generated. There are various situations where this directive is useful:

- To preserve a net that would otherwise be removed as a result of optimization. You might want to preserve the net for simulation results or to obtain a different synthesis implementation.
- To prevent duplicate cells from being merged during optimization. You apply the directive to the nets connected to the input of the cells you want to preserve.
- As a placeholder to apply the -through option of the define_multicycle_path or define_false_path timing constraint. This allows you to specify a unique path as a multiple-cycle or false path. Apply the constraint to the keep buffer.
- To prevent the absorption of a register into a macro. If you apply syn_keep to a reg or signal that will become a sequential object, the tool keeps the register and does not absorb it into a macro.

syn_keep with Multiple Nets in Verilog

In the following statement, syn_keep only applies to the last variable in the wire declaration, which is net c:

```
wire a,b,c /* synthesis syn_keep=1 */;
```

To apply syn_keep to all the nets, use one of the following methods:

- Declare each individual net separately as shown below.

```
wire a /* synthesis syn_keep=1 */;
wire b /* synthesis syn_keep=1 */;
wire c /* synthesis syn_keep=1 */;
```

- Use Verilog 2001 parenthetical comments, to declare the syn_keep attribute as a single line statement.

```
(* syn_keep=1 *) wire a,b,c;
```

- For more information, see Attribute Examples Using Verilog 2001 Parenthetical Comments, on page 368.

syn_keep and SystemVerilog Data Types

The SystemVerilog data types behave like logic or reg, and SystemVerilog allows them to be assigned either inside or outside an always block. If you want to use syn_keep to preserve a net with a SystemVerilog data type, like bit, byte, longint or shortint for example, you must make sure that continuous assigns are made inside an always block, not outside.

The following table shows examples of SystemVerilog datatype assignments:

Assignment in always block, syn_keep works	`assign keep1_wireand_out;`
	`assign keep2_wireand_out;`
	`always @(*) begin`
	`  keep1_bitand_out;`
	`  keep2_bitand_out;`
	`  keep1_byteand_out;`
	`  keep2_byteand_out;`
	`  keep1_longintand_out;`
	`  keep2_longintand_out;`
	`  keep1_shortintand_out;`
	`  keep2_shortintand_out;`

Assignment outside always block, syn_keep does not work	`assign keep1_wireand_out;`
	`assign keep2_wireand_out;`
	`assign keep1_bitand_out;`
	`assign keep2_bitand_out;`
	`assign keep1_byteand_out;`
	`assign keep2_byteand_out;`
	`assign keep1_longintand_out;`
	`assign keep2_longintand_out;`
	`assign keep1_shortintand_out;`
	`assign keep2_shortintand_out;`

For information about supported SystemVerilog data types, see Data Types, on page 377.

Comparison of syn_keep, syn_preserve, and syn_noprune

Although these directives all work to preserve logic from optimization, syn_keep, syn_preserve, and syn_noprune work on different objects:

syn_keep	Only works on nets and combinational logic. It ensures that the wire is kept during synthesis, and that no optimizations cross the wire. This directive is usually used to prevent unwanted optimizations and to ensure that manually created replications are preserved. When applied to a register, the register is preserved and not absorbed into a macro.
syn_preserve	Ensures that registers are not optimized away.
syn_noprune	Ensures that a black box is not optimized away when its outputs are unused (i.e., when its outputs do not drive any logic).

See Preserving Objects from Being Optimized Away, on page 335 in the *User Guide* for more information.

Verilog Syntax and Example

object **/* synthesis syn_keep = 1 */ ;**

object is a wire or reg declaration. Make sure that there is a space between the object name and the beginning of the comment slash (/).

Here is the source code used to produce the results shown in Effect of Using syn_keep, on page 83.

```
module example2(out1, out2, clk, in1, in2);
output out1, out2;
input clk;
input in1, in2;
wire and_out;
wire keep1 /* synthesis syn_keep=1 */;
wire keep2 /* synthesis syn_keep=1 */;
reg out1, out2;
assign and_out=in1&in2;
assign keep1=and_out;
assign keep2=and_out;

always @(posedge clk)begin;
   out1<=keep1;
   out2<=keep2;
end
endmodule
```

VHDL Syntax and Example

attribute syn_keep of *object* **:** *objectType* **is true ;**

where *object* is a single or multiple-bit signal. See VHDL Attribute and Directive Syntax, on page 554 for different ways to specify VHDL attributes and directives.

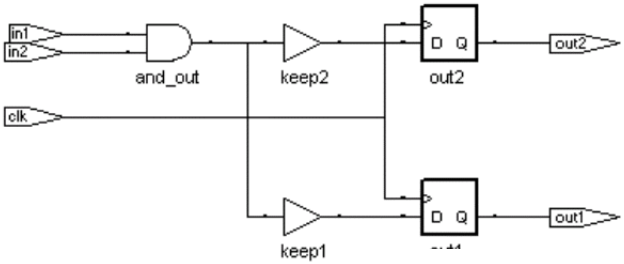
Here is the source code used to produce the schematics shown in Effect of Using syn_keep, on page 83.

```
entity example2 is
   port (in1, in2 : in bit;
         clk : in bit;
         out1, out2 : out bit );
end example2;

architecture rt1 of example2 is
attribute syn_keep : boolean;
signal and_out, keep1, keep2: bit;
attribute syn_keep of keep1, keep2 : signal is true;
begin
and_out <= in1 and in2;
keep1 <= and_out;
keep2 <= and_out;
   process(clk)
   begin
      if (clk'event and clk = '1') then
         out1 <= keep1;
         out2 <= keep2;
      end if;
   end process;
end rt1;
```

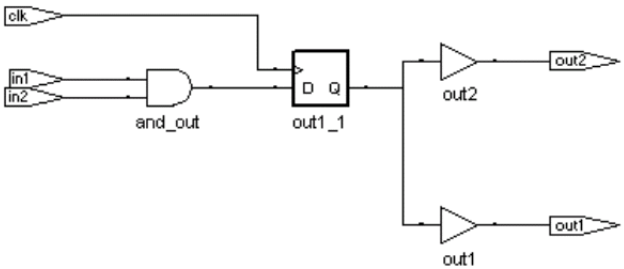
Effect of Using syn_keep

When you use syn_keep on duplicate logic, the tool retains it instead of optimizing it away. The following figure shows the Technology view for two versions of a design.

In the first, syn_keep is set on the nets connected to the inputs of the registers out1 and out2, to prevent sharing. The second figure shows the same design without syn_keep. Setting syn_keep on the input wires for the registers ensures that the design has duplicate registered outputs for out1 and out2. If you do not apply syn_keep to keep1 and keep2, the software optimizes out1 and out2, and only has one register.



With syn_keep



Without syn_keep

syn_loc

Attribute

The syn_loc attribute specifies the location (placement) of ports.

Vendor	Technology
Microsemi	SmartFusion, ProASIC and older families

syn_loc Values

Value	Description
Pin numbers	Assigns pin numbers to ports.

Description

Specifies pin locations for I/O pins and cores, and forward-annotates this information to the place-and-route tool. This attribute can only be specified in a top-level source file or a constraint file.

syn_loc Syntax

Default	Global Attribute	Object
Not Applicable	No	Port

pinNumbers is a comma-separated list of pin or placement numbers. Refer to the vendor data book for valid values.

FDC	**define_attribute** *portDesignName* **{syn_loc} {***pinNumbers***}**	FDC Example
Verilog	*object* **/* synthesis syn_loc = "***pinNumbers***" */**	Verilog Example
VHDL	**attribute syn_loc of** *object* : *objectType* **is "***pinNumbers***" ;**	VHDL Example

FDC Example

	Enable	Object Type	Object	Attribute	Value	Value Type	Description
1	✔		out1[2:0]	syn_loc	P14 P12,P11	string	Assign the object location
2							

The following are examples of using this attribute:

Microsemi
```
define_attribute {CR_DIN[3:0]} syn_loc {M7, Y6, B6, D10}
```

Microsemi
```
define_attribute {CR_DIN[3:0]} syn_loc
{M7, Y6, B6, D10}
```

You can also specify locations for individual bus bits with this attribute:

Microsemi

```
define_attribute {CR_DIN[3]} syn_loc {M7}
define_attribute {CR_DIN[2]} syn_loc {Y6}
define_attribute {CR_DIN[1]} syn_loc {B6}
define_attribute {CR_DIN[0]} syn_loc {D10}
```

Specify the b: prefix and the bit slice:

```
define_attribute {b:CR_DIN[0]} syn_loc {D10}
define_attribute {b:CR_DIN[1]} syn_loc {B6}
define_attribute {b:CR_DIN[2]} syn_loc {Y6}
define_attribute {b:CR_DIN[3]} syn_loc {M7}
```

Verilog Example

Microsemi
```
input [3:0] CR_DIN /* synthesis syn_loc = "M7,Y6, B6, D10"
```

```
module test(a ,b, clk, out1);
   input clk;
   input [2:0]a;
   input [2:0]b;
   output reg [2:0] out1/* synthesis syn_loc = "P14,P12,P11*/;

always@(posedge clk)
begin
   out1 <= a + b;
end
endmodule
```

VHDL Example

Microsemi

```
attribute syn_loc : string;
attribute syn_loc of CR_DIN : signal is "M7,Y6, B6, D10";
```

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

entity test is
   generic (s : integer := 2);

   port (
   clk: in std_logic;
   in1: in std_logic_vector(s downto 0);
   in2: in std_logic_vector(s downto 0);
   d_out: out std_logic_vector(5 downto 0) );
   attribute syn_loc : string;
   attribute syn_loc of d_out:signal is"P14,P12,P11,P5,P21,P13";
end test;

architecture beh of test is
begin
   process (clk)
   begin
      if rising_edge(clk) then
         d_out <= in1 & in2;
      end if;
   end process;
end beh;
```

syn_looplimit

Directive

VHDL only

Specifies a loop iteration limit for while loops in the design when the loop index is a variable, not a constant. If your design requires a variable loop index, use the syn_looplimit directive to specify a limit for the compiler. If you do not, you can get a "while loop not terminating" compiler error. The limit cannot be an expression. The higher the value you set, the longer the runtime. To override the default limit of 2000 in the RTL, use the Loop Limit option on the VHDL tab of the Implementation Options panel. See VHDL Panel, on page 192 in the *Command Reference*.

Verilog applications use the loop_limit directive (see loop_limit, on page 27).

VHDL Syntax and Example

attribute syn_looplimit : *integer*;
attribute syn_looplimit of *labelName* **: label is** *value*;

The following is an example where the loop limit is set to 5000:

```
library IEEE;
use std.textio.all;
use ieee.std_logic_textio.all;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity initram is
   port (rAddr, wAddr, dataIn  : in integer;
         clk: in bit;
         we : in bit;
         dataOut  : out integer );
end;

architecture rtl of initram is
subtype smallint is integer range 0 to 3000;
type intAry is array (0 to 3000) of smallint;
function load( name : string) return intAry is
attribute syn_looplimit : integer;
attribute syn_looplimit of myloop: label is 5000;
```

```
variable t : intAry ;
variable data : smallint ;
variable dataLine : line ;
variable i : natural ;
file dataFile : text open READ_MODE is name ;

begin
   myloop: while ( not endfile(dataFile) ) loop
      readline(dataFile,dataLine);
      read(dataLine,data);
      t(i) := data;
      i := i + 1;
   end loop myloop;

return t;
end load;

-----------------------------------------------------

signal ram : intAry := load("data.txt");
signal rAddr_reg : integer ;
begin
   process (clk) begin
      if (clk'event and clk='1') then
         rAddr_reg <= rAddr;
         if(we = '1') then
            ram(wAddr) <= dataIn;
         end if;
      end if;
end process;

dataOut <= ram(rAddr_reg);
end RTL ;
```

The data.txt file in the example is a large data file with each entry representing an iteration for the loop.

syn_maxfan

Attribute

Overrides the default (global) fanout guide for an individual input port, net, or register output.

Vendor	Technology	Default
Microsemi	All	None

syn_maxfan Value

value	Integer for the maximum fanout

Description

syn_maxfan overrides the global fanout for an individual input port, net, or register output. You set the default Fanout Guide for a design through the Device panel on the Implementation Options dialog box or with the set_option -fanout_limit command or -fanout_guide in the project file. Use the syn_maxfan attribute to specify a different (local) value for individual I/Os.

Generally, syn_maxfan and the default fanout guide are suggested guidelines only, but in certain cases they function as hard limits.

- When they are guidelines, the synthesis tool takes them into account, but does not always respect them absolutely. The synthesis tool does not respect the syn_maxfan limit if the limit imposes constraints that interfere with optimization.

You can apply the syn_maxfan attribute to the following:

- Registers or instances. You can also apply it to a module or entity. If you attach the attribute to a lower-level module or entity that is subsequently optimized during synthesis, the synthesis tool moves the syn_maxfan attribute up to the next higher level. If you do not want syn_maxfan moved up during optimization, set the syn_hier attribute for the entity or module to hard. This prevents the module or entity from being flattened when the design is optimized.

- Ports or nets. If you apply the attribute to a net, the synthesis tool creates a KEEPBUF component and attaches the attribute to it to prevent the net itself from being optimized away during synthesis.

The syn_maxfan attribute is often used along with the syn_noclockbuf attribute on an input port that you do not want buffered. There are a limited number of clock buffers in a design, so if you want to save these special clock buffer resources for other clock inputs, put the syn_noclockbuf attribute on the clock signal. If timing for that clock signal is not critical, you can turn off buffering completely to save area. To turn off buffering, set the maximum fanout to a very high number; for example, 1000.

Similarly, you use syn_maxfan with the syn_replicate attribute in certain technologies to control replication.

syn_maxfan Syntax

Global	Object Type
No	Registers, instances, ports, nets

FDC	**define_attribute {***object***} syn_maxfan {***integer***}**	FDC Example
Verilog	*object* **/* synthesis syn_maxfan = "***value***" */ ;**	Verilog Example
VHDL	**attribute syn_maxfan of** *object* **:** *objectType* **is "***value***" ;**	VHDL Example

FDC Example

define_attribute {*object***} syn_maxfan {***integer***}**

Enable	Object Type	Object	**Attribute**	Value	Value Type	Description
✔	<any>	<Global>	syn_maxfan	1	integer	Overrides the defaul...

Verilog Example

object **/* synthesis syn_maxfan = "***value***" */ ;**

For example:

```
module syn_maxfan (clk,rst,a,b,c);
input clk,rst;
input [7:0] a,b;
output reg [7:0] c;

reg d/* synthesis syn_maxfan=3 */;

always @ (posedge clk)
   begin
      if(rst)
         d <= 0;
      else
         d <= ~d;
   end

always @ (posedge d)
   begin
      c <= a^b;
   end

endmodule
```

VHDL Example

attribute syn_maxfan of *object* **:** *objectType* **is "***value***" ;**

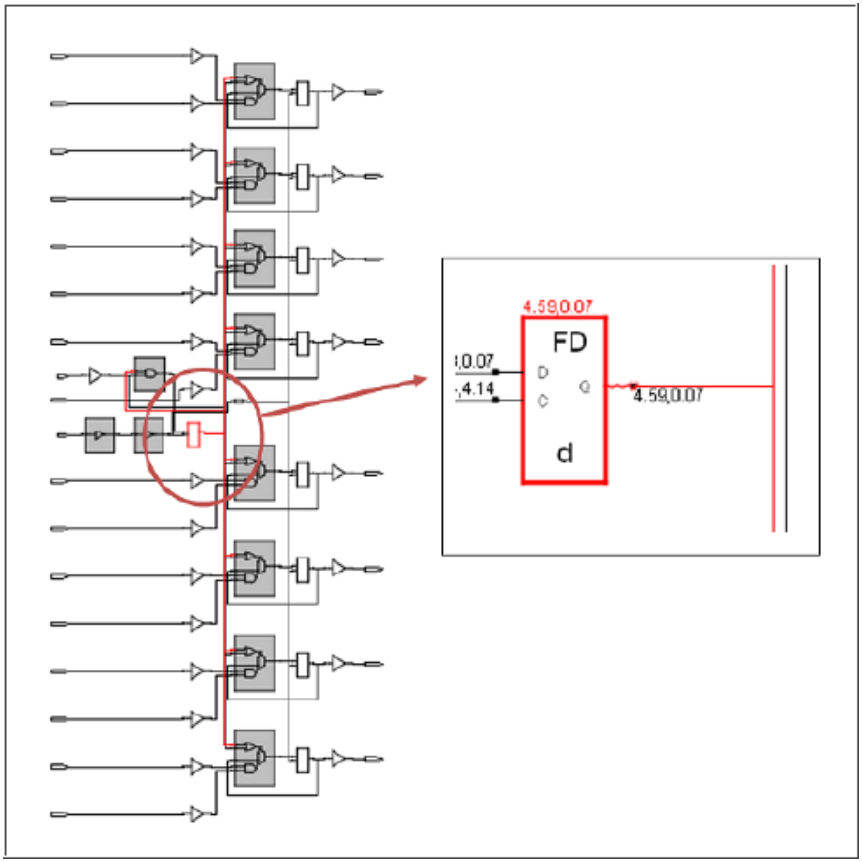
See VHDL Attribute and Directive Syntax, on page 554 for different ways to specify VHDL attributes and directives.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity maxfan is
   port ( a : in std_logic_vector(7 downto 0);
      b : in std_logic_vector(7 downto 0);
      rst : in std_logic;
      clk : in std_logic;
      c : out std_logic_vector(7 downto 0) );
end maxfan;

architecture rtl of maxfan is
signal d : std_logic;

attribute syn_maxfan : integer;
attribute syn_maxfan of d : signal is 3;
```

```
begin

process (clk)
   begin
   if (clk'event and clk = '1') then
      if (rst = '1') then
      d <= '0';
      else
      d <= not d;
   end if;
   end if;
end process;

process (d)
   begin

   if (d'event and d = '1') then

   c <= a and b;
   end if;
   end process;

end rtl;
```

