Synopsys Synplify Pro for Lattice

Language Support Reference Manual

November 2018

SYNOPSYS®

Third-Party Links

Synopsys, Inc.
690 East Middlefield Road
Mountain View, CA 94043
www.synopsys.com

November 2018

Contents

Chapter 2: SystemVerilog Language Support

CHAPTER 1

Verilog Language Support

This chapter discusses Verilog support for the Synopsys tool. SystemVerilog support is described separately in Chapter 2, *SystemVerilog Language Support*. This chapter includes the following topics:

Support for Verilog Language Constructs

This section describes support for various Verilog language constructs:

- Supported and Unsupported Verilog Constructs, on page 14
- Ignored Verilog Language Constructs, on page 15

Supported and Unsupported Verilog Constructs

The following table lists the supported and unsupported Verilog constructs. If the tool encounters an unsupported construct, it generates an error message and stops.

Supported Verilog Constructs	Unsupported Verilog Constructs
Net types: wire, tri, tri0, tri1	Net types: trireg, triand, trior, wand, wor, charge strength
Register types: • reg, integer, time (64-bit reg) • arrays of reg	Register types: real
Gate primitive, module, and macromodule instantiations	Built-in unidirectional and bidirectional switches, and pull-up/pull-down
inputs, outputs, and inouts to a module	UDPs and specify blocks
All operators +, -, *, /, %, **, <, >, <=, >=, ==, !=, ===, !==, ==?, !=?, &&, ||, !, ~, &, ~&, |, ~|, ^~, ~^, ^, <<, >>, ?:, { }, {{ }} (See Operators, on page 25 for additional details.)	Net names: release, and hierarchical net names (for simulation only)
Procedural statements: assign, if-else-if, case, casex, casez, for, repeat, while, forever, begin, end, fork, join	Procedural statements: deassign, wait

Procedural assignments: • always blocks, user tasks, user functions (See always Blocks for Combinational Logic, on page 75) • Blocking assignments = • Non-blocking assignments <= Do not use = with <= for the same register. Use parameter override: # and defparam (down one level of hierarchy only).	• Named events and event triggers
Continuous assignments	
Compiler directives: `begin_keywords, `celldefine, `define, `endcelldefine, `endif, `end_keywords, `ifdef, `ifndef, `else, `elsif, `include, `line, `undef	
Miscellaneous: • Parameter ranges • Local declarations to begin-end block • Variable indexing of bit vectors on the left and right sides of assignments	

Ignored Verilog Language Constructs

When it encounters certain Verilog constructs, the tool ignores them and continues the synthesis run. The following constructs are ignored:

- delay, delay control, and drive strength
- scalared, vectored
- initial block
- Compiler directives (except for `begin_keywords, `celldefine, `define, `endcelldefine, `endif, `end_keywords, `ifdef, `ifndef, `else, `elsif, `include, `line, `undef, which are supported)
- Calls to system tasks and system functions (they are only for simulation)

Data Types

Verilog data types can be categorized into the following general types:

- Net Data Types, on page 16
- Register Data Types, on page 16
- Miscellaneous Data Types, on page 17

Net Data Types

Net data types are used to model physical connections. The following net types are supported:

wire	Connects elements; used with nets driven by a single gate or continuous assignment
tri	Connects elements; used when a net includes more than one driver
tri0	Models resistive pulldown device (its value is 0 when no driver is present)
tri1	Models resistive pullup device (its value is 1 when no driver is present)

While the Verilog compiler allows the use of tri0 and tri1 nets, these nets are treated as wire net types during synthesis, and any variable declared as a tri0 or tri1 net type behaves as a wire net type. A warning is issued in the log file alerting you that a tri0 or tri1 variable is being treated as a wire net type and that a simulation mismatch is possible.

Register Data Types

The supported register data types are outlined in the following table:

reg	A 1-bit wide data type; when more than one bit is required, a range declaration is included
integer	A 32-bit wide data type that cannot include a range declaration
time	A 64-bit wide data type that stores simulation time as an unsigned number; a range declaration is not allowed

Miscellaneous Data Types

The following data types are also supported:

parameter	Specifies a constant value for a variable (see Creating a Scalable Module, on page 70)
localparam	A local constant parameter (see Localparams, on page 45)
genvar	A Verilog 2001 temporary variable used for index control within a generate loop (see Generate Statement, on page 46)

Built-in Gate Primitives

You can create hardware by directly instantiating built-in gates into your design (in addition to instantiating your own modules). The built-in Verilog gates are called primitives.

Syntax

gateTypeKeyword [*instanceName*] **(***portList***) ;**

The gate type keywords for simple and tristate gates are listed in the following tables. The *instanceName* is a unique instance name and is optional. The signal names in the *portList* can be given in any order with the restriction that all outputs must precede any inputs. For tristate gates, outputs come first, then inputs, and then enable. The following tables list the supported keywords.

Keyword (Simple Gates)	Definition
buf	buffer
not	inverter
and	and gate
nand	nand gate
or	or gate
nor	nor gate
xor	exclusive or gate
xnor	exclusive nor gate

Keyword (Tristate Gates)	Definition
bufif1	tristate buffer with logic one enable
bufif0	tristate buffer with logic zero enable
notif1	tristate inverter with logic one enable
notif0	tristate inverter with logic zero enable

Port Definitions

Port signals are defined as input, output, or bidirectional and are referred to as the port list for the module. The three signal declarations are input, output, and inout as described in the following table.

input	An input signal to the module
output	An output signal from the module
inout	A bidirectional signal to/from the module

Statements

Statement types include loop statements, case statements, and conditional statements as described in the ensuing subsections.

loop Statements

Loop statements are used to modify blocks of procedural statements. The loop statements include for, repeat, while, and forever as described in the following table:

for	Continues to execute a given statement until the expression becomes true; the first assignment is executed initially and then the expression is evaluated repeatedly
repeat	Executes a given statement a fixed number of times; the number of executions is defined by the expression following the repeat keyword.
while	Executes a given statement until the expression becomes true

forever	Continuously repeats the ensuing statement

case Statements

Case statements select one statement from a list of statements based on the value of the case expression. A case statement is introduced with a case, casex, or casez keyword and is terminated with an endcase statement. A case statement can include a default condition that is taken when none of the case select expressions is valid.

case	allow branching on multiple conditional expressions based on case statement matching
casex	allows branching of multiple conditional expression matching where any 'x' (unknown) or 'z' value appearing in the case expression is treated as a don't care
casez	allows branching of multiple conditional expression matching where any 'z' (high impedance) value appearing in the case expression is treated as a don't care
endcase	terminates a case, casex, or casez statement
default	assigns a case expression to a default condition when there are no other matching conditions

Conditional Statements

Conditional statements are used to determine which statement is to be executed based on a conditional expression. The conditional statements include if, else, and else if. The simplified syntax for these conditional statements is either:

```
if (conditionalExpression)
   statement1;
else
   statement2;
```

or

```
if (conditionalExpression)
    statement1;
else if (conditionalExpression);
    statement2;
else
    statement3;
```

The if statement can be used in one of two ways:

- as a single "if-else" statement shown in the first simplified syntax
- as a multiple "if-else-if" statement shown in the second simplified syntax

In the first syntax, when *conditionalExpression* evaluates true, *statement1* is executed, and when *conditionalExpression* evaluates false, *statement2* is executed.

In the second syntax, when *conditionalExpression* evaluates true, *statement1* is executed as in the first syntax example. However, when *conditionalExpression* evaluates false, the second conditional expression (else if) is evaluated and, depending on the result, either *statement2* or *statement3* is executed.

Blocks

Blocks delimit a set of statements. The block is typically introduced by a keyword that identifies the start of the block, and is terminated by an end keyword that identifies the end of the block.

module/endmodule Block

The module/endmodule block is the basic compilation unit in Verilog. Modules are introduced with the module (or macromodule) keyword and are terminated by the endmodule keyword. For more information, see Verilog Module Template, on page 69. The following example shows the basic module syntax.

```
module add (out, in1, in2);output out;
input in1, in2;
assign out = in1 & in2;
endmodule
```

begin/end Block

A begin/end block provides a method of grouping multiple statements into a always block. The statements within this block are executed in the order listed. When a timing control statement is included within the block, execution of the next statement is delayed until after the timing delay. The following example illustrates a begin/end block:

```
module tmp (in1, in2, out1, out2);
input in1, in2;
output out1, out2;
reg out1, out2;

always@(in1, in2)
begin
   out1 =(in1 & in2);
   out2 =(in1 | in2);
end
endmodule
```

fork/join Block

A fork/join block provides a method of grouping multiple statements into a an always block. The statements within this block are executed simultaneously. With parallel blocks, because all statements are executed at the same time, mutually dependent statements are not allowed. The following example illustrates a fork/join block:

```
module tmp (in1, in2, out1, out2);
input in1, in2;
output out1, out2;
reg out1, out2;

always@(in1, in2)
fork
   out1 =(in1 & in2);
   out2 =(in1 | in2);
join
endmodule fork, join
```

generate/endgenerate Block

A generate block is created using one of the generate-loop, generate-conditional, or generate-case format. The block is introduced with the keyword generate and terminated with the keyword endgenerate. For more information, see Generate Statement, on page 46.

Compiler Directives

Compiler directives control compilation within an EDA environment. These directives are prefixed with an accent grave (‘) or “tick mark.” Compiler directives are not Verilog statements and, as such, do not require the semicolon terminator. A compiler directive remains active until it is modified or disabled by another directive. The following table lists the supported compiler directives:

`begin_keywords	Specifies a pair of directives `begin_keywords and `end_keywords to identify keywords reserved within a block of source code, based on a specific version of IEEE Std 1364 or IEEE Std 1800. See `begin_keywords and `end_keywords, on page 23 for details.
'celldefine	Identifies the source code limited by 'cellname and 'endcelldefine as a cell.
'define	Creates a macro for text substitution
'else	Indicates an alternative to the previous `ifdef or `ifndef condition
'elsif	Indicates an alternative to the previous `ifdef or `ifndef condition
'endcelldefine	Identifies the source code limited by 'cellname and 'endcelldefine as a cell.
'endif	Indicates the end of an `ifdef or `ifndef conditional procedural statement
`end_keywords	Specifies a pair of directives `begin_keywords and `end_keywords to identify keywords reserved within a block of source code, based on a specific version of IEEE Std 1364 or IEEE Std 1800. See `begin_keywords and `end_keywords, on page 23 for details.
'ifdef	Executes a conditional procedural statement based on a defined macro
'ifndef	Executes a conditional procedural statement in the absence of a text macro

'include	File inclusion; the contents of the referenced file are inserted at the location of the 'include directive.
'line	Maintains the reference to the line numbers for the original source or include file. See 'line Compiler Directive, on page 24 for details.
'undef	Removes the definition of a previously defined text macro

`begin_keywords and `end_keywords

The syntax for the 'begin_keywords and 'end_keywords directives is as follows:

```
keywords_directive ::= `begin_keywords "version_specifier"
version_specifier ::=
| 1800-2012
| 1800-2009
| 1800-2005
| 1364-2005
| 1364-2001
| 1364-2001 -noconfig
| 1364-1995
endkeywords_directive ::= `end_keywords
```

Functional requirements for these directives include the following support:

- The `begin_keywords and `end_keywords directives can be specified outside a design element, such as a module, interface, configuration, or package.
- Source file boundaries can be extended from the start of the `begin_keywords until the matching `end_keywords directive is met or to the end of the compilation unit.
- The `begin_keywords and `end_keywords directive pairs can be nested.
- Different version specifiers can be set as follows:
  – When version_specifier=1364-1995, compiler uses Verilog V95 standard
  – When version_specifier=1364-2***, compiler uses Verilog V2001 standard
  – When version_specifier=1800-****, compiler uses Verilog SYSV standard

Example

```
`begin_keywords "1364-2001"
// Use IEEE Std 1364-2001 Verilog keywords
module m2 (...);
reg [63:0] logic; // OK: "logic" is not a keyword in 1364-2001

...

endmodule
`end_keywords
```

'line Compiler Directive

The compiler may need to keep track of the Verilog/SystemVerilog source or include files and the line numbers in these files, because the original files can lose this information when other processes modify them. The compiler maintains the current line and file name to generate debug messages in the log file. The 'line directive is used to reset the current line number and file name to a different line number and file name relative to the current file. Typically, this is added by machine-generated code or when source code is displayed for debugging.

The syntax used for the 'line directive is shown below:

> 'line *number* "*fileName*" *level*

Where:

- *number* – A positive integer that specifies the new line number that points to the specified file.
- *fileName* – File name string that can be specified with a full or relative path name.
- *level* – This parameter can be specified with the values 0, 1, or 2:
  - 0 – Any line of the original source or include file
  - 1 – First line of the original source or include file
  - 2 – First line after the original source or include file exited

Example

Here is an example of how the 'line compiler directive is used with the original test.v file:

```
'line 1 "test.v" 0
input a;
```

test.v

```
1 module test(a,b,c);
2 input a,b;
3 output c;
4 assign c = a & b;
5 endmodule
```

The 'line compiler directives added by machine-generated Verilog code indicate their original file numbers, which are useful for debugging. For example:

```
'line 2 "test.v" 0
input a;
'line 2 "test.v" 0
input b;
'line 3 "test.v" 0
output c;
'line 4 "test.v" 0
assign c = a & b;
```

Operators

Arithmetic Operators

Arithmetic operators can be used with all data types.

Symbol	Usage	Function
+	a + b	a plus b
-	a - b	a minus b
*	a * b	a multiplied by b

/	a / b	a divided by b
%	a % b	a modulo b
**	a **b	a to the power of b

The / and % operators are supported for compile-time constants and constant powers of two. For the modulus operator (%), the result takes the sign of the first operand.

The exponential operation for `a**b` is supported, where:

Operand	Can be a ...
a	• Constant (positive/negative) • Dynamic variable (signed/unsigned)
b	• Constant (positive/negative) • Dynamic variable (positive/negative integer)

Exponential operation includes the following limitations:

- The exponent cannot be a negative number when the base operand is a dynamic value.
- The following conditions generate an error:
  - Operand a is a dynamic variable and operand b is a negative constant.
  - Operands a and b are dynamic variables.
  - Operand a is a constant power of 2 and negative (for example, -2, -4, -6 ...) and operand b is a dynamic signal (signed/unsigned).
  - Operand a is a constant non power of 2 and positive/negative (for example, 1/-1, 3/-3, 5/-5 ...) and operand b is a dynamic signal (signed/unsigned).

    For the two previous conditions, the compiler only supports operand a as a power of 2 positive integer (for example, 2, 4, 6 ...) and operand b as a dynamic signal (signed/unsigned).

Relational Operators

Relational operators compare expressions. The value returned by a relational operator is 1 if the expression evaluates true or 0 if the expression evaluates false.

Symbol	Usage	Function
<	a < b	a is less than b
>	a > b	a is greater than b
<=	a <= b	a is less than or equal to b
=>	a => b	a equal to or greater than b

Equality Operators

The equality operators compare expressions. When a comparison fails, the result is 0, otherwise it is 1. When both operands of a logical equality (==) or logical inequality (!=) contain an unknown value (x) or high-impedance (z) value, the result of the comparison is unknown (x); otherwise the result is either true or false.

When an operands of case equality (===) or case inequality (!==) contains an unknown value (x) or high-impedance (z) value, the result is calculated bit-by-bit.

Symbol	Usage	Function
==	m == n	m is equal to n
!=	m != n	m is not equal to n
===	m === n	m is identical to n
!==	m !== n	m is not identical to n

When an equality (==) or inequality (!=) operator includes unknown bits (for example, A==4'b10x1 or A!=4'b111z), the Synopsys Verilog compiler assumes that the output is always False. This assumption contradicts the LRM which states that the output should be x (unknown) and can result in a possible simulation mismatch.

Wildcard Equality Operators

The wildcard equality operators (==? and !=?) compare expressions and perform bit-wise comparisons between the two operands. When the right-side operand contains an unknown value (x) or high-impedance (z) value for a given bit position, the compiler treats them as wildcards. The wildcard bit can match any value (0, 1, x, or z) that corresponds to the bit of the left-side operand to which it is being compared. All the other bits are compared for logical equality or inequality operations.

For the wildcard operation below:

```
sig1 ==? 3'b10x
```

The compiler implements the following behavior:

```
sig1 == 3'b100 | | sig1 == 3'b101
```

Note that the Synopsys Verilog compiler does not support wildcard equality operators with two variable operands.

Logical Operators

Logical operators connect expressions. The result a logical operation is 0 if false, 1 if true, or x (unknown) if ambiguous. The negation operator (!) changes a nonzero or true value of the operand to 0 or a zero or false value to 1; an ambiguous value results in x (unknown) value.

Symbol	Usage	Function
&&	a && b	a and b
||	a || b	a or b
!	!a	not a

Bitwise Operators

Bitwise operators are described in the following table:

Symbol	Usage	Function
~	~m	Invert each bit
&	m & n	AND each bit
|	m | n	OR each bit
^	m ^ n	Exclusive OR each bit
~^, ^~	m ~^ n m ^~ n	Exclusive NOR each bit

Unary Reduction Operators

Unary reduction operators are described in the following table:

Symbol	Usage	Function
&	&m	AND all bits
~&	~&m	NAND all bits
|	|m	OR all bits
~|	~|m	NOR all bits
^	^m	Exclusive OR all bits
^~, ~^	~^m ^~m	Exclusive NOR all bits

Miscellaneous Operators

Miscellaneous operators are described in the following table:

Symbol	Usage	Function
? :	sel? m:n	If sel is true, select m
{ }	{m,n}	Concatenate m to n
{ { } }	{n{m}}	Replicate m *n* times

Procedural Assignments

The Verilog procedure may be an always or initial statement, task, or function. Assignment statements for procedural assignments always appear within the procedures and can execute concurrently with other procedures.

Verilog 2001 Support

You can choose the Verilog standard to use for a design or given files within a design: Verilog '95 or Verilog 2001. See File Options Popup Menu Command, on page 498 and *Setting Verilog and VHDL Options*, on page 96 of the *User Guide*. The tool supports the following Verilog 2001 features:

Feature	Description
Combined Data, Port Types (ANSI C-style Modules)	Module data and port type declarations can be combined for conciseness.
Comma-separated Sensitivity List	Commas are allowed as separators in sensitivity lists (as in other Verilog lists).
Wildcards (*) in Sensitivity List	Use @* or @(*) to include all signals in a procedural block to eliminate mismatches between RTL and post-synthesis simulation.
Signed Signals	Data types net and reg, module ports, integers of different bases and signals can all be signed. Signed signals can be assigned and compared. Signed operations can be performed for vectors of any length.
Inline Parameter Assignment by Name	Assigns values to parameters by name, inline.
Constant Function	Builds complex values at elaboration time.
Configuration Blocks	Specifies a set of rules that defines the source description applied to an instance or module.
Localparams	A constant that cannot be redefined or modified.
$signed and $unsigned Built-in Functions	Built-in Verilog 2001 function that converts types between signed and unsigned.
$clog2 Constant Math Function	Returns the value of the log base-2 for the argument passed.
Generate Statement	Creates multiple instances of an object in a module. You can use generate with loops and conditional statements.
Automatic Task Declaration	Dynamic allocation and release of storage for tasks.

Feature	Description
Multidimensional Arrays	Groups elements of the declared element type into multi-dimensional objects.
Variable Partial Select	Supports indexed part select expressions (**+**: and **-**:), which use a variable range to provide access to a word or part of a word.
Cross-Module Referencing	Accesses elements across modules.
ifndef and elsif Compiler Directives	'ifndef and 'elsif compiler directive support.

Combined Data, Port Types (ANSI C-style Modules)

In Verilog 2001, you can combine module data and port type declarations to be concise, as shown below:

Verilog '95

```
module adder_16 (sum, cout, cin, a, b);
output [15:0] sum;
output cout;
input [15:0] a, b;
input cin;
reg [15:0] sum;
reg cout;
wire [15:0] a, b;
wire cin;
```

Verilog 2001

```
module adder_16(output reg [15:0] sum, output reg cout,
input wire cin, input wire [15:0] a, b);
```

Comma-separated Sensitivity List

In Verilog 2001, you can use commas as separators in sensitivity lists (as in other Verilog lists).

Verilog '95

```
always @(a or b or cin)
   sum = a - b - cin;
always @(posedge clock or negedge reset)
   if (!reset)
     q <= 0;
   else
     q <= d;
```

Verilog 2001

```
always @(a, b or cin)
   sum = a - b - cin;
always @(posedge clock, negedge reset)
   if (!reset)
     q <= 0;
   else
     q <= d;
```

Wildcards (*) in Sensitivity List

In Verilog 2001, you can use @* or @(*) to include all signals in a procedural block, eliminating mismatches between RTL and post-synthesis simulation.

Verilog '95

```
always @(a or b or cin)
   sum = a - b - cin;
```

Verilog 2001

```
// Style 1:
always @(*)
   sum = a - b - cin;
// Style 2:
always @*
   sum = a - b - cin;
```

Signed Signals

In Verilog 2001, data types net and reg, module ports, integers of different bases and signals can all be signed. You can assign and compare signed signals, and perform signed operations for vectors of any length.

Declaration

```
module adder (output reg signed [31:0] sum,
   wire signed input [31:0] a, b;
```

Assignment

```
wire signed [3:0] a = 4'sb1001;
```

Comparison

```
wire signed [1:0] sel;
parameter p0 = 2'sb00, p1 = 2'sb01, p2 = 2'sb10, p3 = 2'sb11;
case sel
   p0: ...
   p1: ...
   p2: ...
   p3: ...
endcase
```

Inline Parameter Assignment by Name

In Verilog 2001, you can assign values to parameters by name, inline:

```
module top( /* port list of top-level signals */ );
   dff #(.param1(10), .param2(5)) inst_dff(q, d, clk);
endmodule
```

where:

```
module dff #(parameter param1=1, param2=2) (q, d, clk);
   input d, clk;
   output q;
...
endmodule
```

Constant Function

In Verilog 2001, you can use constant functions to build complex values at elaboration time.

Example – Constant function

```
module ram
// Verilog 2001 ANSI parameter declaration syntax
   #(parameter depth=129,
   parameter width=16 )
// Verilog 2001 ANSI port declaration syntax
(input clk, we,
   // Calculate addr width using Verilog 2001 constant function
   input [clogb2(depth)-1:0] addr,
   input [width-1:0] di,
   output reg [width-1:0] do );
function integer clogb2;
   input [31:0] value;
      for (clogb2=0; value>0; clogb2=clogb2+1)
      value = value>>1;
endfunction
reg [width-1:0] mem[depth-1:0];

always @(posedge clk) begin
   if (we)
      begin
         mem[addr]<= di;
         do<= di;
      end
      else
         do<= mem[addr];
      end
endmodule
```

Localparam

In Verilog 2001, localparam (constants that cannot be redefined or modified) follow the same parameter rules in regard to size and sign. Unlike parameter, localparam cannot be overridden by a defparam from another module.

Example:

```
parameter ONE = 1
localparam TWO=2*ONE
localparam [3:0] THREE=TWO+1;
localparam signed [31:0] FOUR=2*TWO;
```

Configuration Blocks

Verilog configuration blocks define a set of rules that explicitly specify the exact source description to be used for each instance in a design.
A configuration block is defined outside the module and multiple configuration blocks are supported.

Syntax

config *configName***;**
    design *libraryIdentifier***.***moduleName***;**
    default liblist *listofLibraries***;**
    *configurationRule***;**
endconfig

Design Statement

The design statement specifies the library and module for which the configuration rule is to defined.

design *libraryIdentifier***.***moduleName***;**
    libraryIdentifier **:-** Library Name
    moduleName **:-** Module Name

Default Statement

The default liblist statement lists the library from which the definition of the module and sub-modules can be selected. A use clause cannot be used in this statement.

default liblist *listof_Libraries***;**
    listofLibraries **:-** List of Libraries

Configuration Rule Statement

In this section, rules are defined for different instances or cells in the design. The rules are defined using instance or cell clauses.

- instance clause – specifies the particular source description for a given instance in the design.
- cell clause – specifies the source description to be picked for a particular cell/module in a given design.

A configuration rule can be defined as any of the following:

- instance clause with liblist

  `instance` *moduleName*`.`*instance* `liblist` *listofLibraries*`;`

- instance clause with use clause

  `instance` *moduleName*`.`*instance* `use` *libraryIdentifier*`.[`*cellName* | *configName*`];`

- cell clause with liblist

  `cell` *cellName* `liblist` *listofLibraries*`;`

- cell clause with use clause

  `cell` *cellName* `use` *libraryIdentifier*`.[`*cellName* | *configName*`];`

Configuration Block Examples

The following examples illustrate Verilog 2001 configuration blocks.

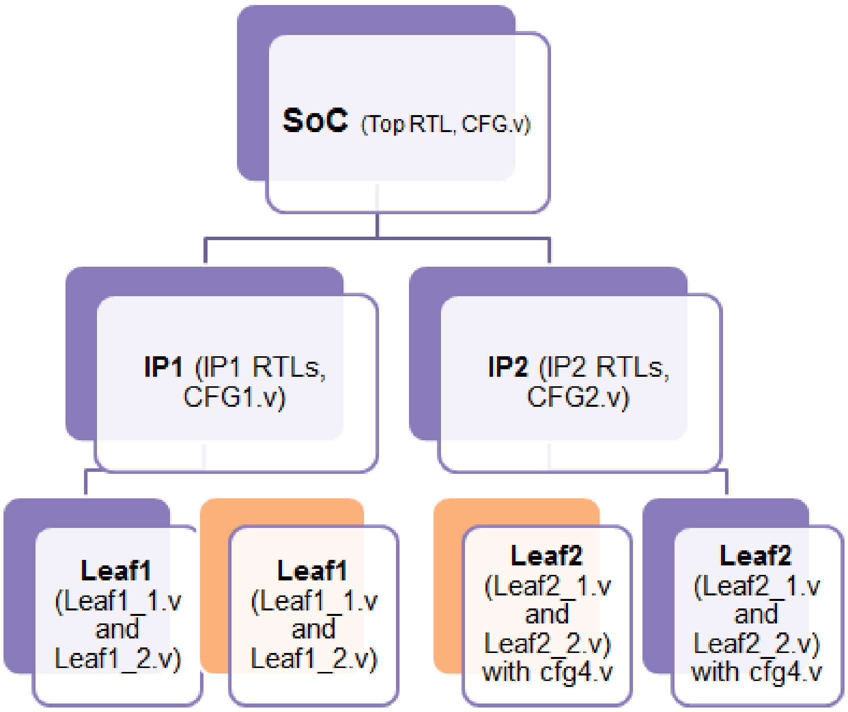
Example – Configuration with instance clause

The following example has different definitions for the leaf module compiled into the multlib and xorlib libraries; configuration rules are defined specifically for instance u2 in the top module to have the definition of leaf module as XOR (by default the leaf definition is multiplier). This example uses an instance clause with liblist to define the configuration rule.

```
//********Leaf module with the Multiplication definition

// Multiplication definition is compiled to the library "multlib"
// Command to be added in the design file to compile a
// specific HDL to a specific library is
// add_file -verilog -lib multlib "leaf_mult.v"
```

```
module leaf
(
//Input Port
   input [7:0] d1,
   input [7:0] d2,
//Output Port
   output reg [15:0] dout
);

always@*
   dout = d1 * d2;
endmodule //EndModule

//********Leaf module with the XOR definition

// XOR definition is compiled to the library "xorlib"
// Command to be added in the design file to compile a
// specific HDL to a specific library is
// add_file -verilog -lib xorlib "leaf_xor.v"

module leaf
(
//Input Port
   input [7:0] d1,
   input [7:0] d2,
//Output Port
   output reg[15:0] dout
);

always@(*)
   dout = d1 ^ d2;
endmodule //EndModule

//********Top module definition

// Top module definition is compiled to the library "TOPLIB"
// Command to be added in the design file to compile a
// specific HDL to a specific library is
// add_file -verilog -lib TOPLIB "top.v"

module top
(
//Input Port
   input [7:0] d1,
   input [7:0] d2,
   input [7:0] d3,
   input [7:0] d4,
//Output Port
   output [15:0] dout1,
   output [15:0] dout2
);

leaf
u1
(
   .d1(d1),
   .d2(d2),
   .dout(dout1)
);

leaf
u2
(
   .d1(d3),
   .d2(d4),
   .dout(dout2)
);

endmodule //End Module

//********Configuration Definition

// Configuration definition is compiled to the library "TOPLIB"
// Command to be added in the design file to compile a
// specific HDL to a specific library is
// add_file -verilog -lib TOPLIB "cfg.v"

config cfg;
   design TOPLIB.top;
   default liblist multlib xorlib TOPLIB; //By default the leaf
      // definition is Multiplication definition
   instance top.u2 liblist xorlib; //For instance u2 the default
      // definition is overridden and the "leaf" definition is
      // picked from "xorlib" which is XOR.
endconfig //EndConfiguration
```

Basically, configuration blocks can be represented by the top-level design with hierarchy shown as follows:


SoC (Top RTL, CFG.v)
IP1 (IP1 RTLs, CFG1.v)
IP2 (IP2 RTLs, CFG2.v)
Leaf1 (Leaf1_1.v and Leaf1_2.v)
Leaf1 (Leaf1_1.v and Leaf1_2.v)
Leaf2 (Leaf2_1.v and Leaf2_2.v) with cfg4.v
Leaf2 (Leaf2_1.v and Leaf2_2.v) with cfg4.v

Example – Configuration with cell clause

In the following example, different definitions of the leaf module are compiled into the multlib and xorlib libraries; a configuration rule is defined for cell leaf that picks the definition of the cell from the multlib library. This example uses a cell clause with a use clause to define the configuration rule.

```
//********Configuration Definition
// Configuration definition is compiled to the library "TOPLIB"
// Command to be added in the design file to compile a
// specific HDL to a specific library is
// add_file -verilog -lib TOPLIB "cfg.v"
```

```
config cfg;
   design TOPLIB.top;
   default liblist xorlib multlib TOPLIB; //By default the leaf
      // definition uses the XOR definition
   cell leaf use multlib.leaf; //Definition of the instances u1 and
u2
      // will be Multiplier which is picked from "multlib"
endconfig //EndConfiguration
```

Example – Hierarchical reference of the module inside the configuration

Similar to the previous example, different definitions of leaf are compiled into the multlib, addlib, and xorlib libraries; suppose the adder and submodule definitions are also included in the code. The configuration rule is defined for instance u2 that is referenced in the hierarchy as the lowest instance module using an instance clause.

```
//********Leaf module with the ADDER definition

// ADDER definition is compiled to the library "addlib"
// Command to be added in the design file to compile a
// specific HDL to a specific library is
// add_file -verilog -lib addlib "leaf_add.v"

module leaf
(
//Input Port
   input [7:0] d1,
   input [7:0] d2,
//Output Port
   output [15:0] dout
);

assign dout = d1 + d2;
endmodule

//********Submodule definition

// Submodule definition is compiled to the library "SUBLIB"
// Command to be added in the design file to compile a
// specific HDL to a specific library is
// add_file -verilog -lib SUBLIB "sub.v"

module sub
(
//Input Port
   input [7:0] d1,
   input [7:0] d2,
   input [7:0] d3,
   input [7:0] d4,
//Output Port
   output [15:0] dout1,
   output [15:0] dout2
);

leaf
u1
(
   .d1(d1),
   .d2(d2),
   .dout(dout1)
);

leaf
u2
(
   .d1(d3),
   .d2(d4),
   .dout(dout2)
);
endmodule //End Module
```

The configuration is defined as follows:

```
//********Configuration Definition

// Configuration definition is compiled to the library "TOPLIB"
// Command to be added in the design file to compile a
// specific HDL to a specific library is
// add_file -verilog -lib TOPLIB "cfg.v"

config cfg;
   design TOPLIB.top;
   default liblist addlib multlib xorlib TOPLIB SUBLIB; //By
   default,
      //the leaf definition uses the ADDER definition
   instance top.u1.u2 liblist xorlib multlib; //For instances u2 is
      //referred hierarchy to lowest instances and the default
   definition
      //is overridden by XOR definition for this instanceendconfig
//EndConfiguration
```

Multiple Configuration Blocks

When using multiple configurations, if a configuration for the top level exists, the configuration is implemented; lower level configurations do not apply unless the top-level configuration includes an instance clause that maps an instance to another configuration.