Effect of Using syn_maxfan

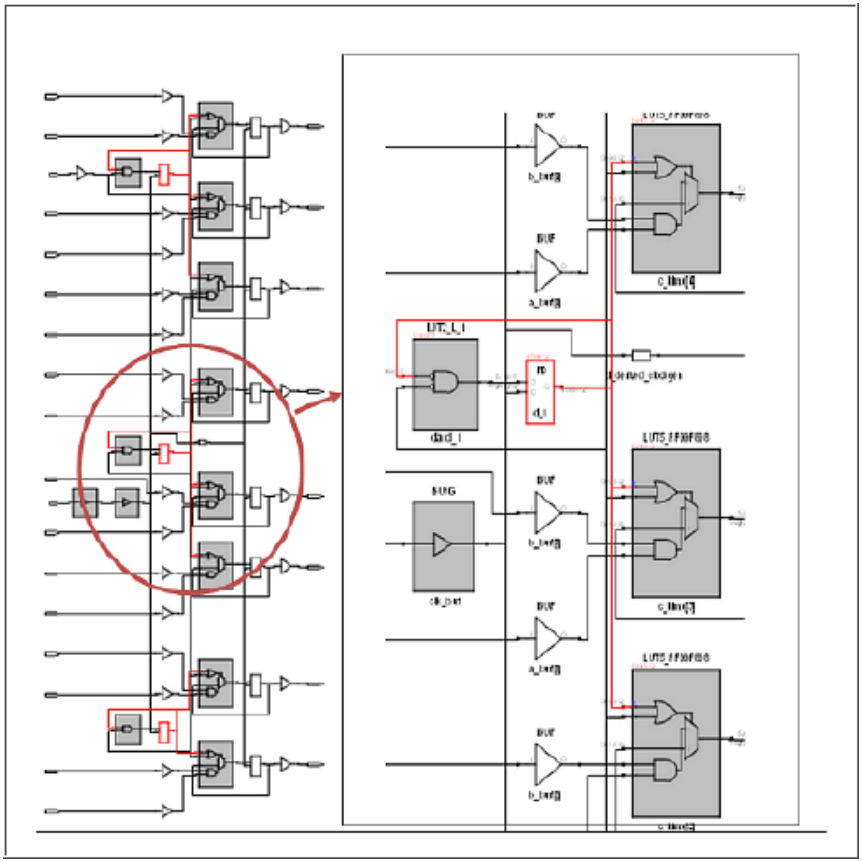
Before applying syn_maxfan:



After applying syn_maxfan:

After applying attribute syn_maxfan, the register “d” replicated three times (shown in red) because its actual fanout is 8, but we have restricted it to 3.

Verilog	reg d/* synthesis syn_maxfan=3 */;
VHDL	attribute syn_maxfan of d : signal is 3;



syn_multstyle

Attribute

Determines how multipliers are implemented.

Vendor	Device	Values
Microsemi	SmartFusion2, IGLOO2	dsp | logic

syn_multstyle Values

Value	Description	Default
dsp	*Microsemi* Implements the multipliers as DSP blocks.	X

Description

This attribute specifies whether the multipliers are implemented as dedicated hardware blocks or as logic.

syn_multstyle Syntax

Global Attribute	Object
Yes	Module or instance

The following shows the attribute syntax when specified in different files:

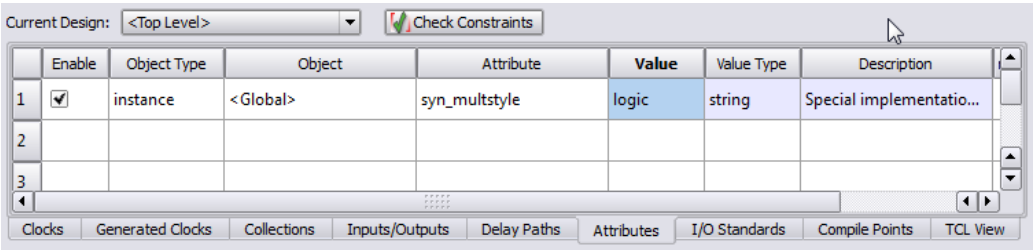
FDC	`define_attribute {instance} syn_multstyle {logic | dsp}`
	Global attribute:
	`define_global_attribute syn_multstyle {logic | dsp }`
		SCOPE Example

Verilog	`input net /* synthesis syn_multstyle = "logic | dsp " */;`	Verilog Example
VHDL	`attribute syn_multstyle of instance : signal is "logic | dsp";`	VHDL Example

See VHDL Attribute and Directive Syntax, on page 554 for different ways to specify VHDL attributes and directives.

SCOPE Example

This SCOPE example specifies that the multipliers be globally implemented as logic:



This example specifies that multipliers be implemented as logic.

```
define_attribute {temp[15:0]} syn_multstyle {logic}
```

Verilog Example

```
module mult(a,b,c,r,en);
input [7:0] a,b;
output [15:0] r;
input [15:0] c;
input en;
wire [15:0] temp /* synthesis syn_multstyle="logic" */;
assign temp = a*b;
assign r = en ? temp: c;
endmodule
```

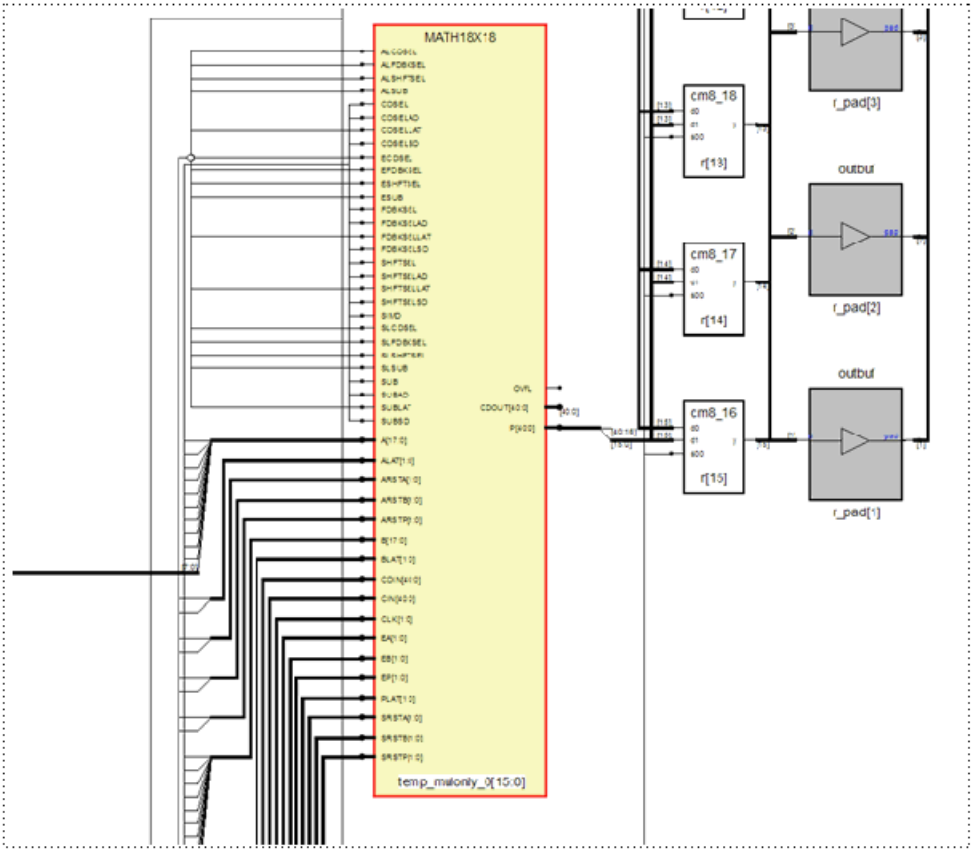
VHDL Example

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

entity mult is
   port (clk : in std_logic ;
      a : in std_logic_vector(7 downto 0) ;
      b : in std_logic_vector(7 downto 0) ;
      c : out std_logic_vector(15 downto 0))
end mult ;

architecture rtl of mult is
signal mult_i : std_logic_vector(15 downto 0) ;
attribute syn_multstyle : string ;
attribute syn_multstyle of mult_i : signal is "logic" ;
begin
mult_i <= std_logic_vector(unsigned(a)*unsigned(b)) ;
   process(clk)
   begin
      if (clk'event and clk = '1') then
         c <= mult_i ;
      end if ;
   end process ;
end rtl ;
```

Effect of Using syn_multstyle in a Microsemi Design

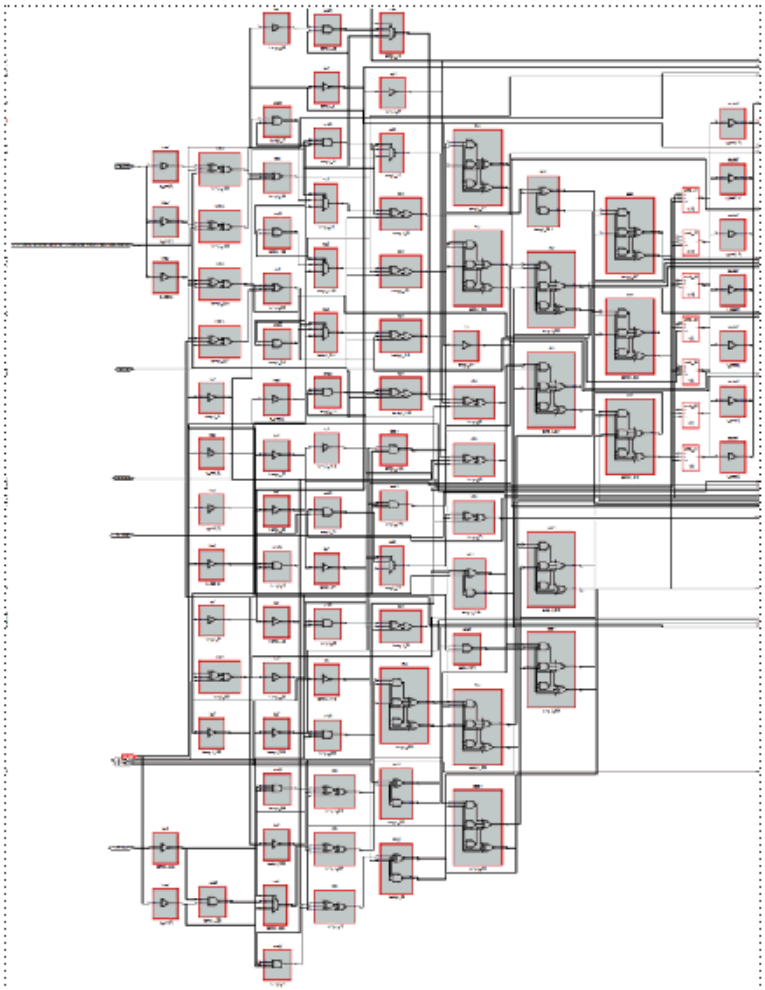
In a Microsemi design, you can specify that the multipliers be implemented as logic or as dedicated DSP blocks. The following figure shows a multiplier implemented as DSP:

Verilog	`wire [15:0] temp /* synthesis syn_multstyle = "dsp"*/;`
VHDL	`attribute syn_multstyle of mult_i : signal is "dsp";`



The following figure shows the same Microsemi design with the multiplier implemented as logic when the attribute is set to logic:

Verilog	`wire [15:0] temp /* synthesis syn_multstyle = "logic"*/;`
VHDL	`attribute syn_multstyle of mult_i : signal is "logic";`



syn_netlist_hierarchy

Attribute.

Determines if the generated netlist is to be hierarchical or flat.

Vendor	Technology
Microsemi	ProASIC, IGLOO families

syn_netlist_hierarchy Values

Value	Description	Default
1/true	Allows hierarchy generation	Default
0/false	Flattens hierarchy in the netlist	

Description

A global attribute that controls the generation of hierarchy in the EDIF or VM output netlist when assigned to the top-level module in your design. The default (1/true) allows hierarchy generation, and setting the attribute to 0/false flattens the hierarchy and produces a completely flattened output netlist.

Syntax Specification

Global	Object
Yes	Module/Architecture

FDC	**define_global_attribute syn_netlist_hierarchy {0|1}**	SCOPE Example
Verilog	*object* **/* synthesis syn_netlist_hierarchy = 0|1 */ ;**	Verilog Example
VHDL	**attribute syn_netlist_hierarchy of** *object* **:** *objectType* **is true|false ;**	VHDL Example

SCOPE Example

Enable	Object Type	Object	**Attribute**	Value	Value Type	Description
☑	global	<Global>	syn_netlist_hierarchy	1	boolean	Enable hierarchy reconstruction

Verilog Example

```
module fu_add(input a,b,cin,output su,cy);
assign su = a ^ b ^ cin;
assign cy = (a & b) | ((a^b) & cin);
endmodule 4

module rca_adder#(parameter width =4)
   (input[width-1:0]A,B,input CIN,
    output[width-1:0]SU,output COUT );
wire[width-2:0]CY;
fu_add FA0(.su(SU[0]),.cy(CY[0]),.cin(CIN),.a(A[0]),.b(B[0]));
fu_add FA1(.su(SU[1]),.cy(CY[1]),.cin(CY[0]),.a(A[1]),.b(B[1]));
fu_add FA2(.su(SU[2]),.cy(CY[2]),.cin(CY[1]),.a(A[2]),.b(B[2]));
fu_add FA3(.su(SU[3]),.cy(COUT),.cin(CY[2]),.a(A[3]),.b(B[3]));
endmodule

module rp_top#(parameter width =16)
   (input[width-1:0]A1,B1,input CIN1,
    output[width- 1:0]SUM,output COUT1) /*synthesis
      syn_netlist_hierarchy=0*/;
wire[2:0]CY1;
rca_adder RA0 (.SU(SUM[3:0]),.COUT(CY1[0]),.CIN(CIN1),
   .A(A1[3:0]),.B(B1[3:0]));
rca_adder RA1(.SU(SUM[7:4]),.COUT(CY1[1]),.CIN(CY1[0]),
   .A(A1[7:4]),.B(B1[7]));
```

```
rca_adder RA2 (.SU(SUM[11:8]),.COUT(CY1[2]),.CIN(CY1[1]),
   .A(A1[11:8]),.B(B1[11:8]));
rca_adder RA3(.SU(SUM[15:12]),.COUT(COUT1),.CIN(CY1[2]),
   .A(A1[15:12]),.B(B1[15:12]));
endmodule
```

VHDL Example

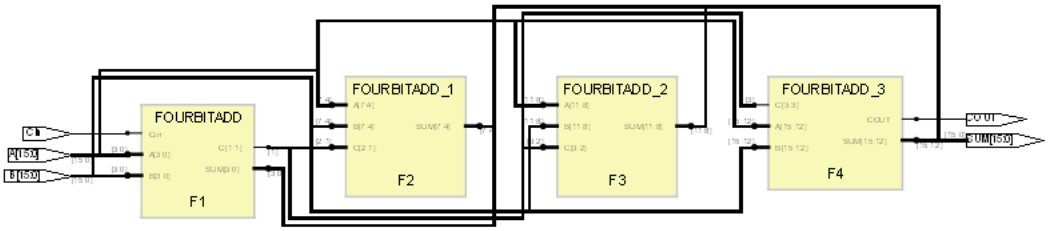
```
library ieee;
use ieee.std_logic_1164.all;

entity FULLADDER is
   port (a, b, c : in std_logic;
         sum, carry: out std_logic);
end FULLADDER;

architecture fulladder_behav of FULLADDER is
begin
   sum <= (a xor b) xor c ;
   carry <= (a and b) or (c and (a xor b));
end fulladder_behav;

library ieee;
use ieee.std_logic_1164.all;

entity FOURBITADD is
   port (a, b : in std_logic_vector(3 downto 0);
         Cin  : in std_logic;
         sum  : out std_logic_vector (3 downto 0);
         Cout, V : out std_logic );
end FOURBITADD;

architecture fouradder_structure of FOURBITADD is
signal c: std_logic_vector (4 downto 1);
component FULLADDER
   port (a, b, c: in std_logic;
         sum, carry: out std_logic);
end component;
begin
   FA0: FULLADDER
      port map (a(0), b(0), Cin, sum(0), c(1));
   FA1: FULLADDER
      port map (a(1), b(1), C(1), sum(1), c(2));
   FA2: FULLADDER
      port map (a(2), b(2), C(2), sum(2), c(3));
```

```
   FA3: FULLADDER
      port map (a(3), b(3), C(3), sum(3), c(4));
   V <= c(3) xor c(4);
   Cout <= c(4);
end fouradder_structure;

library ieee;
use ieee.std_logic_1164.all;

entity BITADD is
   port (A, B: in std_logic_vector(15 downto 0);
         Cin : in std_logic;
         SUM : out std_logic_vector (15 downto 0);
         COUT: out std_logic );
end BITADD;

architecture adder_structure of BITADD is
attribute syn_netlist_hierarchy : boolean;
attribute syn_netlist_hierarchy of adder_structure:
   architecture is false;
signal C: std_logic_vector (4 downto 1);

component FOURBITADD
   port (a, b: in std_logic_vector(3 downto 0);
         Cin : in std_logic;
         sum : out std_logic_vector (3 downto 0);
         Cout, V: out std_logic);
end component;

begin
   F1: FOURBITADD
      port map (A(3 downto 0),B(3 downto 0),
                Cin, SUM(3 downto 0),C(1) );
   F2: FOURBITADD
      port map (A(7 downto 4),B(7 downto 4),
                C(1), SUM(7 downto 4),C(2) );
   F3: FOURBITADD
      port map (A(11 downto 8),B(11 downto 8),
                C(2), SUM(11 downto 8),C(3) );
   F4: FOURBITADD
      port map (A(15 downto 12),B(15 downto 12),
                C(3), SUM(15 downto 12),C(4) );
   COUT <= c(4);
end adder_structure;
```

Effect of Using syn_netlist_hierarchy

Without applying the attribute (default is to allow hierarchy generation) or setting the attribute to 1/true creates a hierarchical netlist.

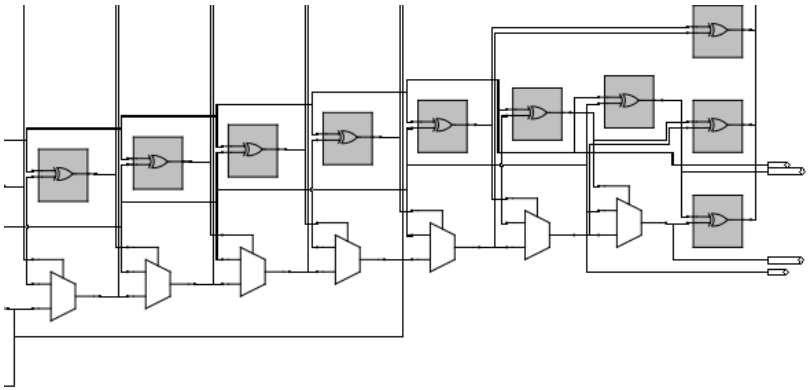
Verilog	`output[width-1:0]SUM,output COUT1)` `/*synthesis syn_netlist_hierarchy=1*/;`
VHDL	`attribute syn_netlist_hierarchy of adder_structure :` `architecture is true ;`



Applying the attribute with a value of 0/false creates a flattened netlist.

Verilog
```
output[width-1:0]SUM,output COUT1)
/*synthesis syn_netlist_hierarchy=0*/;
```

VHDL
```
attribute syn_netlist_hierarchy of adder_structure :
architecture is false ;
```



syn_hier flatten and syn_netlist_hierarchy

The syn_hier=flatten attribute and the syn_netlist_hierarchy=false attributes both flatten hierarchy, but work slightly differently. Use the syn_netlist_hierarchy attribute if you want a completely flattened netlist (this attribute flattens all levels of hierarchy). When you set syn_hier=flatten, you flatten the hierarchical levels below the component on which it is set, but you do not flatten the current hierarchical level where it is set. Refer to syn_hier, on page 62 for information about this attribute.

syn_noarrayports

Attribute

Specifies that the ports of a design unit be treated as individual signals (scalars), not as buses (arrays) in the output file.

Constraint File Syntax and Example

define_global_attribute syn_noarrayports {0|1}

For example:

```
define_global_attribute syn_noarrayports {1}
```

Verilog Syntax and Example

object **/* synthesis syn_noarrayports = 0 | 1 ;**

Where *object* is a module declarations. For example:

```
module adder8(cout, sum, a, b, cin)
   /* synthesis syn_noarrayports = 1 */;

// Other code
```

VHDL Syntax and Example

attribute syn_noarrayports of *object* **:** *objectType* **is true | false ;**

where *object* is an architecture name. The data type is Boolean. See VHDL Attribute and Directive Syntax, on page 554 for different ways to specify VHDL attributes and directives.

In this example, the ports of adder8 are treated as scalars during synthesis.

```
architecture adder8 of adder8 is
attribute syn_noarrayports : boolean;
attribute syn_noarrayports of adder8 : architecture is true;

-- Other code
```

syn_noclockbuf

Attribute

Turns off automatic clock buffer usage.

Vendor	Technology
Microsemi	all

syn_noclockbuf Values

Value	Description
0/false (Default)	Turns on clock buffering.
1/true	Turns off clock buffering.

Description

The synthesis tool uses clock buffer resources, if they exist in the target module, and puts them on the highest fanout clock nets. You can turn off automatic clock buffer usage by using the syn_noclockbuf attribute. For example, you can put a clock buffer on a lower fanout clock that has a higher frequency and a tighter timing constraint.

You can turn off automatic clock buffering for nets or specific input ports. Set the Boolean value to 1 or true to turn off automatic clock buffering.

You can attach this attribute to a port or net in any hard architecture or module whose hierarchy will not be dissolved during optimization.

Constraint File Syntax and Example

Global Support	Object
Yes	module/architecture

define_attribute {*clock_port*} syn_noclockbuf {0|1}

define_global_attribute syn_noclockbuf {0|1}

For example:

```
define_attribute {clk} syn_noclockbuf {1}
define_global_attribute syn_noclockbuf {1}
```

FDC Example

The `syn_noclockbuf` attribute can be applied in the scope window as shown:

	Enabled	Object Type	Object	Attribute	Value	Val Type	Description
1	☑	global	<global>	syn_noclockbuf	1	boolean	Use normal input buffer

Verilog Syntax and Examples

object **/* synthesis syn_noclockbuf = 1 | 0 */ ;**

```
module ckbufg (d,clk,rst,set,q);
input d,rst,set;
input clk   /*synthesis syn_nocclockbuf=1*/;
output reg q;
always@(posedge clk)
begin
if(rst)
q<=0;
else if(set)
q<=1;
else
q<=d;
end
endmodule
```

VHDL Syntax and Examples

attribute syn_noclockbuf of *object* **:** *objectType* **is true | false ;**

```
library IEEE;
use IEEE.std_logic_1164.all;
entity d_ff_srss is
port ( d,clk,reset,set : in STD_LOGIC;
            q : out STD_LOGIC);
attribute syn_noclockbuf: Boolean;
attribute syn_noclockbuf of clk : signal is false;
end d_ff_srss;
architecture d_ff_srss of d_ff_srss is
begin
process(clk)
begin
if clk'event and clk='1' then
if reset='1' then
q <= '0';
elsif set='1' then
q <= '1';
else
q <= d;
end if;
end if;
end process;
end d_ff_srss;
```

Global Support

When `syn_noclockbuf` attribute is applied globally, global buffers are inferred by default. If the `syn_noclockbuf` attribute value is set to ‘1’, global buffers are not inferred.

syn_noprune

Directive

Prevents optimizations for instances and black-box modules (including technology-specific primitives) with unused output ports.

Vendor	Technology	Global	Object
All	All	No	Verilog module/instance VHDL architecture/component

syn_noprune Values

Value	Description
0 | false (Default)	Allows instances and black-box modules with unused output ports to be optimized away.
1 | true	Prevents optimizations for instances and black-box modules with unused output ports.

Description

Use this attribute to prevent the removal of instances, black-box modules, and technology-specific primitives with unused output ports during optimization.

By default, the synthesis tool removes any module that does not drive logic as part of the synthesis optimization process. If you want to keep such an instance in the design, use the syn_noprune directive on the instance or module, along with syn_hier set to hard.

The syn_noprune directive does not prevent a hierarchy from being dissolved or flattened. To ensure that hierarchies are preserved in a design with multiple hierarchies, you must specify the syn_noprune directive and set syn_hier to fixed for all levels of the hierarchy. See Verilog Example 3: Hierarchical Design, on page 113 for an example.

For further information about this and other directives used for preserving logic, see Comparison of syn_keep, syn_preserve, and syn_noprune, on page 81, and Preserving Objects from Being Optimized Away, on page 335 in the *User Guide*.

syn_noprune Syntax

Verilog	*object* **/* synthesis syn_noprune = 1 */ ;**	Verilog Examples
VHDL	**attribute syn_noprune : boolean** **attribute syn_noprune of** *object* **:** *objectType* **is true;**	VHDL Examples

Verilog Examples

This section contains code snippets and an example.

Verilog Example 1: Module Declaration

syn_noprune can be applied in two places: on the module declaration of syn_noprune or in the top-level instantiation. The most common place to use syn_noprune is in the declaration of the module. By placing it here, all instances of the module are protected.

```
module syn_noprune (a,b,c,d,x,y); /* synthesis syn_noprune=1 */;

// Other code
```

The results for this example are shown in Effects of using syn_noprune: Example 1, on page 118.

```
my_design
   my_design1 (out, in, clk_in) /* synthesis syn_noprune=1 */,
   my_design2 (out, in, clk_in),
   my_design3 (out, in, clk_in) /* synthesis syn_noprune=1 */;

module top(a1,b1,c1,d1,y1,clk);
output y1;
input a1,b1,c1,d1;
input clk;
wire x2,y2;
reg y1;
syn_noprune u1(a1,b1,c1,d1,x2,y2) /* synthesis syn_noprune=1 */;

always @(posedge clk)
   y1<= a1;

endmodule
```

```
module syn_noprune (a,b,c,d,x,y)/* synthesis syn_hier="hard" */;
output x,y;
input a,b,c,d;
endmodule
```

Verilog Example 2: Black Box Declaration

Here is a snippet showing syn_noprune used on black box instances. If your design uses multiple instances with a single module declaration, the synthesis comment must be placed before the comma (,) following the port list for each of the instances.

```
my_design my_design1(out,in,clk_in) /* synthesis syn_noprune=1 */;
my_design my_design2(out,in,clk_in) /* synthesis syn_noprune=1 */;
```

In this example, only the instance my_design2 will be removed if the output port is not mapped.

The results for the following code example, where syn_noprune is used on an instance and a black box, is shown in Effects of Using syn_noprune: Example 2, on page 119.

```
module top
   (input a, b, c, d, e, clk,
   output o1);
reg o2_noprunereg /* synthesis syn_noprune = 1*/ ;
wire o3_wire;
assign o1 = a & b;
always @(posedge clk)
   begin
      o2_noprunereg = c & d & e;
   end
noprune_bb U1 (a, o3_wire) /* synthesis syn_noprune = 1*/ ;
endmodule
module noprune_bb ( input in1, output o1 );
endmodule
```

Verilog Example 3: Hierarchical Design

In the example below, syn_noprune1 and syn_noprune2 are intermediate modules in a hierarchical design. You must apply syn_hier = fixed attribute to them if you want the lowest-level modules, syn_noprune3 and syn_noprune4, to be preserved.

```
module top(a1,b1,c1,d1,y1,clk,a2,b2,c2,d2);
   output y1;
   input a1,b1,c1,d1;
   input a2,b2,c2,d2;
   input clk;
   wire x2,y2,x3,y3;
   reg y1;
   syn_noprune1 u1(a1,b1,c1,d1,x2,y2);
   syn_noprune1 u2(a2,b2,c2,d,x3,y3);
always @(posedge clk)
y1<= a1;
endmodule

module syn_noprune1 (a,b,c,d,x,y)/* synthesis syn_noprune=1
      syn_hier = "fixed" */;
   output x,y;
   input a,b,c,d;

   syn_noprune2 uut (.*);
endmodule

module syn_noprune2 (a,b,c,d,x,y)/* synthesis syn_noprune=1
      syn_hier = "fixed"*/;
   output x,y;
   input a,b,c,d;

   syn_noprune3 uut1 (.*);
   syn_noprune4 uut2 (.*);
endmodule

module syn_noprune3 (a,b,c,d,x)/* synthesis syn_black_box
      syn_noprune=1 */;
   output x;
   input a,b,c,d;
endmodule

module syn_noprune4 (a,b,c,d,y)/* synthesis syn_black_box
      syn_noprune=1 */;
   output y;
   input a,b,c,d;
endmodule
```

VHDL Examples

This section contains code snippets and an example.

Architecture Declaration

The syn_noprune attribute is normally associated with the names of architectures. Once it is associated, any component instantiation of the architecture (design unit) is protected from being deleted.

```
library synplify;
architecture mydesign of rtl is

attribute syn_noprune : boolean;
attribute syn_noprune of mydesign : architecture is true;

-- Other code
```

Component Declaration

Here is an example:

```
architecture top_arch of top is
component gsr
   port (gsr : in std_logic);
end component;

attribute syn_noprune : boolean;
attribute syn_noprune of gsr: component is true;
```

See Instantiating Black Boxes in VHDL, on page 552, for more information.

Component Instance Declaration

The syn_noprune attribute works the same on component instances as with a component declaration.

```
architecture top_arch of top is
component gsr
   port (gsr : in bit);
end component;

attribute syn_noprune : boolean;
attribute syn_noprune of u1_gsr: label is true;
```

Example 1

The results for this example are shown in Effects of using syn_noprune: Example 1, on page 118.

```
my_design
   my_design1 (out, in, clk_in) /* synthesis syn_noprune=1 */,
   my_design2 (out, in, clk_in),
   my_design3 (out, in, clk_in) /* synthesis syn_noprune=1 */;

module top(a1,b1,c1,d1,y1,clk);
output y1;
input a1,b1,c1,d1;
input clk;
wire x2,y2;
reg y1;
syn_noprune u1(a1,b1,c1,d1,x2,y2) /* synthesis syn_noprune=1 */;

always @(posedge clk)
   y1<= a1;

endmodule

module syn_noprune (a,b,c,d,x,y)/* synthesis syn_hier="hard" */;
output x,y;
input a,b,c,d;
endmodule

library ieee;
use ieee.std_logic_1164.all;

entity noprune is
   port (a, b, c,d : in std_logic;
         x,y : out std_logic );
end noprune;

architecture behave of noprune is
attribute syn_hier : string;
attribute syn_hier of behave : architecture is "hard" ;
begin
   x <= a and b;
   y <= c and d;
end behave;

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

```
entity top is
   port (a1, b1 : in std_logic;
         c1,d1,clk : in std_logic;
         y1 :out std_logic );
end ;

architecture behave of top is
component noprune
port (a, b, c, d : in std_logic;
      x,y : out std_logic );
end component;

signal x2,y2 : std_logic;
attribute syn_noprune : boolean;
attribute syn_noprune of u1 : label is true;
begin
   u1: noprune port map(a1, b1, c1, d1, x2, y2);
   process begin
      wait until (clk = '1') and clk'event;
      y1 <= a1;
   end process;
end;
```

VHDL Black Box Example

The results for this example are shown in Effect of Using syn_noprune: Example 3, on page 120.