The following code examples define how multiple configuration blocks can be configured. The top-level design file includes the configuration shown below:

```
#######################################################
#design files
add_file -verilog -lib work123 "top.v"
add_file -verilog -lib work123 "sub1_1.v"
add_file -verilog -lib lib2 "sub1_2.v"
add_file -verilog -lib lib3 "sub2_1.v"
add_file -verilog -lib work123 "sub2_2.v"
add_file -verilog -lib lib5 "sub3_1.v"
add_file -verilog -lib work123 "sub3_2.v"
add_file -verilog -lib lib7 "sub4_2.v"
add_file -verilog -lib work123 "sub4_1.v"
add_file -verilog "cfg1.v"
add_file -verilog -lib cfg1 "cfg2.v"
add_file -verilog -lib cfg2 "cfg3.v"
#######################################################
```

Example Top Module – Multiple Configurations

Example Submodule 1_1 – Multiple Configurations

Example Submodule 1_2 – Multiple Configurations

Example Submodule 2_1 – Multiple Configurations

Example Submodule 2_2 – Multiple Configurations

Example Submodule 3_1 – Multiple Configurations

Example Submodule 3_2 – Multiple Configurations

Example Submodule 4_1 – Multiple Configurations

Example Submodule 4_2 – Multiple Configurations

Example Configuration 1 – Multiple Configurations

Example Configuration 2 – Multiple Configurations

Example Configuration – Multiple Configurations

Configuration with Generate Statements

An instance or cell clause can be defined within a generate statement. A configuration rule specifies how this instance or cell is to be configured; where the generated instance or cell for the submodule is compiled into the work1 library and the top module is compiled into the work2 library. See the example below:

Example Configurations with Generate (Instance Clause)

Example 1A - Submodule Definition

Example 1B - Top-Level Module Definition

Example 1C – Configuration Definition

```
config cfg1;
design work2.top;
instance top.blk1.inst liblist work1;
endconfig
```

OR

```
config cfg1;
design work2.top;
cell sub use work1.sub;
endconfig
```

Configuration Block Limitations

Configuration blocks do not support the following:

- Nested configuration
- A use clause with the cell name or library name omitted
- Mixed HDL configuration
- Multiple top levels in the design clause
- Parameter override for the configuration

Localparams

In Verilog 2001, localparams (constants that cannot be redefined or modified) follow the same parameter rules in regard to size and sign.

Example:

```
parameter ONE = 1
localparam TWO=2*ONE
localparam [3:0] THREE=TWO+1;
localparam signed [31:0] FOUR=2*TWO;
```

$signed and $unsigned Built-in Functions

In Verilog 2001, the built-in Verilog 2001 functions can be used to convert types between signed and unsigned.

```
c = $signed (s); /* Assign signed valued of s to c. */
d = $unsigned (s); /* Assign unsigned valued of s to d. */
```

$clog2 Constant Math Function

Verilog-2005 includes the $clog2 constant math function which returns the value of the log base-2 for the argument passed. This system function can be used to compute the minimum address width necessary to address a memory of a given size or the minimum vector width necessary to represent a given number of states.

Syntax

$clog2(*argument***)**

In the above syntax, argument is an integer or vector.

Example 1 – Constant Math Function Counter

Example 2 – Constant Math Function RAM

Generate Statement

The newer Verilog 2005 generate statement is now supported in Verilog 2001. The defparam, parameter, and function and task declarations within generate statements are supported. In addition, the naming scheme for registers and instances is enhanced to include closer correlation to specified generate symbolic hierarchies. Generated data types have unique identifier names and can be referenced hierarchically. Generate statements are created using one of the following three methods: generate-loop, generate-conditional, or generate-case.

```
// for loop
generate
begin:G1
   genvar i;
   for (i=0; i<=7; i=i+1)
   begin :inst
      adder8 add (sum [8*i+7 : 8*i], c0[i+1],
         a[8*i+7 : 8*i], b[8*i+7 : 8*i], c0[i]);
   end
end
endgenerate

// if-else
generate
   if (adder_width < 8)
      ripple_carry # (adder_width) u1 (a, b, sum);
   else
      carry_look_ahead # (adder_width) u1 (a, b, sum);
endgenerate

// case
parameter WIDTH=1;
generate
   case (WIDTH)
      1: adder1 x1 (c0, sum, a, b, ci);
      2: adder2 x1 (c0, sum, a, b, ci);
      default: adder # width (c0, sum, a, b, ci);
   endcase
endgenerate
```

Automatic Task Declaration

In Verilog 2001, tasks can be declared as automatic to dynamically allocate new storage each time the task is called and then automatically release the storage when the task exits. Because there is no retention of tasks from one call to another as in the case of static tasks, the potential conflict of two concurrent calls to the same task interfering with each other is avoided. Automatic tasks make it possible to use recursive tasks.

This is the syntax for declaring an automatic task:

task automatic *taskName* **(***argument* **[,** *argument* **, ...]) ;**

Arguments to automatic tasks can include any language-defined data type (reg, wire, integer, logic, bit, int, longint, or shortint) or a user-defined datatype (typedef, struct, or enum). Multidimensional array arguments are not supported.

Automatic tasks can be synthesized but, like loop constructs, the tool must be able to statically determine how many levels of recursive calls are to be made. Automatic (recursive) tasks are used to calculate the factorial of a given number.

Example

```
module automatic_task (input byte in1,
   output bit [8:0] dout);
parameter FACT_OP = 3;
bit [8:0] dout_tmp;

task automatic factorial(input byte operand,
   output bit [8:0] out1);
integer nFuncCall = 0;
begin
   if (operand == 0)
   begin
      out1 = 1;
   end
   else
   begin
      nFuncCall++;
      factorial((operand-1), out1);
      out1 = out1 * operand;
   end
end
endtask

always_comb
factorial(FACT_OP,dout_tmp);
assign dout = dout_tmp + in1 ;
endmodule
```

Multidimensional Arrays

In Verilog 2001, arrays are declared by specifying the element address ranges after the declared identifiers. Use a constant expression, when specifying the indices for the array. The constant expression value can be a positive integer, negative integer, or zero. Refer to the following examples.

2-dimensional wire object	my_wire is an eight-bit-wide vector with indices from 5 to 0. `wire [7:0] my_wire [5:0];`
3-dimensional wire object	my_wire is an eight-bit-wide vector with indices from 5 to 0 whose indices are from 3 down to 0. `wire [7:0] my_wire [5:0] [3:0];`
3-dimensional wire object	my_wire is an eight-bit-wide vector (-4 to 3) with indices from -3 to 1 whose indices are from 3 down to 0. `wire [-4:3] my_wire [-3:1] [3:0];`

These examples apply to register types too:

```
reg [3:0] mem[7:0]; // A regular memory of 8 words with 4
   //bits/word.

reg [3:0] mem[7:0][3:0]; // A memory of memories.
```

There is a Verilog restriction which prohibits bit access into memory words. Verilog 2001 removes all such restrictions. This applies equally to wires types. For example:

```
wire[3:0] my_wire[3:0];

assign y = my_wire[2][1]; // refers to bit 1 of 2nd word (word
   //does not imply storage here) of my_wire.
```

Variable Partial Select

In Verilog 2001, indexed partial select expressions (+: and -:), which use a variable range to provide access to a word or part of a word, are supported. The software extracts the size of the operators at compile time, but the index expression range can remain dynamic. You can use the partial select operators to index any non-scalar variable.

The syntax to use these operators is described below.

vectorName [*baseExpression* **+:** *widthExpression*]
vectorName [*baseExpression* **-:** *widthExpression*]

vectorName	Name of vector. Direction in the declaration affects the selection of bits
baseExpression	Indicates the starting point for the array. Can be any legal Verilog expression.
+:	The +: expression selects bits starting at the *baseExpression* while adding the *widthExpression*. Indicates an upward slicing.
-:	The -: expression selects bits starting at the *baseExpression* while subtracting the *widthExpression*. Indicates a downward slicing.
widthExpression	Indicates the width of the slice. It must evaluate to a constant at compile time. If it does not, you get a syntax error.

This is an example using partial select expressions:

```
module part_select_support (down_vect, up_vect, out1, out2, out3);
output [7:0] out1;
output [1:0] out2;
output [7:0] out3;
input [31:0] down_vect;
input [0:31] up_vect;
wire [31:0] down_vect;
wire [0:31] up_vect;
wire [7:0] out1;
wire [1:0] out2;
wire [7:0] out3;
wire [5:0] index1;
assign index1 = 1;
assign out1 = down_vect[index1+:8]; // resolves to [8:1]
assign out2 = down_vect[index1-:8]; // should resolve to [1:0],
   // but resolves to constant 2'b00 instead
assign out3 = up_vect[index1+:8]; // resolves to [1:8]
endmodule
```

For the Verilog code above, the following description explains how to validate partial select assignments to out2:

- The compiler first determines how to slice down_vect.
  - down_vect is an array of [31:0]
  - assign out2 = down_vect [1 -: 8] will slice down_vect starting at value 1 down to -6 as [1 : -6], which includes [1, 0, -1, -2, -3, -4, -5, -6]
- Then, the compiler assigns the respective bits to the outputs.
  - out2 [0] = down_vect [-6]
    out2 [1] = down_vect [-5]
  - Negative ranges cannot be specified, so out2 is tied to “00”.
  - Therefore, change the following expression in the code to:
    assign out2 = down_vect [1 -: 2], which resolves to down_vect [1,0]

Cross-Module Referencing

Cross-module referencing (XMR) is a method of accessing an element across modules in Verilog and SystemVerilog. Verilog supports accessing elements across different scopes using the hierarchical reference (.) operator. Cross-module referencing can also be done on the variable of any of the data types available in SystemVerilog.

Cross-module referencing support includes:

- Downward Cross-Module Referencing
- Upward Cross-Module Referencing
- Cross-Module Referencing of Generate Blocks
- Cross-Module Referencing Generate Block Examples
- Cross-Module Referencing Limitations

Downward Cross-Module Referencing

In downward cross-module referencing, you reference elements of lower-level modules in the higher-level modules through instantiated names. This is the syntax for a downward cross-module reference:

port/variable **=** *inst1***.***inst2***.***value***;** // XMR Read

*inst1***.***inst2***.***port/variable* **=** *value***;** // XMR Write

In this syntax, *inst1* is the name of an instance instantiated in the top module and *inst2* is the name of an instance instantiated in *inst1*. *Value* can be a constant, parameter, or variable. *Port/variable* is defined/declared once in the current module.

Example – Downward Read Cross-Module Reference

Example – Downward Write Cross-Module Reference

Upward Cross-Module Referencing

In upward cross-module referencing, a lower-level module references items in a higher-level module in the hierarchy through the name of the top module.

This is the syntax for an upward reference from a lower module:

port/variable **=** *top***.***inst1***.***inst2***.***value*; // XMR Read

*top***.***inst1***.***inst2***.***port/variable* **=** *value*; // XMR Write

The starting reference is the top-level module. In this syntax, *top* is the name of the top-level module, *inst1* is the name of an instance instantiated in top module and *inst2* is the name of an instance instantiated in *inst1*. Value can be a constant, parameter, or variable. *Port/variable* is the one defined/declared in the current module.

Example – Upward Read Cross-Module Reference

Cross-Module Referencing of Generate Blocks

For cross-module referencing of generate blocks, signals can be referenced into, within, and from generate blocks to elements outside its boundary. Support includes:

- Upward read or write cross-module referencing into, within, and from generate blocks.
- Downward read or write cross-module referencing into, within, and from generate blocks.
- Cross-module referencing supports different types of generate blocks, such as, generate blocks using a for/if/case statement.
- Cross-module referencing into or from a generate block of any hierarchy.

Cross-Module Referencing Generate Block Examples

Cross-module referencing of generate blocks are supported for modules shown in the following examples.

Example 1A - XMR of a Generate Block

This code example implements cross-module referencing of the generate block in the top-level module.

Example 1B - XMR of a Generate Block

Here is the code example of the sub-module, for which write and read cross-module referencing occurs from the top-level module above.

Example 2A– XMR of Generate Block with an if Statement

This code example implements cross-module referencing of the generate block using an if statement in the top-level design.

Example 2B– XMR of Generate Block with an if Statement

Here is the code example of the sub-module that is referenced from the top-level generate block.

Example 3A: XMR of Generate Block with a for Statement

This code example implements cross-module referencing of the generate block using a for statement in the top-level design.

Example 3B: XMR of Generate Block with a for Statement

Here is the code example of the sub-module that is referenced from the top-level generate block.

Example 4A: XMR of Generate Block with case Statements

This code example implements cross-module referencing of the generate block using case statements in the top-level design.

Example 4B: XMR of Generate Block with case Statements

Here is the code example of the sub-module that is referenced from the top-level generate block.

Cross-Module Referencing Limitations

The following limitations currently exist with cross-module referencing:

- Cross-module referencing through an array of instances is not supported.
- In upward cross-module referencing, the reference must be an absolute path (an absolute path is always from the top-level module).
- Functions and tasks cannot be accessed through cross-module reference notation.

- You can only use cross-module referencing with Verilog/SystemVerilog elements. You cannot access VHDL elements with hierarchical references.
- To access VHDL hierarchical references, it is recommended that you do this using the hypersource/connect mechanism. For details, see Using Hyper Source, on page 557.

ifndef and elsif Compiler Directives

Verilog 2001 supports the `ifndef and `elsif compiler directives. Note that the `ifndef directive is the opposite of `ifdef.

```
module top(output out);
   `ifndef a
      assign out = 1'b01;
   `elsif b
      assign out = 1'b10;
   `else
      assign out = 1'b00;
   `endif
endmodule
```

Verilog Synthesis Guidelines

This section provides guidelines for synthesis using Verilog and covers the following topics:

- General Synthesis Guidelines, on page 56
- Library Support in Verilog, on page 57
- Constant Function Syntax Restrictions, on page 61
- Multi-dimensional Array Syntax Restrictions, on page 62
- Signed Multipliers in Verilog, on page 63
- Verilog Language Guidelines: always Blocks, on page 64
- Initial Values in Verilog, on page 65
- Cross-language Parameter Passing in Mixed HDL, on page 68
- Library Directory Specification for the Verilog Compiler, on page 68

General Synthesis Guidelines

Some general guidelines are presented here to help you synthesize your Verilog design. See Verilog Module Template, on page 69 for additional information.

- Top-level module – The tool picks the last module compiled that is not referenced in another module as the top-level module. Module selection can be overridden from the Verilog panel of the Implementation Options dialog box.
- Simulate your design before synthesis to expose logic errors. Logic errors that you do not catch are passed through the tool, and the synthesized results will contain the same logic errors.
- Simulate your design after placement and routing – Have the place-and-route tool generate a post placement and routing (timing-accurate) simulation netlist, and do a final simulation before programming your devices.

- Avoid asynchronous state machines – To use the tool for asynchronous state machines, make a netlist of technology primitives from your target library.
- Level-sensitive latches – For modeling level-sensitive latches, use continuous assignment statements.

Library Support in Verilog

Verilog libraries are used to compile design units; this is similar to VHDL libraries. Use the libraries in Verilog to support mixed-HDL designs, where the VHDL design includes instances of a Verilog module that is compiled into a specific library. Library support in Verilog can be used with Verilog 2001 and SystemVerilog designs.

Compiling Design Units into Libraries

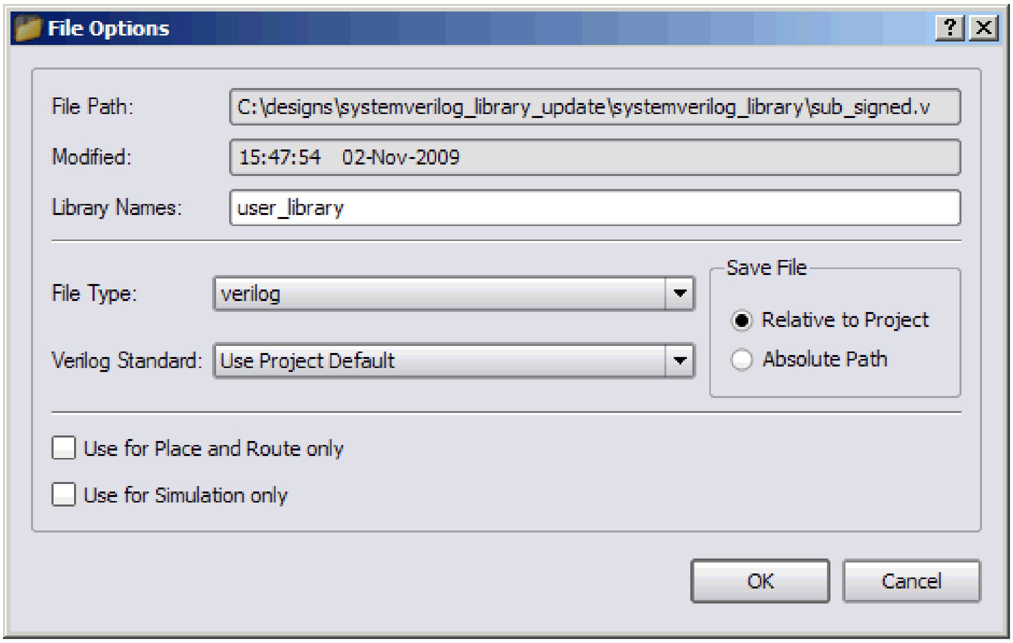
By default, the Verilog source files are compiled into the work library. You can compile these Verilog source files into any user-defined library.

To compile a Verilog file into a user-defined library:

1. Select the file in the Project view.

   The library name appears next to the filename; it directly follows the filename.

2. Right-click and select File Options from the popup menu. Specify the name for your library in the Library Names field. You can:
   – Compile multiple files into the same library.
   – Also compile the same file into multiple libraries.


File Options

File Path: C:\designs\systemverilog_library_update\systemverilog_library\sub_signed.v

Modified: 15:47:54 02-Nov-2009

Library Names: user_library

File Type: verilog

Verilog Standard: Use Project Default

Save File: [✓ Relative to Project] Absolute Path

☐ Use for Place and Route only

☐ Use for Simulation only

Searching for Verilog Design Units in Mixed-HDL Designs

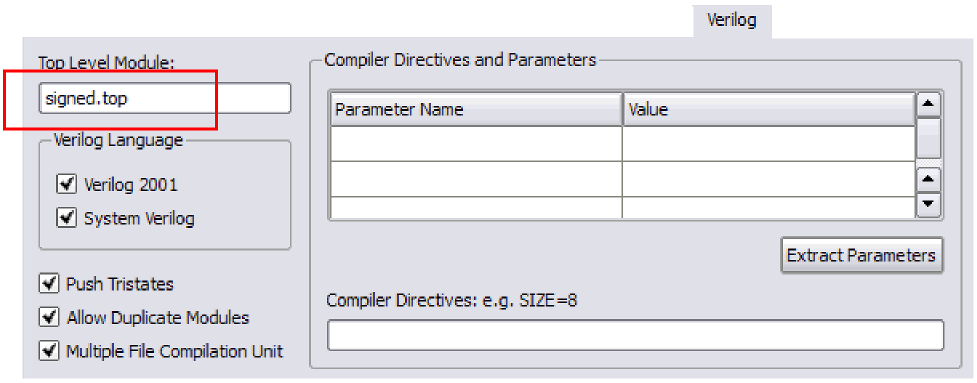
When a VHDL file references a Verilog design unit, the compiler first searches the corresponding library for which the VHDL file was compiled. If the Verilog design unit is not found in the user-defined library for which the VHDL file was compiled, the compiler searches the work library and then all the other Verilog libraries.

Therefore, to use a specific Verilog design unit in the VHDL file, compile the Verilog file into the same user-defined library for which the corresponding VHDL file was compiled. You cannot use the VHDL library clause for Verilog libraries.

Specifying the Verilog Top-level Module

To set the Verilog top-level module for a user-defined library, use *libraryName.moduleName* in the Top Level Module field on the Verilog tab of the Implementation Options dialog box. You can also specify the following equivalent Tcl command:

```
set_option -top_module "signed.top"
```


Verilog
Top Level Module:
Compiler Directives and Parameters
Parameter Name
Value
Extract Parameters
Compiler Directives: e.g. SIZE=8

Top Level Module: signed.top

Verilog Language

- [✓] Verilog 2001
- [✓] System Verilog

- [✓] Push Tristates
- [✓] Allow Duplicate Modules
- [✓] Multiple File Compilation Unit

Limitations

The following functions are not supported:

- Direct Entity Instantiation
- Configuration for Verilog Instances

Example 1: Specifying Verilog Top-level Module—Compiled to the Non-work Library

```
//top_unsigned.v compiled into a user defined library - "unsigned"
//add_file -verilog -lib unsigned "./top_unsigned.v"
module top ( input unsigned [7:0] a, b,
output unsigned [15:0] result );
assign result = a * b;
endmodule
```

```
//top_signed.v compiled into a user defined library - "signed"
//add_file -verilog -lib signed "./top_signed.v"
module top ( input signed [7:0] a, b,
output signed [15:0] result );
assign result = a * b;
endmodule
```

To set the top-level module from the signed library:

- Specify the prefix library name for the module in the Top Level Module option in the Verilog panel of the Implementation Options dialog box.
- `set_option -top_module "signed.top"`

Example 2: Referencing Verilog Module from VHDL

This example includes two versions of the Verilog `sub` module that are compiled into the signed_lib and unsigned_lib libraries. The compiler uses the sub module from unsigned_lib when the top.vhd is compiled into unsigned_lib.

```
//Sub module sub in sub_unsigned is compiled into unsigned_lib
//add_file -verilog -lib unsigned_lib "./sub_unsigned.v"
module sub ( input unsigned [7:0] a, b,
output unsigned [15:0] result );
assign result = a * b;
endmodule

//Sub module sub in sub_signed is compiled into signed_lib
//add_file -verilog -lib signed_lib "./sub_signed.v"
module sub ( input signed [7:0] a, b,
output signed [15:0] result );
assign result = a * b;
endmodule

//VHDL Top module top is compiled into unsigned_lib library
// add_file -vhdl -lib unsigned_lib "./top.vhd"
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY top IS
GENERIC(
size_t : integer := 8
);
   PORT( a_top : IN std_logic_vector(size_t-1 DOWNTO 0);
   b_top : IN std_logic_vector(size_t-1 DOWNTO 0);
   result_top : OUT std_logic_vector(2*size_t-1 DOWNTO 0)
);
END top;
```

```
ARCHITECTURE RTL OF top IS
component sub
   PORT(a : IN std_logic_vector(7 DOWNTO 0);
   b : IN std_logic_vector(7 DOWNTO 0);
   result : OUT std_logic_vector(15 DOWNTO 0));
END component;
BEGIN
U1 : sub
   PORT MAP (
      a => a_top,
      b => b_top,
   result => result_top
);
END ARCHITECTURE RTL;
```

Constant Function Syntax Restrictions

For Verilog 2001, the syntax for constant functions is identical to the existing function definitions in Verilog. Restrictions on constant functions are as follows:

- No hierarchal references are allowed
- Any function calls inside constant functions must be constant functions
- System tasks inside constant functions are ignored
- System functions inside constant functions are illegal
- Any parameter references inside a constant function should be visible
- All identifiers, except arguments and parameters, should be local to the constant function
- Constant functions are illegal inside the scope of a generate statement

Multi-dimensional Array Syntax Restrictions

For Verilog 2001, the following examples show multi-dimensional array syntax restrictions.

```
reg [3:0] arrayb [7:0][0:255];

arrayb[1] = 0;
// Illegal Syntax - Attempt to write to elements [1][0]..[1][255]

arrayb[1][12:31] = 0;
// Illegal Syntax - Attempt to write to elements [1][12]..[1][31]

arrayb[1][0] = 0;
// Okay. Assigns 32'b0 to the word referenced by indices [1][0]

Arrayb[22][8] = 0;
// Semantic Error, There is no word 8 in 2nd dimension.
```

When using multi-dimension arrays, the association is always from right-to-left while the declarations are left-to-right.

Example 1

```
module test (input a,b, output z, input clk, in1, in2);
reg tmp [0:1][1:0];

always @(posedge clk)
begin
   tmp[1][0] <= a ^ b;
   tmp[1][1] <= a & b;
   tmp[0][0] <= a | b;
   tmp[0][1] <= a &~ b;
end
assign z = tmp[in1][in2];

endmodule
```

Example 2

```
module bb(input [2:0] in, output [2:0] out)
   /* synthesis syn_black_box */;
endmodule

module top(input [2:0] in, input [2:1] d1, output [2:0] out);
wire [2:0] w1[2:1];
wire [2:0] w2[2:1];
```

```
generate
begin : ABCD
   genvar i;
   for(i=1; i < 3; i = i+1)
   begin : CDEF
      assign w1[i] = in;
      bb my_bb(w1[i], w2[i]);
   end
end
endgenerate
assign out = w2[d1];

endmodule
```

Signed Multipliers in Verilog

This section applies only to those using Verilog compilers earlier than version 2001.

The software contains an updated signed multiplier module generator. A signed multiplier is used in Verilog whenever you multiply signed numbers. Because earlier versions of Verilog compilers do not support signed data types, an example is provided on how to write a signed multiplier in your Verilog design:

```
module smul4(a, b, clk, result);
input [3:0]a;
input [3:0]b;
input clk;
output [7:0]result;
wire [3:0] inputa_signbits, inputb_signbits;
reg [3:0]inputa;
reg [3:0]inputb;
reg [7:0]out, result;
assign inputa_signbits = {4{inputa[3]}};
assign inputb_signbits = {4{inputb[3]}};

always @(inputa or inputb or inputa_signbits or inputb_signbits)
begin
   out = {inputa_signbits,inputa} * {inputb_signbits,inputb};
end
```

```
always @(posedge clk)
begin
   inputa = a;
   inputb = b;
   result = out;
end

endmodule
```

Verilog Language Guidelines: always Blocks

An always block can have more than one event control argument, provided they are all edge-triggered events or all signals; these two kinds of arguments cannot be mixed in the same always block.

Examples

```
// OK: Both arguments are edge-triggered events
always @(posedge clk or posedge rst)

// OK: Both arguments are signals
always @(A or B)

// No good: One edge-triggered event, one signal
always @(posedge clk or rst)
```

An always block represents either sequential logic or combinational logic. The one exception is that you can have an always block that specifies level-sensitive latches and combinational logic. Avoid this style, however, because it is error prone and can lead to unwanted level-sensitive latches.

An event expression with posedge/negedge keywords implies edge-triggered sequential logic; and without posedge/negedge keywords implies combinational logic, a level-sensitive latch, or both.

Each sequential always block is triggered from exactly one clock (and optional sets and resets).

You must declare every signal assigned a value inside an always block as a reg or integer. An integer is a 32-bit quantity by default, and is used with the Verilog operators to do two's complement arithmetic.

Syntax:

integer [*msb***:***lsb*] *identifier***;**

Avoid combinational loops in always blocks. Make sure all signals assigned in a combinational always block are explicitly assigned values every time the always block executes, otherwise the tool needs to insert level-sensitive latches in the design to hold the last value for the paths that do not assign values. This is a common source of errors, so the tool issues a warning message that latches are being inserted into your design.

You will get an error message if you have combinational loops in your design that are not recognized as level-sensitive latches by the tool (for example if you have an asynchronous state machine).

It is illegal to have a given bit of the same reg or integer variable assigned in more than one always block.

Assigning a 'bx to a signal is interpreted as a “don't care” (there is no 'bx value in hardware); the tool then creates the hardware with the most efficient design.

Initial Values in Verilog

In Verilog, you can now store and pass initial values that the software previously ignored. Initial values specified in Verilog only affect the compiler output. This ensures that the synthesis results match the simulation results. For initial values for RAM, see Initial Values for RAMs, on page 200.

Initial Values for Registers

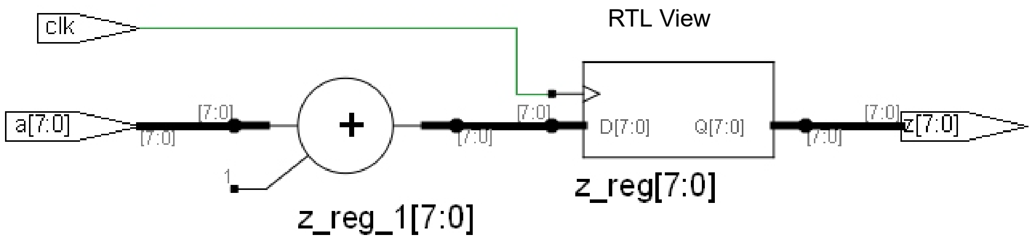
The compiler reads the procedural assign statements with initial values. It then stores the values, propagates them to inferred logic, and passes them down stream. The initial values only affect the output of the compiler; initial value properties are not forward-annotated to the final netlist.

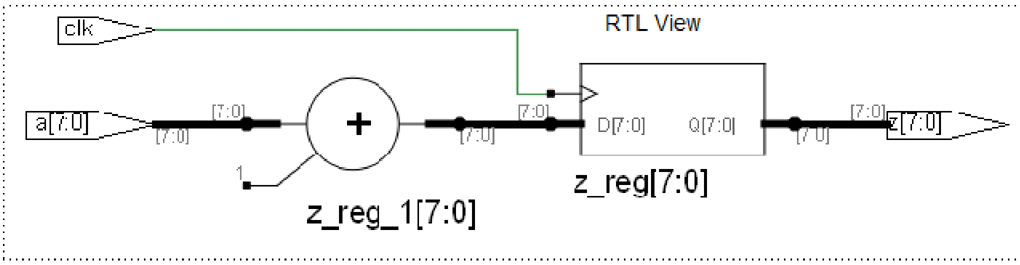
If synthesis removes an unassigned register that has an initial value, the initialization values are still propagated forward. If bits of a register are unassigned, the compiler removes the unassigned bits and propagates the initial value.

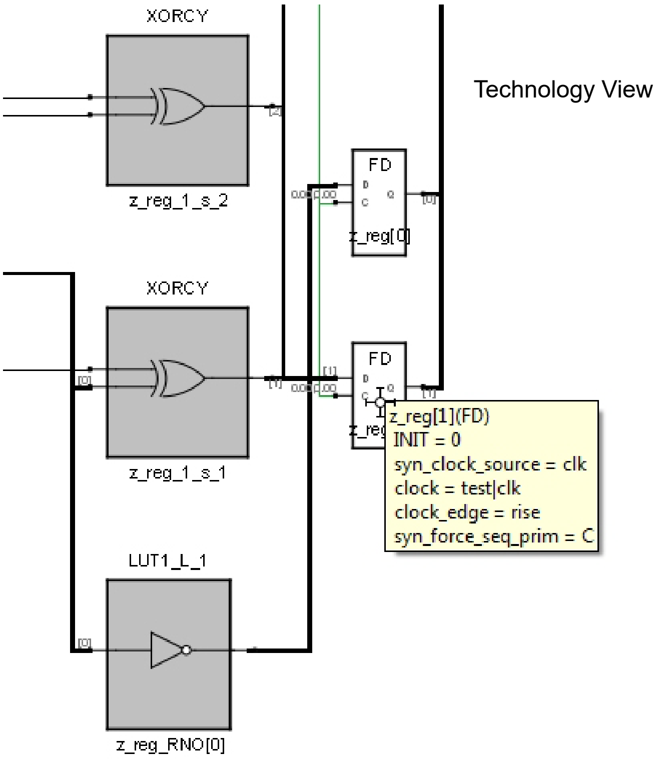
To illustrate, assume register one does not receive any input (an initial value is not specified). If the register is not initialized, it is subsequently removed during the optimization process. However, if the register is initialized to a value of 1 as in the example below, the compiler keeps the register during synthesis.

```
module test (
   input clk,
   input [7:0] a,
   output [7:0] z );
reg [7:0] z_reg = 8'hf0 ;
reg one = 1'd1;

always@(posedge clk)
   z_reg <= a + one;
   assign z = z_reg;
endmodule
```

The following figures show the HDL Analyst views.