Example 3

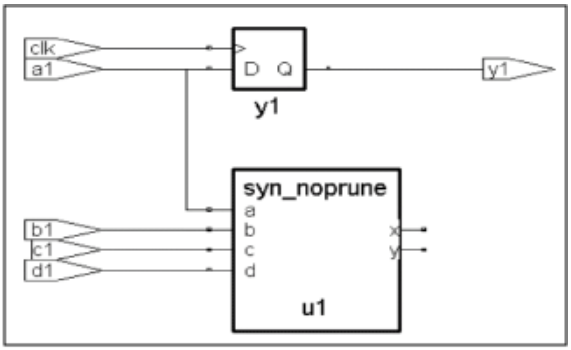
```
library ieee;
use ieee.std_logic_1164.all;

entity top is
   port (
   clk : in  std_logic;
   a, b, c, d : in std_logic;
   out_a : out std_logic);
end entity top;

architecture arch of top is
   component noprune_bb
      port(
      din : in std_logic;
      dout : out std_logic);
   end component noprune_bb;
```

```
   signal o1_noprunereg : std_logic;
   signal o2_reg : std_logic;

   attribute syn_noprune : boolean;
   attribute syn_noprune of U1: label is true;
   attribute syn_noprune of o1_noprunereg : signal is true;

begin
   process(clk)
      begin
         if rising_edge(clk) then
            o1_noprunereg <= b and  c;
            out_a <= a;
      end if;
   end process;
   U1: noprune_bb port map (d, o2_reg);
end architecture arch;
```

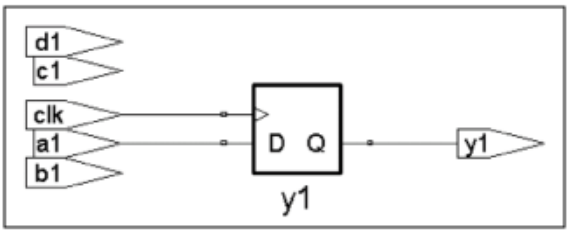
Effects of using syn_noprune: Example 1

The following figure shows the HDL Analyst view for two versions of a design: one version using syn_noprune on black box instance U1, one version without syn_noprune.

With syn_noprune, module U1 is preserved in the design. Without syn_noprune, the module is optimized away. See the examples in Verilog Examples, on page 112and VHDL Examples, on page 115 for the corresponding code.



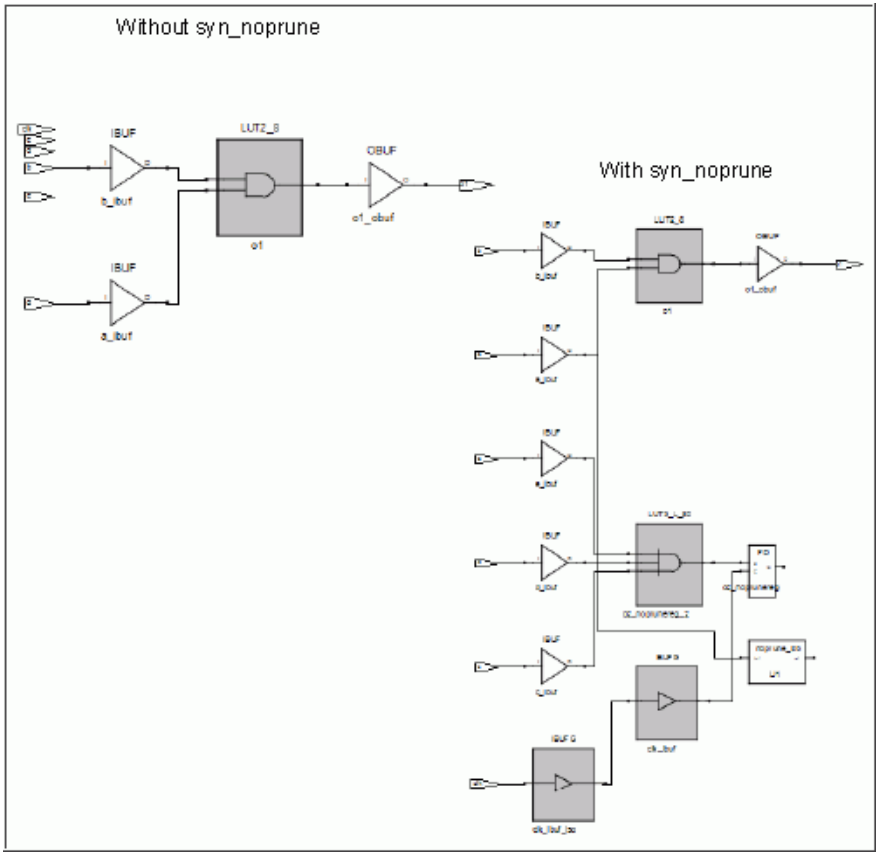
With syn_noprune



Without syn_noprune

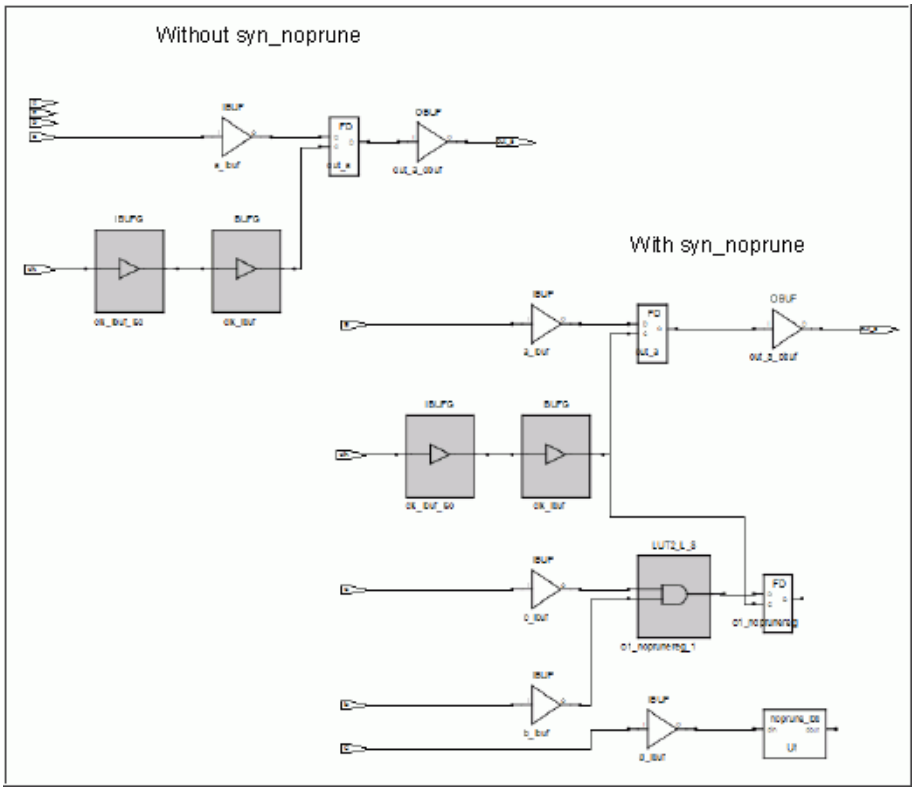
Effects of Using syn_noprune: Example 2

The following Technology views show that the instance and black box module are not optimized away when syn_noprune is applied. For the corresponding Verilog code, see Verilog Example 2: Black Box Declaration, on page 113.



Effect of Using syn_noprune: Example 3

In the following VHDL code example, syn_noprune is applied on both an instance and black box module with unused outputs. For the corresponding code, see For the corresponding VHDL code, see VHDL Black Box Example, on page 117.



syn_pad_type

Attribute

Specifies an I/O buffer standard.

Vendor	Technology
Microsemi	Axcelerator, IGLOO, and ProASIC and newer families

syn_pad_type Values

Value	Description
{buffer}_{standard} For example: IBUF_LVCMOS_18	Specifies the port I/O standard.

Description

Specifies an I/O buffer standard. Refer to I/O Standards, on page 183 and to the vendor-specific documentation for a list of I/O buffer standards available for the selected device family.

syn_pad_type Syntax

Default	Global Attribute	Object
Not Applicable	No	Port

FDC	**define_io_standard -default** *portType* **{***port***} -delay_type** *portType* **syn_pad_type {***io_standard***}** For example: `define_io_standard {p} -delay_type output syn_pad_type {LVCMOS_18}`	FDC Example
Verilog	*object* ***/* synthesis syn_pad_type =*** *io_standard* ****/***	Verilog Example
VHDL	**attribute syn_pad_type of** *object* : *objectType* **is** *io_standard* **;**	VHDL Example

FDC Example

	Enable	Object Type	Object	Attribute	Value
1	☑	port	p:output	syn_pad_type	LVCMOS_18
2					

-default_portType_	*PortType* can be input, output, or bidir. Setting default_input, default_output, or default_bidir causes all ports of that type to have the same I/O standard applied to them.
-delay_type *portType*	*PortType* can be input, output, or bidir.
syn_pad_type {*io_standard***}**	Specifies I/O standard (see following table).

Constraint File Examples

To set...	Use this syntax...
The default for all input ports to the AGP1X pad type	`define_io_standard -default_input -delay_type input syn_pad_type {AGP1X}`
All output ports to the GTL pad type	`define_io_standard -default_output -delay_type output syn_pad_type {GTL}`
All bidirectional ports to the CTT pad type	`define_io_standard -default_bidir -delay_type bidir syn_pad_type {CTT}`

The following are examples of pad types set on individual ports. You cannot assign pad types to bit slices.

```
define_io_standard {in1} -delay_type input
   syn_pad_type {LVCMOS_15}

define_io_standard {out21} -delay_type output
   syn_pad_type {LVCMOS_33}

define_io_standard {bidirbit} -delay_type bidir
   syn_pad_type {LVTTL_33}
```

Verilog Example

```
module top (clk,A,B,PC,P);

input clk;
input A ;
input B,PC;
output reg P/* synthesis syn_pad_type = "OBUF_LVCMOS_18" */;

reg a_d,b_d;
reg m;

always @(posedge clk)
   begin
      a_d <= A;
      b_d <= B;
      m   <= a_d + b_d;
      P   <= m + PC;
   end

endmodule
```

VHDL Example

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

library synplify;
use synplify.attributes.all;

entity top is
   port (clk : in std_logic ;
   A : in std_logic_vector(1 downto 0);
```

```
   B : in std_logic_vector(1 downto 0);
   PC : in std_logic_vector(1 downto 0);
   P : out std_logic_vector(1 downto 0));

attribute syn_pad_type : string;
attribute syn_pad_type of P : signal is "OBUF_LVCMOS_18";
end top ;

architecture rtl of top is
signal m : std_logic_vector(1 downto 0);

begin
   process(clk)
      begin
         if (clk'event and clk = '1') then
            m <= A + B;
            P <= m + PC;
         end if ;
   end process ;
end rtl ;
```

Effect of Using syn_pad_type

The following figure shows the netlist output after the attribute is applied:

Verilog	`output reg P /*synthesis syn_pad_type = "OBUF_LVCMOS_18"*/;`
VHDL	`attribute syn_pad_type of P : signal is "OBUF_LVCMOS_18";`

Net list

```
95          )
96          (instance m_2_4 (viewRef PRIM (cellRef LUT2_L (libraryRef VIRTEX)))
97           (property INIT (string "4'h6"))
98          )
99          (instance P_2_2 (viewRef PRIM (cellRef LUT2_L (libraryRef VIRTEX)))
100          (property INIT (string "4'h6"))
101         )
102         (instance P_obuf (viewRef PRIM (cellRef OBUF (libraryRef VIRTEX)))
103          (property IOSTANDARD (string "LVCMOS18"))
104         )
105         (instance PC_ibuf (viewRef PRIM (cellRef IBUF (libraryRef VIRTEX)))
```

P&R Files

```
We can see the effect of syn_pad_type in the following P&R files

<projectdirectory>\rev_1\pr_1\top.pad(412):
T17|P|IOB|IO_L1P_GC_24|OUTPUT|LVCMOS18|24|12|SLOW||||UNLOCATED|NO|NONE|

<projectdirectory>\rev_1\pr_1\top_pad.txt(413
|T17  |P    |IOB    |IO_L1P_GC_24    |OUTPUT  |LVCMOS18  |24   |12    |SLOW   |  |  |  |UNLOCATED |NO
```

syn_preserve

Directive

Prevents sequential optimizations such as constant propagation, inverter push-through, and FSM extraction.

syn_preserve Values

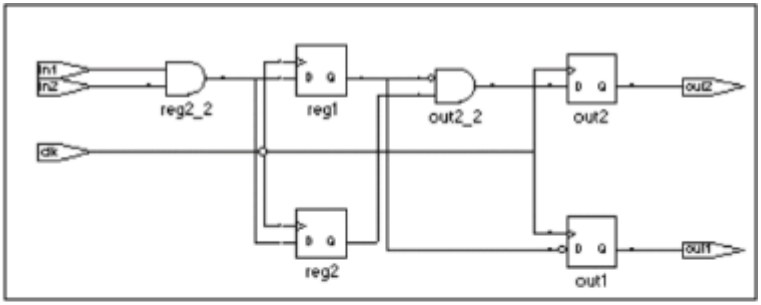
Value	Description
1 | true	Preserves register logic.
0 | false (Default)	Optimizes registers as needed.

Description

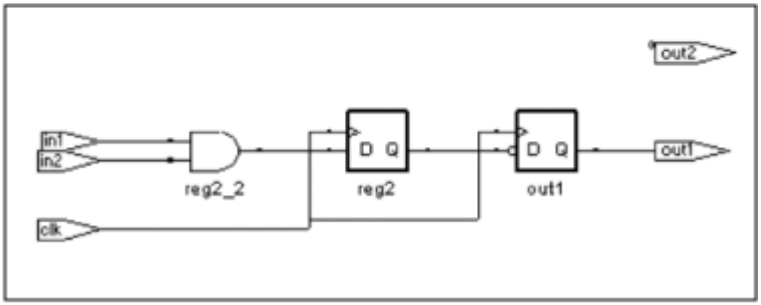
The syn_preserve directive controls whether objects are optimized away. Use syn_preserve to retain registers for simulation, or to preserve the logic of registers driven by a constant 1 or 0. You can set syn_preserve on individual registers or on the module/architecture so that the directive is applied to all registers in the module.

For example, assume that the input of a flip-flop is always driven to the same value, such as logic 1. By default, the synthesis tool ties that signal to VCC and removes the flip-flop. Using syn_preserve on the registered signal prevents the removal of the flip-flop. This is useful when you are not finished with the design but want to do a preliminary run to find the area utilization.

Another use for this attribute is to preserve a particular state machine. When the FSM compiler is enabled, it performs various state-machine optimizations. Use syn_preserve to retain a particular state machine and prevent it from being optimized away.



With syn_preserve



Without syn_preserve

When registers are removed during synthesis, the tool issues a warning message in the log file. For example:

```
@W:...Register bit out2 is always 0, optimizing ...
```

The syn_preserve directive is similar to syn_keep and syn_noprune, in that it preserves logic. For more information, see Comparison of syn_keep, syn_preserve, and syn_noprune, on page 81, and Preserving Objects from Being Optimized Away, on page 335 in the *User Guide*.

syn_preserve Syntax

Verilog	*object* **/* synthesis syn_preserve = 0 |1 */**	Verilog Example
VHDL	**attribute syn_preserve of** *object* : *objectType* **is true | false;**	VHDL Examples

Verilog Example

In the following example, syn_preserve is applied to all registers in the module to prevent them from being optimized away. For the results, see Effect of using syn_preserve, on page 131.