RTL View
clk
a[7:0]
[7:0]
+
1
z_reg_1[7:0]
D[7:0]
Q[7:0]
z_reg[7:0]
z[7:0]


Technology View
XORCY
z_reg_1_s_2
FD
z_reg[0]
XORCY
z_reg_1_s_1
FD
LUT1_L_1
z_reg_RNO[0]
z_reg[1](FD)
INIT = 0
syn_clock_source = clk
clock = test|clk
clock_edge = rise
syn_force_seq_prim = C

Cross-language Parameter Passing in Mixed HDL

The compiler supports the passing of parameters for integers, natural numbers, real numbers, and strings from Verilog to VHDL. The compiler also supports the passing of these same generics from VHDL to Verilog.

Library Directory Specification for the Verilog Compiler

Currently, if a module is instantiated in a module top without a module definition, the Verilog compiler errors out. Verilog simulators provide a command line switch (-y *libraryDirectory*) to specify a set of library directories which the compiler searches.

Library directories are specified in the Library Directories section in the Verilog panel of the Implementations Options dialog box.

Example:

If the design has one Verilog file specified

```
module foo(input a, b, output z);

foobar u1 (a, b, z);

endmodule
```

Then, if foobar.v exists in one of the specified directories, it is loaded into the compiler.

Verilog Module Template

Hardware designs can include combinational logic, sequential logic, state machines, and memory. These elements are described in the Verilog module. You also can create hardware by directly instantiating built-in gates into your design (in addition to instantiating your own modules).

Within a Verilog module you can describe hardware with one or more continuous assignments, always blocks, module instantiations, and gate instantiations. The order of these statements within the module is irrelevant, and all execute concurrently. The following is the Verilog module template:

```
module <top_module_name>(<port_list>);

/* Port declarations. followed by wire,
   reg, integer, task and function declarations */

/* Describe hardware with one or more continuous assignments,
   always blocks, module instantiations and gate instantiations */

// Continuous assignment
wire <result_signal_name>;
assign <result_signal_name> = <expression>;

// always block
always @(<event_expression>)

begin
   // Procedural assignments
   // if statements
   // case, casex, and casez statements
   // while, repeat and for loops
   // user task and user function calls
end

// Module instantiation
<module_name> <instance_name> (<port_list>);

// Instantiation of built-in gate primitive
gate_type_keyword (<port_list>);

endmodule
```

The statements between the begin and end statements in an always block execute sequentially from top to bottom. If you have a fork-join statement in an always block, the statements within the fork-join execute concurrently.

A disable statement can be included to terminate an active procedure within a module. As shown in the example, including a disable statement in the begin/end block prevents the out2 =(in1 | in2) expression from being executed.

```
always@(in1, in2)
begin : comb1
   out1 =(in1 & in2);
disable comb1;
   out2 =(in1 | in2);
endendmodule
```

You can add comments in Verilog by preceding your comment text with // (two forward slashes). Any text from the slashes to the end of the line is treated as a comment, and is ignored by the tool. To create a block comment, start the comment with /* (forward slash followed by asterisk) and end the comment with */ (asterisk followed by forward slash). A block comment can span any number of lines but cannot be nested inside another block comment.

Scalable Modules

This section describes creating and using scalable Verilog modules. The topics include:

- Creating a Scalable Module, on page 70
- Using Scalable Modules, on page 71
- Using Hierarchical defparam, on page 73

Creating a Scalable Module

You can create a Verilog module that is scalable, so that it can be stretched or shrunk to handle a user-specified number of bits in the port list buses.

Declare parameters with default parameter values. The parameters can be used to represent bus sizes inside a module.

Syntax

parameter *parameterName* **=** *value* **;**

You can define more than one parameter per declaration by using comma-separated *parameterName* **=** *value* pairs.

Example

```
parameter size = 1;
parameter word_size = 16, byte_size = 8;
```

Using Scalable Modules

To use scalable modules, instantiate the scalable module and then override the default parameter value with the defparam keyword. Give the instance name of the module you are overriding, the parameter name, and the new value.

Syntax

defparam *instanceName***.***parameterName* **=** *newValue* **;**

Example

```
big_register my_register (q, data, clk, rst);
defparam my_register.size = 64;
```

Combine the instantiation and the override in one statement. Use a # (hash mark) immediately after the module name in the instantiation, and give the new parameter value. To override more than one parameter value, use a comma-separated list of new values.

Syntax

moduleName **# (***newValuesList***)** *instanceName* **(***portList* **);**

Example

```
big_register #(64) my_register (q, data, clk, rst);
```

Creating a Scalable Adder

```
module adder(cout, sum, a, b, cin);

/* Declare a parameter, and give a default value */
parameter size = 1;
output cout;

/* Notice that sum, a, and b use the value of the size parameter */
output [size-1:0] sum;
input [size-1:0] a, b;
input cin;
assign {cout, sum} = a - b - cin;
endmodule
```

Scaling by Overriding a Parameter Value with defparam

You can instantiate a Verilog module for the VHDL entity adder and override its size parameter using the following statement highlighted in the Verilog code:

```
module adder8(cout, sum, a, b, cin);
output cout;
output [7:0] sum;
input [7:0] a, b;
input cin;
adder my_adder (cout, sum, a, b, cin);

// Creates my_adder as an eight bit adder
defparam my_adder.size = 8;
endmodule
```

Scaling by Overriding the Parameter Value with

```
module adder16(cout, sum, a, b, cin);
output cout;
```

You can define a parameter at this level of hierarchy and pass that value down to a lower-level instance. In this example, a parameter called my_size is declared. You can declare a parameter with the same name as the lower level name (size) because this level of hierarchy has a different name range than the lower level and there is no conflict – but there is no correspondence between the two names either, so you must explicitly pass the parameter value down through the hierarchy.

```
parameter my_size = 16;    // I want a 16-bit adder
output [my_size-1:0] sum;
input [my_size-1:0] a, b;
input cin;

/* my_size overrides size inside instance my_adder of adder */
// Creates my_adder as a 16-bit adder
adder #(my_size) my_adder (cout, sum, a, b, cin);
endmodule
```

Using Hierarchical defparam

The defparam statement is used to specify constant expressions. For example, the defparam statement can be used to define the width of variables or specify time delays. The compiler supports defparam to override parameter values for modules at the current level or multiple levels of hierarchy.

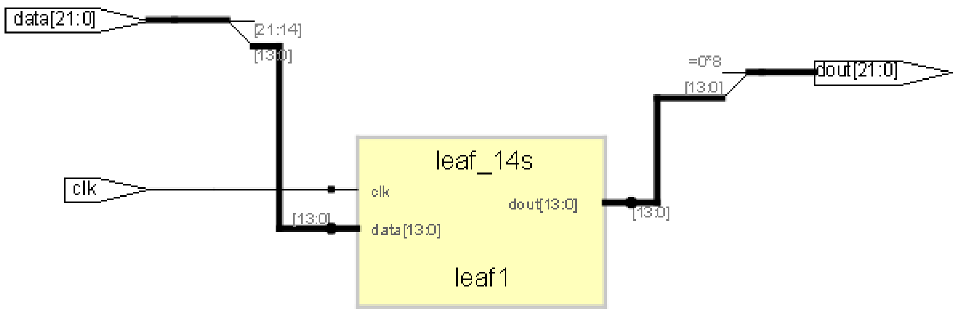
Syntax

defparam *hierarchicalPath* **=** *constantExpression*

For example: `defparam i1.i2.i3.parameter = constant`

Example: Hierarchical defparam

For the leaf module, the RTL view below shows that the input and output data widths are [0:14] in the HDL Analyst tool.



Combinational Logic

Combinational logic is hardware with output values based on some function of the current input values. There is no clock, and no saved states. Most hardware is a mixture of combinational and sequential logic.

You create combinational logic with an always block and/or continuous assignments.

Combinational Logic Examples

The following combinational logic synthesis examples are included in the *installDirectory*/examples/verilog/common_rtl/combinat directory:

- Adders
- ALU
- Bus Sorter
- 3-to-8 Decoder
- 8-to-3 Priority Encoders
- Comparator
- Multiplexers (concurrent signal assignments, case statements, or if-then-else statements can be used to create multiplexers; the tool automatically creates parallel multiplexers when the conditions in the branches are mutually exclusive)
- Parity Generator
- Tristate Drivers

always Blocks for Combinational Logic

Use the Verilog always blocks to model combinational logic as shown in the following template.

```
always @(event_expression)
begin
   // Procedural assignment statements,
   // if, case, casex, and casez statements
   // while, repeat, and for loops
   // task and function calls
end
```

When modeling combinational logic with always blocks, keep the following in mind:

- The always block must have exactly one event control (@(*event_ expression*)) in it, located immediately after the always keyword.
- List all signals feeding into the combinational logic in the event expression. This includes all signals that affect signals that are assigned inside the always block. List all signals on the right side of an assignment inside an always block. The tool assumes that the sensitivity list is complete, and generates the desired hardware. However, it will issue a warning message if any signals on the right side of an assignment inside an always block are not listed, because your pre- and post-synthesis simulation results might not match.
- You must explicitly declare as reg or integer all signals you assign in the always block.

Note: Make sure all signals assigned in a combinational always block are explicitly assigned values each time the always block executes. Otherwise, the tool must insert level-sensitive latches in your design to hold the last value for the paths that do not assign values. This will occur, for instance, if there are combinational loops in your design. This often represents a coding error. The software issues a warning message that latches are being inserted into your design because of combinational loops. You will get an error message if you have combinational loops in your design that are not recognized as level-sensitive latches by the tool.

Event Expression

Every always block must have one event control (@(*event_expression*)), that specifies the signal transitions that trigger the always block to execute. This is analogous to specifying the inputs to logic on a schematic by drawing wires to gate inputs. If there is more than one signal, separate the names with the or keyword.

Syntax

always @ (*signal1* **or** *signal2* ...**)**

Example

```
/* The first line of an always block for a multiplexer that
   triggers when 'a', 'b' or 'sel' changes */
always @(a or b or sel)
```

Locate the event control immediately after the always keyword. Do not use the posedge or negedge keywords in the event expression; they imply edge-sensitive sequential logic.

Example: Multiplexer

See also Comma-separated Sensitivity List, on page 33.

```
module mux (out, a, b, sel);
output out;
input a, b, sel;
reg out;

always @(a or b or sel)
begin
   if (sel)
      out = a;
   else
      out = b;
end
endmodule
```

Continuous Assignments for Combinational Logic

Use continuous assignments to model combinational logic. To create a continuous assignment:

1. Declare the assigned signal as a wire using the syntax:

   wire [*msb* **:***lsb*] *result_signal* **;**

2. Specify your assignment with the assign keyword, and give the expression (value) to assign.

   assign *result_signal* = *expression* **;**

   or ...

   Combine the wire declaration and assignment into one statement:

   wire [*msb* **:***lsb*] *result_signal* = *expression* **;**

Each time a signal on the right side of the equal sign (=) changes value, the expression re-evaluates, and the result is assigned to the signal on the left side of the equal sign. You can use any of the built-in operators to create the expression.

The bus range [msb **:** lsb] is only necessary if your signal is a bus (more than one bit wide).

All outputs and inouts to modules default to wires; therefore the wire declaration is redundant for outputs and inouts and assign *result_signal* = *expression* is sufficient.

Example: Bit-wise AND

```
module bitand (out, a, b);
output [3:0] out;
input [3:0] a, b;
/* This wire declaration is not required because "out" is an
   output in the port list */
wire [3:0] out;
assign out = a & b;
endmodule
```

Example: 8-bit Adder

```
module adder_8 (cout, sum, a, b, cin);
output cout;
output [7:0] sum;
input cin;
input [7:0] a, b;
assign {cout, sum} = a - b - cin;
endmodule
```

Signed Multipliers

A signed multiplier is inferred whenever you multiply signed numbers in Verilog 2001 or VHDL. However, Verilog 95 does not support signed data types. If your Verilog code does not use the Verilog 2001 standard, you can implement a signed multiplier in the following way:

```
module smul4(a, b, clk, result);
input [3:0]a;
input [3:0]b;
input clk;
output [7:0]result;
reg [3:0]inputa;
reg [3:0]inputb;
reg [7:0]out, result;

always @(inputa or inputb)
begin
   out = {{4{inputa[3]}},inputa} * {{4{inputb[3]}},inputb};
end

always @(posedge clk)
begin
   inputa = a;
   inputb = b;
   result = out;
end

endmodule
```

Sequential Logic

Sequential logic is hardware that has an internal state or memory. The state elements are either flip-flops that update on the active edge of a clock signal or level-sensitive latches that update during the active level of a clock signal.

Because of the internal state, the output values might depend not only on the current input values, but also on input values at previous times. A state machine is sequential logic where the updated state values depend on the previous state values. There are standard ways of modeling state machines in Verilog. Most hardware is a mixture of combinational and sequential logic.

You create sequential logic with always blocks and/or continuous assignments.

Sequential Logic Examples

The following sequential logic synthesis examples are included in the *installDirectory*/examples/verilog/common_rtl/sequentl directory:

- Flip-flops and level-sensitive latches
- Counters (up, down, and up/down)
- Register file
- Shift registers
- State machines

For additional information on synthesizing flip-flops and latches, see these topics:

- Flip-flops Using always Blocks, on page 80
- Level-sensitive Latches, on page 81
- Sets and Resets, on page 83
- SRL Inference, on page 88

Flip-flops Using always Blocks

To create flip-flops/registers, assign values to the signals in an always block, and specify the active clock edge in the event expression.

always Block Template

```
always @(event_expression)
begin
   // Procedural statements
end
```

The always block must have one event control (@(*event_expression*)) immediately after the always keyword that specifies the clock signal transitions that trigger the always block to execute.

Syntax

always @ (*edgeKeyword clockName***)**

where *edgeKeyword* is posedge (for positive-edge triggered) or negedge (for negative-edge triggered).

Example

```
always @(posedge clk)
```

Assignments to Signals in always Blocks

When assigning signals in an always block:

- Explicitly declare, as a reg or integer, any signal you assign inside an always block.
- Any signal assigned within an edge-triggered always block will be implemented as a register; for instance, signal q in the following example.

Example

```
module dff_or (q, a, b, clk);
output q;
input a, b, clk;
reg q; // Declared as reg, since assigned in always block
```

```
always @(posedge clk)
begin
   q <= a | b;
end
endmodule
```

In this example, the result of a | b connects to the data input of a flip-flop, and the q signal connects to the q output of the flip-flop.

Level-sensitive Latches

The preferred method of modeling level-sensitive latches in Verilog is to use continuous assignment statements.

Example

```
module latchor1 (q, a, b, clk);
output q;
input a, b, clk;

assign q = clk ? (a | b) : q;
endmodule
```

Whenever clk, a, or b change, the expression on the right side re-evaluates. If your clk becomes true (active, logic 1), a | b is assigned to the q output. When the clk changes and becomes false (deactivated), q is assigned to q (holds the last value of q). If a or b changes and clk is already active, the new value a | b is assigned to q.

Although it is simpler to specify level-sensitive latches using continuous assignment statements, you can create level-sensitive latches from always blocks. Use an always block and follow these guidelines for event expression and assignments.

always Block Template

```
always@(event_expression)
begin   // Procedural statements
end
```

Whenever the assignment to a signal is incompletely defined, the event expression specifies the clock signal and the signals that feed into the data input of the level-sensitive latch.

Syntax

always @ (*clockName* **or** *signal1* **or** *signal2* ... **)**

Example

```
always @(clk or data)
begin
   if (clk)
      q <= data;
end
```

The always block must have exactly one event control (@(*event_expression*)) in it, and must be located immediately after the always keyword.

Assignments to Signals in always Blocks

You must explicitly declare as reg or integer any signal you assign inside an always block.

Any incompletely-defined signal that is assigned within a level-triggered always block will be implemented as a latch.

Whenever level-sensitive latches are generated from an always block, the tool issues a warning message, so that you can verify if a given level-sensitive latch is really what you intended. (If you model a level-sensitive latch using continuous assignment then no warning message is issued.)

Example: Creating Level-sensitive Latches You Want

```
module latchor2 (q, a, b, clk);
output q;
input a, b, clk;
reg q;

always @(clk or a or b)
begin
   if (clk)
      q <= a | b;
end
endmodule
```

If clk, a, or b change, and clk is a logic 1, then set q equal to a|b.

What to do when clk is a logic zero is not specified (there is no else in the if statement), so when clk is a logic 0, the last value assigned is maintained (there is an implicit q=q). The tool correctly recognizes this as a level-sensitive latch, and creates a level-sensitive latch in your design. The tool issues a warning message when you compile this module (after examination, you may choose to ignore this message).

Example: Creating Unwanted Level-sensitive Latches

```
module mux4to1 (out, a, b, c, d, sel);
output out;
input a, b, c, d;
input [1:0] sel;
reg out;

always @(sel or a or b or c or d)
begin
   case (sel)
      2'd0: out = a;
      2'd1: out = b;
      2'd3: out = d;
   endcase
end
endmodule
```

In the above example, the sel case value 2'd2 was intentionally omitted. Accordingly, out is not updated when the select line has the value 2'd2, and a level-sensitive latch must be added to hold the last value of out under this condition. The tool issues a warning message when you compile this module, and there can be mismatches between RTL simulation and post-synthesis simulation. You can avoid generating level-sensitive latches by adding the missing case in the case statement; using a “default” case in the case statement; or using the Verilog full_case directive.

Sets and Resets

A set signal is an input to a flip-flop that, when activated, sets the state of the flip-flop to a logic one. Asynchronous sets take place independent of the clock, whereas synchronous sets only occur on an active clock edge.

A reset signal is an input to a flip-flop that, when activated, sets the state of the flip-flop to a logic zero. Asynchronous resets take place independent of the clock, whereas synchronous resets take place only at an active clock edge.

Asynchronous Sets and Resets

Asynchronous sets and resets are independent of the clock. When active, they set flip-flop outputs to one or zero (respectively), without requiring an active clock edge. Therefore, list them in the event control of the always block, so that they trigger the always block to execute, and so that you can take the appropriate action when they become active.

Event Control Syntax

always @ (*edgeKeyword clockSignal* **or** *edgeKeyword resetSignal* **or**
edgeKeyword setSignal **)**

EdgeKeyword is posedge for active-high set or reset (or positive-edge triggered clock) or negedge for active-low set or reset (or negative-edge triggered clock).

You can list the signals in any order.

Example: Event Control

```
// Asynchronous, active-high set (rising-edge clock)
always @(posedge clk or posedge set)

// Asynchronous, active-low reset (rising-edge clock)
always @(posedge clk or negedge reset)

// Asynchronous, active-low set and active-high reset
// (rising-edge clock)
always @(posedge clk or negedge set or posedge reset)
```

Example: always Block Template with Asynch, Active-high reset, set

```
always @(posedge clk or posedge set or posedge reset)
begin
   if (reset) begin

      /* Set the outputs to zero */

   end else if (set) begin

      /* Set the outputs to one */
   end else begin

      /* Clocked logic */
   end
end
```

Example: flip-flop with Asynchronous, Active-high reset and set

```
module dff1 (q, qb, d, clk, set, reset);
input d, clk, set, reset;
output q, qb;
// Declare q and qb as reg because assigned inside always
reg q, qb;

always @(posedge clk or posedge set or posedge reset)
begin
   if (reset) begin
      q <= 0;
      qb <= 1;
   end else if (set) begin
      q <= 1;
      qb <= 0;
   end else begin
      q <= d;
      qb <= ~d;
   end
end
endmodule
```

For simple, single variable flip-flops, the following template can be used.

```
always @(posedge clk or posedge set or posedge reset)

q = reset ? 1'b0 : set ? 1'b1 : d;
```

Synchronous Sets and Resets

Synchronous sets and resets set flip-flop outputs to logic 1 or 0 (respectively) on an active clock edge.

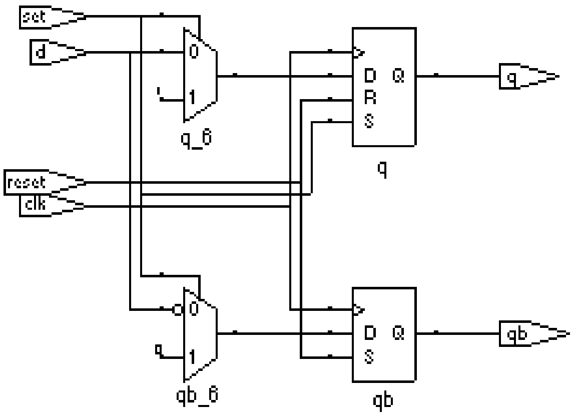
Do not list the set and reset signal names in the event expression of an always block so they do not trigger the always block to execute upon changing. Instead, trigger the always block on the active clock edge, and check the reset and set inside the always block first.

RTL View Primitives

The Verilog compiler can detect and extract the following flip-flops with synchronous sets and resets and display them in the schematic view:

- sdffr – flip-flop with synchronous reset
- sdffs – flip-flop with synchronous set
- sdffrs – flip-flop with both synchronous set and reset
- sdffpat – vectored flip-flop with synchronous set/reset pattern
- sdffre – enabled flip-flop with synchronous reset
- sdffse – enabled flip-flop with synchronous set
- sdffpate – enabled, vectored flip-flop with synchronous set/reset pattern

You can check the name (type) of any primitive by placing the mouse pointer over it in the schematic view: a tooltip displays the name. The following figure shows flip-flops with synchronous sets and resets.



Event Control Syntax

always @ (*edgeKeyword clockName* **)**

In the syntax line, *edgeKeyword* is posedge for a positive-edge triggered clock or negedge for a negative-edge triggered clock.

Example: Event Control

```
// Positive edge triggered
always @(posedge clk)

// Negative edge triggered
always @(negedge clk)
```

Example: always Block Template with Synchronous, Active-high reset, set

```
always @(posedge clk)
begin
   if (reset) begin
      /* Set the outputs to zero */
   end else if (set) begin
      /* Set the outputs to one */
   end else begin
      /* Clocked logic */
   end
end
```

Example: D Flip-flop with Synchronous, Active-high set, reset

```
module dff2 (q, qb, d, clk, set, reset);
input d, clk, set, reset;
output q, qb;
reg q, qb;

always @(posedge clk)
begin
   if (reset) begin
      q <= 0;
      qb <= 1;
   end else if (set) begin
      q <= 1;
      qb <= 0;
   end else begin
      q <= d;
      qb <= ~d;
   end
end
endmodule
```

SRL Inference

Sequential elements can be mapped into SRLs using an initialization assignment in the Verilog code. You can now infer SRLs with initialization values. Enable the System Verilog option on the Verilog tab of the Implementation Options dialog box before you run synthesis.

This is an example of a SRL with no resets. It has four 4-bit wide registers and a 4-bit wide read address. Registers shift when the write enable is 1.

```
module test_srl(clk, enable, dataIn, result, addr);
input clk, enable;
input [3:0] dataIn;
input [3:0] addr;
output [3:0] result;
reg [3:0] regBank[3:0]='{4'h0,4'h1,4'h2,4'h3};
integer i;

always @(posedge clk) begin
   if (enable == 1) begin
      for (i=3; i>0; i=i-1) begin
         regBank[i] <= regBank[i-1];
      end
   regBank[0] <= dataIn;
   end
end

assign result = regBank[addr];
endmodule
```

Verilog State Machines

This section describes Verilog state machines: guidelines for using them, defining state values, and dealing with asynchrony. The topics include:

State Machine Guidelines

A finite state machine (FSM) is hardware that advances from state to state at a clock edge.

The tool works best with synchronous state machines. You typically write a fully synchronous design and avoid asynchronous paths such as paths through the asynchronous reset of a register. See Asynchronous State Machines, on page 92, for information about asynchronous state machines.

- The state machine must have a synchronous or asynchronous reset, to be inferred. State machines must have an asynchronous or synchronous reset to set the hardware to a valid state after power-up, and to reset your hardware during operation (asynchronous resets are available freely in most FPGA architectures).
- You can define state machines using multiple event controls in an always block only if the event control expressions are identical (for example, @(posedge clk)). These state machines are known as implicit state machines. However it is better to use the explicit style described here and shown in Example – FSM Coding Style, on page 90.
- Separate the sequential from the combinational always block statements. Besides making it easier to read, it makes what is being registered very obvious. It also gives better control over the type of register element used.
- Represent states with defined labels or enumerated types.
- Use a case statement in an always block to check the current state at the clock edge, advance to the next state, then set the output values. You can use if statements in an always block, but stay with case statements, for consistency.

- Always use a default assignment as the last assignment in your case statement and set the state variable to 'bx. See Example: default Assignment, on page 90.
- Set encoding style with the syn_encoding directive. This attribute overrides the default encoding assigned during compilation. The default encoding is determined by the number of states. See syn_encoding Values, on page 67 for a list of default and other encodings. When you specify a particular encoding style with syn_encoding, that value is used during the mapping stage to determine encoding style.

  object **/*synthesis syn_encoding="sequential"*/;**

  See syn_encoding, on page 67, for details about the syntax and values.

  One-hot implementations are not always the best choice for state machines, even in FPGAs and CPLDs. For example, one-hot state machines might result in larger implementations, which can cause fitting problems. An example of an FPGA where one-hot implementation can be detrimental is a state machine that drives a large decoder, generating many output signals. In a 16-state state machine, for instance, the output decoder logic might reference sixteen signals in a one-hot implementation, but only four signals in a sequential representation.

Example – FSM Coding Style

Example: default Assignment

```
default: state = 'bx;
```

Assigning 'bx to the state variable (a “don't care” for synthesis) tells the tool that you have specified all the used states in your case statement. Any remaining states are not used, and the tool can remove unnecessary decoding and gates associated with the unused states. You do not have to add any special, non-Verilog directives.

If you set the state to a used state for the default case (for example, default state = state1), the tool generates the same logic as if you assign 'bx, but there will be pre- and post-synthesis simulation mismatches until you reset the state machine. These mismatches occur because all inputs are unknown at start up on the simulator. You therefore go immediately into the default case, which sets the state variable to state1. When you power up the hardware, it can be in a used state, such as state2, and then advance to a state other than state1. Post-synthesis simulation behaves more like hardware with respect to initialization.

State Values

In Verilog, you must give explicit state values for states. You do this using parameter or `define statements. It is recommended that you use parameter, for the following reasons:

- The `define is applied globally whereas parameter definitions are local. With global `define definitions, you cannot reuse common state names that you might want to use in multiple designs, like RESET, IDLE, READY, READ, WRITE, ERROR and DONE. Local definitions make it easier to reuse certain state names in multiple FSM designs. If you work around this restriction by using `undef and then redefining them with `define in the new FSM modules, it makes it difficult to probe the internal values of FSM state buses from a testbench and compare them to state names.
- The tool only displays state names in the FSM Viewer if they are defined using parameter.

Example 1: Using Parameters for State Values

```
parameter state1 = 2'h1, state2 = 2'h2;
...
current_state = state2; // Setting current state to 2'h2
```

Example 2: Using `define for State Values

```
`define state1     2'h1
`define state2     2'h2
...
current_state = `state2; // Setting current state to 2'h2
```

Asynchronous State Machines

Avoid defining asynchronous state machines in Verilog. An asynchronous state machine has states, but no clearly defined clock, and has combinational loops.

Do not use the tool to design asynchronous state machines; the software might remove your hazard-suppressing logic when it performs logic optimization, causing your asynchronous state machines to work incorrectly.

The tool displays a “Found combinational loop” warning message for an asynchronous state machine when it detects combinational loops in continuous assignment statements, always blocks, and built-in gate-primitive logic.

To create asynchronous state machines, do one of the following:

- To use Verilog, make a netlist of technology primitives from your target library. Any instantiated technology primitives are left in the netlist, and not removed during optimization.
- Use a schematic editor (and not Verilog) for the asynchronous state machine part of your design.

The following asynchronous state machine examples generate warning messages.

Example - Asynchronous FSM with Continuous Assignment

Example - Asynchronous FSM with an always Block

Example - READ Address Registered

Example - Data Output Registered

Instantiating Black Boxes in Verilog

Black boxes are modules with just the interface specified; internal information is ignored by the software. Black boxes can be used to directly instantiate:

- Technology-vendor primitives and macros (including I/Os).
- User-designed macros whose functionality was defined in a schematic editor, or another input source. (When the place-and-route tool can merge design netlists from different sources.)

Black boxes are specified with the syn_black_box directive. If the macro is an I/O, use black_box_pad_pin=1 on the external pad pin. The input, output, and delay through a black box are specified with special black box timing directives (see syn_black_box, on page 44).

For most of the technology-vendor architectures, macro libraries are provided (in *installDirectory*/lib/*technology*/*family*.v) that predefine the black boxes for their primitives and macros (including I/Os).

Verilog simulators require a functional description of the internals of a black box. To ensure that the functional description is ignored and treated as a black box, use the translate_off and translate_on directives. See translate_off/translate_on, on page 284 for information on the translate_off and translate_on directives.

If the black box has tristate outputs, you must define these outputs with a black_box_tri_pins directive (see black_box_tri_pins, on page 20).

For information on how to instantiate black boxes and technology-vendor I/Os, see *Defining Black Boxes for Synthesis, on page 398* of the *User Guide*.

PREP Verilog Benchmarks

PREP (Programmable Electronics Performance) Corporation distributes benchmark results that show how FPGA vendors compare with each other in terms of device performance and area. The following PREP benchmarks are included in the *installDirectory*/examples/verilog/common_rtl/prep:

- PREP Benchmark 1, Data Path (prep1.v)
- PREP Benchmark 2, Timer/Counter (prep2.v)
- PREP Benchmark 3, Small State Machine (prep3.v)
- PREP Benchmark 4, Large State Machine (prep4.v)
- PREP Benchmark 5, Arithmetic Circuit (prep5.v)
- PREP Benchmark 6, 16-Bit Accumulator (prep6.v)
- PREP Benchmark 7, 16-Bit Counter (prep7.v)
- PREP Benchmark 8, 16-Bit Pre-scaled Counter (prep8.v)
- PREP Benchmark 9, Memory Map (prep9.v)

The source code for the benchmarks can be used for design examples for synthesis or for doing your own FPGA vendor comparisons.

Hierarchical or Structural Verilog Designs

This section describes the creation and use of hierarchical Verilog designs:

Using Hierarchical Verilog Designs

The software accepts and processes hierarchical Verilog designs. You create hierarchy by instantiating a module or a built-in gate primitive within another module.

The signals connect across the hierarchical boundaries through the port list, and can either be listed by position (the same order that you declare them in the lower-level module), or by name (where you specify the name of the lower-level signals to connect to). Connecting by name minimizes errors, and can be especially advantageous when the instantiated module has many ports.

Creating a Hierarchical Verilog Design

To create a hierarchical design:

1. Create modules.
2. Instantiate the modules within other modules. (When you instantiate modules inside of others, the ones that you have instantiated are sometimes called “lower-level modules” to distinguish them from the “top-level” module that is not inside of another module.)
3. Connect signals in the port list together across the hierarchy either “by position” or “by name” (see the examples, below).

Example: Creating Modules (Interfaces Shown)

```
module mux(out, a, b, sel); // mux
output [7:0] out;
input [7:0] a, b;
input sel;

// mux functionality

endmodule

module reg8(q, data, clk, rst); // Eight-bit register
output [7:0] q;
input [7:0] data;
input clk, rst;
// Eight-bit register functionality
endmodule

module rotate(q, data, clk, r_l, rst); // Rotates bits or loads
output [7:0] q;
input [7:0] data;
input clk, r_l, rst;
// When r_l is high, it rotates; if low, it loads data
// Rotate functionality
endmodule
```

Example: Top-level Module with Ports Connected by Position

```
module top1(q, a, b, sel, r_l, clk, rst);
output [7:0] q;
input [7:0] a, b;
input sel, r_l, clk, rst;
wire [7:0] mux_out, reg_out;

// The order of the listed signals here will match
// the order of the signals in the mux module declaration.
mux mux_1 (mux_out, a, b, sel);
reg8 reg8_1 (reg_out, mux_out, clk, rst);
rotate rotate_1 (q, reg_out, clk, r_l, rst);

endmodule
```

Example: Top-level Module with Ports Connected by Name

```
module top2(q, a, b, sel, r_l, clk, rst);
output [7:0] q;
input [7:0] a, b;
input sel, r_l, clk, rst;
wire [7:0] mux_out, reg_out;

/* The syntax to connect a signal "by name" is:
.<lower_level_signal_name>(<local_signal_name>)
*/
mux mux_1 (.out(mux_out), .a(a), .b(b), .sel(sel));

/* Ports connected "by name" can be in any order */
reg8 reg8_1 (.clk(clk), .data(mux_out), .q(reg_out), .rst(rst));
rotate rotate_1 (.q(q), .data(reg_out), .clk(clk),
   .r_l(r_l), .rst(rst) );
endmodule
```

Include Files

The `include compiler directive can be used to insert the entire contents of a source file within another source file during compilation. The result appears as though the contents of the included source file replaces the `include compiler directive.