```
module mod_preserve (out1,out2,clk,in1,in2)
   /* synthesis syn_preserve=1 */;
output out1, out2;
input clk;
input in1, in2;
reg out1;
reg out2;
reg reg1;
reg reg2;

always@ (posedge clk)begin
reg1 <= in1 &in2;
reg2 <= in1&in2;
out1 <= !reg1;
out2 <= !reg1 & reg2;
end
endmodule
```

This is an example of setting syn_preserve on a state register:

```
reg [3:0] curstate /* synthesis syn_preserve = 1 */ ;
```

VHDL Examples

This section contains some VHDL code examples:

Example 3

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

```
entity simpledff is
   port (q : out std_logic_vector(7 downto 0);
         d : in std_logic_vector(7 downto 0);
         clk : in std_logic );

-- Turn on flip-flop preservation for the q output
attribute syn_preserve : boolean;
attribute syn_preserve of q : signal is true;
end simpledff;

architecture behavior of simpledff is
begin
   process(clk)
   begin
      if rising_edge(clk) then
   -- Notice the continual assignment of "11111111" to q.
         q <= (others => '1');
      end if;
   end process;
end behavior;
```

Example 2

In this example, syn_preserve is used on the signal curstate that is later used in a state machine to hold the value of the state register.

```
architecture behavior of mux is
begin
signal curstate : state_type;
attribute syn_preserve of curstate : signal is true;

-- Other code
```

Example 3

The results for the following example are shown in Effect of using syn_preserve, on page 131.

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

entity mod_preserve is
   port (out1 : out std_logic;
         out2 : out std_logic;
         in1,in2,clk : in std_logic );
end mod_preserve;
```

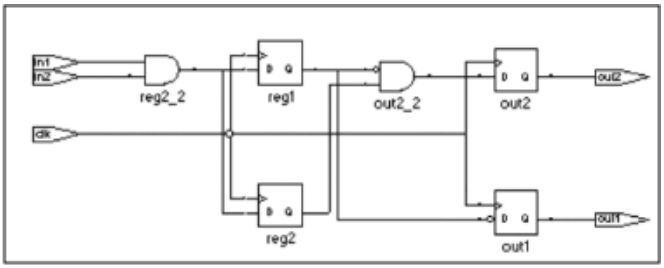
```
architecture behave of mod_preserve is
attribute syn_preserve : boolean;
attribute syn_preserve of behave: architecture is true;
signal reg1 : std_logic;
signal reg2 : std_logic;
begin
   process
   begin
      wait until clk'event and clk = '1';
      reg1 <= in1 and in2;
      reg2 <= in1 and in2;
      out1 <= not (reg1);
      out2 <= (not (reg1) and reg2) ;
   end process;
end behave;
```

Effect of using syn_preserve

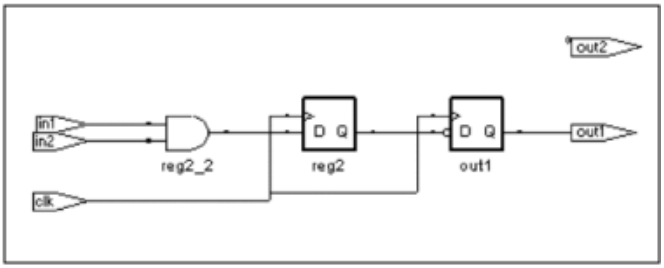
The following figure shows reg1 and out2 are preserved during optimization with syn_preserve.

When syn_preserve is not set, reg1 and reg2 are shared because they are driven by the same source. out2 gets the result of the AND of reg2 and NOT reg1. This is equivalent to the AND of reg1 and NOT reg1, which is a 0. As this is a constant, the tool removes out2 and the output out2 is always 0.

Verilog	mod_preserve /* synthesis syn_preserve = 1 */
VHDL	attribute syn_preserve of behave : architecture is true;



With syn_preserve



Without syn_preserve

syn_probe

Attribute.

Inserts probe points for testing and debugging the internal signals of a design.

syn_probe Values

Value	Description
1/true	Inserts a probe, and automatically derives a name for the probe port from the net name.
0/false	Disables probe generation.
portName	Inserts a probe and generates a port with the specified name. If you include empty square brackets, [ ], the probe names are automatically indexed to the net name.

Description

syn_probe works as a debugging aid, inserting probe points for testing and debugging the internal signals of a design. The probes appear as ports at the top level. When you use this attribute, the tool also applies syn_keep to the net.

You can specify values to name probe ports and assign pins to named ports for selected technologies. Pin-locking properties of probed nets will be transferred to the probe port and pad. If empty square brackets [] are used, probe names will be automatically indexed, according to the index of the bus being probed.

The table below shows how to apply syn_probe values to nets, buses, and bus slices. It indicates what port names will appear at the top level. When the syn_probe value is 0, probe generation is disabled; when syn_probe is 1, the probe port name is derived from the net name.

Net Name	syn_probe Value	Probe Port	Comments
n:ctrl	1	ctrl_probe_1	Probe port name generated by the synthesis tool.
n:ctr	test_pt	test_pt	For string values on a net, the port name is identical to the syn_probe value.
n:aluout[2]	test_pt	test_pt	For string values on a bus slice, the port name is identical to the syn_probe value.
n:aluout[2]	test_pt[ ]	test_pt[2]	The empty square brackets [ ] indicate that port names will be indexed to net names.
n:aluout[2:0]	test_pt[ ]	test_pt[2] test_pt[1] test_pt[0]	The empty square brackets [ ] indicate that port names will be indexed to net names.
n:aluout[2:0]	test_pt	test_pt, test_pt_0, test_pt_1	If a syn_probe value without brackets is applied to a bus, the port names are adjusted.

syn_probe Syntax

Global	Object	Default
No	Net	None

The following table shows the syntax used to define this attribute in different files:

FDC	**define_attribute {n:***netName***} syn_probe {***probePortname***|1|0}**	FDC Example
Verilog	*object* **/* synthesis syn_probe = "***string***" | 1 | 0 */ ;**	Verilog Example
VHDL	**attribute syn_probe of** *object* **: signal is "***string***" | 1 | 0 ;**	VHDL Example

FDC Example

The following examples insert a probe signal into a net and assign pin locations to the ports.

```
define_attribute {n:inst2.DATA0_*[7]} syn_probe {test_pt[]}
define_attribute {n:inst2.DATA0_*[7]} syn_loc
   {14,12,11,5,21,18,16,15}
```

Enable	Object Type	Object	Attribute	Value	Value Type	Description
☑	<any>	<Global>	syn_probe	1	string	Send a signal to out...

Verilog Example

The following example inserts probes on bus alu_tmp [7:0] and assign pin locations to each of the ports inserted for the probes.

```
module alu(out1, opcode, clk, a, b, sel);
output [7:0] out1;
input [2:0] opcode;
input [7:0] a, b;
input clk, sel;
reg [7:0] alu_tmp /* synthesis syn_probe="alu1_probe[]"
   syn_loc="A5,A6,A7,A8,A10,A11,A13,A14" */;
reg [7:0] out1;
// Other code
always @(opcode or a or b or sel)
begin
   case (opcode)
      3'b000:alu_tmp <= a+b;
      3'b000:alu_tmp <= a-b;
      3'b000:alu_tmp <= a^b;
      3'b000:alu_tmp <= sel ? a:b;
      default:alu_tmp <= a|b;
   endcase
end

always @(posedge clk)
out1 <= alu_tmp;
endmodule
```

VHDL Example

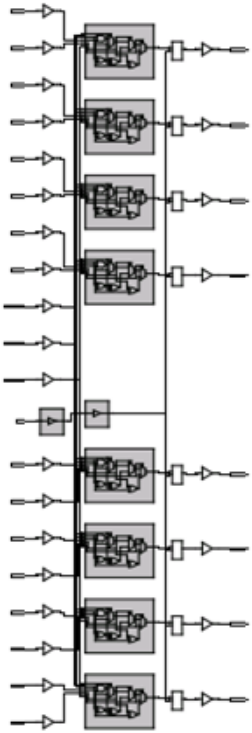
The following example inserts probes on bus alu_tmp(7 downto 0) and assigns pin locations to each of the ports inserted for the probes.

```
library ieee;
use ieee.std_logic_1164.all;
entity alu is
port ( a : in std_logic_vector(7 downto 0);
      b : in std_logic_vector(7 downto 0);
   opcode : in std_logic_vector(2 downto 0);
      clk : in std_logic;
   out1 : out std_logic_vector(7 downto 0) );
end alu;
architecture rtl of alu is
signal alu_tmp : std_logic_vector (7 downto 0);


attribute syn_probe : string;
attribute syn_probe of alu_tmp : signal is "test_pt";
attribute syn_loc : string;
attribute syn_loc of alu_tmp : signal is
   "A5,A6,A7,A8,A10,A11,A13,A14";


begin
   process (clk)
      begin
         if (clk'event and clk = '1') then
         out1 <= alu_tmp;
         end if;
      end process;
   process (opcode,a,b)
      begin
         case opcode is
         when "000"   =>  alu_tmp <= a and b;
         when "001"   =>  alu_tmp <= a or b;
         when "010"   =>  alu_tmp <= a xor b;
         when "011"   =>  alu_tmp <= a nand b;
         when others  =>  alu_tmp <= a nor b;
      end case;
   end process;


end rtl;
```

Effect of Using syn_probe

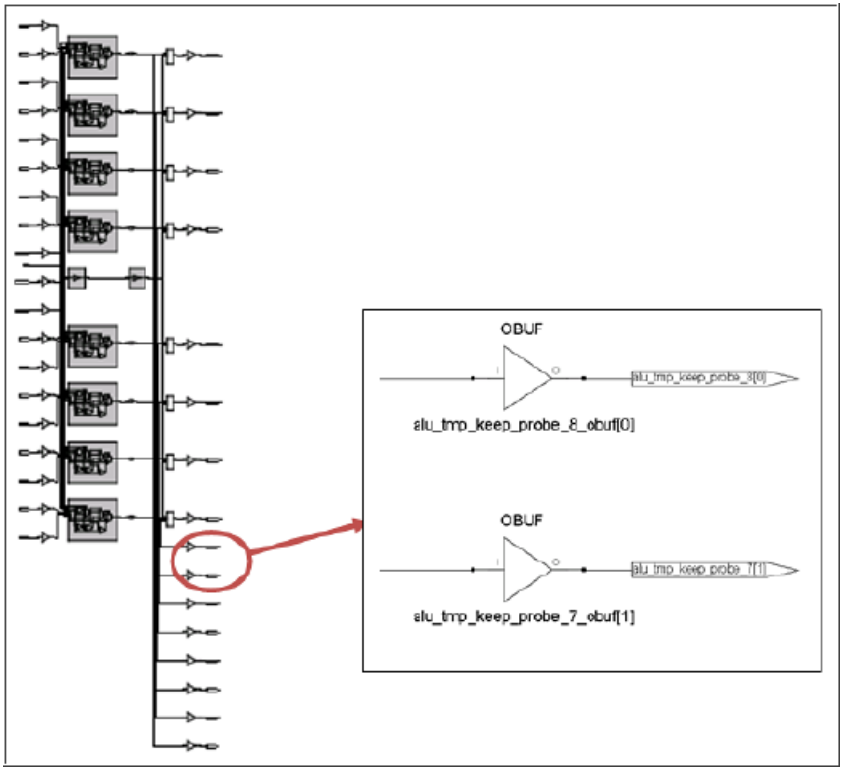
Before applying syn_probe:

Verilog	reg [7:0] alu_tmp /* synthesis syn_probe="0"*/
VHDL	attribute syn_probe of alu_tmp : signal is "0";



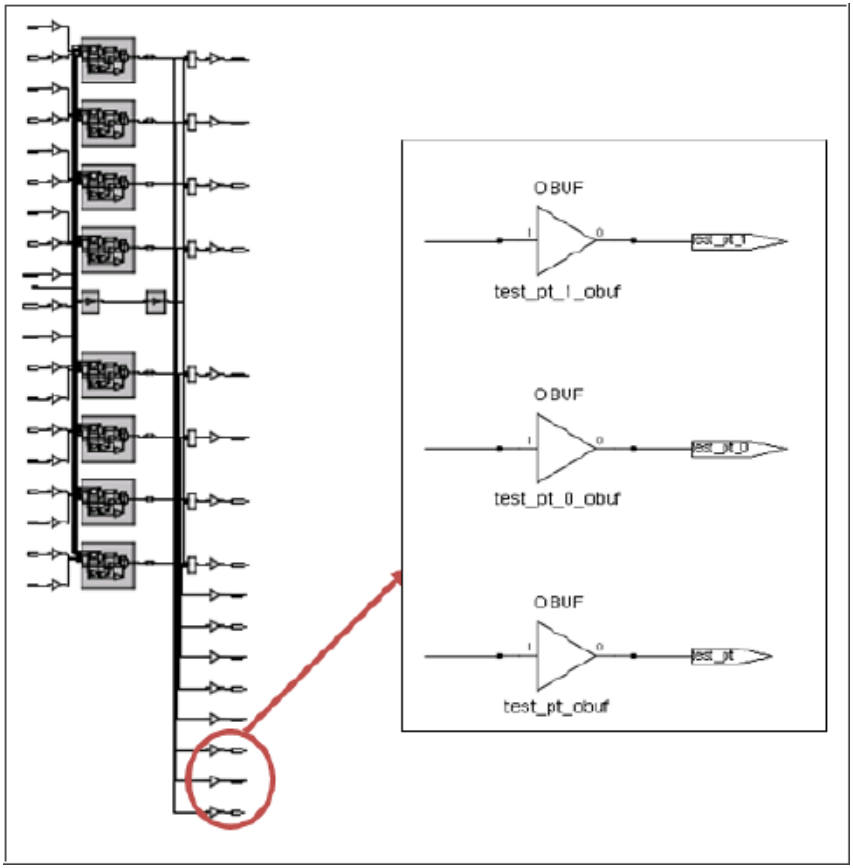
After applying syn_probe with “1”:

Verilog	reg [7:0] alu_tmp /* synthesis syn_probe="1"*/
VHDL	attribute syn_probe of alu_tmp : signal is "1";



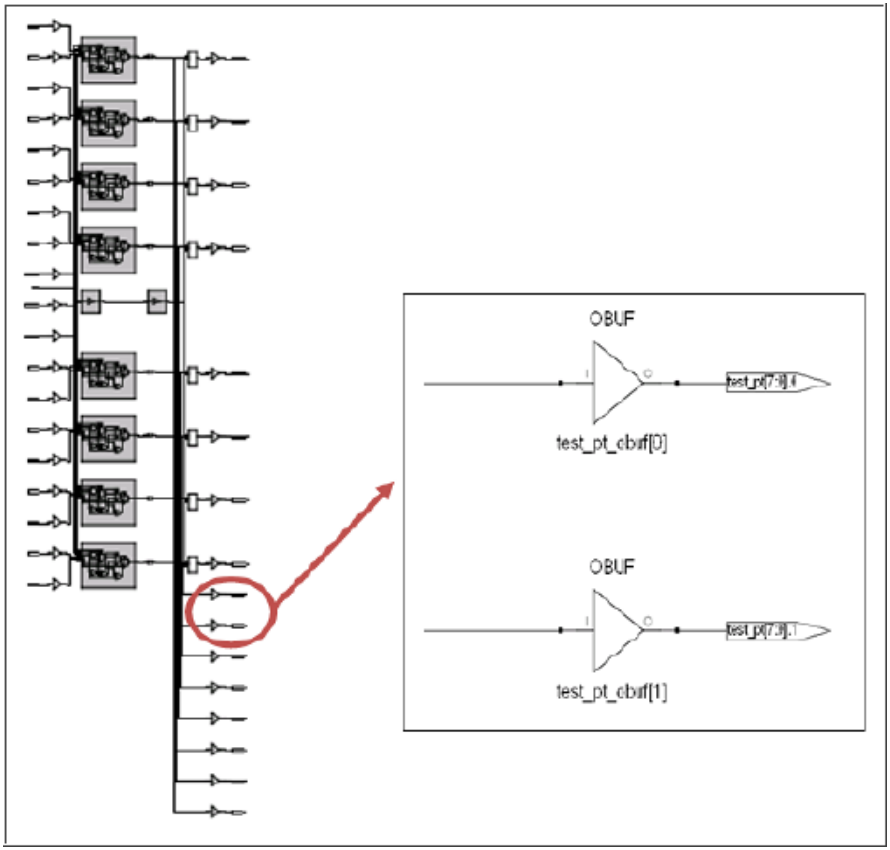
After applying syn_probe with “test_pt”:

Verilog	reg [7:0] alu_tmp /* synthesis syn_probe="test_pt"*/
VHDL	attribute syn_probe of alu_tmp : signal is "test_pt";



After applying syn_probe with “test_pt[]”:

Verilog	reg [7:0] alu_tmp /* synthesis syn_probe="test_pt[]"*/
VHDL	attribute syn_probe of alu_tmp : signal is "test_pt[]";



syn_radhardlevel

Attribute.

Implements designs with high reliability, using radiation-resistant techniques.

Vendor	Technologies	Tool
Microsemi	Anti-fuse (RT, RH and RD radhard devices) ProASIC3, ProASIC3E, ProASIC3L, IGLOO2, SmartFusion2	Synplify Pro

Some high reliability techniques are not available or appropriate for all Microsemi families. Use a design technique that is valid for the project. Contact Microsemi technical support for details.

You can apply syn_radhardlevel globally to the top-level module/architecture or on an individual register output signal (or inferred register in VHDL), and the tool uses the attribute value in conjunction with the Microsemi macro files supplied with the software. For more details about using this attribute, see Specifying syn_radhardlevel in the Source Code, on page 482 and Working with Radhard Designs, on page 481 in the *User Guide*.

syn_radhardlevel = none | cc | tmr | tmr_cc

Value	Description
none	Default. Uses standard design techniques, and does not insert any triple register logic.
cc	Microsemi Anti-fuse Implements combinational cells with feedback as storage, rather than flip-flop or latch primitives.
tmr	Microsemi Anti-fuse, ProASIC3, ProASIC3E, ProASIC3L, SmartFusion2, IGLOO2 Uses triple module redundancy or triple voting to implement registers. Each register is implemented by three flip-flops or latches that “vote” to determine the state of the register. This option can potentially affect area and timing QoR because of the additional logic inserted, so be sure to check your area and timing goals when you use this option.
tmr_cc	Microsemi Anti-fuse Uses triple module redundancy, where each voting register is composed of combinational cells with feedback rather than flip-flop or latch primitives

syn_radhardlevel Syntax (Microsemi)

Name	Global Attribute	Object
syn_radhardlevel	No	Module, architecture, register Verilog: output signal VHDL: architecture, signal

The following table summarizes the syntax in different files:

FDC	define_attribute {*object*} syn_radhardlevel {none|cc|trmr|tmr_cc}	Constraint File Example, on page 143
Verilog	*object* /* synthesis syn_radhardlevel = none|cc|trmr|tmr_cc */	Verilog syn_radhardlevel Example, on page 143
VHDL	attribute syn_rw_conflict_logic : boolean; attribute syn_rw_conflict_logic of *Object* : Object Type is none|cc|trmr|tmr_cc ;	VHDL syn_radhardlevel Example, on page 143

Constraint File Example

```
define_attribute {dataout[3:0]} syn_radhardlevel {tmr}
```

Verilog syn_radhardlevel Example

```
//Top level
module top (clk, dataout, a, b);
input clk;
input a;
input b;
output [3:0] dataout;
M1 inst_M1 (a1, M3_out1, clk, rst, M1_out);
// Other code

//Sub modules subjected to DTMR
module M1 (a1, a2, clk, rst, q)
   /* synthesis syn_radhardlevel="distributed_tmr" */;
input clk;
input signed [15:0] a1,a2;
input clk, rst;
output signed [31:0] q;
// Other code
```

VHDL syn_radhardlevel Example

See VHDL Attribute and Directive Syntax, on page 554 for alternate methods for specifying VHDL attributes and directives.

```
library synplify;
architecture top of top is
attribute syn_radhardlevel : string;
attribute syn_radhardlevel of top: architecture is "tmr";

-- Other code
```

syn_ramstyle

Attribute

Specifies the implementation for an inferred RAM.

Vendor	Devices
Microsemi	ProASIC3, Fusion, SmartFusion2 Older devices

syn_ramstyle Values

Default	Global Attribute	Object
block_ram	Yes	View, module, entity, RAM instance

The values for syn_ramstyle vary with the target technology. The following table lists all the valid syn_ramstyle values, some of which apply only to certain technologies. For details about using syn_ramstyle, see RAM Attributes, on page 309 in the *User Guide*.

block_ram	Specifies that the inferred RAM be mapped to the appropriate device-specific memory. It uses the dedicated memory resources in the FPGA. By default, the software uses deep block RAM configurations instead of wide configurations to get better timing results. Using deeper RAMs reduces the output data delay timing by reducing the MUX logic at the output of the RAMs. By default the software does not use the parity bit for data with this option. Alternatively, you can specify a *ramType* value. See RAM Type Values and Implementations, on page 145 for details of how memory is implemented for different devices.

no_rw_check	By default, the synthesis tool inserts bypass logic around the inferred RAM to avoid simulation mismatches caused by indeterminate output values when reads and writes are made to the same address. When this option is specified, the synthesis tool does not insert glue logic around the RAM. You can use this option on its own or in conjunction with a RAM type value such as M512, or with the power value for supported technologies. You cannot use it with the rw_check option, as the two are mutually exclusive. There are other read-write check controls. See Read-Write Address Checks, on page 146 for details about the differences.
ramType	Specifies a device-specific RAM implementation. Valid values vary from vendor to vendor as they are based on device architecture: • Microsemi: lsram, uram See RAM Type Values and Implementations, on page 145 for details of how memory is implemented for different devices.
registers	Specifies that an inferred RAM be mapped to registers (flip-flops and logic), not technology-specific RAM resources.
rw_check	When enabled, the synthesis tool inserts bypass logic around the RAM to prevent a simulation mismatch between the RTL and post-synthesis simulations. You can use this option on its own or in conjunction with a RAM type value such as M512, or with the power value for supported technologies. You cannot use it with the no_rw_check option, as the two are mutually exclusive. There are other read-write check controls. See Read-Write Address Checks, on page 146 for details about the differences.

RAM Type Values and Implementations

The table lists vendor-specify RAM implementation information, including vendor-specific *ramType* values.

Vendor	Values	Implementation	Technology
Microsemi		Default: block_ram	ProASIC3/ ProASIC3E/ ProASIC3L
	registers	Registers	
		Default: Registers	SmartFusion, Fusion IGLOO+, IGLOO IGLOOe

Vendor	Values	Implementation	Technology
	lsram	RAM1K18	SmartFusion2
	uram	RAM64X18	
	registers	Registers	

Description

The syn_ramstyle attribute specifies the implementation to use for an inferred RAM. You can apply the attribute globally, to a module, or a RAM instance. You can also use syn_ramstyle to prevent the inference of a RAM, by setting it to registers. If your RAM resources are limited, you can map additional RAMs to registers instead of RAM resources using this setting.

The syn_ramstyle values vary with the technology.

Read-Write Address Checks

When reads and writes are made to the same address, the output could be indeterminate, and this can cause simulation mismatches. The synthesis tool offers multiple ways to specify how to handle read-write address checking:

Read Write Control	Use when...
syn_ramstyle	You know your design does not read and write to the same address simultaneously and you want to specify the RAM implementation. The attribute has two mutually-exclusive read-write check options: • Use no_rw_check to eliminate bypass logic. If you enable global RAM inference with the Read Write Check on RAM option, you can use no_rw_check to selectively disable glue logic insertion for individual RAMs. • Use rw_check to insert bypass logic. If you disable global RAM inference with the Read Write Check on RAM option, you can use rw_check to selectively enable glue logic insertion for individual RAMs.
Read Write Check on RAM	You want to globally enable or disable glue logic insertion for all the RAMs in the design.

If there is a conflict, the software uses the following order of precedence:

- syn_ramstyle attribute settings
- Read Write Check on RAM option on the Device panel of the Implementation Options dialog box.

syn_ramstyle Syntax

FDC	define_attribute {*signalname* [*bitRange*]} -syn_ramstyle *value* define_global_attribute syn_ramstyle *value*	FDC Example
Verilog	*object* /* synthesis syn_ramstyle = *value* */	Verilog Example
VHDL	attribute syn_ramstyle of *object* : *objectType* is *value* ;	VHDL Example

FDC Example

	Enabled	Object Type	Object	Attribute	Value	Val Type	Description
1	✓	<any>	<global>	syn_ramstyle	select_ram	string	Special implementation of inferred RAM

If you edit a constraint file to apply syn_ramstyle, be sure to include the range of the signal with the signal name. For example:

```
define_attribute {mem[7:0]} syn_ramstyle {registers};

define_attribute {mem[7:0]} syn_ramstyle {block_ram};
```

Verilog Example

```
module ram4 (datain,dataout,clk);
output [31:0] dataout;
input clk;
input [31:0] datain;
reg [7:0] dataout[31:0] /* synthesis syn_ramstyle="block_ram" */;
// Other code
```

VHDL Example

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

```
entity ram4 is
   port (d : in std_logic_vector(7 downto 0);
         addr : in std_logic_vector(2 downto 0);
         we : in std_logic;
         clk : in std_logic;
         ram_out : out std_logic_vector(7 downto 0) );
end ram4;

library synplify;
architecture rtl of ram4 is
type mem_type is array (127 downto 0) of std_logic_vector (7
   downto 0);
signal mem : mem_type;
-- mem is the signal that defines the RAM

attribute syn_ramstyle : string;
attribute syn_ramstyle of mem : signal is "block_ram";

-- Other code
```

syn_reference_clock

Attribute.

Specifies a clock frequency other than the one implied by the signal on the clock pin of the register.

Vendor	Technology	Default Value	Global	Object
Microsemi	SmartFusion2, ProASIC3, older families	-	-	Register

Description

syn_reference_clock is a way to change clock frequencies other than by using the signal on the clock pin. For example, when flip-flops have an enable with a regular pattern, such as every second clock cycle, use syn_reference_clock to have timing analysis treat the flip-flops as if they were connected to a clock at half the frequency.

To use syn_reference_clock, define a new clock, then apply its name to the registers you want to change.

FDC	**define_attribute {***register***} syn_reference_clock {***clockName***}**	FDC Example

FDC Example

define_attribute {*register***} syn_reference_clock {***clockName***}**

For example:

```
define_attribute {myreg[31:0]} syn_reference_clock {sloClock}
```

You can also use syn_reference_clock to constrain multiple-cycle paths through the enable signal. Assign the find command to a collection (clock_enable_col), then refer to the collection when applying the syn_reference_clock constraint.

The following example shows how you can apply the constraint to all registers with the enable signal en40:

```
define_scope_collection clock_enable_col {find -seq * -filter
   (@clock_enable==en40)}
define_attribute {$clock_enable_col} syn_reference_clock {clk2}
```

Enable	Object Type	Object	Attribute	Value	Value Type	Description
☑	<any>	<Global>	syn_reference_clock	1	string	Override the default ...

Note: You apply syn_reference_clock only in a constraint file; you cannot use it in source code.

Effect of using syn_reference_clock

Before applying attribute:

```
Performance Summary
*******************

Worst slack in design: 499.379

                  Requested     Estimated     Requested     Estimated                 Clock        Clock
Starting Clock    Frequency     Frequency     Period        Period        Slack       Type         Group
-----------------------------------------------------------------------------------------------------------
clk               2.0 MHz       1609.5 MHz    500.000       0.621         499.379     declared     default_clkgroup_0
ref_clk           1.0 MHz       NA            1000.000      NA            NA          declared     default_clkgroup_1
===========================================================================================================
```

After applying attribute:

```
Performance Summary
*******************

Worst slack in design: 999.379

                  Requested     Estimated     Requested     Estimated                 Clock        Clock
Starting Clock    Frequency     Frequency     Period        Period        Slack       Type         Group
-----------------------------------------------------------------------------------------------------------
clk               2.0 MHz       NA            500.000       NA            NA          declared     default_clkgroup_0
ref_clk           1.0 MHz       1609.5 MHz    1000.000      0.621         999.379     declared     default_clkgroup_1
===========================================================================================================
```

syn_replicate

Attribute

Controls replication of registers during optimization.

Vendor	Technologies
Microsemi	SmartFusion and older families

syn_replicate values

Value	Default	Global	Object	Description
0	No	Yes	Register	Disables duplication of registers
1	Yes	Yes	Register	Allows duplication of registers

Description

The synthesis tool automatically replicates registers while optimizing the design and fixing fanouts, packing I/Os, or improving the quality of results.

If area is a concern, you can use this attribute to disable replication either globally or on a per-register basis. When you disable replication globally, it disables I/O packing and other QoR optimizations. When it is disabled, the synthesis tool uses only buffering to meet maximum fanout guidelines.

To disable I/O packing on specific registers, set the attribute to 0. Similarly, you can use it on a register between clock boundaries to prevent replication. Take an example where the tool replicates a register that is clocked by clk1 but whose fanin cone is driven by clk2, even though clk2 is an unrelated clock in another clock group. By setting the attribute for the register to 0, you can disable this replication.

syn_replicate Syntax Specification

FDC	define_global_attribute syn_replicate {0|1};	FDC Example
Verilog	*object* /* synthesis syn_replicate = 1 | 0 */;	Verilog Example
VHDL	attribute syn_replicate : boolean; attribute syn_replicate of *object* : signal is false;	VHDL Example

FDC Example

Enabled	Object Type	Object	Attribute	Value	Val Type	Description	Comment
✓	global	<global>	syn_replicate	0	boolean	Controls replication of registers	

Verilog Example

```
module norep (Reset, Clk, Drive, OK, ADPad, IPad, ADOut);
input Reset, Clk, Drive, OK;
input [6:0] ADOut;
inout [6:0] ADPad;
output [6:0] IPad;
reg [6:0] IPad;
reg DriveA /* synthesis syn_replicate = 0 */;
assign ADPad = DriveA ? ADOut : 32'bz;

always @(posedge Clk or negedge Reset)
   if (!Reset)
      begin
         DriveA <= 0;
         IPad   <= 0;
      end
   else
      begin
         DriveA <= Drive & OK;
         IPad   <= ADPad;
      end
endmodule
```