The included file is compiled with the same options (i.e., Verilog standard or defines) as the file in which it is included. Suppose that the file a.h has option set to vlog_std v95 and is included within top.v, where vlog_std sysv has been added from the Tcl command line. For this example, the compiler uses the sysv option since the command line has higher precedence than the set_option.

synthesis Macro

Use this text macro along with the Verilog `ifdef compiler directive to conditionally exclude part of your Verilog code from being synthesized. The most common use of the synthesis macro is to avoid synthesizing stimulus that only has meaning for logic simulation.

The synthesis (or SYNTHESIS; either all upper case or all lower case is accepted) macro is defined so that when the statement `ifdef synthesis is true, the statements in the `ifdef branch are compiled, and the stimulus statements in the `else branch are ignored.

Note: Because Verilog simulators do *not* recognize a synthesis macro, the compiler for your simulator will use the stimulus in the `else branch.

In the following example, the AND gate in the `ifdef branch is inserted during synthesis when the compiler recognizes the synthesis macro and takes the assign c = a & b branch. Conversely, during simulation, the OR gate is inserted when the simulator ignores the synthesis macro and takes the assign c = a | b branch.

Note: A macro in Verilog has a non-zero value only if it is defined.

```
module top (a,b,c);
   input a,b;
   output c;
`ifdef SYNTHESIS
   assign c = a & b;
`else
   assign c = a | b;
`endif
endmodule
```

text Macro

The directive define creates a macro for text substitution. The compiler substitutes the text of the macro for the string *macroName*. A text macro is defined using arguments that can be customized for each individual use.

The syntax for a text macro definition is as follows.

textMacroDefinition **::= define** *textMacroName macroText*

textMacroName **::=** *textMacroIdentifier*[**(***formalArgumentList***)**]

formalArgumentList **::=** formalArgumentIdentifier **{,** formalArgumentIdentifier**}**

When formal arguments are used to define a text macro, the scope of the formal argument is extended to the end of the macro text. You can use a formal argument in the same manner as an identifier.

A text macro with one or more arguments is expanded by replacing each formal argument with the actual argument expression.

Example 1

```
`define MIN(p1, p2) (p1)<(p2)?(p1):(p2)

module example1(i1, i2, o);
input i1, i2;
output o;
reg o;

always @(i1, i2) begin
o = `MIN(i1, i2);
end
endmodule
```

Example 2

```
`define SQR_OF_MAX(a1, a2) (`MAX(a1, a2))*(`MAX(a1, a2))
`define MAX(p1, p2) (p1)<(p2)?(p1):(p2)

module example2(i1, i2, o);
input i1, i2;
output o;
reg o;

always @(i1, i2) begin
o = `SQR_OF_MAX(i1, i2);
end
endmodule
```

Example 3

Include File ppm_top_ports_def.inc

```
//ppm_top_ports_def.inc

// Single source definition for module ports and signals
// of PPM TOP.
// Input
`DEF_DOT `DEF_IN([7:0]) in_test1 `DEF_PORT(in_test1) `DEF_END
`DEF_DOT `DEF_IN([7:0]) in_test2 `DEF_PORT(in_test2) `DEF_END

// In/Out
// `DEF_DOT `DEF_INOUT([7:0]) io_bus1 `DEF_PORT(io_bus1) `DEF_END

// Output
`DEF_DOT `DEF_OUT([7:0]) out_test2 `DEF_PORT(out_test2)
// No DEF_END here...

`undef DEF_IN
`undef DEF_INOUT
`undef DEF_OUT
`undef DEF_END
`undef DEF_DOT
`undef DEF_PORT
```

Verilog File top.v

```
// top.v

`define INC_TYPE 1
module ppm_top(
 `ifdef INC_TYPE
// Inc file Port def...
   `define DEF_IN(arg1) /* arg1 */
   `define DEF_INOUT(arg1) /* arg1 */
   `define DEF_OUT(arg1) /* arg1 */
   `define DEF_END,
   `define DEF_DOT /* nothing */
   `define DEF_PORT(arg1) /* arg1 */

`include "ppm_top_ports_def.inc"
   `else
   // Non-Inc file Port def, above defines should expand to
   // what is below...
      /* nothing */ /* [7:0] */ in_test1 /* in_test1 */ ,
      /* nothing */ /* [7:0] */ in_test2 /* in_test2 */ ,

   // In/Out
   //`DEF_DOT `DEF_INOUT([7:0]) io_bus1 `DEF_PORT(io_bus1)
`DEF_END

   // Output
      /* nothing */ /* [7:0] */ out_test2 /* out_test2 */
// No DEF_END here...
 `endif
);

   `ifdef INC_TYPE
   // Inc file Signal type def...
   `define DEF_IN(arg1) input arg1
   `define DEF_INOUT(arg1) inout arg1
   `define DEF_OUT(arg1) output arg1
   `define DEF_END;
   `define DEF_DOT /* nothing */
   `define DEF_PORT(arg1) /* arg1 */

`include "ppm_top_ports_def.inc"
   `else
   // Non-Inc file Signal type def, defines should expand to
   // what is below...
      /* nothing */ input [7:0] in_test1 /* in_test1 */ ;
      /* nothing */ input [7:0] in_test2 /* in_test2 */ ;

// In/Out
   //`DEF_DOT `DEF_INOUT([7:0]) io_bus1 `DEF_PORT(io_bus1)`DEF_END

// Output
   /* nothing */ output [7:0] out_test2 /* out_test2) */
// No DEF_END here...
   `endif

 ; /* Because of the 'No DEF_END here...' in line of the include
file. */

   assign out_test2 = (in_test1 & in_test2);

endmodule
```

Verilog Attribute and Directive Syntax

Verilog attributes and directives allow you to associate information with your design to control the way it is analyzed, compiled, and mapped.

- *Attributes* direct the way your design is optimized and mapped during synthesis.
- *Directives* control the way your design is analyzed prior to mapping. They must therefore be included directly in your source code; they cannot be specified in a constraint file like attributes.

Verilog does not have predefined attributes or directives for synthesis. To define directives or attributes in Verilog, attach them to the appropriate objects in the source code as comments. You can use either of the following comment styles:

- Regular line comments
- Block or C-style comments

Each specification begins with the keyword synthesis. The directive or attribute value is either a string, placed within double quotes, or a Boolean integer (0 or 1). Directives, attributes, and their values are-case sensitive and are usually in lower case.

Attribute Syntax and Examples using Verilog Line Comments

Here is the syntax using a regular Verilog comment:

// synthesis *directive* | *attribute* [**= "***value***"**]

This example shows how to use the syn_hier attribute:

```
// synthesis syn_hier = "firm"
```

This example shows the parallel_case directive:

```
// synthesis parallel_case
```

This directive forces a multiplexed structure in Verilog designs. It is implicitly true whenever you use it, which is why there is no associated value.

Attribute Syntax and Examples Using Verilog C-Style Comments

Here is the syntax for specifying attributes and directives with the C-style block comment:

> **/* synthesis** *directive* **|** *attribute* [**= "***value***"**] ***/**

This example shows the syn_hier attribute specified with a C-style comment:

```
/* synthesis syn_hier = "firm" */
```

The following are some other rules for using C-style comments to define attributes:

- If you use C-style comments, you must place the comment *after* the *object* declaration and *before* the semicolon of the statement. For example:

  ```
  module bl_box(out, in) /* synthesis syn_black_box */ ;
  ```

- To specify more than one directive or attribute for a given design object, place them within the same comment, separated by a space. Do *not* use commas as separators. Here is an example where the syn_preserve and syn_state_machine directives are specified in a single comment:

  ```
  module radhard_dffrs(q,d,c,s,r)
     /* synthesis syn_preserve=1 syn_state_machine=0 */;
  ```

- To make source code more readable, you can split long block comment lines by inserting a backslash character (\) followed immediately by a newline character (carriage return). A line split this way is still read as a single line; the backslash causes the newline following it to be ignored. You can split a comment line this way any number of times. However, note these exceptions:
  - The first split cannot occur before the first attribute or directive specification.
  - A given attribute or directive specification cannot be split before its equal sign (=).

  Take this block comment specification for example:

```
/* synthesis syn_probe=1 xc_loc="P20,P21,P22,P23,P24,P25,P26,P27" */;
```

  You cannot split the line before you specify the first attribute, `syn_probe`. You cannot split the line before either of the equal signs (`syn_probe=` or `xc_loc=`). You can split it anywhere within the string value `"P20,P21,P22,P23,P24,P25,P26,P27"`.

Attribute Examples Using Verilog 2001 Parenthetical Comments

Here is the syntax for specifying attributes and directives as Verilog 2001 parenthetical comments:

(* *directive* **|***attribute* **[= "***value***"] *)**

Verilog 2001 parenthetical comments can be applied to:

- individual objects
- multiple objects
- individual objects within a module definition

The following example shows two syn_keep attributes specified as parenthetical comments:

```
module example1(out1, out2, clk, in1, in2);
output out1, out2;
input clk;
input in1, in2;
wire and_out;
(* syn_keep=1 *) wire keep1;
(* syn_keep=1 *) wire keep2;
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
```

For the above example, a single parenthetical comment could be added directly to the reg statement to apply the syn_keep attribute to both out1 and out2:

```
(* syn_keep=1 *) reg out1, out2;
```

The following rules apply when using parenthetical comments to define attributes:

- Always place the comment *before* the design object (and terminating semicolon). For example:

  ```
  (* syn_black_box *) module bl_box(out, in);
  ```

- To specify more than one directive or attribute for a given object, place the attributes within the same parenthetical comment, separated by a space (do *not* use commas as separators). The following example shows the syn_preserve and syn_state_machine directives applied in a single parenthetical comment:

  ```
  (* syn_preserve=1 syn_state_machine=0 *)
     module radhard_dffrs(q,d,c,s,r);
  ```

- Parenthetical comments can be applied to individual objects within a module definition. For example,

  ```
  module example2 (out1, (*syn_preserve=1*) out2, clk, in1, in2);
  ```

  applies a syn_preserve attribute to out2, and

  ```
  module example2 ( (*syn_preserve=1*) out1,
     (*syn_preserve=1*) out2, clk, in1, in2);
  ```

  applies a syn_preserve attribute to both out1 and out2

CHAPTER 2

SystemVerilog Language Support

This chapter describes support for the SystemVerilog standard for the Synopsys tool. For information on the Verilog standard, see Chapter 1, *Verilog Language Support*. SystemVerilog support includes:

Feature Summary

SystemVerilog is a IEEE (P1800) standard with extensions to the IEEE Std.1800-2009 SystemVerilog standard. The extensions integrate features from C, C++, VHDL, OVA, and PSL. The following table summarizes the SystemVerilog features currently supported in the Synopsys FPGA Verilog compilers. See SystemVerilog Limitations, on page 111 for a list of limitations.

Feature	Brief Description
Unsized Literals	Specification of unsized literals as single-bit values without a base specifier.
Data Types • Typedefs • Enumerated Types • Struct Construct • Union Construct • Static Casting	Data types that are a hybrid of both Verilog and C including: • User-defined types that allow you to create new type definitions from existing types • Variables and nets defined with a specific set of named values • Structure data type to represent collections of variables referenced as a single name • Data type collections sharing the same memory location • Conversion of one data type to another data type.
Arrays • Arrays • Arrays of Structures	Packed, unpacked, and multi-dimensional arrays of structures.
Data Declarations • Constants • Variables • Nets • Data Types in Parameters • Type Parameters	Data declarations including constant, variable, net, and parameter data types.

Feature	Brief Description
Operators and Expressions • Operators • Aggregate Expressions • Streaming Operator • Set Membership Operator • Set Membership Case Inside Operator • Type Operator	C assignment operators and special bit-wise assignment operators.
Procedural Statements and Control Flow • Do-While Loops • For Loops • Unnamed Blocks • Block Name on end Keyword • Unique and Priority Modifiers	Procedural statements including variable declarations and block functions.
Processes • always_comb • always_latch • always_ff	Specialized procedural blocks that reduce ambiguity and indicate the intent.
Tasks and Functions • Implicit Statement Group • Formal Arguments • endtask/endfunction Names	Information on implicit grouping for multiple statements, passing formal arguments, and naming end statements for functions and tasks.
Hierarchy • Compilation Units • Packages • Port Connection Constructs • Extern Module	Permits sharing of language-defined data types, user-defined types, parameters, constants, function definitions, and task definitions among one or more compilation units, modules, or interfaces (pkgs)
Interface • Interface Construct • Modports	Interface data type to represent port lists and port connection lists as single name.
System Tasks and System Functions • $bits System Function • Array Querying Functions	Queries to return the number of bits required to hold an expression as a bit stream or array.

Feature	Brief Description
Generate Statement: Conditional Generate Constructs	Generate-loop, generate-conditional, or generate-case statements with defparams, parameters, and function and task declarations. Conditional if-generate and case-generate constructs
Assertions	SystemVerilog assertion support.
Keyword Support	Supported and unsupported keywords.

SystemVerilog Limitations

The following SystemVerilog limitations are present in the release.

Feature	Limitations
Data Types	
• Enumerated Types	• Enumerated type methods do not support name() and cross-module referencing.
• Union Construct	• Union constructs do not support unpacked union, tagged packed union, and tagged unpacked union.
Operators and Expressions	
• Type Operator	• When the $typeof operator uses an expression as its argument, the expression cannot contain any hierarchical references or reference elements of dynamic objects. • The $typeof operator is not supported on complex expressions.
Hierarchy	
• Compilation Units	• Compilation unit elements can only be accessed or read, and cannot appear between module and endmodule statements.
• Packages	• The variables declared in packages can only be accessed or read; package variables cannot be written between a module statement and its end module statement
• Extern Module	• An extern module declaration is not supported within a module.

Feature	Limitations
Interface • Interface Construct • Modports	• An array of interfaces cannot be used as a module port. See Interface, on page 167. • An interface cannot have a multi-dimensional port. • Access of array type elements outside of the interface are not supported. See Compilation Units, on page 160. • For restrictions using interface/modport structures, see Modport Limitations and Non-Supported Features, on page 174.
System Tasks and System Functions • $bits System Function	• Passing an interface member as an argument to the $bits function is not supported • $bits cannot be used within a module instantiation • $bits is not supported with params/localparams
Generate Statement: Conditional Generate Constructs	The generate statement does not support the following functions: • Defparam support for generate instances • Hierarchical access for interface • Hierarchical access of function/task defined within a generate block

Interface

- An array of interfaces cannot be used as a module port.
- An interface cannot have a multi-dimensional port.
- Access of array type elements outside of the interface are not supported. For example:

```
interface ff_if (input logic din, input [7:0] DHAin1,
   input [7:0] DHAin2, output logic dout);
logic [1:0] [1:0] [1:0] DHAout_intf;
```

```
always_comb
DHAout_intf = DHAin1 + DHAin2;

modport write (input din, output dout);
endinterface: ff_if

ff_if ff_if_top(.*);
DHAout = ff_if_top.DHAout_intf;
```

- Modport definitions within a Generate block are not supported. For example:

```
interface myintf_if (input logic [7:0] a , input logic [7:0]  b,
   output logic [7:0] out1, output logic [7:0] out2);
generate
   begin: x
   genvar i;
      for (i = 0;i <= 7;i=i+1)
      begin : u
         modport myinst(input .ma(a[i]), input .mb(b[i]),
            output .mout1(out1[i]) , output .mout2(out2[i]));
      end
   end
endgenerate
endinterface
```

Compilation Unit and Package

- Write access to the variable defined in package/compilation unit is not supported. For example:

```
package MyPack;
typedef struct packed {
   int r;
   longint g;
   byte b;
}  MyStruct;

MyStruct StructMyStruct;
endpackage: MyPack

import MyPack::*;
module top ( ...
...

always@(posedge clk)
StructMyStruct <= '{default:254};
```

Unsized Literals

SystemVerilog allows you to specify unsized literals without a base specifier (auto-fill literals) as single-bit values with a preceding apostrophe ('). All bits of the unsized value are set to the value of the specified bit.

```
'0, '1, 'X, 'x, 'Z, 'z // sets all bits to this value
```

In other words, this feature allows you to fill a register, wire, or any other data types with 0, 1, X, or Z in simple format.

Verilog Example	SystemVerilog equivalent
a = 4'b1111;	a = '1;

Data Types

SystemVerilog makes a clear distinction between an *object* and its *data type*. A data type is a set of values, or a set of operations that can be performed on those values. Data types can be used to declare data objects.

SystemVerilog offers the following data types, which represent a hybrid of both Verilog and C:

Data Type	Description
shortint	2-state, SystemVerilog data type, 16-bit signed integer
int	2-state, SystemVerilog data type, 32-bit signed integer
longint	2-state, SystemVerilog data type, 64-bit signed integer
byte	2-state, SystemVerilog data type, 8-bit signed integer or ASCII character
bit	2-state, SystemVerilog data type, user-defined vector size
logic	4-state, SystemVerilog data type, user-defined vector size

Data types are characterized as either of the following:

- 4-state (4-valued) data types that can hold 1, 0, X, and Z values
- 2-state (2-valued) data types that can hold 1 and 0 values

The following apply when using data types:

- The data types byte, shortint, int, integer and longint default to signed; data types bit, reg, and logic default to unsigned, as do arrays of these types.
- The signed keyword is part of Verilog. The unsigned keyword can be used to change the default behavior of signed data types.
- The Verilog compiler does not generate an error even if a 2-state data type is assigned X or Z. It treats it as a “don't care” and issues a warning.
- The syn_keep directive provides limited support with SystemVerilog data types such as logic, wire, and bit.

Typedefs

You can create your own names for type definitions that you use frequently in your code. SystemVerilog adds the ability to define new net and variable user-defined names for existing types using the typedef keyword.

Example – Simple typedef Variable Assignment

Example – Using Multiple typedef Assignments

Enumerated Types

The tool supports SystemVerilog enumerated types in accordance with SV LRM section: 6.19.

The enumerated types feature allows variables and nets to be defined with a specific set of named values. This capability is particularly useful in state-machine implementation where the states of the state machine can be verbally represented.

Data Types

Enumerated types have a base data type which, by default, is int (a 2-state, 32-bit value). By default, the first label in the enumerated list has a logic value of 0, and each subsequent label is incremented by one.

For example, a variable that has three legal states:

```
enum {WAITE, LOAD, READY} state;
```

The first label in the enumerated list has a logic value of 0 and each subsequent label is incremented by one. In the example above, State is an int type and WAITE, LOAD And READY have 32-bit int values. WAITE is 0, LOAD is 1, and READY is 2.

For this example, an explicit base type of logic is specified that allows the enumerated types of state to more specifically model hardware:

```
enum logic [2:0] {WAITE=3'b001, LOAD=3'b010,READY=3'b100} state;
```

Specifying Ranges

SystemVerilog enumerated types also allow you to specify ranges that are automatically elaborated. Types can be specified as outlined in the following table.

Syntax	Description
name	Associates the next consecutive number with the specified name.
name = C	Associates the constant C to the specified name.
name[*N*]	Generates N named constants in this sequence: *name0, name1,..., nameN-1*. N must be a positive integral number.

Syntax	Description
name[*N*] = C	Optionally assigns a constant to the generated named constants to associate that constant with the first generated named constant. Subsequent generated named constants are associated with consecutive values. N must be a positive integral number.
name[*N*:*M*]	Creates a sequence of named constants, starting with *nameN* and incrementing or decrementing until it reaches named constant *nameM*. N and M are non-negative integral numbers.
name[*N*:*M*] = C	Optionally assigns a constant to the generated named constants to associate that constant with the first generated named constants. Subsequent generated named constants are associated consecutive values. N and M must be positive integral numbers.

The following example declares enumerated variable vr, which creates the enumerated named constants register0 and register1, which are assigned the values 1 and 2, respectively. Next, it creates the enumerated named constants register2, register3, and register4 and assigns them the values 10, 11, and 12.

```
enum { register[2] = 1, register[2:4] = 10 } vr;
```

State-Machine Example

The following is an example state-machine design in SystemVerilog.

Example – State-machine Design

Type Casting Using Enumerated Types

By using enumerated types, you can define a type. For example:

```
typedef enum { red,green,blue,yellow,white,black} Colors;
```

The above definition assigns a unique number to each of the color identifiers and creates the new data type Colors. This new type can then be used to create variables of that type.

Valid assignment would be:

```
Colors c;
```

```
C = green;
```

Enumerated Types in Expressions

Elements of enumerated types can be used in numerical expressions. The value used in the expression is the value specified with the numerical value. For example:

```
typedef enum { red,green,blue,yellow,white,black} Colors;
integer a,b;
a = blue *3 // 6 is assigned to a
b = yellow + green; // 4 is assigned to b
```

Enumerated Type Methods

SystemVerilog provides a set of specialized methods to iterate values of enumerated types. The enumerated type method can be used to conveniently code logic such as a state machine. Apply the enumerated type methods on the specified enumerated type variable, using any of the methods below:

- first – Returns the first member of the enumeration.

  ```
  enum first( );
  ```

- last – Returns the last member of the enumeration.

  ```
  enum last( );
  ```

- next – Returns the next n^{th} enumeration value starting from the current value of the specified variable.

  ```
  enum next( );
  ```

- prev – Returns the previous n^{th} enumeration value starting from the current value of the specified variable.

  ```
  enum prev( );
  ```

- num – Returns the number of elements for the specified enumerations.

  ```
  int num( );
  ```

Note: Only the prev and next constructs support argument values.

The following code example shows that enumeration methods can be used to traverse the FSM, instead of having to explicitly specify the enumeration.

Example - Enumerated Type Method

Enumerated Type Limitations

The compiler does not support enumerated type methods with:

- Enumeration type method of name( )
- Cross-module referencing (XMR)

Struct Construct

SystemVerilog adds several enhancements to Verilog for representing large amounts of data. In SystemVerilog, the Verilog array constructs are extended both in how data can be represented and for operations on arrays. A structure data type has been defined as a means to represent collections of data types. These data types can be either standard data types (such as int, logic, or bit) or, they can be user-defined types (using SystemVerilog typedef). Structures allow multiple signals, of various data types, to be bundled together and referenced by a single name.

Structures are defined under section 4.11 of IEEE Std 1800-2005 (IEEE Standard for SystemVerilog).

In the example structure floating_pt_num below, both characteristic and mantissa are 32-bit values of type bit.

```
struct {
   bit [31:0] characteristic;
   bit [31:0] mantissa;
}  floating_pt_num;
```

Alternately, the structure could be written as:

```
typedef struct {
   bit [31:0] characteristic;
   bit [31:0] mantissa;
}  flpt;
flpt floating_pt_num;
```

In the above sequence, a type flpt is defined using typedef which is then used to declare the variable floating_pt_num.

Assigning a value to one or more fields of a structure is straight-forward.

```
floating_pt_num.characteristic = 32'h1234_5678;

floating_pt_num.mantissa       = 32'h0000_0010;
```

As mentioned, a structure can be defined with fields that are themselves other structures.

```
typedef struct{
   flpt x;
   flpt y;
}  coordinate;
```

Packed Struct

Various other unique features of SystemVerilog data types can also be applied to structures. By default, the members of a structure are *unpacked*, which allows the Synopsys tool to store structure members as independent objects. It is also possible to *pack* a structure in memory without gaps between its bit fields. This capability can be useful for fast access of data during simulation and possibly result in a smaller footprint of your simulation binary.

To pack a structure in memory, use the packed keyword in the definition of the structure:

```
typedef struct packed {
   bit [31:0] characteristic;
   bit [31:0] mantissa;
} flpt;
```

An advantage of using packed structures is that one or more bits from such a structure can be selected as if the structure was a packed array. For instance, flpt[47:32] in the above declaration is the same as characteristic[15:0].

Struct members are selected using the .name syntax as shown in the following two code segments.

```
// segment 1
typedef struct {
   bit [7:0] opcode;
   bit [23:0] addr;
} instruction; // named structure type
instruction IR; // define variable
IR.opcode = 1; //set field in IR.

// segment 2
struct {
   int x,y;
} p;
p.x = 1;
```

Union Construct

A union is a collection of different data types similar to a structure with the exception that members of the union share the same memory location. At any given time, you can write to any one member of the union which can then be read by the same member or a different member of that union.

Union is broadly classified as:

- Packed Union
- Unpacked Union

Currently, only packed unions are supported.

Packed Union

A packed union can only have members that are of the packed type (packed structure, packed array of logic, bit, int, etc.). All members of a packed union must be of equal size.

Syntax

Union packed
{
    member1;
    member2;
} *unionName*;

Unpacked Union

The members of an unpacked union can include both packed and unpacked types (packed/unpacked structures, arrays of packed/unpacked logic, bit, int, etc.) with no restrictions as to the size of the union members.

Syntax

Union
{
    *member1***;**
    *member2***;**
} *unionName***;**

Example 1 – Basic Packed Union (logical operation)

Example 2 – Basic Packed Union (arithmetic operation)

Example 3 – Nested Packed Union

Example 4 – Array of packed Union

Union Construct Limitations

The SystemVerilog compiler does not support the following union constructs:

- unpacked union
- tagged packed union
- tagged unpacked union

Currently, support is limited to packed unions, arrays of packed unions, and nested packed unions.

Static Casting

Static casting allows one data type to be converted to another data type. The static casting operator is used to change the data type, the size, or the sign:

- Type casting – a predefined data type is used as a *castingType* to change the data type.
- Size casting – a positive decimal number is used as a *castingType* to change the number of data bits.
- Sign casting – signed/unsigned are used to change the sign of data type.
- Bit-stream casting – type casting that is applied to unpacked arrays and structs. During bit-stream casting, both the left and right sides of the equation must be the same size. Arithmetic operations cannot be combined with static casting operations as is in the case of singular data types.

Syntax

castingType **'** **(***castingExpression***)**

Example – Type Casting of Singular Data Types

Example – Type Casting of Aggregate Data Types

Example – Bit-stream Casting

Example – Size Casting

Example – Sign Casting

Arrays

Topics in this section include:

Arrays

SystemVerilog uses the term *packed array* to refer to the dimensions declared before the object name (same as Verilog *vector width*). The term *unpacked array* refers to the dimensions declared after the object name (same as Verilog *array dimensions*). For example:

```
reg [7:0] foo1; //packed array
reg foo2 [7:0]; //unpacked array
```

A packed array is guaranteed to be represented as a contiguous set of bits and, therefore, can be conveniently accessed as array elements. While unpacked is not guaranteed to work so, but in terms of hardware, both would be treated or bit-blasted into a single dimension.

```
module test1 (input [3:0] data, output [3:0] dout);
   //example on packed array four-bit wide.

assign dout = data;
endmodule

module test2 (input data [3:0], output dout [3:0]);
//unpacked array of 1 bit by 4 depth;

assign dout = data;
endmodule
```

Multi-dimensional packed arrays unify and extend Verilog's notion of *registers* and *memories*:

```
reg [1:0][2:0] my_var[32];
```

Classical Verilog permitted only one dimension to be declared to the left of the variable name. SystemVerilog permits any number of such *packed* dimensions. A variable of packed array type maps 1:1 onto an integer arithmetic quantity. In the example above, each element of my_var can be used in expressions as a six-bit integer. The dimensions to the right of the name (32 in this case) are referred to as *unpacked* dimensions. As in Verilog-2001, any number of unpacked dimensions is permitted.

The general rule for multi-dimensional packed array is as follows:

```
reg/wire [matrix_n:0] … [matrix_1:0][depth:0][width:0] temp;
```

The general rule for multi-dimensional unpacked array is as follows:

```
reg/wire temp1 [matrix_n:0]… [matrix_1:0][depth:0]; //single bit wide
reg/wire [width_m:0] temp2 [matrix_n:0]… [matrix_1:0][depth:0];
   // width_m bit wide
```

The general rule for multi-dimensional array, mix of packed/unpacked, is as follows:

```
reg/wire [width_m:0] temp3 [matrix:0]… [depth:0];

reg/wire [depth:0][width:0] temp4 [matrix_m:0]… [matrix_1:0]
```

For example, in a multi-dimensional declaration, the dimensions declared following the type and before the name vary more rapidly than the dimensions following the name.

Multi-dimensional arrays can be used as ports of the module.

The following items are now supported for multi-dimensional arrays:

- Assignment of a whole multi-dimensional array to another.
- Access (reading) of an entire multi-dimensional array.
- Assignment of an index (representing a complete dimension) of a multi-dimensional array to another.
- Access (reading) of an index of a multi-dimensional array.
- Assignment of a slice of a multi-dimensional array.
- Access of a slice of a multi-dimensional array.
- Access of a variable part-select of a multi-dimensional array.

In addition, wire declarations are supported for any packed or unpacked data type. This support includes multi-dimensional enum and struct data types in input port declarations (see Nets, on page 130 for more information).

Packed arrays are supported with the access/store mechanisms listed above. Packed arrays can also be used as ports and arguments to functions and tasks. The standard multi-dimensional access of packed arrays is supported.

Unpacked array support is the same as packed array supported stated in items one through seven above.

Example – Multi-dimensional Packed Array with Whole Assignment

Example – Multi-dimensional Packed Array with Partial Assignment

Example – Multi-dimensional Packed Array with Arithmetic Ops

Example – Packed/Unpacked Array with Partial Assignment

Arrays of Structures

SystemVerilog supports multi-dimensional arrays of structures which can be used in many applications to manipulate complex data structures. A multi-dimensional array of structure is a structured array of more than one dimension. The structure can be either packed or unpacked and the array of this structure can be either packed or unpacked or a combination of packed and unpacked. As a result, there are many combinations that define a multi-dimensional array of structure.

A multi-dimensional array of structure can be declared as either anonymous type (inline) or by using a typedef (user-defined data type).

Some applications where multi-dimensional arrays of structures can be used are where multi-channeled interfaces are required such as packet processing, dot-product of floating point numbers, or image processing.

Example – Multi-dimensional Array of Packed and Unpacked Structures Using typedef

Example – Multi-dimensional Array of UnPacked Structures Using typedef

Example – Multi-dimensional Array of Packed Structures Using Anonymous Type

Example – Multi-dimensional Array of Packed Structures Using typedef

Array Querying Functions

SystemVerilog provides system functions that return information about a particular dimension of an array. For information on this function, see Array Querying Functions, on page 181.

Data Declarations

There are several data declarations in SystemVerilog: *literals*, *parameters*, *constants*, *variables*, *nets*, and *attributes*. The following are described here:

- Constants, on page 129
- Variables, on page 129
- Nets, on page 130
- Data Types in Parameters, on page 131
- Type Parameters, on page 131

Constants

Constants are named data variables, which never change. A typical example for declaring a constant is as follows:

```
const a = 10;

const logic  [3:0] load = 4'b1111;

const reg  [7:0] load1 = 8'h0f, dataone = '1;
```

The Verilog compiler generates an error if constant is assigned a value.

```
const shortint a = 10;
assign a = '1;     // This is illegal
```

Variables

Variables can be declared two ways:

Method 1	Method 2

shortint a, b;
logic [1:0] c, d;
	var logic [15:0] a;
	var a,b; // equivalent var logic a, b
	var [1:0] c, d; // equivalent var logic [1:0] c, d
	input var shortint datain1,datain2;
	output var logic [15:0] dataout1,dataout2;

Method 2 uses the keyword var to preface the variable. In this type of declaration, a data type is optional. If the data type is not specified, logic is inferred.