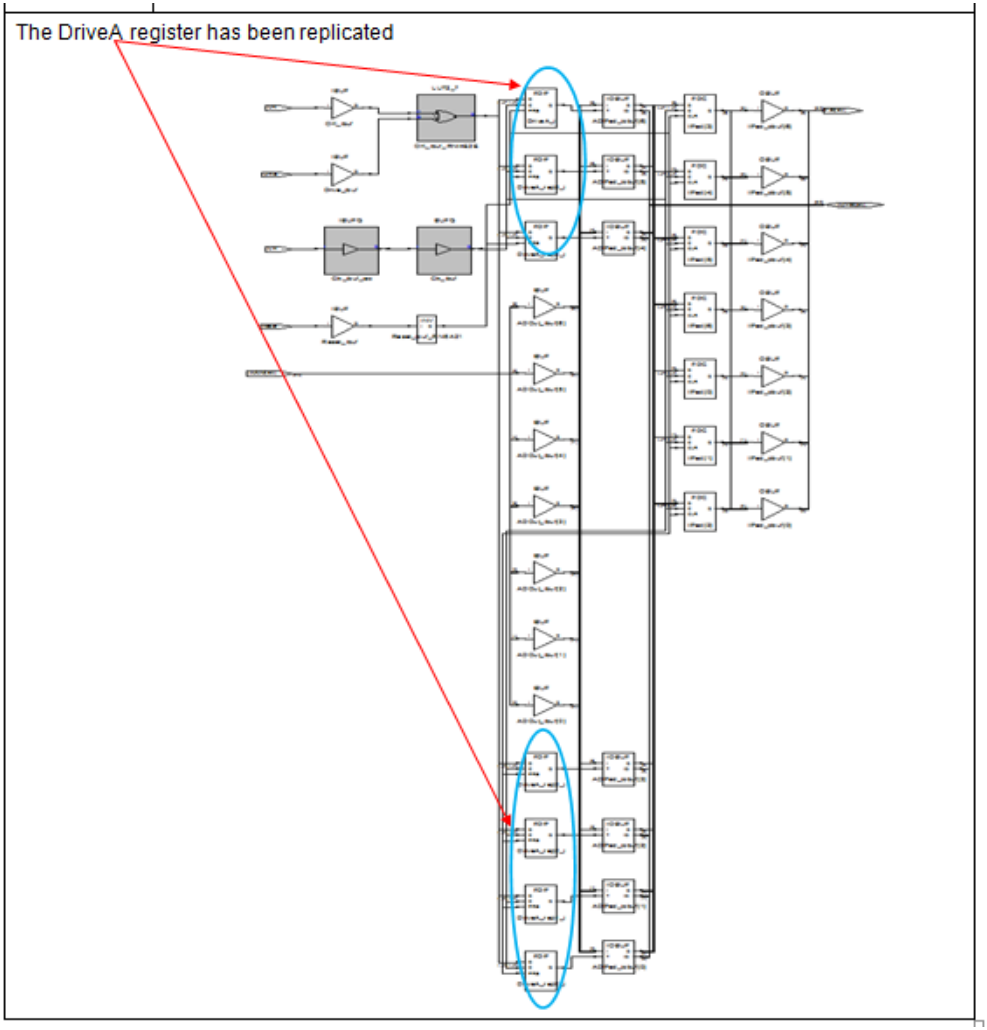
VHDL Example

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

entity norep is
   port (Reset : in std_logic;
         Clk : in std_logic;
         Drive : in std_logic;
         OK : in std_logic;
         ADPad : inout std_logic_vector (6 downto 0);
         IPad : out std_logic_vector (6 downto 0);
         ADOut : in std_logic_vector (6 downto 0) );
end norep;

architecture archnorep of norep is
signal DriveA : std_logic;
attribute syn_replicate : boolean;
attribute syn_replicate of DriveA : signal is false;
begin
ADPad <= ADOut when DriveA='1' else (others => 'Z');
   process (Clk, Reset)
   begin
      if Reset='0' then
         DriveA <= '0';
         IPad <= (others => '0');
      elsif rising_edge(clk) then
         DriveA <= Drive and OK;
         IPad <= ADPad;
      end if;
   end process;
end archnorep;
```

Effect of Using syn_replicate

The following example shows a design without the syn_replicate attribute:

Verilog	reg DriveA /*synthesis syn_replicate=1*/
VHDL	attribute syn_replicate : boolean; attribute syn_replicate of DriveA : signal is true;



When you apply syn_replicate, the registers are not duplicated:

Verilog	reg DriveA /*synthesis syn_replicate=0*/
VHDL	attribute syn_replicate : boolean; attribute syn_replicate of DriveA : signal is false;

syn_resources

Attribute

Microsemi ProASIC3, ProASIC3E, ProASIC3L, IGLOO, IGLOOe, IGLOO+, and Fusion

Specifies the resources used inside a black box. It is applied to Verilog black-box modules and VHDL architectures or component definitions.

Return to Summary of Attributes and Directives.

The value of the attribute is any combination of the following:

Value	Description
blockrams=*integer*	number of RAM resources
corecells=*integer*	number of core cells for Microsemi families only.

The Microsemi families only support resource values of blockrams and corecells.

Constraint File Syntax and Example

define_attribute {v:*moduleName***} syn_resources {blockrams=***integer***}**

define_attribute {v:*moduleName***} syn_resources {blockrams=***integer***|corecells=***integer***}**

You can apply the attribute to more than one kind of resource at a time by separating assignments by a comma (**,**). For example:

```
define_attribute {v:bb} syn_resources {blockrams=10}
define_attribute {v:bb} syn_resources {corecells=50,blockrams=20}
```

Verilog Syntax and Example

object **/* synthesis syn_resources = "***value***" */ ;**

In Verilog, you can only attach this attribute to a module. Here is an example:

```
module bb (o,i) /* synthesis syn_black_box syn_resources =
   "luts=500,regs=463,blockrams=10" */;
input i;
output o;
endmodule

module top_bb (o,i);
input i;
output o;
bb u1 (o,i);
endmodule
```

Verilog Syntax and Example (Microsemi)

object **/* synthesis syn_resources = "***value***" */ ;**

In Verilog, you can only attach this attribute to a module. Here is an example:

```
module bb (o,i) /* synthesis syn_black_box syn_resources =
   "corecells=10,blockrams=5" */;
input i;
output o;
endmodule

module top_bb (o,i);
input i;
output o;
bb u1 (o,i);
endmodule
```

VHDL Syntax and Example (Microsemi)

attribute syn_resources of *object* **:** *objectType* **is "***string***" ;**

See VHDL Attribute and Directive Syntax, on page 554 for different ways to specify VHDL attributes and directives. In VHDL, this attribute can be placed on either an architecture or a component declaration.

```
architecture top of top is
component decoder
  port (clk : in bit;
      a, b : in bit;
      qout : out bit_vector(7 downto 0) );
end component;
```

```
attribute syn_resources : string;
attribute syn_resources of decoder: component is
   "corecells=500,blockrams=10";

-- Other code
```

syn_sharing

Directive

Enables or disables the sharing of operator resources during the compilation stage of synthesis.

Technology	Default Value	Global	Object
All	On	Yes	Component, module

syn_sharing Values

Value	Description
off | false	Does not share resources during the compilation stage of synthesis.
on | true (Default)	Optimizes the design to perform resource sharing during the compilation stage of synthesis.

Description

The syn_sharing directive controls resource sharing during the compilation stage of synthesis. This is a compiler-specific optimization that does not affect the mapper; this means that the mapper might still perform resource sharing optimizations to improve timing, even if syn_sharing is disabled.

You can also specify global resource sharing with the Resource Sharing option in the Project view, from the Project->Implementation Options->Options panel, or with the set_option -resource_sharing Tcl command.

resource sharing globally, you can use the syn_sharing directive to turn on resource sharing for specific modules or architectures. See Sharing Resources, on page 345 in the *User Guide* for a detailed procedure.

syn_sharing Syntax

Verilog	*object* /* synthesis syn_sharing="on | off" */ ;	Verilog Example
VHDL	attribute syn_sharing of *object* : *objectType* is "true | false" ;	VHDL Example

Verilog Example

```
module add (a, b, x, y, out1, out2, sel, en, clk)
   /* synthesis syn_sharing=off */;

input a, b, x, y, sel, en, clk;
output out1, out2;
wire tmp1, tmp2;
assign tmp1 = a * b;
assign tmp2 = x * y;
reg out1, out2;

always@(posedge clk)
   if (en)
      begin
         out1 <= sel ?  tmp1: tmp2;
      end
   else
      begin
         out2 <= sel ? tmp1: tmp2;
   end
endmodule
```

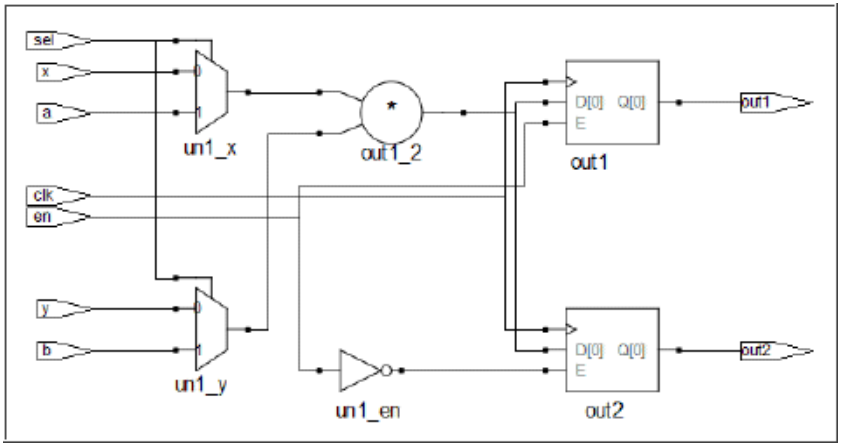
VHDL Example

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

entity add is
   port (a, b : in std_logic_vector(1 downto 0);
         x, y : in std_logic_vector(1 downto 0);
         clk, sel, en: in std_logic;
         out1 : out std_logic_vector(3 downto 0);
         out2 : out std_logic_vector(3 downto 0)
   );
end add;

architecture rtl of add is
signal tmp1, tmp2: std_logic_vector(3 downto 0);
begin
   tmp1 <= a * b;
   tmp2 <= x * y;

attribute syn_sharing : string;
attribute syn_sharing of add : component is “false”;

process(clk) begin
   if clk’event and clk=’1’ then
      if (en=’1’) then
         if (sel=’1’) then
            out1 <= tmp1;
         else
            out1 <= tmp2;
         end if;
      else
         if (sel=’1’) then
            out2 <= tmp1;
         else
            out2 <= tmp2;
         end if;
      end if;
   end if;
end process;
end rtl;
```

Effect of Using syn_sharing

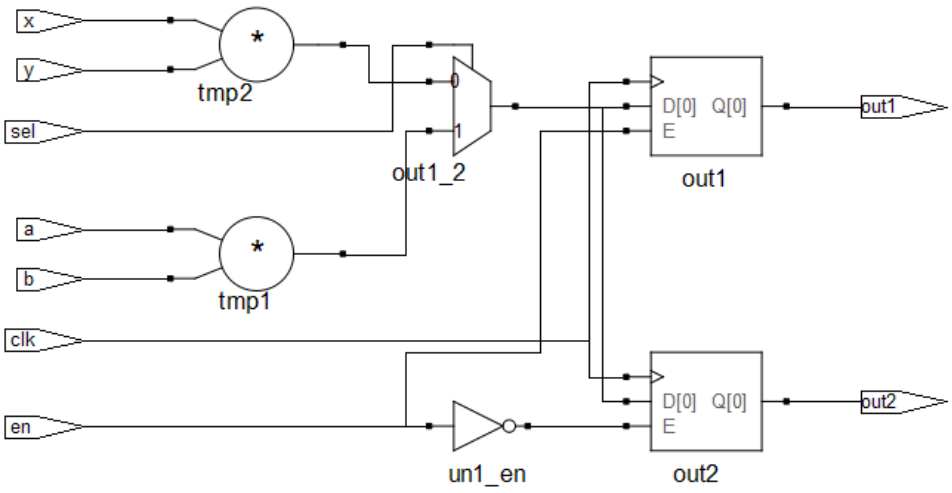
The following example shows the default setting, where resource sharing in the compiler is on:

Verilog	`module add /* synthesis syn_sharing = "on" */;`
VHDL	`attribute syn_sharing of add : component is "true" ;`



The next figure shows the same design when resource sharing is off, and two adders are inferred:

Verilog	`module add /* synthesis syn_sharing = "off" */;`
VHDL	`attribute syn_sharing of add : component is "false" ;`



syn_state_machine

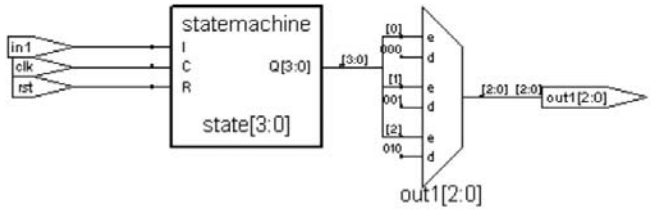
Directive

Enables/disables state-machine optimization on individual state registers in the design. When you disable the FSM Compiler, state-machines are not automatically extracted. To extract some state machines, use this directive with a value of 1 on just those individual state-registers to be extracted. Conversely, when the FSM Compiler is enabled and there are state machines in your design that you do not want extracted, use syn_state_machine with a value of 0 to override extraction on just those individual state registers.

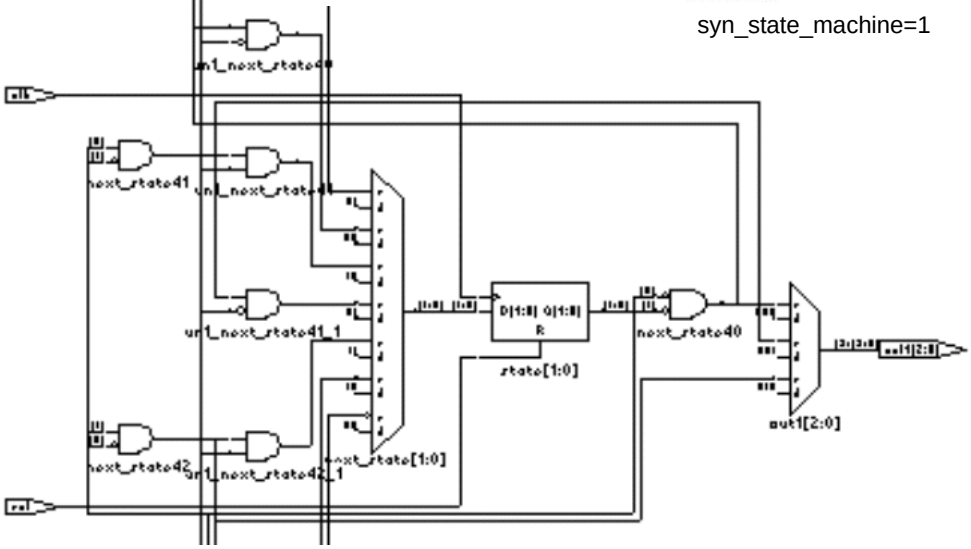
Also, when the FSM Compiler is enabled, all state machines are usually detected during synthesis. However, on occasion there are cases in which certain state machines are not detected. You can use this directive to declare those undetected registers as state machines.

The following figure shows an example of two implementations of a state machine: one with the syn_state_machine directive enabled, the other with the directive disabled.

syn_state_machine=0



syn_state_machine=1



See the following HDL syntax and example sections for the source code used to generate the schematics above. See also:

- syn_encoding, on page 43 for information on overriding default encoding styles for state machines.
- For VHDL designs, syn_encoding Compared to syn_enum_encoding, on page 56 for usage information about these two directives.

Verilog Syntax and Examples

object **/* synthesis syn_state_machine = 0 | 1 */ ;**

where *object* is a state register. Data type is Boolean: 0 does not extract an FSM, 1 extracts an FSM.

Following is an example of syn_state_machine applied to register OUT.

```
module prep3 (CLK, RST, IN, OUT);
input CLK, RST;
input [7:0] IN;
output [7:0] OUT;
reg [7:0] OUT;
reg [7:0] current_state /* synthesis syn_state_machine=1 */;

// Other code
```

Here is the source code used for the example in the previous figure.

```
module FSM1 (clk, in1, rst, out1);
input        clk, rst, in1;
output [2:0] out1;

`define s0 3'b000
`define s1 3'b001
`define s2 3'b010
`define s3 3'bxxx

reg [2:0] out1;
reg [2:0] state /* synthesis syn_state_machine = 1 */;
reg [2:0] next_state;

always @(posedge clk or posedge rst)
   if (rst) state <= `s0;
   else     state <= next_state;
```

```
// Combined Next State and Output Logic
always @(state or in1)
   case (state)
      `s0 : begin
         out1 <= 3'b000;
         if (in1) next_state <= `s1;
         else next_state <= `s0;
      end
      `s1 : begin
         out1 <= 3'b001;
         if (in1) next_state <= `s2;
         else next_state <= `s1;
      end
      `s2 : begin
         out1 <= 3'b010;
         if (in1) next_state <= `s3;
         else next_state <= `s2;
      end
      default : begin
         out1 <= 3'bxxx;
         next_state <= `s0;
      end
   endcase
endmodule
```

VHDL Syntax and Examples

attribute syn_state_machine of *object* **:** *objectType* **is true|false ;**

where *object* is a signal that holds the value of the state machine. For example:

```
attribute syn_state_machine of current_state: signal is true;
```

See VHDL Attribute and Directive Syntax, on page 554 for different ways to specify VHDL attributes and directives.

Following is the source code used for the example in the previous figure.