Typical module declaration:

```
module test01 (input var shortint datain1,datain2,
   output var logic [15:0] dataout1,dataout2);
```

A variable can be initialized as follows:

```
var  a = 1'b1;
```

Nets

Nets are typically declared using the wire keyword. Any 4-state data type can be used to declare a net. When using wire with struct and union constructs, each member of the construct must be a 4-state data type.

Syntax

wire *4stateDataType identifierName*;

Example – Logic Type Defined as a Wire Type

```
module top (
   input wire logic [1:0] din1,din2, // logic defined as wire
   output logic [1:0] dout);
   assign dout = din1 + din2;
endmodule
```

Example – struct Defined as a Wire Type

```
typedef struct { logic [4:1] a;
}  MyStruct;

module top (
   input wire MyStruct [1:0] din [1:0] [1:0], // structure
      // defined as wire
   output wire MyStruct [1:0] dout [1:0] [1:0] ); // structure
      // defined as wire
assign dout = din;
endmodule
```

Restrictions

Using wire with a 2-state data type (for example, int or bit) results in the following error message:

CG1205 | Net data types must be 4-state values

A lexical restriction also applies to a net or port declaration in that the net type keyword wire cannot be followed by reg.

Data Types in Parameters

In SystemVerilog with different data types being introduced, the *parameter* can be of any data type (i.e., language-defined data type, user-defined data type, and packed/unpacked arrays and structures). By default, parameter is the int data type.

Syntax

parameter *dataType varaibleName* **=** *value*

In the above syntax, *dataType* is a language-defined data type, user-defined data type, or a packed/unpacked structure or array.

Example – Parameter is of Type longint

Example – Parameter is of Type enum

Example – Parameter is of Type structure

Example – Parameter is of Type longint Unpacked Array

Type Parameters

SystemVerilog includes the ability for a parameter to also specify a data type. This capability allows modules or instances to have data whose type is set for each instance – these *type* parameters can have different values for each of their instances.

Note: Overriding a type parameter with a defparam statement is illegal.

Syntax

parameter type *typeIdentifierName* **=** *dataType***;**

localparam type *typeIdentifierName* **=** *dataType***;**

In the above syntax, *dataType* is either a language-defined data type or a user-defined data type.

Example - Type Parameter of Language-Defined Data Type

Example - Type Parameter of User-Defined Data Type

Example - Type Local Parameter

Operators and Expressions

Topics in this section include:

Operators

SystemVerilog includes the C assignment operators and special bit-wise assignment operators:

+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, <<<=, >>>=

An assignment operator is semantically equivalent to a blocking assignment with the exception that the expression is only evaluated once.

Operator Example	**Same as**
`A += 2;`	`A = A + 2;`
`B -= B;`	`B = B - A;`
`C *= B;`	`C = C * B;`
`D /= C;`	`D = D / C;`
`E %= D;`	`E = E % D;`
`F &= E;`	`F = F & E;`
`G |= F;`	`G = G | F;`
`H ^= G;`	`H = H ^ G;`
`I <<= H;`	`I = I << H;`

Operator Example	Same as
`J >>= I;`	`J = J  >> I;`
`K <<<=J;`	`K = K  <<< J;`
`L >>>=K;`	`L = L  >>> K;`

Increment and Decrement Operators

In addition, SystemVerilog also has the increment/decrement operators *i++*, *i--*, *++i*, and *--i*.

Operator Example	Same as
`A++;`	A = A + 1;
`A--;`	A = A - 1;
`++A;`	Increment first and then use A
`--A;`	Decrement first and then use A

In the following code segment, out1 gets r1 and out2 gets the twice-decremented value of out1:

```
always @(*)
   begin
      out1 = r1--;
      out2 = --r1;
   end
```

Aggregate Expressions

Aggregate expressions (aggregate pattern assignments) are primarily used to initialize and assign default values to unpacked arrays and structures.

Syntax

SystemVerilog aggregate expressions are constructed from braces; an apostrophe prefixes the opening (left) brace.

'{ *listofValues* **}**

In the syntax, *listofValues* is a comma-separated list. SystemVerilog also provides a mechanism to initialize all of the elements of an unpacked array by specifying a default value within the braces using the following syntax:

> **'{ default:** *value* **}**
> **'{ data type:***value* **}**
> **'{ index:***value* **}**

The aggregate (pattern) assignment can be used to initialize any of the following.

- a 2-dimensional unpacked array under a reset condition (see Initializing Unpacked Array Under Reset Condition example).
- all the elements of a 2-dimensional unpacked array to a default value using the default keyword under a reset condition (see Initializing Unpacked Array to Default Value example).
- a specific data type using the keyword for *type* instead of default (see Initializing Specific Data Type example).
- unpacked elements of ports that can be passed to a submodule during instantiations (see Aggregate on Port example). For example:

```
sub u1(.temp('{'0,'1,'1,0})
```

Example – Aggregate on Ports

Currently, you must enable the Beta Features for Verilog on the Verilog tab of the Implementation Options panel to use this feature. Otherwise, the compiler generates an error message.

Example: Aggregate on Ports (Submodule)

Example: Aggregate on Ports (Top-Level Module)

Aggregate (pattern) assignment can also be specified in a package (see Aggregate Assignment in Package example) and in a compilation unit (see Aggregate Assignment in Compilation Unit example).

Example – Initializing Unpacked Array Under Reset Condition

Example – Initializing Unpacked Array to Default Value

Example – Initializing Specific Data Type

Example – Aggregate Assignment in Package

Example – Aggregate Assignment in Compilation Unit

Streaming Operator

The streaming operator (>> or <<) packs the bit-stream type to a particular sequence of bits in a user-specified order. Bit-stream types can be any integral, packed or unpacked type or structure. The streaming operator can be used on either the left or right side of the expression.

The streaming operator determines the order of bits in the output data stream:

- The left-to-right operator (>>) arranges the output data bits in the same order as the input bit stream
- The right-to-left operator (<<) arranges the output data bits in reverse order from the input bit stream

Syntax

streamingExpression **::= {** *streamOperator* [*sliceSize*] *streamConcatenation* **}**

streamOperator **::= >> | <<**

sliceSize **::=** *dataType* **|** *constantExpression*

streamConcatenation **::= {***streamExpression* {**,** *streamExpression*} **}**

streamExpression **::=** *arrayRangeExpression*

When an optional *sliceSize* value is included, the stream is broken up into the slice-size segments prior to performing the specified streaming operation. By default, the *sliceSize* value is 1.

Usage

The streaming operator is used to:

- Reverse the entire data stream
- Bit-stream from one data type to other

When the slice size is larger than the data stream, the stream is left-justified and zero-filled on the right. If the data stream is larger than the left side variable, an error is reported.

Example – Packed type inputs/outputs with RHS operator

Example – Unpacked type inputs/outputs with RHS operator

Example – Packed type inputs/outputs with LHS operator

Example – Slice-size streaming with RHS operator

Example – Slice-size streaming with LHS slice operation

Set Membership Operator

The set membership operator, also referred to as the *inside* operator, returns the value TRUE when the expression value (i.e., the LHS of the operator) is present in the value list of the RHS operator. If the expression value is not present in the RHS operator, returns FALSE.

Syntax

(*expressionValue***) inside {***listofValues***}**

expressionValue **::=** *singularExpression*

listofValues **::=** *rangeofValues***,** *expressions***,** *arrayofAggregateTypes*

Example – Inside operator with dynamically changing input at LHS operator

Example – Inside operator with expression at LHS operator

Example – Inside operator with dynamically changing input at LHS and RHS operators

Example – Inside operator with array of parameter at LHS operator

Set Membership Case Inside Operator

With the case inside operator, a case expression is compared to each case item. Also, when using this operator, the case items can include an open range. The comparison returns TRUE when the case expression matches a case item, otherwise it returns FALSE.

Syntax

```
[unique|priority] case (caseExpression) inside
    (caseItem) : statement ;
    (caseItem) : statement ;
        .
        .
        .
    [default : statement ;]
endcase
```

In the above syntax, *caseItem* can be:

- a list of constants
- an open range
- a combination of a list of constants and an open range

The case inside operator supports the following optional modifiers:

- unique – each *caseItem* is unique and there are no overlapping *caseItems*. If there is an overlapping *caseItem*, a warning is issued.
- priority – the case statement is prioritized and all possible legal cases are covered by the case statement. If the *caseExpression* fails to match any of the *caseItems*, a warning is issued.

Example – Case Inside

```
module top# (
   parameter byte p1[2:1][4:1] = '{'{0,2,4,6},'{1,3,5,7}} )
//Input
(  input logic[4:1]sel,a,b,
//Output
   output logic[3:1] q );

always_comb begin
   case (sel) inside
      8,p1[1],10,12,14:q <= a;
      p1[2],9,11,13,15:q <= b;
   endcase
end
endmodule
```

Example – Unique Case Inside

```
module top# (
   parameter byte p1[2:1][4:1] = '{'{15,14,13,12},'{0,1,2,3}} )
//Input
(  input logic[4:1]sel1,sel2,
   input byte a,b,
//Output
   output byte q );

generate begin
   always@(*) begin
      unique case (sel1^sel2) inside
         p1 : q = a+b;
         [4:7],13,14,15 : q = a ^ b;
         [9:12],8 : q = a*b;
      endcase
   end
end
endgenerate
endmodule
```

Example – Priority Case Inside

```
typedef enum logic[4:1] {s[0:15]} EnumDt;

module top (
   input logic reset,
   input logic clock,
   input logic x,
   input logic[2:1] y,
   output logic[3:1] op );
EnumDt state;

always@(posedge reset or posedge clock)
begin
   if (reset == 1'b1)
   begin
      op <= 3'b000;
      state <= s0;
   end
   else
   begin
      priority case (state) inside
         [s0:s2],s12 : begin
            if (x == 1'b0  && y == 1'b0)
            begin
               state <= s3;
               op <= 3'b001;
            end
            else
            begin
               state <= s2;
               op <= 3'b000;
            end
         end
         [s3:s5] : begin
            if(x == 1'b1  && y== 1'b0)
            begin
               state <= s7;
               op <= 3'b010;
            end
            else
            begin
               state <= s9;
               op <= 3'b110;
            end
         end
         [s6:s8],s13 : begin
```

```
                if(x == 1'b0  &&  y== 1'b1)
                begin
                   state <= s11;
                   op <= 3'b011;
                end
                else if (x == 1'b0 && y == 1'b1)
                begin
                   state <= s4;
                   op <= 3'b010;
                end
             end
             [s9:s11] : begin
                if(x == 1'b1  && y== 1'b1 )
                begin
                   state <= s5;
                   op <= 3'b100;
                end
                else if (x == 1'b0 && y == 1'b1)
                begin
                   state <= s0;
                   op <= 3'b111;
                end
             end
             default : begin
                state <= s1;
                op <= 3'b111;
             end
          endcase
       end
    end
    endmodule
```

Type Operator

SystemVerilog provides a type operator as a way of referencing the data type of a variable or an expression.

Syntax

type(*dataType* **|** *expression***)**

dataType – a user-defined data type or language-defined data type

expression – any expression, variable, or port

An e*xpression* inside the type operator results in a self-determined type of expression; the expression is not evaluated. Also the *expression* cannot contain any hierarchical references.

Data Declaration

The type operator can be used while declaring signals, variables, or ports of a module/interface or a member of that interface.

Example – Using Type Operator to Declare Input/Output Ports

```
typedef logic signed[4:1]logicdt;
// Module top
module top(
   input type(logicdt) d1,
   output type(logicdt) dout1 );
type(logicdt) sig;
var type(logicdt) sig1;
assign sig  = d1;
assign sig1= d1+1'b1;
assign dout1= sig + sig1;
endmodule
```

Data Type Declaration

Defining of the user-defined data type can have the type operator, wherein a variable or another user-defined data type can be directly referenced while defining a data type using the type operator. The data type can be defined in the compilation unit, package, or inside the module or interface.

Example – Using Type Operator to Declare Unpacked Data Type

```
typedef logic[4:1] logicdt;
typedef type(logicdt)Unpackdt[2:1];

module top(
   input Unpackdt d1,
   output Unpackdt dout1 );
assign dout1[2] = d1[2];
assign dout1[1] = d1[1];
endmodule
```

Type Casting

The type operator can be used to directly reference the data type of a variable or port, or can be user-defined and used in type casting to convert either signed to unsigned or unsigned to signed.

Example – Using Type Operator to Reference Data Type

```
typedef logic [20:0]dt;
//Module top
module top (
   input byte d1,d2,
   output int unsigned dout1 );
assign dout1 = type(dt)'(d1 * d2);
endmodule
```

Defining Type Parameter/Local Parameter

The type operator can be used when defining a Type parameter to define the data type. The definition can be overridden based on user requirements.

Example – Using Type Operator to Declare Parameter Type Value

```
// Module top
module top(
   input byte a1,
   input byte a2,
   output shortint dout1 );
parameter type dtype = type(a1);
dtype sig1;
assign sig1 = a1;
assign dout1 = ~sig1;
endmodule
```

Comparison and Case Comparison

The type operator can be used to compare two types when evaluating a condition or a case statement.

Example – Using Type Operator in a Comparison

```
// Module top
module top(
   input byte d1,
   input shortint d2,
   output shortint dout1);

always_comb begin
   if(type(d1) == type(d2))
      dout1 = d1;
   else
      dout1 = d2;
end
endmodule
```

Limitations

The type operator is not supported on complex expressions (for example type(d1*d2)).

$typeof Operator

Verilog (IEEE Std 1800-2012) LRM no longer supports the $typeof operator. However, the tool can support the $typeof operator in accordance with SystemVerilog (IEEE Std 1800-2012) LRM section: 6.23. SystemVerilog provides the $typeof system function used to assign or override a type parameter or as a comparison with another $typeof operator, which is evaluated during elaboration.

Syntax

typeofFunction ::=

$typeof (*dataType*) – A user-defined data type or language-defined data type

$typeof (*expression*) – Any expression, variable, or port

For example:

```
bit [12:0] A_bus;
parameter type bus_t = $typeof(A_bus);
```

Example: $type Operator

For this test case:

- Parameter mtype is defined as logic signed [7:0].
- Input and output ports (din and dout) are defined as type mtype.
- Parameter mtype1 is created after the $typeof operator is applied to input port din.
- As a result, sig1 is also defined with parameter mtype1.

$typeof Operator Limitations

The compiler does not support the following $typeof conditions:

- When the $typeof operator uses an *expression* as its argument, the *expression* cannot contain any hierarchical references or reference elements of dynamic objects.
- The $typeof operator is not supported on complex expressions. For example:

  $typeof (d1 + 4'h4 - 2'b01)

Procedural Statements and Control Flow

Topics in this section include

Do-While Loops

The while statement executes a loop for as long as the loop-control test is true. The control value is tested at the *beginning* of each pass through the loop. However, a while loop does not execute at all if the test on the control value is false the first time the loop is encountered. This top-testing behavior can require extra coding prior to beginning the while loop, to ensure that any output variables of the loop are consistent.

SystemVerilog enhances the for loop and adds a do-while loop, the same as in C. The control on the do-while loop is tested at the *end* of each pass through the loop (instead of at the beginning). This implies that each time the loop is encountered in the execution flow, the loop statements are executed at least once.

Because the statements within a do-while loop are going to execute at least once, all the logic for setting the outputs of the loop can be placed inside the loop. This bottom-testing behavior can simplify the coding of while loops, making the code more concise and more intuitive.

Example – Simple Do-while Loop

Example – Do-while with If Else Statement

Example – Do-while with Case Statement

For Loops

SystemVerilog simplifies declaring local variables for use in for loops. The declaration of the for loop variable can be made within the for loop. This eliminates the need to define several variables at the module level, or to define local variables within named begin...end blocks as shown in the following example.

Example – Simple for Loop

A variable defined as in the example above, is local to the loop. References to the variable name within the loop see the local variable, however, reference to the same variable outside the loop encounters an error. This type of variable is created and initialized when the for loop is invoked, and destroyed when the loop exits.

SystemVerilog also enhances for loops by allowing more than one initial assignment statement. Multiple initial or step assignments are separated by commas as shown in the following example.

Example – For Loop with Two Variables

Foreach Loops

SystemVerilog allows foreach loops to iterate through array elements. The foreach loop automatically declares its loop control variables, determines the starting and ending indices of the array, and determines the direction of indexing (count up). The arguments specified with the foreach loop identifies the designation for the type of array (fixed-size or associative). It is followed by a list of loop variables enclosed in square brackets; where each loop variable in the square bracket corresponds to one dimension of the array.

Example – Foreach Loop Example

Unnamed Blocks

SystemVerilog allows local variables to be declared in unnamed blocks.

Example – Local Variable in Unnamed Block

Block Name on end Keyword

SystemVerilog allows a block name to be defined after the end keyword when the name matches the one defined on the corresponding begin keyword. This means, you can name the start and end of a begin statement for a block. The additional name does not affect the block semantics, but does serve to enhance code readability by documenting the statement group that is being completed.

Example – Including Block Name with end Keyword

Unique and Priority Modifiers

SystemVerilog adds unique and priority modifiers to use in case statements. The Verilog full_case and parallel_case statements are located inside of comments and are ignored by the Verilog simulator. For synthesis, full_case and parallel_-case directives instruct the tool to take certain actions or perform certain optimizations that are unknown to the simulator.

To prevent discrepancies when using full_case and parallel_case directives and to ensure that the simulator has the same understanding of them as the synthesis tool, use the priority or unique modifier in the case statement. The priority and unique keywords are recognized by all tools, including the Verilog simulators, allowing all tools to have the same information about the design.

The following table shows how to substitute the SystemVerilog unique and priority modifiers for Verilog full_case and parallel_case directives for synthesis.

Verilog using full_case, parallel_case	**SystemVerilog using unique/priority case modifiers**

`case (...)`
`...`
`endcase`
	`case (...)`
	`...`
	`endcase`

`case (...) //full_case`
`...`
`endcase`
	`priority case (...)`
	`...`
	`endcase`

`case (...) //parallel_case`
`...`
`endcase`
	`unique case (...)`
	`...`
	`default : ...`
	`endcase`

`case (...) //full_case parallel_case`
`...`
`endcase`
	`unique case (...)`
	`...`
	`endcase`

Example – Unique Case

Example – Priority Case

Processes

In Verilog, an "if" statement with a missing "else" condition infers an unintentional latch element, for which the Synopsys compiler currently generates a warning. Many commercially available compilers do not generate any warning, causing a serious mismatch between intention and inference. SystemVerilog adds three specialized procedural blocks that reduce ambiguity and clearly indicate the intent:

- always_comb, on page 150
- always_latch, on page 152
- always_ff, on page 153

Use them instead of the Verilog general purpose always procedural block to indicate design intent and aid in the inference of identical logic across synthesis, simulation, and verification tools.

always_comb

The SystemVerilog always_comb process block models combinational logic, and the logic inferred from the always_comb process must be combinational logic. The Synopsys compiler warns you if the behavior does not represent combinational logic.

The semantics of an always_comb block are different from a normal always block in these ways:

- It is illegal to declare a sensitivity list in tandem with an always_comb block.
- An always_comb statement cannot contain any block, timing, or event controls and fork, join, or wait statements.

Note the following about the always_comb block:

- There is an inferred sensitivity list that includes all the variables from the RHS of all assignments within the always_comb block and variables used to control or select assignments See Examples of Sensitivity to LHS and RHS of Assignments, on page 152.
- The variables on the LHS of the expression should not be written by any other processes.

- The always_comb block is guaranteed to be triggered once at time zero after the initial block is executed.
- always_comb is sensitive to changes within the contents of a function and not just the function arguments, unlike the always@(*) construct of Verilog 2001.

Example – always_comb Block

Invalid Use of always_comb Block

The following code segments show use of the construct that are *NOT VALID*.

```
always_comb @(a or b) //Wrong. Sensitivity list is inferred not
   //declared
begin
   foo;
end

always_comb
begin
   @clk out <=in; //Wrong to use trigger within this always block
end

always_comb
begin
   fork //Wrong to use fork-join within this always block
   out <=in;
   join
end

always_comb
begin
   if(en)mem[waddr]<=data; //Wrong to use trigger conditions
      //within this block
end
```

Examples of Sensitivity to LHS and RHS of Assignments

In the following code segment, sensitivity only to the LHS of assignments causes problems.

```
always @(y)
   if (sel)
      y= a1;
   else
      y= a0;
```

In the following code segment, sensitivity only to the RHS of assignments causes problems.

```
always @(a0, a1)
   if (sel)
      y= a1;
   else
      y= a0;
```

In the following code segment, sensitivity to the RHS of assignments and variables used in control logic for assignments produces correct results.

```
always @(a0, a1, sel)
   if (sel)
      y= a1;
   else
      y= a0;
```

always_latch

The SystemVerilog always_latch process models latched logic, and the logic inferred from the always_latch process must only be latches (of any kind). The Synopsys compiler warns you if the behavior does not follow the intent.

Note the following:

- It is illegal for always_latch statements to contain a sensitivity list, any block, timing, or event controls, and fork, join, or wait statements.
- The sensitivity list of an always_latch process is automatically inferred by the compiler and the inferring rules are similar to the always_comb process (see always_comb, on page 150).

Example – always_latch Block

Invalid Use of always_latch Block

The following code segments show use of the construct that are *NOT VALID*.

```
always_latch
begin
   if(en)
      treg<=1;
   else
      treg<=0; //Wrong to use fully specified if statement
end

always_latch
begin
   @(clk)out <=in; //Wrong to use trigger events within this
      //always block
end
```

always_ff

The SystemVerilog always_ff process block models sequential logic that is triggered by clocks. The compiler warns you if the behavior does not represent the intent. The always_ff process has the following restrictions:

- An always_ff block must contain only one event control and no blocking timing controls.
- Variables on the left side of assignments within an always_ff block must not be written to by any other process.

Example – always_ff Block

Invalid Use of always_ff Block

The following code segments show use of the construct that are *NOT VALID*.

```
always_ff @(posedge clk or negedge rst)
begin
   if(rst)
      treg<=in; //Illegal; wrong polarity for rst in the
         //sensitivity list and the if statement
end
```

```
always_ff
begin
   @(posedgerst)treg<=0;
   @(posedgeclk)treg<=in; //Illegal; two event controls
end

always_ff @(posedge clk or posedge rst)
begin
   treg<=0; //Illegal; not clear which trigger is to be
               // considered clk or rst
end
```

Tasks and Functions

Support for task and function calls includes the following:

- Implicit Statement Group
- Formal Arguments, on page 155
- endtask/endfunction Names, on page 158

Implicit Statement Group

Multiple statements in the task or function definition do not need to be placed within a begin…end block. Multiple statements are implicitly grouped, executed sequentially as if they are enclosed in a begin…end block.

```
/* Statement grouping */
function int incr2(int a);
   incr2 = a + 1;
   incr2 = incr2 + 1;
endfunction
```

Formal Arguments

This section includes information on passing formal arguments when calling functions or tasks. Topics include:

- Passing Arguments by Name
- Default Direction and Type
- Default Values

Passing Arguments by Name

When a task or function is called, SystemVerilog allows for argument values to be passed to the task/function using formal argument names; order of the formal arguments is not important. As in instantiations in Verilog, named argument values can be passed in any order, and are explicitly passed through to the specified formal argument. The syntax for the named argument passing is the same as Verilog's syntax for named port connections to a module instance. For example:

```
/* General functions */
function [1:0] inc(input [1:0] a);
   inc = a + 1;
endfunction
function [1:0] sel(input [1:0] a, b, input s);
   sel = s ? a : b;
endfunction

/* Tests named connections on function calls */
assign z0 = inc(.a(a));
assign z2 = sel(.b(b), .s(s), .a(a));
```

Default Direction and Type

In SystemVerilog, input is the default direction for the task/function declaration. Until a formal argument direction is declared, all arguments are assumed to be inputs. Once a direction is declared, subsequent arguments will be the declared direction, the same as in Verilog.

The default data type for task/function arguments is logic, unless explicitly declared as another variable type. (In Verilog, each formal argument of a task/function is assumed to be reg). For example:

```
/* Tests default direction of argument */
function int incr1(int a);
   incr1 = a + 1;
endfunction
```

In this case, the direction for a is input even though this is not explicitly defined.

Default Values

SystemVerilog allows an optional default value to be defined for each formal argument of a task or function. The default value is specified using a syntax similar to setting the initial value of a variable. For example:

```
function int testa (int a = 0, int b, int c = 1);
   testa = a + b + c;
endfunction

task testb (int a = 0, int b, int c = 1, output int d);
   d = a + b + c;
endtask
```

When a task/function is called, it is not necessary to pass a value to the arguments that have default argument values. If nothing is passed to the task/function for that argument position, the default value is used. Specifying default argument values allows a task/function definition to be used in multiple ways. Verilog requires that a task/function call have the exact same number of argument expressions as the number of formal arguments. SystemVerilog allows the task/function call to have fewer argument expressions than the number of formal arguments. A task/function call must pass a value to an argument, if the formal definition of the argument does not have a default value. Consider the following examples:

```
/* functions With positional associations and missing arguments */
assign a = testa(,5); /* Same as testa(0,5,1) */
assign b = testa(2,5); /* Same as testa(2,5,1) */
assign c = testa(,5,); /* Same as testa(0,5,1) */
assign d = testa(,5,7); /* Same as testa(0,5,7) */
assign e = testa(1,5,2); /* Same as testa(1,5,2) */

/* functions With named associations and missing arguments */
assign k = testa(.b(5)); /* Same as testa(0,5,1) */
assign l = testa(.a(2),.b(5)); /* Same as testa(2,5,1) */
assign m = testa(.b(5)); /* Same as testa(0,5,1) */
assign n = testa(.b(5),.c(7)); /* Same as testa(0,5,7) */
assign o = testa(.a(1),.b(5),.c(2)); /* Same as testa(1,5,2) */
```

In general, tasks are not supported outside the scope of a procedural block (even in previous versions). This is primarily due to the difference between tasks and function.

Here are some task examples using default values:

```
always @(*)
begin
/* tasks With named associations and missing arguments */
testb(.b(5),.d(f)); /* Same as testb(0,5,1) */
testb(.a(2),.b(5),.d(g)); /* Same as testb(2,5,1) */
testb(.b(5),.d(h)); /* Same as testb(0,5,1) */
testb(.b(5),.c(7),.d(i)); /* Same as testb(0,5,7) */
testb(.a(1),.b(5),.c(2),.d(j)); /* Same as testb(1,5,2) */

/* tasks With positional associations and missing arguments */
testb(,5,,p); /* Same as testb(0,5,1) */
testb(2,5,,q); /* Same as testb(2,5,1) */
testb(,5,,r); /* Same as testb(0,5,1) */
testb(,5,7,s); /* Same as testb(0,5,7) */
testb(1,5,2,t); /* Same as testb(1,5,2) */
```

endtask/endfunction Names

SystemVerilog allows a name to be specified with the endtask or endfunction keyword. The syntax is:

endtask : *taskName*

endfunction : *functionName*

The space before and after the colon is optional. The name specified must be the same as the name of the corresponding task or function as shown in the following example.

```
/* Function w/ statement grouping, also has an endfunction label */

function int incr3(int a);
   incr3 = a + 1;
   incr3 = incr3 + 1;
   incr3 = incr3 + 1;
endfunction : incr3

/* Test with a task - also has an endtask label */
task task1;
input [1:0] in1,in2,in3,in4;
output [1:0] out1,out2;
   out1 = in1 | in2;
   out2 = in3 & in4;
endtask : task1
```

```
/* Test with a task - some default values */
task task2(
input [1:0] in1=2'b01,in2= 2'b10,in3 = 2'b11,in4 = 2'b11,
output [1:0] out1 = 2'b10,out2);

   out2 = in3 & in4;
endtask : task2

/* Tests default values for arguments */
function int dflt0(input int a = 0, b = 1);
   dflt0 = a + b;
endfunction

/* Call to function with default direction */
assign z1 = incr1(3);
assign z3 = incr2(3);
assign z4 = incr3(3);
assign z9 = dflt0();
assign z10 = dflt0(.a(7), .b());
always @(*)
begin
   task1(.in1(in1), .out2(z6), .in2(in2), .out1(z5),
      .in3(in3), .in4(in4));
   task1(in5, in6, in7, in8, z7, z8);
   task2(in5, in6, in7, in8, z11, z12);
   task2(in5, in6, , , z13, z14);
   task2(.out1(z15), .in1(in5), .in2(in6), .out2(z16),
      .in3(in7), .in4(in8));
   task2(.out2(z18), .in2(in6), .in1(in5), .in3(),
      .out1(z17), .in4());
end
```

Hierarchy

Topics in this section include:

- Compilation Units, below
- Packages, on page 162
- Port Connection Constructs, on page 163
- Extern Module, on page 166

Compilation Units

Compilation units allow declarations to be made outside of a package, module, or interface boundary. These units are visible to all modules that are compiled at the same time.

A compilation unit's scope exists only for the source files that are compiled at the same time; each time a source file is compiled, a compilation unit scope is created that is unique to only that compilation.

Syntax

//$unit definitions

declarations;

//End of $unit

module ();
. . .
. . .
. . .
endmodule

In the above syntax, declarations can be variables, nets, constants, user-defined data types, tasks, or functions

Usage

Compilation units can be used to declare variables and nets, constants, user-defined data types, tasks, and functions as noted in the following examples.

A variable can be defined within a module as well as within a compilation unit. To reference the variable from the compilation unit, use the **$unit::***variableName* syntax. To resolve the scope of a declaration, local declarations must be searched first followed by the declarations in the compilation unit scope.

Example – Compilation Unit Variable Declaration

Example – Compilation Unit Net Declaration

Example – Compilation Unit Constant Declaration

Example – Compilation Unit User-defined Datatype Declaration

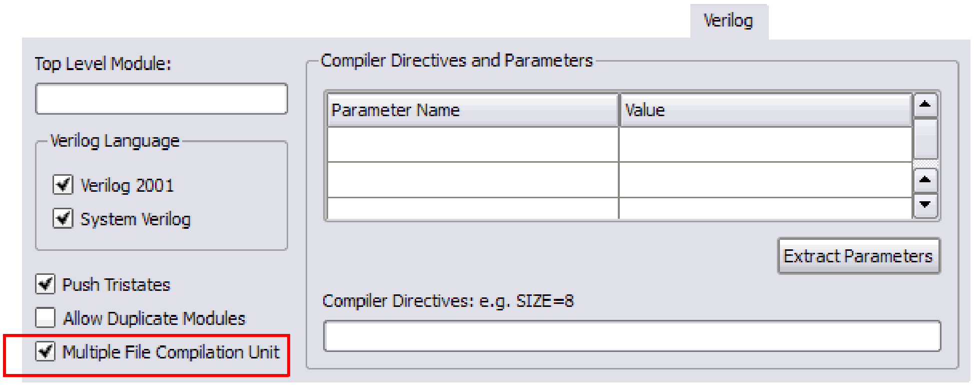
Example – Compilation Unit Task Declaration

Example – Compilation Unit Function Declaration

Example – Compilation Unit Access

Example – Compilation Unit Scope Resolution

To use the compilation unit for modules defined in multiple files, enable the Multiple File Compilation Unit check box on the Verilog tab of the Implementation Options dialog box as shown below.



[✓] Verilog 2001
[✓] System Verilog
[✓] Push Tristates
[ ] Allow Duplicate Modules
[✓] Multiple File Compilation Unit

You can also enable this compiler directive by including the following Tcl command in your design file:

```
set_option -multi_file_compilation_unit 1
```

Compilation Unit Limitations

Compilation unit elements can only be accessed or read, and cannot appear between module and endmodule statements.

Packages

Packages permit the sharing of language-defined data types, typedef user-defined types, parameters, constants, function definitions, and task definitions among one or more compilation units, modules, or interfaces. The concept of packages is leveraged from the VHDL language.

Syntax

SystemVerilog packages are defined between the keywords package and endpackage.

package packageIdentifier;

packageItems

endpackage : packageIdentifier

PackageItems include user-defined data types, parameter declarations, constant declarations, task declarations, function declarations, and import statements from other packages. To resolve the scope of any declaration, the local declarations are always searched before declarations in packages.

Referencing Package Items

As noted in the following examples, package items can be referenced by:

- Direct reference using a scope resolution operator (::). The scope resolution operator allows referencing a package by the package name and then selecting a specific package item.
- Importing specific package items using an import statement to import specific package items into a module.
- Importing package items using a wildcard (*) instead of naming a specific package item.

Example – Direct Reference Using Scope Resolution Operator (::)

Example – Importing Specific Package Items

Example – Wildcard (*) Import Package Items

Example – User-defined Data Types (typedef)

Example – Parameter Declarations

Example – Constant Declarations

Example – Task Declarations

Example – Function Declarations

Example – import Statements from Other Packages

Example – Scope Resolution

Package Limitations

The variables declared in packages can only be accessed or read; package variables cannot be written between a module statement and its end module statement.

Port Connection Constructs

Instantiating modules with a large number of ports is unnecessarily verbose and error-prone in Verilog. The SystemVerilog *.name* and ".*" constructs extend the 1364 Verilog feature of allowing named port connections on instantiations, to implicitly instantiate ports.

.name Connection

The SystemVerilog *.name* connection is semantically equivalent to a Verilog named port connection of type *.port_identifier(name)*. Use the *.name* construct when the name and size of an instance port are the same as those on the module. This construct eliminates the requirement to list a port name twice when both the port name and signal name are the same and their sizes are the same as shown below:

```
module myand(input [2:0] in1, in2, output [2:0] out);
...
endmodule

module foo (….ports….)
wire [2:0] in1, out;
wire [7:0] tmp;
wire [7:0] in2 = tmp;
myand mand1(.in1, .out, .in2(tmp[2:0]));  // valid
```

Note: SystemVerilog *.name* connection is currently not supported for mixed-language designs.

Restrictions to the *.name* feature are the same as the restrictions for named associations in Verilog. In addition, the following restrictions apply:

- Named associations and positional associations cannot be mixed:

  ```
  myand mand2(.in1, out, tmp[2:0]);
  ```

- Sizes must match in mixed named and positional associations. The example below is not valid because of the size mismatch on in2.

  ```
  myand mand3(.in1, .out, .in2);
  ```

- The identifier referred by the *.name* must not create an implicit declaration, regardless of the compiler directive ‘*default_nettype*.
- You cannot use the *.name* connection to create an implicit cast.
- Currently, the *.name* port connection is not supported for mixed HDL source code.

.* Connection

The SystemVerilog “.*” connection is semantically identical to the default *.name* connection for every port in the instantiated module. Use this connection to implicitly instantiate ports when the instance port names and sizes match the connecting module’s variable port names and sizes. The implicit .* port connection syntax connects all other ports on the instantiated module.Using the .* connection facilitates the easy instantiation of modules with a large number of ports and wrappers around IP blocks.

The ".*" connection can be freely mixed with *.name* and *.port_identifier*(*name*) type connections. However, it is illegal to have more than one ".*" expression per instantiation.

The use of ".*" facilitates easy instantiation of modules with a large number of ports and wrappers around IP blocks as shown in the code segment below:

```
module myand(input [2:0] in1, in2, output [2:0] out);
...
endmodule

module foo (….ports….)
wire [2:0] in1, in2, out;
wire [7:0] tmp;

myand and1(.*); // Correct usage, connect in1, in2, out
myand and2(.in1, .*) // Correct usage, connect in2 and out
myand and3(.in1(tmp[2:0]), .*); // Correct Usage, connect
   // in2 and out
myand and5(.in1, .in2, .out, .*); //Correct Usage, ignore the .*
```

Note: SystemVerilog “.*” connection is currently not supported for mixed-language designs.

Restrictions to the .* feature are the same as the restrictions for the *.name* feature. See *.name Connection, on page 164*. In addition, the following restrictions apply:

- Named associations and positional associations cannot be mixed. For example

  ```
  myand and4(in1, .*);
  ```

  is illegal (named and positional connections cannot be mixed)

- Named associations where there is a mismatch of variable sizes or names generate an error.
- You can only use the .* once per instantiation, although you can mix the .* connection with *.name* and *.port_identifier(name)* type connections.
- If you use a .* construction but all remaining ports are explicitly connected, the compiler ignores the .* construct.
- Currently, the .* port connection is not supported for mixed HDL source code.

Extern Module

SystemVerilog simplifies the compilation process by allowing you to specify a prototype of the module being instantiated. The prototype is defined using the extern keyword, followed by the declaration of the module and its ports. Either the Verilog-1995 or the Verilog-2001 style of module declaration can be used for the prototype.

The extern module declaration can be made in any module, at any level of the design hierarchy. The declaration is only visible within the scope in which it is defined. Support is limited to declaring extern module outside the module.

Syntax

extern module *moduleName* **(***direction port1*, *direction portVector port2*, *direction port3***);**

Example 1 – Extern Module Instantiation

Example 2 – Extern Module Reference

Extern Module Limitations

An extern module declaration is not supported within a module.

Interface

Topics in this section include:

- Interface Construct
- Modports, on page 174
- Modport Limitations and Non-Supported Features, on page 174

Interface Construct

SystemVerilog includes enhancements to Verilog for representing port lists and port connection lists characterized by name repetition with a single name to reduce code size and simplify maintenance. The interface and modport structures in SystemVerilog perform this function. The interface construct includes all of the characteristics of a module with the exception of module instantiation; support for interface definitions is the same as the current support for module definitions. Interfaces can be instantiated and connected to client modules using generates.

Interface Definition: Internal Logic and Hierarchical Structure

Per the SystemVerilog standard, an interface definition can contain any logic that a module can contain with the exception that interfaces cannot contain module instantiations. An interface definition can contain instantiations of other interfaces. Like modules, interface port declaration lists can include interface-type ports. Synthesis support for interface logic is the same as the current support for modules.

Port Declarations and Port Connections for Interfaces

Per the SystemVerilog standard, interface port declaration and port connection syntax/semantics are identical to those of modules.

Interface Member Types

The following interface member types are visible to interface clients:

- 4-State var types: reg, logic, integer
- 2-State var types: bit, byte, shortint, int, longint

- Net types: wire, wire-OR, and wire-AND
- Scalars and 1-dimensional packed arrays of above types
- Multi-dimensional packed and unpacked arrays of above types
- SystemVerilog struct types

Interface Member Access

The members of an interface instance can be accessed using the syntax:

interfaceRef.*interfaceMemberName*

In the above syntax, *interfaceRef* is either:

- the name of an interface-type port of the module/interface containing the member access
- the name of an interface instance that is instantiated directly within the module/interface containing the member access.

Note that reference to interface members using full hierarchical naming is not supported and that only the limited form described above for instances at the current level of hierarchy is supported.

Access to an interface instance by clients at lower levels of the design hierarchy is achieved by connecting the interface instance to a compatible interface-type port of a client instance and connecting this port to other compatible interface-type ports down the hierarchy as required. This chaining of interface ports can be done to an arbitrary depth. Interface instances can be accessed only by clients residing at the same or lower levels of the design hierarchy.

Interface-Type Ports

Interface-type ports are supported as described in the SystemVerilog standard, and generic interface ports are supported. A modport qualifier can appear in either a port declaration or a port connection as described in the SystemVerilog standard. Interface-type ports:

- can appear in either module or interface port declarations
- can be used to access individual interface items using "." syntax:

  interfacePortname.*interfaceMemberName*

- can be connected directly to compatible interface ports of module/interface instances

Interface/Module Hierarchy

Interfaces can be instantiated within either module or interface definitions. See Interface Member Access, on page 168 for additional details on hierarchical interface port connections.

Interface Functions and Tasks

Import-only functions and tasks (using import keyword in modport) are supported.

Element Access Outside the Interface

Interface can have a collection of variables or nets, and this collection can be of a language-defined data type, user-defined data type, or array of language and user-defined data type. All of these variables can be accessed outside the interface.

The following example illustrates accessing a 2-dimensional structure type defined within the interface that is being accessed from another module.

Example – Accessing a 2-dimensional Structure

```
typedef struct
{
   byte st1;
}Struct1D_Dt[1:0][1:0];

//Interface Definition
interface intf(
   input bit clk,
   input bit rst
);

   Struct1D_Dt i1; //2D - Structure type
   modport MP( input i1,input clk,input rst); //Modport Definition
endinterface

//Sub1 Module definition
module sub1(
   intf INTF1, //Interface
   input int d1
);

   assign INTF1.i1[1][1].st1 = d1[7:0];
   assign INTF1.i1[1][0].st1 = d1[15:8];
   assign INTF1.i1[0][1].st1 = d1[23:16];
   assign INTF1.i1[0][0].st1 = d1[31:24];
endmodule

//Sub2 Module definition
module sub2(
   intf.MP IntfMp, //Modport Interface
   output byte dout[3:0]
);

always_ff@(posedge IntfMp.clk)
begin
   if(IntfMp.rst)
   begin
      dout <= '{default:'1};
   end
   else begin
      dout[3] <= IntfMp.i1[1][1].st1;
      dout[2] <= IntfMp.i1[1][0].st1;
      dout[1] <= IntfMp.i1[0][1].st1;
      dout[0] <= IntfMp.i1[0][0].st1;
   end
end
endmodule

//Top Module definition
module top(
   input bit clk,
   input bit rst,
   input int d1,
   output byte dout[3:0]
);
intf intu1(.clk(clk),.rst(rst));
sub1 sub1u1(.INTF1(intu1),.d1(d1));
sub2 sub2u1(.IntfMp(intu1.MP),.dout(dout));
endmodule
```

Nested Interface

With the nested interface feature, nesting of interface is possible by either instantiating one interface in another or by using one interface as a port in another interface. Generic interface is not supported for nested interface; array of interface when using interface as a port also is not supported.

The following example illustrates the use of a nested interface. In the example, one interface is instantiated within another interface and this top-level interface is used in the modules.

Example – Nested Interface

```
//intf1 Interface definition
interface intf1;
   byte i11;
   byte i12;
endinterface

//IntfTop Top Interface definition
interface IntfTop;
   intf1 intf1_u1(); //Interface instantiated
   shortint i21;
endinterface
```

```
//Sub1 Module definition
module sub1(
   input byte d1,
   input byte d2,
   IntfTop intfN1
);
assign intfN1.intf1_u1.i11 = d1; //Nested interface being accessed
assign intfN1.intf1_u1.i12 = d2; //Nested interface being accessed
endmodule

//Sub2 Module definition
module sub2(
   IntfTop intfN2
);
assign intfN2.i21 = intfN2.intf1_u1.i11 + intfN2.intf1_u1.i12;
//Nested
   //interface being accessed
endmodule

//Sub3 Module definition
module sub3(
   IntfTop intfN3,
   output shortint dout
);
assign dout = intfN3.i21;
endmodule

//Top Module definition
module top(
   input byte d1,
   input byte d2,
   output shortint dout
);
IntfTop IntfTopU1();
   sub1 sub1U1(.d1(d1),.d2(d2),.intfN1(IntfTopU1));
   sub2 sub2U1(.intfN2(IntfTopU1));
   sub3 sub3U1(.intfN3(IntfTopU1), .dout(dout));
endmodule
```

Arrays of Interface Instances

In Verilog, multiple instances of the same module can be created using the array of instances concept. This same concept is extended for the interface construct in SystemVerilog to allow multiple instances of the same interface to be created during component instantiation or during port declaration. These arrays of interface instances and slices of interface instance arrays can be passed as connections to arrays of module instances across modules.

The following example illustrates the use of array of interface instance both during component instantiation and during port declaration.

Example – Array of Interface During Port Declaration

```
//intf Interface Definition
interface intf;
   byte i1;
endinterface

//Sub1 Module definition
module sub1(
   intf IntfArr1 [3:0], //Array of interface during port
declaration
   input byte d1[3:0]
);
assign IntfArr1[0].i1 = d1[0];
assign IntfArr1[1].i1 = d1[1];
assign IntfArr1[2].i1 = d1[2];
assign IntfArr1[3].i1 = d1[3];
endmodule

//Sub2 Module definition
module sub2(
   intf IntfArr2[3:0], //Array of interface during port
declaration
   output byte dout[3:0]
);
assign dout[0] = IntfArr2[0].i1;
assign dout[1] = IntfArr2[1].i1;
assign dout[2] = IntfArr2[2].i1;
assign dout[3] = IntfArr2[3].i1;
endmodule
```

```
//Top module definition
module top(
   input byte d1[3:0],
   output byte dout[3:0]
);
intf intfu1[3:0](); //Array of interface instances
   sub1 sub1u1(intfu1,d1);
   sub2 sub2u1(intfu1,dout);
endmodule
```

Modports

Modport expressions are supported, and modport selection can be done in either the port declaration of a client module or in the port connection of a client module instance.

If a modport is associated with an interface port or instance through a client module, the module can only access the interface members enumerated in the modport. However, per the SystemVerilog standard, a client module is not constrained to use a modport, in which case it can access any interface members.

Modport Keywords

The input, output, inout, and import access modes are parsed without errors. The signal direction for input, output, and inout is ignored during synthesis, and the correct signal polarity is inferred from how the interface signal is used within the client module. The signal polarity keywords are ignored because the precise semantics are currently not well-defined in the SystemVerilog standard, and simulator support has yet to be standardized.

Example – Instantiating an interface Construct

Modport Limitations and Non-Supported Features

The following restrictions apply when using interface/modport structures:

- Declaring interface within another interface is not supported
- Do not code RAM RTL within a SystemVerilog interface definition, since this can prevent extraction of the RAM primitive. However, the RAM can be defined within a module.

- Direction information in modports has no effect on synthesis.
- Exported (export keyword) interface functions and tasks are not supported.
- Virtual interfaces are not supported.
- Full hierarchical naming of interface members is not supported.
- Modports defined within generate statements are not supported.

System Tasks and System Functions

Topics in this section include:

$bits System Function

SystemVerilog supports a $bits system function which returns the number of bits required to hold an expression as a bit stream. The syntax is:

$bits(*datatype***)**

$bits(*expression***)**

In the above syntax, *datatype* can be any language-defined data type (reg, wire, integer, logic, bit, int, longint, or shortint) or user-defined datatype (typedef, struct, or enum) and *expression* can be any value including packed and unpacked arrays.

The $bits system function is synthesizable and can be used with any of the following applications:

- Port Declaration
- Variable Declaration
- Constant Definition
- Function Definition

System tasks and system functions are described in Section 22 of IEEE Std 1800-2005 (IEEE Standard for SystemVerilog); $bits is described in Section 22.3.

Example – $bits System Function

Example – $bits System Function within a Function

$bit System FunctionLimitations

The $bits system function is not supported under the following conditions:

- Passing an interface member as an argument to the $bits function is not supported. In the example

  ```
  parameter logic[2:0] din = $bits(ff_if_0.din);
  ```

  interface instance ff_if_0.din is one of the ports of the modport. To avoid the limitation, use the actual value as the argument in place of the interface member.

- $bits cannot be used within a module instantiation:

  ```
  module Top
     (output foo);
     Intf intf();
     Foo #(.PARAM($bits(intf.i))) Foo (.foo);
  endmodule : Top
  ```

- $bits is not supported with params/localparams:

  ```
  localparam int WIDTH = $bits(ramif.port0_out);
  ```

$countbits System Function

SystemVerilog supports a $countbits system function which counts the number of bits that have a specified value in a bus. The syntax is:

$countbits (expression, control_bit {, control_bit})

In the above syntax, *expression* can be a constant or a variable and *control_bit* can be 0, 1, X, or Z. It returns a value equal to the number of bits in *expression* whose values match one of the *control_bit* values.

Syntax Examples

$countbits (*expression*, '1)	Returns the number of bits in *expression* having value 1.
$countbits (*expression*, '1, '0)	Returns the number of bits in *expression* having values 1 or 0.
$countbits (*expression*, 'Z)	When *control_bit* is set to X or Z, only constants may be used (for example, parameters or localparams). If they are used with a variable, the return value will always be 0.

Example

```
module test (
   input [3:0] in1,
   output [31:0] out1,out2
   );

assign out1 = $countbits(in1,'1); // 1 counter
assign out2 = $countbits(in1,'0); // 0 counter
endmodule
```

$countones System Function

SystemVerilog function support includes the $countones system function. This function checks for specific characteristics on a particular signal and return a single-bit value.

$countones returns true when the number of ones in a given expression matches a predefined value.

Syntax Example

$countones (*expression***)**

Example - Ones Count

In the following example, a 4-bit count is checked for two and only two bits set to 1 which, when present, returns true.

```
module top(
   input clk,
   input rst,
   input byte d1,
   output byte dout
);
logic[3:0] count;

always_ff@(posedge clk)begin
   if(rst)
      count <= '0;
   else
      count <= count + 1'b1;
end

assign dout = $countones(count) == 3'd2 ? d1 : ~d1;
endmodule
```

$onehot and $onehot0 System Functions

SystemVerilog function support includes the $onehot and $onehot0 system functions. These functions check for specific characteristics on a particular signal and return a single-bit value.

$onehot returns true when only one bit of the expression is true.

$onehot0 returns true when no more than one bit of the expression is high (either one bit high or no bits are high).

Syntax Example

$onehot (*expression***)**

$onehot0 (*expression***)**

Example 1 - System Function within if Statement

The following example shows a $onehot/$onehot0 function used inside an if statement and ternary operator.

```
module top
   (
   //Input
   input byte d1,
   input byte d2,
   input shortint d3,

   //Output
   output byte dout1,
   output byte dout2
   );
byte sig1;
assign sig1 = d1 + d2;

//Use of $onehot
always_comb begin
   if($onehot(sig1))
      dout1 = d3[7:0];
   else
      dout1 = d3[15:8];
end

byte sig2;
assign sig2 = d1 ^ d2;
//Use of $onehot0
assign dout2 = $onehot0(sig2)? d3[7:0] : d3[15:8];
endmodule
```

Example 2 - System Function with Expression

The following example includes an expression, which is evaluated to a single-bit value, as an argument to a system function.

```
module top
   (
   //Input
   input byte d1,
   input byte d2,
   input shortint d3,
   //Output
   output byte dout1,
   output byte dout2
   );

//Use of $onehot with Expression inside onehot function
always@*
begin
   if($onehot((d1 == d2) ? d1[3:0] : d1[7:4]))
      dout1 = d3[7:0];
   else
      dout1 = d3[15:8];
end

//Use of $onehot0 with AND operation inside onehot function
assign dout2 = $onehot0(d1 & d2)? d3[7:0] : d3[15:8];
endmodule
```

Array Querying Functions

SystemVerilog provides system functions that return information about a particular dimension of an array.

Syntax

arrayQuery **(***arrayIdentifier*[,*dimensionExpression*]**);**
arrayQuery **(***dataTypeName*[,*dimensionExpression*]**);**
$dimensions | $unpacked_dimensions (*arrayIdentifier* **|** *dataTypeName***)**

In the above syntax, *arrayQuery* is one of the following array querying functions:

- **$left** – returns the left bound (MSB) of the dimension.
- **$right** – returns the right bound (LSB) of the dimension.
- **$low** – returns the lowest value of the left and right bound dimension.
- **$high** – returns the highest value of the left and right bound dimension.
- **$size** – returns the number of elements in a given dimension.
- **$increment** – returns a value "1" when the left bound is greater than or equal to the right bound, else it returns a value "-1".

In the third syntax example, $dimensions returns the total number of packed and unpacked dimensions in a given array, and $unpacked_dimensions returns the total number of unpacked dimensions in a given array. The variable *dimensionExpression*, by default, is "1". The order of dimension expression increases from left to right for both unpacked and packed dimensions, starting with the unpacked dimension for a given array.

Example 1 – Array Querying Function $left and $right Used on Packed 2D-data Type

Example 2 – Array Querying Function $low and $high Used on Unpacked 3D-data Type

Example 3 – Array Querying Function $size and $increment Used on a Mixed Array

Example 4 – Array Querying Function $dimensions and $unpacked_dimensions Used on a Mixed Array

Example 5 – Array Querying Function with Data Type as Input

Generate Statement

The tool supports the Verilog 2005 generate statement, which conforms to the Verilog 2005 LRM. The defparam, parameter, and function and task declarations within generate statements are supported. The naming scheme for registers and instances is also enhanced to include closer correlation to specified generate symbolic hierarchies. Generated data types have unique identifier names and can be referenced hierarchically. Generate statements are created using one of the following three methods: generate-loop, generate-conditional, or generate-case.

Note: The generate statement is a Verilog 2005 feature; to use this statement with the tool, you must enable SystemVerilog for your design.

Example 1 – Shift Register Using generate-for

Example 2 – Accessing Variables Declared in a generate-if

Example 3 – Accessing Variables Declared in a generate-case

Limitations

The following generate statement functions are not currently supported:

- Defparam support for generate instances
- Hierarchical access for interface
- Hierarchical access of function/task defined within a generate block

Note: Whenever the generate statement contains symbolic hierarchies separated by a hierarchy separator (.), the instance name includes the (\) character before this hierarchy separator (.).

Conditional Generate Constructs

The if-generate and case-generate conditional generate constructs allow the selection of, at most, one generate block from a set of alternative generate blocks based on constant expressions evaluated during elaboration. The generate and endgenerate keywords are optional.

Generate blocks in conditional generate constructs can be either named or unnamed and can consist of only a single item. It is not necessary to enclose the blocks with begin and end keywords; the block is still a generate block and, like all generate blocks, comprises a separate scope and a new level of hierarchy when it is instantiated. The if-generate and case-generate constructs can be combined to form a complex generate scheme.

Note: Conditional generate constructs are a Verilog 2005 feature; to use these constructs with the tool, you must enable SystemVerilog for your design.

Example 1 – Conditional Generate: if-generate

```
// test.v
module test
#  (parameter width = 8,
    sel = 2 )
```

```
   (input clk,
    input [width-1:0] din,
    output [width-1:0] dout1,
    output [width-1:0] dout2);

if(sel == 1)
   begin:sh
   reg [width-1:0] sh_r;

   always_ff @ (posedge clk)
      sh_r <= din;
   end
else
   begin:sh
      reg [width-1:0] sh_r1;
      reg [width-1:0] sh_r2;

   always_ff @ (posedge clk)
   begin
      sh_r1 <= din;
      sh_r2 <= sh_r1;
   end
end

assign dout1 = sh.sh_r1;
assign dout2 = sh.sh_r2;

endmodule
```

Example 2 – Conditional Generate: case-generate

```
// top.v
module top
#  (parameter mod_sel = 3,
    mod_sel2 = 3,
    width1 = 8,
    width2 = 16 )

   (input [width1-1:0] a1,
    input [width1-1:0] b1,
    output [width1-1:0] c1,
    input [width2-1:0] a2,
    input [width2-1:0] b2,
    output [width2-1:0] c2 );
```

```
case(mod_sel)
   0:
      begin:u1
         my_or u1(.a(a1),.b(b1),.c(c1) );
      end
   1:
      begin:u1
         my_and u2(.a(a2),.b(b2),.c(c2) );
      end
   default:
      begin:u1
         my_or u1(.a(a1),.b(b1),.c(c1) );
      end
endcase

case(mod_sel2)
   0:
      begin:u3
         my_or u3(.a(a1),.b(b1),.c(c1) );
      end
   1:
      begin:u4
         my_and u4(.a(a2),.b(b2),.c(c2) );
      end
   default:
      begin:def
         my_and u2(.a(a2),.b(b2),.c(c2) );
      end
endcase
endmodule

// my_and.v
module my_and
# (parameter width2 = 16 )

   (input [width2-1:0] a,
    input  [width2-1:0] b,
    output [width2-1:0] c
   );

assign c = a & b;
endmodule

// my_or.v
module  my_or
#  (parameter width = 8)
```

```
    (input [width-1:0] a,
     input [width-1:0] b,
     output [width-1:0] c );

assign c = a | b;
endmodule
```

Assertions

The parsing of SystemVerilog Assertions (SVA) is supported as outlined in the following table.

Assertion Construct	Support Level	Comment
Immediate assertions	Supported	
Concurrent assertions	Partially Supported, Ignored	Multiclock properties are not supported
Boolean expressions	Partially Supported, Ignored	In the boolean expressions, $rose function having a clocking event is not supported.
Sequence	Supported, ignored	
Declaring sequences	Partially Supported, Ignored	Sequence with ports declared in global space is not supported
Sequence operations	Partially Supported, Ignored	All variations of first_match, within and intersect in a sequence is not supported.
Manipulating data in a sequence	Partially Supported, Ignored	More than one assignment in the parenthesis is not supported.
Calling subroutines on sequence match	Partially Supported, Ignored	Calling of more than one tasks is not supported
System functions	Partially Supported	System functions $onehot, $onehot1, $countones and $countbits supported; $isunknown not supported
Declaring properties	Partially Supported, Ignored	Declaring of properties in a package and properties with ports declared in global space are not supported
Multiclock support	Partially Supported, Ignored	
Expect statement	Not Supported	
Final blocks	Partially Supported, Ignored	
Property blocks	Supported, Ignored	

Assertion Construct	Support Level	Comment
Checker	Partially Supported, Ignored	
Default clocking and default disable iff	Supported, Ignored	
Let statement	Partially Supported, Ignored	
Program	Partially Supported, Ignored	

Keyword Support

This table lists supported SystemVerilog keywords for the tool:

always_comb	always_ff	always_latch	assert*
assume*	automatic	bind*	bit
break	byte	checker*	clocking*
const	continue	cover*	do
endchecker*	endclocking*	endinterface	endproperty*
endsequence*	enum	expect*	extern
final*	function	global*	import
inside	int	interface	intersect*
let*	logic	longint	modport
packed	package	parameter	priority
property*	restrict*	return	sequence*
shortint	struct	task	throughout*
timeprecision*	timeunit*	typedef	union
unique	void	within*	

* Reserved keywords for SystemVerilog assertion parsing; cannot be used as identifiers or object names

CHAPTER 3

VHDL Language Support

This chapter discusses how you can use the VHDL language to create HDL source code for the Synopsys tool:

Language Constructs

This section generally describes how the tool pertains to the different VHDL language constructs. The topics include:

Supported VHDL Language Constructs

The following is a compact list of language constructs that are supported.

- Entity, architecture, and package design units
- Function and procedure subprograms
- All IEEE library packages, including:
  - std_logic_1164
  - std_logic_unsigned
  - std_logic_signed
  - std_logic_arith
  - numeric_std and numeric_bit
  - standard library package (std)
- In, out, inout, buffer, linkage ports
- Signals, constants, and variables
- Aliases
- Integer, physical, and enumeration data types; subtypes of these
- Arrays of scalars and records
- Record data types
- File types
- All operators (-, -, *, /, **, mod, rem, abs, not, =, /=, <, <=, >, >=, and, or, nand, nor, xor, xnor, sll, srl, sla, sra, rol, ror, &)

Note: With the ** operator, arguments are compiler constants. When the left operand is 2, the right operand can be a variable.

- Sequential statements: signal and variable assignment, wait, if, case, loop, for, while, return, null, function, and procedure call
- Concurrent statements: signal assignment, process, block, generate (for and if), component instantiation, function, and procedure call
- Component declarations and four methods of component instantiations
- Configuration specification and declaration
- Generics; attributes; overloading
- Next and exit looping control constructs
- Predefined attributes: t'base, t'left, t'right, t'high, t'low, t'succ, t'pred, t'val, t'pos, t'leftof, t'rightof, integer'image, a'left, a'right, a'high, a'low, a'range, a'reverse_range, a'length, a'ascending, s'stable, s'event
- Unconstrained ports in entities
- Global signals declared in packages

Unsupported VHDL Language Constructs

If any of these constructs are found, an error message is reported and the synthesis run is canceled.

- Register and bus kind signals
- Guarded blocks
- Expanded (hierarchical) names
- User-defined resolution functions. The tool only supports the resolution functions for std_logic and std_logic_vector.
- Slices with range indices that do not evaluate to constants

Partially-supported VHDL Language Constructs

When one of the following constructs is encountered, compilation continues, but will subsequently error out if logic must be generated for the construct.

- real data types (real data expressions are supported in VHDL-2008 IEEE float_pkg.vhd) – real data types are supported as constant declarations or as constants used in expressions as long as no floating point logic must be generated
- access types – limited support for file I/O
- null arrays – null arrays are allowed as operands in concatenation expressions

Ignored VHDL Language Constructs

The compiler ignores the following constructs in your design. If found, the tool parses and ignores the construct (provided that no logic is required to be synthesized) and continues with the synthesis run.

- disconnect
- report
- initial values on inout ports
- assert on ports and signals
- after

VHDL Language Constructs

This section describes the synthesis language support that the tool provides for each VHDL construct. The language information is taken from the most recent VHDL Language Reference Manual (Revision ANSI/IEEE Std 1076-1993). The section names match those from the LRM, for easy reference.

Data Types

Predefined Enumeration Types

Enumeration types have a fixed set of unique values. The two predefined data types – bit and Boolean, as well as the std_logic type defined in the std_logic_1164 package are the types that represent hardware values. You can declare signals and variables (and constants) that are vectors (arrays) of these types by using the types bit_vector, and std_logic_vector. You typically use std_logic and std_logic_vector, because they are highly flexible for synthesis and simulation.

std_logic Values	Treated by the tool as ...
'U' (uninitialized)	don't care
'X' (forcing unknown)	don't care
'0' (forcing logic 0)	logic 0
'1' (forcing logic 1)	logic 1
'Z' (high impedance)	high impedance
'W' (weak unknown)	don't care
'L' (weak logic 0)	logic 0
'H' (weak logic 1)	logic 1
'-' (don't care)	don't care

bit Values	Treated by the tool as ...
'0'	logic 0
'1'	logic 1

boolean Values	Treated by the tool as ...
false	logic 0
true	logic 1

User-defined Enumeration Types

You can create your own enumerated types. This is common for state machines because it allows you to work with named values rather than individual bits or bit vectors.

Syntax

type *type_name* **is (***value_list***);**

Examples

```
type states is (state0, state1, state2, state3);
type traffic_light_state is (red, yellow, green);
```

Integers

An integer is a predefined VHDL type that has 32 bits. When you declare an object as an integer, restrict the range of values to those you are using. This results in a minimum number of bits for the implementation and on ports.

Data Types for Signed and Unsigned Arithmetic

For signed arithmetic, you have the following choices:

- Use the std_logic_vector data type defined in the std_logic_1164 package, and the package std_logic_signed.
- Use the signed data type, and signed arithmetic defined in the package std_logic_arith.
- Use an integer subrange (for example: signal mysig: integer range -8 to 7). If the range includes negative numbers, the tool uses a two's-complement bit vector of minimum width to represent it (four bits in this example). Using integers limits you to a 32-bit range of values, and is typically only used to represent small buses. Integers are most commonly used for indexes.

- Use the signed data type from the numeric_std or numeric_bit packages.

For unsigned arithmetic, you have the following choices:

- Use the std_logic_vector data type defined in the std_logic_1164 package and the package std_logic_unsigned.
- Use the unsigned data type and unsigned arithmetic defined in the package std_logic_arith.
- Use an integer subrange (for example: signal mysig: integer range 0 to 15). If the integers are restricted to positive values, the tool uses an unsigned bit vector of minimum width to represent it (four bits in this example). Using integers limits you to a 32-bit range of values, and is typically only used to represent small buses (integers are most commonly used for indexes).
- Use the unsigned data type from the numeric_std or numeric_bit packages.

Physical Types

A physical type is a numeric type for representing a physical quantity such as time. The declaration of a physical type includes the specification of a base unit and possibly a number of secondary units that are multiples of the base unit. The syntax for declaring physical types is:

type *physical_type* **is** *range_constraint*
  units
    *base_unit***;**
    *unit_definitions***;**
    ...
  end units

The following example illustrates a physical-type definition:

```
type time is range -2_147_483_647 to 2_147_483_647
   units
      fs;
      ps = 1000 fs;
      ns = 1000 ps;
      us = 1000 ns;
      ms = 1000 us;
      sec = 1000 ms;
      min = 60 sec;
   end units;
```

Arrays

An array is a composite object made up of elements that are of the same subtype.