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

entity FSM1 is
   port (clk,rst,in1 : in std_logic;
         out1 : out std_logic_vector (2 downto 0) );
end FSM1;

architecture behave of FSM1 is
type state_values is ( s0, s1, s2,s3 );
signal state, next_state: state_values;
attribute syn_state_machine : boolean;
attribute syn_state_machine of state : signal is false;

begin
   process (clk, rst)
   begin
      if rst = '1' then
         state <= s0;
      elsif rising_edge(clk) then
         state <= next_state;
      end if;
   end process;

   process (state, in1) begin
      case state is
         when s0 =>
            out1 <= "000";
            if in1 = '1' then next_state <= s1;
               else next_state <= s0;
            end if;
         when s1 =>
            out1 <= "001";
            if in1 = '1' then next_state <= s2;
               else next_state <= s1;
            end if;
         when s2 =>
            out1 <= "010";
            if in1 = '1' then next_state <= s3;
               else next_state <= s2;
            end if;
         when others =>
            out1 <= "XXX"; next_state <= s0;
      end case;
   end process;
end behave;
```

syn_tco<*n*>

Directive

Used with the syn_black_box directive; supplies the clock to output timing-delay through a black box.

The syn_tco<n> directive is one of several directives that you can use with the syn_black_box directive to define timing for a black box. See syn_black_box, on page 37 for a list of the associated directives.

Constraint File Syntax and Example

The syn_tco<*n*> directive can be entered as an attribute using the Attribute panel of the SCOPE editor. The information in the object, attribute, and value fields must be manually entered. This is the constraint file syntax for the directive:

define_attribute {v:*blackboxModule***} syn_tco***n* **{**[**!**]*clock***->***bundle***=***value***}**

For details about the syntax, see the following table:

v:	Constraint file syntax that indicates that the directive is attached to the view.
blackboxModule	The symbol name of the black-box.
n	A numerical suffix that lets you specify different clock to output timing delays for multiple signals/bundles.
!	The optional exclamation mark indicates that the clock is active on its falling (negative) edge.
clock	The name of the clock signal.
bundle	A bundle is a collection of buses and scalar signals. The objects of a bundle must be separated by commas with no intervening spaces. A valid bundle is A,B,C, which lists three signals. To assign values to bundles, use the following syntax. The values are in ns. [**!**]*clock***->***bundle***=***value*
value	Clock to output delay value in ns.

Constraint file example:

```
define_attribute {v:RCV_CORE} syn_tco1 {CLK-> R_DATA_OUT[63:0]=20}
define_attribute {v:RCV_CORE) syn_tco2 {CLK-> DATA_VALID=30}
```

Verilog Syntax and Example

object **/* syn_tco***n* **= "[!]***clock* **->** *bundle* **=** *value***" */ ;**

See Constraint File Syntax and Example, on page 169 for syntax explanations. The following example defines syn_tco<*n*> and other black-box constraints:

```
module ram32x4(z,d,addr,we,clk);
/* synthesis syn_black_box syn_tco1="clk->z[3:0]=4.0"
   syn_tpd1="addr[3:0]->z[3:0]=8.0"
   syn_tsu1="addr[3:0]->clk=2.0"
   syn_tsu2="we->clk=3.0" */
output [3:0] z;
input [3:0] d;
input [3:0] addr;
input we;
input clk;
endmodule
```

VHDL Syntax and Examples

attribute syn_tco*n* **of** *object* **:** *objectType* **is "** [**!**]*clock* **->** *bundle* **=** *value***" ;**

In VHDL, there are ten predefined instances of each of these directives in the synplify library: syn_tpd1, syn_tpd2, syn_tpd3, … syn_tpd10. If you are entering the timing directives in the source code and you require more than 10 different timing delay values for any one of the directives, declare the additional directives with an integer greater than 10. For example:

```
attribute syn_tco11 : string;
attribute syn_tco12 : string;
```

See Constraint File Syntax and Example, on page 169 for other syntax explanations.

See VHDL Attribute and Directive Syntax, on page 554 for alternate methods for specifying VHDL attributes and directives.

The following example defines syn_tco<*n*> and other black-box constraints:

```
-- A USE clause for the Synplify Attributes package
-- was included earlier to make the timing constraint
-- definitions visible here.
architecture top of top is
component Dpram10240x8
   port (
-- Port A
      ClkA, EnA, WeA: in  std_logic;
      AddrA : in  std_logic_vector(13 downto 0);
      DinA  : in  std_logic_vector(7 downto 0);
      DoutA : out std_logic_vector(7 downto 0);
-- Port B
      ClkB, EnB: in  std_logic;
      AddrB : in  std_logic_vector(13 downto 0);
      DoutB : out std_logic_vector(7 downto 0) );
end component;

attribute syn_black_box : boolean;
attribute syn_tsu1      : string;
attribute syn_tsu2      : string;
attribute syn_tco1      : string;
attribute syn_tco2      : string;
attribute syn_isclock   : boolean;
attribute syn_black_box of Dpram10240x8 : component is true;
attribute syn_tsu1 of Dpram10240x8 : component is
   "EnA,WeA,AddrA,DinA -> ClkA = 3.0";
```

```
attribute syn_tco1 of Dpram10240x8 : component is
   "ClkA -> DoutA[7:0] = 6.0";
attribute syn_tsu2 of Dpram10240x8 : component is
   "EnB,AddrB -> ClkB = 3.0";
attribute syn_tco2 of Dpram10240x8 : component is
   "ClkB -> DoutB[7:0] = 13.0";

-- Other code
```

Verilog-Style Syntax in VHDL for Black Box Timing

In addition to the syntax used in the code above, you can also use the following Verilog-style syntax to specify black-box timing constraints:

```
attribute syn_tco1 of inputfifo_coregen : component is
   "rd_clk->dout[48:0]=3.0";
```

syn_tpd<*n*>

Directive

Used with the syn_black_box directive; supplies information on timing propagation for combinational delay through a black box.

The syn_tpd<n> directive is one of several directives that you can use with the syn_black_box directive to define timing for a black box. See syn_black_box, on page 37 for a list of the associated directives.

Constraint File Syntax and Example

You can enter the syn_tpd<*n*> directive as an attribute using the Attribute panel of the SCOPE editor. The information in the object, attribute, and value fields must be manually entered. This is the constraint file syntax:

define_attribute {v:*blackboxModule***} syn_tpd***n* **{***bundle***->***bundle***=***value***}**

For details about the syntax, see the following table:

v:	Constraint file syntax that indicates that the directive is attached to the view.
blackboxModule	The symbol name of the black-box.
n	A numerical suffix that lets you specify different input to output timing delays for multiple signals/bundles.
bundle	A bundle is a collection of buses and scalar signals. The objects of a bundle must be separated by commas with no intervening spaces. A valid bundle is A,B,C, which lists three signals. The values are in ns. **"***bundle***->***bundle***=***value***"**
value	Input to output delay value in ns.

Constraint file example:

```
define_attribute {v:MEM} syn_tpd1 {MEM_RD->DATA_OUT[63:0]=20}
```

Verilog Syntax and Example

object **/* syn_tpd***n* **= "***bundle* **->** *bundle* **=** *value***" */ ;**

See Constraint File Syntax and Example, on page 173 for an explanation of the syntax. This is an example of syn_tpd<*n*> along with some of the other black-box timing constraints:

```
module ram32x4(z,d,addr,we,clk); /* synthesis syn_black_box
   syn_tpd1="addr[3:0]->z[3:0]=8.0"
   syn_tsu1="addr[3:0]->clk=2.0"
   syn_tsu2="we->clk=3.0" */
output [3:0] z;
input [3:0] d;
input [3:0] addr;
input we;
input clk;
endmodule
```

VHDL Syntax and Examples

attribute syn_tpd*n* **of** *object* **:** *objectType* **is "***bundle* **->** *bundle* **=** *value***" ;**

In VHDL, there are 10 predefined instances of each of these directives in the synplify library, for example: syn_tpd1, syn_tpd2, syn_tpd3, … syn_tpd10. If you are entering the timing directives in the source code and you require more than 10 different timing delay values for any one of the directives, declare the additional directives with an integer greater than 10. For example:

```
attribute syn_tpd11 : string;
attribute syn_tpd11 of bitreg : component is
   "di0,di1 -> do0,do1 = 2.0";
attribute syn_tpd12 : string;
attribute syn_tpd12 of bitreg : component is
   "di2,di3 -> do2,do3 = 1.8";
```

See Constraint File Syntax and Example, on page 173 for an explanation of the syntax.

See VHDL Attribute and Directive Syntax, on page 554 for different ways to specify VHDL attributes and directives.

The following is an example of assigning syn_tpd<*n*> along with some of the black box constraints. See Verilog-Style Syntax in VHDL for Black Box Timing, on page 172 for another way.

```
-- A USE clause for the Synplify Attributes package was included
-- earlier to make the timing constraint definitions visible here.
architecture top of top is
component rcf16x4z
   port (ad0, ad1, ad2, ad3 : in std_logic;
         di0, di1, di2, di3 : in std_logic;
         clk, wren, wpe : in std_logic;
         tri : in std_logic;
         do0, do1, do2, do3 : out std_logic );
end component;

attribute syn_tpd1 of rcf16x4z : component is
   "ad0,ad1,ad2,ad3 -> do0,do1,do2,do3 = 2.1";
attribute syn_tpd2 of rcf16x4z : component is
   "tri -> do0,do1,do2,do3 = 2.0";
attribute syn_tsu1 of rcf16x4z : component is
   "ad0,ad1,ad2,ad3 -> clk = 1.2";
attribute syn_tsu2 of rcf16x4z : component is
   "wren,wpe -> clk = 0.0";
-- Other code
```

syn_tristate

Directive

Specifies that an output port, on a module defined as a black box, is a tristate. Use this directive to eliminate multiple driver errors if the output of a black box has more than one driver. A multiple driver error is issued unless you use this directive to specify that the outputs are tristate.

Verilog Syntax and Examples

object **/* synthesis syn_tristate = 1 */ ;**

where *object* can be black-box output ports. For example:

```
module BUFE(O, I, E); /* synthesis syn_black_box */
   output O /* synthesis syn_tristate = 1 */;

// Other code
```

syn_tsu<*n*>

Directive

Used with the syn_black_box directive; supplies information on timing setup delay required for input pins (relative to the clock) in the black box.

The syn_tsu<n> directive is one of several directives that you can use with the syn_black_box directive to define timing for a black box. See syn_black_box, on page 37 for a list of the associated directives.

Constraint File Syntax and Example

The syn_tsu<*n*> directive can be entered as an attribute using the Attribute panel of the SCOPE editor. The information in the object, attribute, and value fields must be manually entered. The constraint file syntax for the directive is:

define_attribute {v:*blackboxModule***} syn_tsu***n* **{***bundle***->**[**!**]*clock***=***value***}**

For details about the syntax, see the following table:

v:	Constraint file syntax that indicates that the directive is attached to the view.
blackboxModule	The symbol name of the black-box.
n	A numerical suffix that lets you specify different clock to output timing delays for multiple signals/bundles.
bundle	A collection of buses and scalar signals. The objects of a bundle must be separated by commas with no intervening spaces. A valid bundle is A,B,C, which lists three signals. The values are in ns. This is the syntax to define a bundle: *bundle***->**[**!**]*clock***=***value*
!	The optional exclamation mark indicates that the clock is active on its falling (negative) edge.
clock	The name of the clock signal.
value	Input to clock setup delay value in ns.

Constraint file example:

```
define_attribute {v:RTRV_MOD} syn_tsu4 {RTRV_DATA[63:0]->!CLK=20}
```

Verilog Syntax and Example

object **/* syn_tsu***n* **= "***bundle* **->** [**!**]*clock* **=** *value***" */ ;**

For syntax explanations, see Constraint File Syntax and Example, on page 177.

This is an example that defines syn_tsu<*n*> along with some of the other black-box constraints:

```
module ram32x4(z,d,addr,we,clk);
/* synthesis syn_black_box syn_tpd1="addr[3:0]->z[3:0]=8.0"
   syn_tsu1="addr[3:0]->clk=2.0" syn_tsu2="we->clk=3.0" */
output [3:0] z;
input [3:0] d;
input [3:0] addr;
input we;
input clk;
endmodule
```

VHDL Syntax and Examples

attribute syn_tsu*n* **of** *object* **:** *objectType* **is "***bundle* **->** [**!**]*clock* **=** *value***" ;**

In VHDL, there are 10 predefined instances of each of these directives in the synplify library, for example: syn_tsu1, syn_tsu2, syn_tsu3, … syn_tsu10. If you are entering the timing directives in the source code and you require more than 10 different timing delay values for any one of the directives, declare the additional directives with an integer greater than 10:

```
attribute syn_tsu11 : string;
attribute syn_tsu11 of bitreg : component is
   "di0,di1 -> clk = 2.0";
attribute syn_tsu12 : string;
attribute syn_tsu12 of bitreg : component is
   "di2,di3 -> clk = 1.8";
```

For other syntax explanations, see Constraint File Syntax and Example, on page 177.

See VHDL Attribute and Directive Syntax, on page 554 for different ways to specify VHDL attributes and directives. For this directive, you can also use the following Verilog-style syntax to specify it, as described in Verilog-Style Syntax in VHDL for Black Box Timing, on page 172.

The following is an example of assigning syn_tsu<*n*> along with some of the other black-box constraints:

```
-- A USE clause for the Synplify Attributes package
-- was included earlier to make the timing constraint
-- definitions visible here.
architecture top of top is
component rcf16x4z
   port (ad0, ad1, ad2, ad3 : in std_logic;
         di0, di1, di2, di3 : in std_logic;
         clk, wren, wpe : in std_logic;
         tri : in std_logic;
         do0, do1, do2, do3 : out std_logic );
end component;

attribute syn_tco1 of rcf16x4z : component is
   "ad0,ad1,ad2,ad3 -> do0,do1,do2,do3 = 2.1";
attribute syn_tpd2 of rcf16x4z : component is
   "tri -> do0,do1,do2,do3 = 2.0";
attribute syn_tsu1 of rcf16x4z : component is
   "ad0,ad1,ad2,ad3 -> clk = 1.2";
attribute syn_tsu2 of rcf16x4z : component is
   "wren,wpe -> clk = 0.0";
-- Other code
```

translate_off/translate_on

Directive

Allows you to synthesize designs originally written for use with other synthesis tools without needing to modify source code. All source code that is between these two directives is ignored during synthesis.

Another use of these directives is to prevent the synthesis of stimulus source code that only has meaning for logic simulation. You can use translate_off/translate_on to skip over simulation-specific lines of code that are not synthesizable.

When you use translate_off in a module, synthesis of all source code that follows is halted until translate_on is encountered. Every translate_off must have a corresponding translate_on. These directives cannot be nested, therefore, the translate_off directive can only be followed by a translate_on directive.

Note: See also, pragma translate_off/pragma translate_on, on page 31. These directives are implemented the same in the source code.

Verilog Syntax and Example

The Verilog syntax for these directives is as follows:

/* synthesis translate_off */

/* synthesis translate_on */

For example:

```
module test(input a, b, output c);

//synthesis translate_off
assign c=a&b

//synthesis translate_on
assign c=a|b;
endmodule
```

For SystemVerilog designs, you can alternatively use the synthesis_off/synthesis_on directives. The directives function the same as the translate_off/translate_on directives to ignore all source code contained between the two directives during synthesis.

For Verilog designs, you can use the synthesis macro with the Verilog ‘ifdef directive instead of the translate on/off directives. See synthesis Macro, on page 361 for information.

VHDL Syntax and Example

For VHDL designs, you can alternatively use the synthesis_off/synthesis_on directives. Select Project->Implementation Options->VHDL and enable the Synthesis On/Off Implemented as Translate On/Off option. This directs the compiler to treat the synthesis_off/on directives like translate_off/on and ignore any code between these directives.

See VHDL Attribute and Directive Syntax, on page 554 for different ways to specify VHDL attributes and directives.

The following is the VHDL syntax for translate-off/translate_on:

synthesis translate_off

synthesis translate_on

For example:

```
architecture behave of ram4 is
begin

-- synthesis translate_off
stimulus: process (clk, a, b)

-- Source code you DO NOT want synthesized

end process;
-- synthesis translate_on

-- Other source code you WANT synthesized
```

VHDL Syntax and Example

attribute syn_sharing of *object* **:** *objectType* **is " true | false" ;**

where *object* is an architecture name. See VHDL Attribute and Directive Syntax, on page 554 for different ways to specify VHDL attributes and directives.

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

entity alu is
   port (a, b : in std_logic_vector (7 downto 0);
         opcode: in std_logic_vector (1 downto 0);
         clk: in std_logic;
         result: out std_logic_vector (7 downto 0) );
end alu;

architecture behave of alu is
-- Turn on resource sharing for the architecture.
attribute syn_sharing of behave : architecture is "on";
begin
-- Behavioral source code for the design goes here.
end behave;
```

Index

Symbols

A

B

C

D

E

F

T

V

W