> **type** *typeName* **is array (range) of** *elementType*
>
> **type** *typeName* **is array (type range <>) of** *elementType*

Each of the elements within the array is indexed by one or more indices belonging to specified discrete types. The number of indices is the number of dimensions (that is, a one-dimensional array has one index, a two-dimensional array has two indices, etc.). The order of indices is significant and follows the order of dimensions in the type declaration.

An array can be either constrained or unconstrained. An array is said to be constrained if the size of the array is constrained. The size of the array can be constrained using a discrete type mark or a range. In both cases, the number of the elements in the array is known during the compilation.

An array is said to be unconstrained if its size is unconstrained. The size of an unconstrained array is declared in the form of the name of the discrete type with an unconstrained range. The number of elements of an unconstrained array type is unknown. The size of a particular object is specified only when it is declared.

The standard package contains declarations of two one-dimensional unconstrained predefined array types: string and bit_vector. Elements of the string type are of the type character and are indexed by positive values, and the elements of the bit_vector type are of the type bit and are indexed by natural values.

Declaring and Assigning Objects in VHDL

VHDL objects (object classes) include signals (and ports), variables, and constants. The tool does not support the file object class.

Naming Objects

VHDL is case insensitive. A VHDL name (identifier) must start with a letter and can be followed by any number of letters, numbers, or underscores ('_'). Underscores cannot be the first or last character in a name and cannot be used twice in a row. No special characters such as '$', '?', '*', '-', or '!', can be used as part of a VHDL identifier.

Syntax

object_class object_name **:** *data_type* [**:=** *initial_value*] **;**

- In the above syntax:
- object_class is a signal, variable, or constant.
- object_name is the name (the identifier) of the object.
- data_type can be any predefined data type (such as bit or std_logic_vector) or user-defined data type.

Assignment Operators

<= Signal assignment operator.

:= Variable assignment and initial value operator.

Ranges

A range specifies a subset of values of a scalar type.

range *leftBound* **to** *rightBound*

range *leftBound* **downto** *rightBound*

range <>

A range can be either ascending or descending. A range is called ascending when it is specified with the keyword to as the direction and the left bound value is smaller than the right bound (otherwise the range is null). A range is called descending when the range is specified with the keyword downto as the direction and the left bound is greater than the right bound (otherwise the range is null). A range can be null range if the set contains no values.

Dynamic Range Assignments

The tools support VHDL assignments with dynamic ranges, which are defined as follows:

A(b downto c) <= D(e downto f);

A and D are constrained variables or signals, and b, c, e, and f are constants (generics) or variables. Dynamic range assignments can be used for RHS, LHS, or both.

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

entity test is
   port (data_out: out std_logic_vector(63 downto 0);
         data_in: in std_logic_vector(63 downto 0);
         selector: in NATURAL range 0 to 7);
end test;

architecture rtl of test is
begin
   data_out((selector*8)+7 downto (selector*8))
      <= data_in((selector*8)+7 downto (selector*8));
end rtl;
```

Dynamic Slicing Support

Dynamic slicing can be applied for the following conditions:

- Comparison statements
- Concatenations

Example 1: Comparison of Dynamic Slices Within an if Statement

This code example shows dynamic slices compared in an if statement.

Example 2: Dynamic Slices with Concatenation Statements

For this example, dynamic slice operands are supported in a concatenation statement. The operands must be signals/variables/ports or slices/indices of signals/variables/ports.

Dynamic Range Assignment Limitations

Currently, the following limitations apply to dynamic range assignments:

- There is no support for selected signal assignment; i.e., with *expression* Select.
- Only comparisons with if statements are supported; no other comparison statements can be used.

Null Ranges

A null range is a range that specifies an empty subset of values. A range specified as *m* to *n* is a null range when *m* is greater than *n*, and a range specified as *n* downto *m* is a null range when *n* is less than *m*.

Support for null ranges allows ports with negative ranges to be compiled successfully. During compilation, any port declared with a null range and its related logic are removed by the compiler.

In the following example, port a_in1 (-1 to 0) is a null range and is subsequently removed by the compiler.

```
-- top.vhd
library ieee;
use ieee.std_logic_1164.all;

entity top is
generic (width : integer := 0);
   port (a_in1 : in std_logic_vector(width -1 downto 0);
         b_in1 : in std_logic_vector(3 downto 0);
         c_out1 : out std_logic_vector(3 downto 0));
end top;

architecture struct of top is
component sub is
   port (a_in1 : in std_logic_vector(width -1 downto 0);
         b_in1 : in std_logic_vector(3 downto 0);
         c_out1 : out std_logic_vector(3 downto 0));
end component;

begin
   UUT : sub port map (a_in1 => a_in1, b_in1 => b_in1,
      c_out1 => c_out1);
end struct;
```

```
-- sub.vhd
library ieee;
use ieee.std_logic_1164.all;

entity sub is
generic (width : integer := 0);
   port (a_in1 : in std_logic_vector(width -1 downto 0);
         b_in1 : in std_logic_vector(3 downto 0);
         c_out1 : out std_logic_vector(3 downto 0));
end sub;

architecture rtl of sub is
begin
   c_out1 <= not (b_in1 & a_in1);
end rtl;
```

Signals and Ports

In VHDL, the port list of the entity lists the I/O signals for the design using the syntax:

port (port_declaration**);**

where *port_declaration* is any of the following:

portSignalName **: in** *portSignalType* **:=** *initialValue*

portSignalName **: out** *portSignalType* **:=** *initialValue*

portSignalName **: inout** *portSignalType* **:=** *initialValue*

portSignalName **: buffer** *portSignalType* **:=** *initialValue*

portSignalName **: linkage** *portSignalType* **:=** *initialValue*

Ports of mode in can be read from, but not assigned (written) to. Ports of mode out can be assigned to, but not read from. Ports of mode inout are bidirectional and can be read from and assigned to. Ports of mode buffer are like inout ports, but can have only one associated internal driver. With ports of mode linkage, the value of the port can be read or updated, but only by appearing as an actual corresponding to an interface object of mode linkage.

Internal signals are declared in the architecture declarative area and can be read from or assigned to anywhere within the architecture.

Examples

```
signal my_sig1 : std_logic; -- Holds a single std_logic bit
begin -- An architecture statement area
my_sig1 <= '1'; -- Assign a constant value '1'

-- My_sig2 is a 4-bit integer
   signal my_sig2 : integer range 0 to 15;
begin -- An architecture statement area
my_sig2 <= 12;

-- My_sig_vec1 holds 8 bits of std_logic, indexed from 0 to 7
   signal my_sig_vec1 : std_logic_vector (0 to 7);
begin -- An architecture statement area

-- Simple signal assignment with a literal value
my_sig_vec1 <= "01001000";

-- 16 bits of std_logic, indexed from 15 down to 0
   signal my_sig_vec2 : std_logic_vector (15 downto 0);
begin -- An architecture statement area

-- Simple signal assignment with a vector value
my_sig_vec2 <= "0111110010000101";

-- Assigning with a hex value FFFF
my_sig_vec2 <= X"FFFF";

-- Use package Std_Logic_Signed
   signal my_sig_vec3 : signed (3 downto 0);
begin -- An architecture statement area

-- Assigning a signed value, negative 7
my_sig_vec3 <= "1111";

-- Use package Std_Logic_Unsigned
   signal my_sig_vec4 : unsigned (3 downto 0);
begin -- An architecture statement area

-- Assigning an unsigned value of 15
my_sig_vec4 <= "1111";

-- Declare an enumerated type, a signal of that type, and
-- then make an valid assignment to the signal
   type states is (state0, state1, state2, state3);
   signal current_state : states;
begin -- An architecture statement area
current_state <= state2;
```

```
-- Declare an array type, a signal of that type, and
-- then make a valid assignment to the signal
   type array_type is array (1 downto 0) of
      std_logic_vector (7 downto 0);
   signal my_sig: array_type;
begin -- An architecture statement area
my_sig <= ("10101010","01010101");
```

Variables

VHDL variables are declared within a process or subprogram and then used internally. Generally, variables are not visible outside the process or subprogram where they are declared unless passed as a parameter to another subprogram.

Example

```
process (clk) -- What follows is the process declaration area
   variable my_var1 : std_logic := '0'; -- Initial value '0'

begin -- What follows is the process statement area
   my_var1 := '1';
end process;
```

Example

```
process (clk, reset)
-- Set the initial value of the variable to hex FF
   variable my_var2 : std_logic_vector (1 to 8) := X"FF";

begin
-- my_var2 is assigned the octal value 44
   my_var2 := O"44";
end process;
```

VHDL Constants

VHDL constants are declared in any declarative region and can be used within that region. The value of a constant cannot be changed.

Example

```
package my_constants is
   constant num_bits : integer := 8;

-- Other package declarations

end my_constants;
```

Aliases

An alias declares an alternative name for any existing object which can be a signal, variable, constant, or file.

alias *aliasName* **:** *aliasType* **is** *objectName***;**

Aliases can also be used for non-objects, virtually everything that has been previously declared with the exception of labels, loop parameters, and generate parameters. An alias does not define a new object; it is just a specific name assigned to some existing object.

Aliases are typically used to assign specific names to vector slices to improve readability of the specification. When an alias denotes a slice of an object and no subtype indication is given, the subtype of the object is viewed as if it was of the subtype specified by the slice.alias alias_name : alias_type is object_name;

Libraries and Packages

When you want to synthesize a design in VHDL, include the HDL files in the source files list of your tool. Often your VHDL design will have more than one source file. List all the source files in the order you want them compiled; the files at the top of the list are compiled first.

Compiling Design Units into Libraries

All design units in VHDL, including your entities and packages are compiled into libraries. A library is a special directory of entities, architectures and/or packages. You compile source files into libraries by adding them to the source file list. In VHDL, the library you are compiling has the default name work. All entities and packages in your source files are automatically compiled into work. You can keep source files anywhere on your disk, even though you add them to libraries. You can have as many libraries as are needed.

1. To add a file to a library, select the file in the Project view.

   The library structure is maintained in the Project view. The name of the library where a file belongs appears on the same line as the filename, and directly in front of it.

2. Choose Project -> Set Library from the menu bar, then type the library name. You can add any number of files to a library.

3. If you want to use a design unit that you compiled into a library (one that is no longer in the default work library), you must use a library clause in the VHDL source code to reference it.

   For example, if you add a source file for the entity ram16x8 to library my_rams, and instantiate the 16x8 RAM in the design called top_level, you must add library my_rams; just before defining top_level.

Predefined Packages

The tool supports the two standard libraries, std and ieee, that contain packages containing commonly used definitions of data types, functions, and procedures. These libraries and their packages are built in to the tool, so you do not compile them. The predefined packages are described in the following table.

Library	Package	Description
std	standard	Defines the basic VHDL types including bit and bit_vector
ieee	std_logic_1164	Defines the 9-value std_logic and std_logic_vector types
ieee	numeric_bit	Defines numeric types and arithmetic functions. The base type is BIT.

Library	**Package**	**Description**
ieee	numeric_std	Defines arithmetic operations on types defined in std_logic_1164
ieee	std_logic_arith	Defines the signed and unsigned types, and arithmetic operations for the signed and unsigned types
ieee	std_logic_signed	Defines signed arithmetic for std_logic and std_logic_vector
ieee	std_logic_unsigned	Defines unsigned arithmetic for std_logic and std_logic_vector

The tool also has vendor-specific built-in macro libraries for components like gates, counters, flip-flops, and I/Os. The libraries are located in *installDirectory*/lib/*vendorName*. Use the built-in macro libraries to instantiate vendor macros directly into the VHDL designs and forward-annotate them to the output netlist. Refer to the appropriate vendor support chapter for more information.

Additionally, the tool library contains an attributes package of built-in attributes and timing constraints (*installDirectory*/lib/vhd/synattr.vhd) that you can use with VHDL designs. The package includes declarations for timing constraints (including black-box timing constraints), vendor-specific attributes and synthesis attributes. To access these built-in attributes, add the following two lines to the beginning of each of the VHDL design units that uses them:

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

If you want the addition operator (+) to take two std_ulogic or std_ulogic_vector as inputs, you need the function defined in the std_logic_arith package (the cdn_arith.vhd file in *installDirectory*/lib/vhd/). Add this file to the design. To successfully compile, the VHDL file that uses the addition operator on these input types must have include the following statement:

```
use work.std_logic_arith.all;
```

Accessing Packages

To gain access to a package include a library clause in your VHDL source code to specify the library the package is contained in, and a use clause to specify the name of the package. The library and use clauses must be included immediately before the design unit (entity or architecture) that uses the package definitions.

Syntax

> **library** *library_name***;**
> **use** *library_name***.***package_name***.all;**

To access the data types, functions and procedures declared in std_logic_1164, std_logic_arith, std_logic_signed, or std_logic_unsigned, you need a library ieee clause and a use clause for each of the packages you want to use.

Example

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

-- Other code
```

Library and Package Rules

To access the standard package, no library or use clause is required. The standard package is predefined (built-in) in VHDL, and contains the basic data types of bit, bit_vector, Boolean, integer, real, character, string, and others along with the operators and functions that work on them.

If you create your own package and compile it into the work library to access its definitions, you still need a use clause before the entity using them, but not a library clause (because work is the default library.)

To access packages other than those in work and std, you must provide a library and use clause for each package as shown in the following example of creating a resource library.

```
-- Compile this in library mylib
library ieee;
use ieee.std_logic_1164.all;
```

```
package my_constants is
constant max: std_logic_vector(3 downto 0):= "1111";
   .
   .
   .
end package;

-- Compile this in library work
library ieee, mylib;
use ieee.std_logic_1164.all;
use mylib.my_constants.all;

entity compare is
   port (a: in std_logic_vector(3 downto 0);
         z: out std_logic);
end compare;

architecture rtl of compare is
begin
   z <= '1' when (a = max) else '0';
end rtl;
```

The rising_edge and falling_edge functions are defined in the std_logic_1164 package. If you use these functions, your clock signal must be declared as type std_logic.

Instantiating Components in a Design

No library or use clause is required to instantiate components (entities and their associated architectures) compiled in the default work library. The files containing the components must be listed in the source files list before any files that instantiate them.

To instantiate components from the built-in technology-vendor macro libraries, you must include the appropriate use and library clauses in your source code. Refer to the section for your vendor for more information.

To create a separate resource library to hold your components, put all the entities and architectures in one source file, and assign that source file to the library components in the design. To access the components from your source code, put the clause library components; before the designs that instantiate them. There is no need for a use clause. The design file must include both the files that create the package components and the source file that accesses them.

Package Definitions

A package is a unit that groups various declarations to be shared among several designs. Packages are stored in libraries for greater convenience. A package consists of package declaration as shown in the following syntax:

> **package** *packageName* **is**
>
> *package_declarations*
>
> **end package** *packageName***;**

The purpose of a package is to declare shareable types, subtypes, constants, signals, files, aliases, component attributes, and groups. Once a package is defined, it can be used in multiple independent designs. Items declared in a package declaration are visible in other design units if the use clause is applied.

Literals

There are five classes of literals: numeric literals, enumeration literals, string literals, bit-string literals, and the literal null.

Numeric Literals

The class of numeric literals includes abstract literals (which include integer literals and real literals) and physical literals. A real literal includes a decimal point, while an integer literal does not. When a real or integer literal is written in the conventional decimal notation, it is called a decimal literal.

When a number is written in exponential form, the letter E of the exponent can be written either in lowercase or in uppercase. If the exponential form is used for an integer number, then a zero exponent is allowed.

Abstract literals can be written in the form of based literals. In such cases, the base is specified explicitly (in decimal literals, the base is implicitly ten). The base in a based literal must be at least two and no more than sixteen. The base is specified in decimal notation.

The digits used in based literals can be any decimal digits (0..9) or a letter (either in upper or lower case). The meaning of based notation is as in decimal literals, with the exception of base.

A physical literal consists of an abstract numeric literal followed by an identifier that denotes the unit of the given physical quantity.

Enumeration literals

The enumeration literals are literals of enumeration types, used in type declaration and in expressions that evaluate to a value of an enumeration type. This class of literals includes identifiers and character literals. Reserved words cannot be used in identifiers, unless they are a part of extended identifiers that start and end with a backslash.

String Literals

String literals are made up of a sequence of graphic characters (letters, digits, and special characters) enclosed between double quotation marks. This class of literals is usually used for warnings or reports that are displayed during simulation.

Bit-String Literals

Bit-string literals represent values of string literals that denote sequences of extended digits, the range of which depends on the specified base.

The base specifier determines the base of the digits: the letter B denotes binary digits (0 or 1), letter O denotes octal digits (0 to 7), and letter X denotes hexadecimal (digits 0 to 9 and letters A to F, case insensitive). Underlines can be used to increase readability and have no impact on the value.

All values specified as bit-string literals are converted into binary representation without underlines. Binary strings remain unchanged (only underlines are removed), each octal digit is converted into three bits and each hexadecimal character is converted into four bits.

Operators

The tool supports the creation of expressions using all predefined VHDL operators:

Arithmetic Operator	Description
-	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation (supported for compile-time constants and when left operand is 2; right operand can be a variable)
abs	Absolute value
mod	Modulus
rem	Remainder

Relational Operator	Description
=	Equal (if either operand has a bit with an 'X' or 'Z' value, the result is 'X')
/=	Not equal (if either operand has a bit with an 'X' or 'Z' value, the result is 'X')
<	Less than (if, because of unknown bits in the operands, the relation is ambiguous, then the result is the unknown value 'X')
<=	Less than or equal to (if, because of unknown bits in the operands, the relation is ambiguous, then the result is the unknown value 'X')
>	Greater than (if, because of unknown bits in the operands, the relation is ambiguous, then the result is the unknown value 'X')
>=	Greater than or equal to (if, because of unknown bits in the operands, the relation is ambiguous, then the result is the unknown value 'X')

Logical Operator	Description
and	and
or	or
nand	nand
nor	nor
xor	xor
xnor	xnor
not	not (takes only one operand)

Shift Operator	Description
sll	Shift left logical – logically shifted left by R index positions
srl	Shift right logical – logically shifted right by R index positions
sla	Shift left arithmetic – arithmetically shifted left by R index positions
sra	Shift right arithmetic – arithmetically shifted right by R index positions
rol	Rotate left logical – rotated left by R index positions
ror	Rotate right logical – rotated right by R index positions

Miscellaneous Operator	Description
-	Identity
-	Negation
&	Concatenation

Note: Initially, X's are treated as “don’t-cares”, but they are eventually converted to 0's or 1's in a way that minimizes hardware.

Large Time Resolution

The support of predefined physical time types includes the expanded range from –2147483647 to +2147483647 with units ranging from femtoseconds, and secondary units ranging up to an hour. Predefined physical time types allow selection of a wide number range representative of time type.

Example 1 – Using Large Time Values in Comparisons

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

entity test is
   generic (INTERVAL1 : time := 1000 fs;
            INTERVAL2 : time := 1 ps;
            INTERVAL3 : time := 1000 ps;
            INTERVAL4 : time := 1 ns
   );

   port (a : in std_logic_vector(3 downto 0);
         b : in std_logic_vector(3 downto 0);
         c : out std_logic_vector(3 downto 0);
         d : out std_logic_vector(3 downto 0)
   );
end test;

architecture RTL of test is
begin
   c <= (a and b) when (INTERVAL1 = INTERVAL2) else
      (a or b);
   d <= (a xor b) when (INTERVAL3 /= INTERVAL4) else
      (a nand b);
end RTL;
```

Example 2 – Using Large Time Values in Constant Calculations

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

entity test is
   generic (Interval : time := 20 ns;
            CLK_PERIOD : time := 8 ns );
   port (en : in std_logic;
         a : in std_logic_vector(10 downto 0);
         b : in std_logic_vector(10 downto 0);
         a_in : in std_logic_vector(7 downto 0);
         b_in : in std_logic_vector(7 downto 0);
         dummyOut : out std_logic_vector(7 downto 0);
         out1 : out std_logic_vector(10 downto 0));
end entity;

architecture behv of test is
   constant my_time : positive := (Interval / 2 ns);
   constant CLK_PERIOD_PS : real := real(CLK_PERIOD / 1 ns);
   constant RESULT : positive := integer(CLK_PERIOD_PS);
   signal dummy : std_logic_vector (RESULT-1 downto 0);
   signal temp : std_logic_vector (my_time downto 0);
begin
   process (a, b)
   begin
      temp <= a and b;
      out1 <= temp;
   end process;
dummy <= (others => '0') when en = '1' else
   (a_in or b_in);
dummyOut <= dummy;
end behv;
```

Example 3 – Using Large Time Values in Generic Calculations

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

entity test is
   generic (clk_period : time := 6 ns);
   port (rst_in : in std_logic;
         in1 : in std_logic;
         CLK_PAD : in std_logic;
         RST_DLL : in std_logic;
         dout : out std_logic;
         CLK_out : out std_logic;
         LOCKED : out std_logic);
end test;

architecture STRUCT of test is
   signal CLK, CLK_int, CLK_dcm : std_logic;
   signal clk_dv : std_logic;
   constant clk_prd : real := real(clk_period / 2.0 ns);
begin
U1 : IBUFG port map (I => CLK_PAD, O => CLK_int);
U2 : DCM generic map
(CLKDV_DIVIDE  => clk_prd)
   port map (CLKFB => CLK,
            CLKIN => CLK_int,
            CLKDV => clk_dv,
            DSSEN => '0',
            PSCLK => '0',
            PSEN => '0',
            PSINCDEC => '0',
            RST => RST_DLL,
            CLK0 => CLK_dcm,
            LOCKED => LOCKED);
U3 : BUFG port map (I => CLK_dcm, O => CLK);
CLK_out <= CLK;
   process (clk_dv , rst_in, in1)
   begin
      if (rst_in = '1') then
         dout <= '0';
      elsif (clk_dv'event and clk_dv = '1') then
         dout <= in1;
      end if;
   end process;
end architecture STRUCT;
```

VHDL Process

The VHDL keyword process introduces a block of logic that is triggered to execute when one or more signals change value. Use processes to model combinational and sequential logic.

process Template to Model Combinational Logic

```
<optional_label> : process (<sensitivity_list>)

-- Declare local variables, data types,
-- and other local declarations here

begin
   -- Sequential statements go here, including:
   --    signal and variable assignments
   --    if and case statements
   --    while and for loops
   --    function and procedure calls
end process  <optional_label>;
```

Sensitivity List

The sensitivity list specifies the signal transitions that trigger the process to execute. This is analogous to specifying the inputs to logic on a schematic by drawing wires to gate inputs. If there is more than one signal, separate the names with commas.

A warning is issued when a signal is not in the sensitivity list but is used in the process, or when the signal is in the sensitivity list but not used by the process.

Syntax

process (*signal1***,** *signal2***, ...);**

A process can have only one sensitivity list, located immediately after the keyword process, or one or more wait statements. If there are one or more wait statements, one of these wait statements must be either the first or last statement in the process.

List all signals feeding into the combinational logic (all signals that affect signals assigned inside the process) in the sensitivity list. If you forget to list all signals, the tool generates the desired hardware, and reports a warning message that you are not triggering the process every time the hardware is changing its outputs, and therefore your pre- and post-synthesis simulation results might not match.

Any signals assigned in the process must either be outputs specified in the port list of the entity or declared as signals in the architecture declarative area. Any variables assigned in the process are local and must be declared in the process declarative area.

Note: Make sure all signals assigned in a combinational process are explicitly assigned values each time the process executes. Otherwise, the synthesis tool must insert level-sensitive latches in your design, in order to hold the last value for the paths that don't assign values (if, for example, you have combinational loops in your design). This usually represents coding error, so synthesis issues a warning message that level-sensitive latches are being inserted into the design because of combinational loops. You will get an error message if you have combinational loops in your design that are not recognized as level-sensitive latches.

Common Sequential Statements

This section describes common sequential statements.

Procedures

A procedure is a form of a subprogram that contains local declarations and a sequence of statements. A procedure can be called from any place within the architecture. The procedure definition consists of two parts:

- the procedure declaration, which contains the procedure name and the parameter list required when the procedure is called; the procedure declaration consists of the procedure name and the formal parameter list. In the procedure specification, the identifier and optional formal parameter list follow the reserved word procedure.
- the procedure body, which consists of the local declarations and statements required to execute the procedure; the procedure body defines the procedure's algorithm composed of sequential statements. When the procedure is called, execution of the sequence of statements declared within the procedure body begins. The procedure body consists of the subprogram declarative part following the reserved word is with the subprogram statement part placed between the reserved words begin and end.

The basic syntax for a procedure is:

> **procedure** *procedureName* **(***formalParameterList***)**
>
> **procedure** *procedureName* (*formalParameterList*) **is**
>   *procedureDeclarations*
> **begin**
>   *sequential statements*
> **end procedure** *procedureName***;**

if-then-else Statement

Syntax

> **if** *condition1* **then**
>   *sequential_statement(s)*
> [**elsif** *condition2* **then**
>   *sequential_statement(s)*]
> [**else**
>   *sequential_statement(s)*]
> **end if;**

The else and elsif clauses are optional.

Example

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

entity mux is
   port (output_signal : out std_logic;
         a, b, sel : in std_logic);
end mux;

architecture if_mux of mux is
begin
   process (sel, a, b)
   begin
      if sel = '1' then
         output_signal <= a;
      elsif sel = '0' then
         output_signal <= b;
      else
         output_signal <= 'X';
      end if;
   end process ;
end if_mux;
```

case Statement

Syntax

case *expression* **is**
    when *choice1* **=>** *sequential_statement(s)*
    when *choice2* **=>** *sequential _statement(s)*

-- *Other case choices*

    when *choiceN* **=>** *sequential_statement(s)*
end case;

Note: VHDL requires that the expression match one of the given choices. To create a default, have the final choice be when others => sequential_statement(s) or null. (Null means not to do anything.)

Example

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

entity mux is
   port (output_signal : out std_logic;
         a, b, sel : in std_logic);
end mux;

architecture case_mux of mux is
begin
   process (sel, a, b)
   begin
      case sel is
         when '1' =>
            output_signal <= a;
         when '0' =>
            output_signal <= b;
         when others =>
            output_signal <= 'X';
      end case;
   end process;
end case_mux;
```

Note: To test the condition of matching a bit vector, such as `"0-11"`, that contains one or more don't-care bits, do *not* use the equality relational operator (=). Instead, use the std_match function (in the ieee.numeric_std package), which succeeds (evaluates to true) whenever all of the explicit constant bits (0 or 1) of the vector are matched, regardless of the values of the bits in the don't-care (-) positions. For example, use the condition `std_match(a, "0-11")` to test for a vector with the first bit unset (0) and the third and fourth bits set (1).

Loop Statement

Loop statements are used to repeatedly execute a sequence of sequential statements. The basic syntax for a loop is:

[*loop_label* **:**]*iteration_scheme* **loop**
    sequential statements
        [**next** [*label*] [**when** *condition*]**;**
        [**exit** [*label*] [**when** *condition*]**;**
end loop [*loop_label*]**;**

Loop labels are optional, but can be useful when writing nested loops. The next and exit statements are sequential statements that can only be used within a loop:

- The next statement terminates the remainder of the current loop iteration and causes execution to proceed to the next loop iteration.
- The exit statement terminates the loop and omits the remainder of the statements. Execution continues with the next statement after the loop is exited.

There are three loop iteration types:

- basic loop
- while … loop

- for … loop

Basic Loop Statement

The basic loop has no iteration scheme and is executed continuously until it encounters an exit or next statement. The basic loop must include at least one wait statement. As an example, assume a 4-bit counter that counts from 0 to 15. When it reaches 15, it begins over from 0. A wait statement is used to cause the loop to execute every time the clock transitions from 0 to 1.

While-Loop Statement

The while … loop evaluates a true-false condition. When the condition is true, the loop repeats, otherwise the loop is skipped and execution halted. The syntax for the while… loop is:

```
[loop_label :] while condition loop
    sequential statements
        [next [label] [when condition];
        [exit [label] [when condition];
end loop [loop_label];
```

The condition of the loop is tested prior to each iteration (including the first iteration). If the result is false, the loop is terminated.

For-Loop Statement

The for ... loop uses an integer iteration scheme to determine the number of iterations. The syntax is:

```
[loop_label :] for identifier in range loop
    sequential statements
        [next [label] [when condition];
        [exit [label] [when condition];
end loop [loop_label];
```

The *identifier* is automatically declared by the loop itself and does not need to be declared separately. The value of *identifier* can only be read within the loop and is not accessible outside the loop; its value cannot be assigned or changed in contrast to the while ... loop that can accept variables that are modified inside the loop.

The *range* is a computable integer range in one of the following forms, in which *integer_expression* must evaluate to an integer:

integer_expression **to** *integer_expression*

integer_expression **downto** *integer_expression*

The for statement also supports loop bounds where one or both of the loops are non-constant values.

Example: VHDL for Loop With Non-Constant Bounds

Next and Exit Statements

The next statement causes execution to jump to the next iteration of a loop statement and then proceed with the next iteration. The next statement syntax is:

next [*label*] [**when** *condition*]**;**

The when keyword is optional and executes the next statement when its condition evaluates to the Boolean value TRUE.

The exit statement omits the remaining statements, terminating the loop entirely and continuing with the next statement after the exited loop. The exit statement syntax is::

exit [*label*] [**when** *condition*]**;**

The when keyword is optional and executes the next statement when its condition evaluates to the Boolean value TRUE.

Return Statement

The return statement ends the execution of a subprogram (procedure or function) in which it appears and causes an unconditional jump to the end of a subprogram.

return *expression***;**

A return statement can only be used within a procedure or function body. When a return statement appears within nested subprograms, the return applies to the innermost subprogram (i.e., the jump is performed to the next end procedure or end function clause).

Assertion Statement

An assertion statement checks if a given condition is true and, if not, performs some action.

```
assert condition
   report string
   severity severityLevel;
```

The *condition* specified in an assert statement must evaluate to a boolean value (true or false). If it is false, it is said that an assertion violation has occurred. The expression specified in the report clause is the message of predefined type string to be reported when the assertion violation occurs.

If the severity clause is present, it must specify an expression of predefined type *severity level* which determines the severity level of the assertion violation. The severity-level type is specified in the standard package and includes the following values: NOTE, WARNING, ERROR, and FAILURE. If the severity clause is omitted, it is implicitly assumed to be ERROR.

When an assertion violation occurs, the report is issued and displayed on the screen. The severity level defines the degree to which the violation of the assertion affects operation of the corresponding process:

- NOTE – used to pass informative messages
- WARNING – used in unusual conditions in which the operation can be continued, but with unpredictable results
- ERROR – used when the assertion violation makes continuation of the operation not feasible
- FAILURE – used when the assertion violation is a fatal error and the operation must be immediately terminated

Block Statement

A block statement groups concurrent statements with an architecture to improve readability of the specification.

```
block_label : block
   declarations
begin
   concurrent statements
end block block_label;
```

Each block is assigned a label placed just before the block reserved word. This same label can be optionally repeated at the end of the block immediately following the end block reserved words.

A block statement can be preceded by two optional parts: a header and a declarative part. The declarative part introduces any of the declarations possible for an architecture including declarations of subprograms, types, subtypes, constants, signals, shared variables, files, aliases, components, attributes, configurations, use clauses and groups. These declarations are local to the block and are not visible outside of the block.

A block header can also include port and generic declarations (similar to an entity), as well as port and generic map declarations. The purpose of port map and generic map statements is to map signals and other objects declared outside of the block into the ports and generic parameters that have been declared within the block.

The statements part may contain any concurrent constructs allowed in an architecture. In particular, other block statements can be used here. This way, a kind of hierarchical structure can be introduced into a single architecture body (for additional information, see Configuration Declaration, on page 277.

Concurrent Signal Assignments

There are three types of concurrent signal assignments in VHDL.

- Simple
- Selected (with-select-when)
- Conditional (when-else)

Use the concurrent signal assignment to model combinational logic. Put the concurrent signal assignment in the architecture body. You can have any number of statements to describe your hardware implementation. Because all statements are concurrently active, the order you place them in the architecture body is not significant.

Re-evaluation of Signal Assignments

Every time any signal on the right side of the assignment operator (<=) changes value (including signals used in the expressions, values, choices, or conditions), the assignment statement is re-evaluated, and the result is assigned to the signal on the left side of the assignment operator. You can use any of the predefined operators to create the assigned value.

Simple Signal Assignments

Syntax

signal **<=** *expression***;**

Example

```
architecture simple_example of simple is
begin
   c <= a nand b;
end simple_example;
```

Selected Signal Assignments

Syntax

with *expression* **select**
signal **<=** *value1* **when** *choice1***,**
    value2 **when** *choice2***,**
    .
    .
    .
    valueN **when** *choiceN***;**

Example

```
library ieee;
use ieee.std_logic_1164.all;
entity mux is
   port (output_signal : out std_logic;
         a, b, sel : in std_logic);
end mux;
```

```
architecture with_select_when of mux is
begin
   with_sel_select
   output_signal <= a when '1',
      b when '0',
      'X' when others;
end with_select_when;
```

Conditional Signal Assignments

Syntax

signal **<=** *value1* **when** *condition1* **else**
value2 **when** *condition2* **else**
valueN-1 **when** *conditionN-1* **else**
*valueN***;**

Example

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

entity mux is
   port (output_signal: out std_logic;
         a, b, sel: in std_logic);
end mux;

architecture when_else_mux of mux is
begin
output_signal <= a when sel = '1' else
   b when sel = '0' else
   'X';
end when_else_mux;
```

Note: To test the condition of matching a bit vector, such as `"0-11"`, that contains one or more don't-care bits, do *not* use the equality relational operator (=). Instead, use the std_match function (in the ieee.numeric_std package), which succeeds (evaluates to true) whenever all of the explicit constant bits (0 or 1) of the vector are matched, regardless of the values of the bits in the don't-care (-) positions. For example, use the condition `std_match(a, "0-11")` to test for a vector with the first bit unset (0) and the third and fourth bits set (1).

Resource Sharing

When you have mutually exclusive operators in a case statement, the tool shares resources for the operators in the case statements. For example, automatic sharing of operator resources includes adders, subtractors, incrementors, decrementors, and multipliers.

Combinational Logic

Combinational logic is hardware with output values based on some function of the current input values. There is no clock and no saved states. Most hardware is a mixture of combinational and sequential logic.

Create combinational logic with concurrent signal assignments and/or processes.

Sequential Logic

Sequential logic is hardware that has an internal state or memory. The state elements are either flip-flops that update on the active edge of a clock signal, or level-sensitive latches, that update during the active level of a clock signal.

Because of the internal state, the output values might depend not only on the current input values, but also on input values at previous times. State machines are made of sequential logic where the updated state values depend on the previous state values. There are standard ways of modeling state machines in VHDL. Most hardware is a mixture of combinational and sequential logic.

Create sequential logic with processes and/or concurrent signal assignments.

Component Instantiation in VHDL

A structural description of a design is made up of component instantiations that describe the subsystems of the design and their signal interconnects. The tool supports four major methods of component instantiation:

- Simple component instantiation (described below)
- Selected component instantiation
- Direct entity instantiation
- Configurations described in Configuration Specification, on page 273

Simple Component Instantiation

In this method, a component is first declared either in the declaration region of the architecture, or in a package of (typically) component declarations, and then instantiated in the statement region of the architecture. By default, the synthesis process binds a named entity (and architecture) in the working library to all component instances that specify a component declaration with the same name.

Syntax

label **:** [**component**] *declaration_name*
  [**generic map (***actual_generic1***,** *actual_generic2***, ...)**]
  [**port map (** *port1***,** *port2***, ...)**] **;**

The use of the reserved word component is optional in component instantiations.

Example: VHDL 1987

```
architecture struct of hier_add is
component add
   generic (size : natural);
   port (a : in bit_vector(3 downto 0);
         b : in bit_vector(3 downto 0);
         result : out bit_vector(3 downto 0));
end component;
```

```
begin
-- Simple component instantiation
add1: add
   generic map(size => 4)
   port map(a => ain,
      b => bin,
      result => q);

-- Other code
```

Example: VHDL 1993

```
architecture struct of hier_add is
component add
   generic (size : natural);
   port (a : in bit_vector(3 downto 0);
         b : in bit_vector(3 downto 0);
         result : out bit_vector(3 downto 0));
end component;

begin
-- Simple component instantiation
add1: component add -- Component keyword new in 1993
   generic map(size => 4)
   port map(a => ain,
      b => bin,
      result => q);

-- Other code
```

Note: If no entity is found in the working library named the same as the declared component, all instances of the declared component are mapped to a black box and the error message “Unbound component mapped to black box” is issued.

VHDL Selected Name Support

Selected Name Support (SNS) is provided in VHDL for constants, operators, and functions in library packages. SNS eliminates ambiguity in a design referencing elements with the same names, but that have unique functionality when the design uses the elements with the same name defined in multiple packages. By specifying the library, package, and specific element (constant, operator, or function), SNS designates the specific constant, operator, or function used. This section discusses all facets of SNS. Complete VHDL examples are included to assist you in understanding how to use SNS effectively.

Constants

SNS lets you designate the constant to use from multiple library packages. To incorporate a constant into a design, specify the library, package, and constant. Using this feature eliminates ambiguity when multiple library packages have identical names for constants and are used in an entity-architecture pair.

The following example has two library packages available to the design constants. Each library package has a constant defined by the name C1 and each has a different value. SNS is used to specify the constant (see work.PACKAGE.C1 in the constants example below).

```
-- CONSTANTS PACKAGE1
library IEEE;
use IEEE.std_logic_1164.all;
package PACKAGE1 is
   constant C1: std_logic_vector := "10001010";
end PACKAGE1;

-- CONSTANTS PACKAGE2
library IEEE;
use IEEE.std_logic_1164.all;
package PACKAGE2 is
   constant C1: std_logic_vector := "10110110";
end PACKAGE2;

-- CONSTANTS EXAMPLE
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity CONSTANTS is
   generic (num_bits : integer := 8);
      port (a,b: in std_logic_vector (num_bits -1 downto 0);
            out1, out2: out std_logic_vector (num_bits -1 downto 0)
            );
end CONSTANTS;
```

```
architecture RTL of CONSTANTS is
begin
   out1 <= a - work.PACKAGE1.Cl; -Example of specifying SNS
   out2 <= b - work.PACKAGE2.C1; -Example of specifying SNS
end RTL;
```

In the above design, outputs out1 and out2 use two C1 constants from two different packages. Although each output uses a constant named C1, the constants are not equivalent. For out1, the constant C1 is from PACKAGE1. For out2, the constant C1 is from PACKAGE2. For example:

`out1 <= a - work.PACKAGE1.Cl;` is equivalent to `out1 <= a - "10001010";`

whereas:

`out2 <= b - work.PACKAGE2.Cl;` is equivalent to `out2 <= b - "10110110";`

The constants have different values in different packages. SNS specifies the package and eliminates ambiguity within the design.

Functions and Operators

Functions and operators in VHDL library packages customarily have overlapping naming conventions. For example, the add operator in the IEEE standard library exists in both the std_logic_signed and std_logic_unsigned packages. Depending upon the add operator used, different values result. Defining only one of the IEEE library packages is a straightforward solution to eliminate ambiguity, but applying this solution is not always possible. A design requiring both std_logic_signed and std_logic_unsigned package elements must use SNS to eliminate ambiguity.

Functions

In the following example, multiple IEEE packages are declared in a 256x8 RAM design. Both std_logic_signed and std_logic_unsigned packages are included. In the RAM definition, the signal address_in is converted from type std_logic_vector to type integer using the CONV_INTEGER function, but which CONV_INTEGER function will be called? SNS determines the function to use. The RAM definition clearly declares the std_logic_unsigned package as the source for the CONV_INTEGER function.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;
use IEEE.numeric_std.all;

entity FUNCTIONS is
   port (address : in std_logic_vector(7 downto 0);
         data_in : in std_logic_vector(7 downto 0);
         data_out : out std_logic_vector(7 downto 0);
         we : in std_logic;
         clk : in std_logic);

end FUNCTIONS;

architecture RTL of FUNCTIONS is
type mem_type is array (255 downto 0) of
   std_logic_vector (7 downto 0);
signal mem: mem_type;
signal address_in: std_logic_vector(7 downto 0);
begin
data_out <= mem(IEEE.std_logic_unsigned.CONV_INTEGER(address_in));
   process (clk)
   begin
      if rising_edge(clk) then
         if (we = '1') then
            mem(IEEE.std_logic_unsigned.CONV_INTEGER(address_in))
               <= data_in;
         end if;
      address_in <= address;
      end if;
   end process;
end RTL;
```

Operators

In this example, comparator functions from the IEEE std_logic_signed and std_logic_unsigned library packages are used. Depending upon the comparator called, a signed or an unsigned comparison results. In the assigned outputs below, the op1 and op2 functions show the valid SNS syntax for operator implementation.

```
library IEEE;
use IEEE.std_logic_1164.std_logic_vector;
use IEEE.std_logic_signed.">";
use IEEE.std_logic_unsigned.">";
```

```
entity OPERATORS is
   port (in1 :std_logic_vector(1 to 4);
         in2 :std_logic_vector(1 to 4);
         in3 :std_logic_vector(1 to 4);
         in4 :std_logic_vector(1 to 4);
         op1,op2 :out boolean);
end OPERATORS;

architecture RTL of OPERATORS is
begin
   process(in1,in2,in3,in4)
   begin

      --Example of specifying SNS
      op1 <= IEEE.std_logic_signed.">"(in1,in2);

      --Example of specifying SNS
      op2 <= IEEE.std_logic_unsigned.">"(in3,in4);
   end process;
end RTL;
```

User-defined Function Support

SNS is not limited to predefined standard IEEE packages and packages supported by the tool; SNS also supports user-defined packages. You can create library packages that access constants, operators, and functions in the same manner as the packages supported by IEEE or synthesis.

The following example incorporates two user-defined packages in the design. Each package includes a function named func. In PACKAGE1, func is an XOR gate, whereas in PACKAGE2, func is an AND gate. Depending on the package called, func results in either an XOR or an AND gate. The function call uses SNS to distinguish the function that is called.

```
-- USER DEFINED PACKAGE1
library IEEE;
use IEEE.std_logic_1164.all;
package PACKAGE1 is
   function func(a,b: in std_logic) return std_logic;
end PACKAGE1;

package body PACKAGE1 is
   function func(a,b: in std_logic) return std_logic is
begin
   return(a xor b);
end func;
end PACKAGE1;

-- USER DEFINED PACKAGE2
library IEEE;
use IEEE.std_logic_1164.all;

package PACKAGE2 is
   function func(a,b: in std_logic) return std_logic;
end PACKAGE2;

package body PACKAGE2 is
   function func(a,b: in std_logic) return std_logic is
begin
   return(a and b);
end func;
end PACKAGE2;

-- USER DEFINED FUNCTION EXAMPLE
library IEEE;
use IEEE.std_logic_1164.all;

entity USER_DEFINED_FUNCTION is
   port (in0: in std_logic;
         in1: in std_logic;
         out0: out std_logic;
         out1: out std_logic);
end USER_DEFINED_FUNCTION;

architecture RTL of USER_DEFINED_FUNCTION is
begin
   out0 <= work.PACKAGE1.func(in0, in1);
   out1 <= work.PACKAGE2.func(in0, in1);
end RTL;
```

Demand Loading

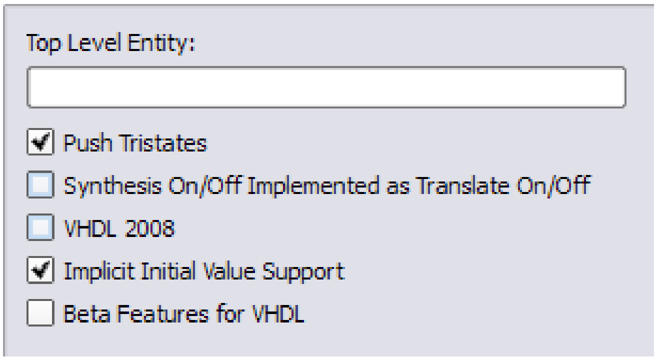
In the previous section, the user-defined function example successfully uses SNS to determine the func function to implement. However, neither PACKAGE1 nor PACKAGE2 was declared as a use package clause (for example, work.PACKAGE1.all;). How could func have been executed without a use package declaration? A feature of SNS is demand loading: this loads the necessary package without explicit use declarations. Demand loading lets you create designs using SNS without use package declarations, which supports all necessary constants, operators, and functions.

VHDL Implicit Data-type Defaults

Type default propagation avoids potential simulation mismatches that are the result of differences in behavior with how initial values for registers are treated in the synthesis tools and how they are treated in the simulation tools.

With implicit data-type defaults, when there is no explicit initial-value declaration for a signal being registered, the VHDL compiler passes an init value through a syn_init property to the mapper, and the mapper then propagates the value to the respective register. Compiler requirements are based on specific data types. These requirements can be broadly grouped based on the different data types available in the VHDL language.

Implicit data-type defaults are enabled on the VHDL panel of the Implementation Options dialog box or through a -supporttypedflt argument to a set_option command.


Top Level Entity:

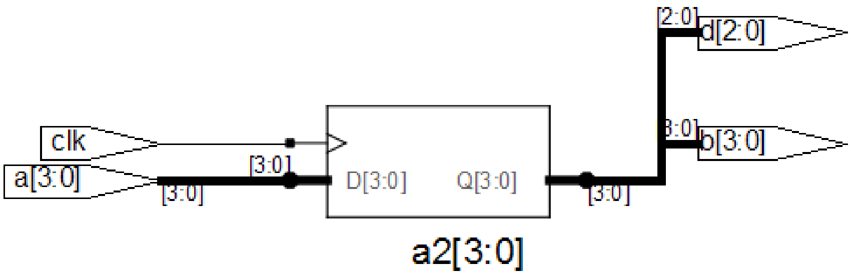
- [✓] Push Tristates
- [ ] Synthesis On/Off Implemented as Translate On/Off
- [ ] VHDL 2008
- [✓] Implicit Initial Value Support
- [ ] Beta Features for VHDL

To illustrate the use of implicit data-type defaults, consider the following example.

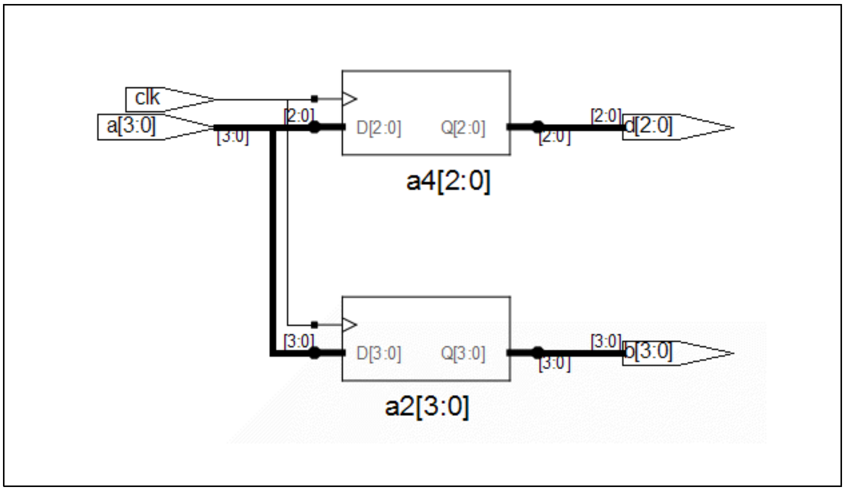
```
library ieee;
use ieee.std_logic_1164.all;

entity top is
   port (clk:in std_logic;
         a : in integer range 1 to 8;
         b : out integer range 1 to 8;
         d : out positive range 1 to 7);
end entity top;
```

```
architecture rtl of top is
signal a1,a2 : integer range 1 to 8;
signal a3,a4 : positive range 1 to 7;
begin
a1 <= a;
a3 <= a;
b <= a2;
d <= a4;
   process(clk)
   begin
      if (rising_edge(clk))then
         a2 <= a1;
         a4 <= a3;
      end if;
   end process;
end rtl;
```

In the above example, two signals (a2 and a4) with different type default values are registered. Without implicit data-type defaults, if the values of the signals being registered are not the same, the compiler merges the redundant logic into a single register as shown in the figure below.



Enabling implicit data-type defaults prevents certain compiler and mapper optimizations to preserve both registers as shown in the following figure.



Example – Impact on Integer Ranges

The default value for the integer type when a range is specified is the minimum value of the range specified, and size is the upper limit of that range. With implicit data-type defaults, the compiler is required to propagate the minimum value of the range as the init value to the mapper. Consider the following example:

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

entity top is
   port (clk,set:in std_logic;
         a : in integer range -6 to 8;
         b : out integer range -6 to 8);
end entity top;

architecture rtl of top is
signal a1,a2: integer range -6 to 8;
begin
a1 <= a ;
   process(clk,set)
   begin
      if (rising_edge(clk))then
         if set = '1' then
            a2 <= a;
         else
            a2 <= a1;
         end if;
      end if;
   end process;
b <= a2;
end rtl;
```

In the example,

```
signal a1, a2: integer range -6 to 8;
```

the default value is -6 (FA in 2’s complement) and the range is -6 to 8. With a total of 15 values, the size of the range can be represented in four bits.

Example – Impact on RAM Inferencing

When inferencing a RAM with implicit data-type defaults, the compiler propagates the type default values as init values for each RAM location. The mapper must check if the block RAMs of the selected technology support initial values and then determine if the compiler-propagated init values are to be considered. If the mapper chooses to ignore the init values, a warning is issued stating that the init values are being ignored. Consider the following VHDL design:

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

entity top is
   port (clk : in std_logic;
         addr : in std_logic_vector (6 downto 0);
         din : in positive;
         wen : in std_logic;
         dout : out positive);
end top;

architecture behavioral of top is
-- RAM
type tram is array(0 to 127) of positive;
signal ram : tram ;
begin
-- Contents of RAM has initial value = 1
   process (clk)
   begin
      if clk'event and clk = '1' then
         if wen = '1' then
            ram(conv_integer(addr)) <= din_sig;
         end if;
      dout <= ram(conv_integer(addr));
      end if;
   end process;
end behavioral;
```

In the above example:

- The type of signal a1 is bit_vector
- The default value for type integer is 1 when no range is specified

Accordingly, a value of x00000001 is propagated by the compiler to the mapper with a syn_init property.

VHDL Synthesis Guidelines

This section provides guidelines for synthesis using VHDL. The following topics are covered:

General Synthesis Guidelines

Some general guidelines are presented here to help you synthesize your VHDL design.

- Top-level entity and architecture. The tool chooses the top-level entity and architecture – the last architecture for the last entity in the last file compiled. Entity selection can be overridden from the VHDL panel of the Implementation Options dialog box. Files are compiled in the order they appear – from top to bottom in the design source files list.
- Simulate your design before synthesis because it exposes logic errors. Logic errors that are not caught are passed through the tool, and the synthesized results will contain the same logic errors.

- Simulate your design after placement and routing. Have the place-and-route tool generate a post placement and routing (timing-accurate) simulation netlist, and do a final simulation before programming your devices.
- Avoid asynchronous state machines. To use the tool for asynchronous state machines, make a netlist of technology primitives from your target library.
- For modeling level-sensitive latches, it is simplest to use concurrent signal assignments.

VHDL Language Guidelines

This section discusses VHDL language guidelines.

Processes

- A process must have either a sensitivity list or one wait statement.
- Each sequential process can be triggered from exactly one clock and only one edge of clock (and optional sets and resets).
- Avoid combinational loops in processes. Make sure all signals assigned in a combinational process are explicitly assigned values every time the process executes; otherwise, the tool needs to insert level-sensitive latches in your design to hold the last value for the paths that do not assign values. This might represent a mistake on your part, so synthesis issues a warning message that level-sensitive latches are being inserted into your design. You will get an warning message if you have combinational loops in your design that are not recognized as level-sensitive latches (for example, if you have an asynchronous state machine).

Assignments

- Assigning an 'X' or '-' to a signal is interpreted as a “don't care”, so the tool creates the hardware that is the most efficient design.

Data Types

- Integers are 32-bit quantities. If you declare a port as an integer data type, specify a range (for example, my_input: in integer range 0 to 7). Otherwise, your synthesis result file will contain a 32-bit port.
- Enumeration types are represented as a vector of bits. The encoding can be sequential, gray, or one hot. You can manually choose the encoding for ports with an enumeration type.

Model Template

You can place any number of concurrent statements (signal assignments, processes, component instantiations, and generate statements) in your architecture body as shown in the following example. The order of these statements within the architecture is not significant, as all can execute concurrently.

- The statements between the begin and the end in a process execute sequentially, in the order you type them from top to bottom.
- You can add comments in VHDL by proceeding your comment text with two dashes "--". Any text from the dashes to the end of the line is treated as a comment, and ignored by the tool.

```
-- List libraries/packages that contain definitions you use
library <library_name>;
use <library_name>.<package_name>.all;

-- The entity describes the interface for your design.
entity <entity_name> is
   generic (<define_interface_constants_here>);
      port (<port_list_information_goes_here>);
end <entity_name>;

-- The architecture describes the functionality (implementation)
-- of your design
architecture <architecture_name> of <entity_name> is

-- Architecture declaration region.
-- Declare internal signals, data types, and subprograms here
-- If you will create hierarchy by instantiating a
-- component (which is just another architecture), then
-- declare its interface here with a component declaration;
component <entity_name_instantiated_below>
   port (<port_list_information_as_defined_in_the_entity>);
end component;

begin -- Architecture body, describes functionality

-- Use concurrent statements here to describe the functionality
-- of your design. The most common concurrent statements are the
-- concurrent signal assignment, process, and component
-- instantiation.

-- Concurrent signal assignment (simple form):
<result_signal_name> <= <expression>;

-- Process:
process <sensitivity list>)
-- Declare local variables, data types,
-- and other local declarations here
begin
-- Sequential statements go here, including:
   -- signal and variable assignments
   -- if and case statements
   -- while and for loops
   -- function and procedure calls
end process;

-- Component instantiation
<instance_name> : <entity_name>
   generic map (<override values here >)
   port map (<port list>);
end <architecture_name>;
```

Constraint Files for VHDL Designs

In previous versions of the software, all object names output by the compiler were converted to lower case. This means that any constraints files created by dragging from the schematic view or through the constraints file GUI contained object names using only lower case. Case is preserved on design object names. If you use mixed-case names in your VHDL source, for constraints to be applied correctly, you must manually update any older constraint files or re-create constraints in the constraint editor.

Creating Flip-flops and Registers Using VHDL Processes

It is easy to create flip-flops and registers using a process in your VHDL design.

process Template

```
process (<sensitivity list>)
begin
   <sequential statement(s)>
end;
```

To create a flip-flop:

1. List your clock signal in the sensitivity list. Recall that if the value of any signal listed in the sensitivity list changes, the process is triggered, and executes. For example,

   ```
   process (clk)
   ```

2. Check for rising_edge or falling_edge as the first statement inside the process. For example,

   ```
   process (clk)
   begin
      if rising_edge(clk) then
         <sequential statement(s)>
   ```

   or

   ```
   process (clk)
   begin
      if falling_edge(clk) then
         <sequential statement(s)>
   ```

   Alternatively, you could use an if clk'event and clk = '1' then statement to test for a rising edge (or if clk'event and clk = '0' then for a falling edge). Although these statements work, for clarity and consistency, use the rising_edge and falling_edge functions from the VHDL 1993 standard.

3. Set your flip-flop output to a value, with no delay, if the clock edge occurred. For example, q <= d ;.

Complete Example

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

entity dff_or is
   port (a, b, clk: in std_logic;
         q: out std_logic);
end dff_or;

architecture sensitivity_list of dff_or is
begin
   process (clk) -- Clock name is in sensitivity list
   begin
      if rising_edge(clk) then
         q <= a or b;
      end if;
   end process;
end sensitivity_list;
```

In this example, if clk has an event on it, the process is triggered and starts executing. The first statement (the if statement) then checks to see if a rising edge has occurred for clk. If the if statement evaluates to true, there was a rising edge on clk and the q output is set to the value of a or b. If the clk changes from 1 to 0, the process is triggered and the if statement executes, but it evaluates to false and the q output is not updated. This is the functionality of a flip-flop, and synthesis correctly recognizes it as such and connects the result of the a or b expression to the data input of a D-type flip-flop and the q signal to the q output of the flip-flop.

Note: The signals you set inside the process will drive the data inputs of D-type flip-flops.

Clock Edges

There are many ways to correctly represent clock edges within a process including those shown here.

The typical rising clock edge representation is:

```
rising_edge(clk)
```

Other supported rising clock edge representations are:

```
clk = '1' and clk'event
clk'last_value = '0' and clk'event
clk'event and clk /= '0'
```

The typical falling clock edge representation is:

```
falling_edge(clk)
```

Other supported falling clock edge representations are:

```
clk = '0' and clk'event
clk'last_value = '1' and clk'event
clk'event and clk /= '1'
```

Incorrect or Unsupported Representations for Clock Edges

Rising clock edge:

```
clk = '1'
clk and clk'event -- Because clk is not a Boolean
```

Falling clock edge:

```
clk = '0'
not clk and clk'event -- Because clk is not a Boolean
```

Defining an Event Outside a Process

The 'event attribute can be used outside of a process block. For example, the process block

```
process (clk,d)
begin
   if (clk='1' and clk'event) then
      q <= d;
   end if;
end process;
```

can be replaced by including the following line outside of the process statement:

```
q <= d when (clk='1' and clk'event);
```

Using a WAIT Statement Inside a Process

The tool supports a wait statement inside a process to create flip-flops, instead of using a sensitivity list.

Example

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

entity dff_or is
   port (a, b, clk: in std_logic;
         q: out std_logic);
end dff_or;

architecture wait_statement of dff_or is
begin
   process -- Notice the absence of a sensitivity list.
   begin
-- The process waits here until the condition becomes true
      wait until rising_edge(clk);
         q <= a or b;
   end process;
end wait_statement;
```

Rules for Using wait Statements Inside a Process

- It is illegal in VHDL to have a process with a wait statement and a sensitivity list.
- The wait statement must either be the first or the last statement of the process.

Clock Edge Representation in wait Statements

The typical rising clock edge representation is:

```
wait until rising_edge(clk);
```

Other supported rising clock edge representations are:

```
wait until clk = '1' and clk'event
wait until clk'last_value = '0' and clk'event
wait until clk'event and clk /= '0'
```

The typical falling clock edge representation is:

```
wait until falling_edge(clk)
```

Other supported falling clock edge representations are:

```
wait until clk = '0' and clk'event
wait until clk'last_value = '1' and clk'event
wait until clk'event and clk /= '1'
```

Level-sensitive Latches Using Concurrent Signal Assignments

To model level-sensitive latches in VHDL, use a concurrent signal assignment statement with the conditional signal assignment form (also known as when-else).

Syntax

signal **<=** *value1* **when** *condition1* **else**
value2 **when** *condition2* **else**
valueN-1 **when** *conditionN-1* **else**
*valueN***;**

Example

In VHDL, you are not allowed to read the value of ports of mode out inside of an architecture that it was declared for. Ports of mode buffer can be read from and written to, but must have no more than one driver for the port in the architecture. In the following port statement example, q is defined as mode buffer.

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

entity latchor1 is
   port (a, b, clk : in std_logic;
-- q has mode buffer so it can be read inside architecture
         q: buffer std_logic);
end latchor1;

architecture behave of latchor1 is
begin
   q <= a or b when clk = '1' else q;
end behave;
```

Whenever clk, a, or b changes, the expression on the right side re-evaluates. If clk becomes true (active, logic 1), the value of a or b is assigned to the q output. When the clk changes and becomes false (deactivated), q is assigned to q (holds the last value of q). If a or b changes, and clk is already active, the new value of a or b is assigned to q.

Level-sensitive Latches Using VHDL Processes

Although it is simpler to specify level-sensitive latches using concurrent signal assignment statements, you can create level-sensitive latches with VHDL processes. Follow the guidelines given here for the sensitivity list and assignments.

process Template

```
process (<sensitivity list>)
begin
   <sequential statement(s)>
end process;
```

Sensitivity List

The sensitivity list specifies the clock signal, and the signals that feed into the data input of the level-sensitive latch. The sensitivity list must be located immediately after the process keyword.

Syntax

process (*clock_name***,** *signal1***,** *signal2***, ...)**

Example

```
process (clk, data)
```

process Template for a Level-sensitive Latch

```
process (<clock, data_signals ... > ...)
begin
   if (<clock> = <active_value>)
      <signals> <= <expression involving data signals>;
   end if;
end process ;
```

All data signals assigned in this manner become logic into data inputs of level-sensitive latches.

Whenever level-sensitive latches are generated from a process, the tool issues a warning message so that you can verify if level-sensitive latches are really what you intended. Often a thorough simulation of your architecture will reveal mistakes in coding style that can cause the creation of level-sensitive latches during synthesis.

Example: Creating Level-sensitive Latches that You Want

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

entity latchor2 is
   port (a, b, clk : in std_logic;
         q: out std_logic);
end latchor2;

architecture behave of latchor2 is
begin
   process (clk, a, b)
   begin
      if clk = '1' then
         q <= a or b;
      end if;
   end process;
end behave;
```

If there is an event (change in value) on either clk, a or b, and clk is a logic 1, set q to a or b.

What to do when clk is a logic 0 is not specified (there is no else), so when clk is a logic zero, the last value assigned is maintained (there is an implicit q=q). The tool correctly recognizes this as a level-sensitive latch, and creates a level-sensitive latch in your design. It will issue a warning message when you compile this architecture, but after examination, this warning message can safely be ignored.

Example: Creating Unwanted Level-sensitive Latches

This design demonstrates the level-sensitive latch warning caused by a missed assignment in the when two => case. The message generated is:

```
"Latch generated from process for signal odd, probably caused by a
missing assignment in an if or case statement".
```

This information will help you find a functional error even before simulation.

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

entity mistake is
   port (inp: in std_logic_vector (1 downto 0);
         outp: out std_logic_vector (3 downto 0);
         even, odd: out std_logic);
end mistake;

architecture behave of mistake is
   constant zero: std_logic_vector (1 downto 0):= "00";
   constant one: std_logic_vector (1 downto 0):= "01";
   constant two: std_logic_vector (1 downto 0):= "10";
   constant three: std_logic_vector (1 downto 0):= "11";
begin
   process (inp)
   begin
      case inp is
         when zero =>
            outp <= "0001";
            even <= '1';
            odd <= '0';
         when one =>
            outp <= "0010";
            even <= '0';
            odd <= '1';
         when two =>
            outp <= "0100";
            even <= '1';
-- Notice that assignment to odd is mistakenly commented out next.

            -- odd <= '0';
         when three =>
            outp <= "1000";
            even <= '0';
            odd <= '1';
      end case;
   end process;
end behave;
```

Signed mod Support for Constant Operands

The tool supports signed mod for constant operands. Additionally, division operators (/, rem, mod), where the operands are compile-time constants and greater than 32 bits, are supported.

Example of using signed mod operator with constant operands

```
LIBRARY ieee; USE ieee.std_logic_1164.ALL;
LIBRARY ieee; USE ieee.numeric_std.all;

ENTITY divmod IS
   PORT (tstvec: out signed(7 DOWNTO 0));
END divmod;

ARCHITECTURE structure OF divmod IS
   CONSTANT NOMINATOR   : signed(7 DOWNTO 0) := "10000001";
   CONSTANT DENOMINATOR : signed(7 DOWNTO 0) := "00000011";
   CONSTANT RESULT      : signed(7 DOWNTO 0) := NOMINATOR mod
      DENOMINATOR;
BEGIN
   tstvec <= result;

END ARCHITECTURE structure;
```

Example of a signed division with a constant right operand.

```
LIBRARY ieee; USE ieee.std_logic_1164.ALL;
LIBRARY ieee; USE ieee.numeric_std.all;

ENTITY divmod IS
   PORT (tstvec: out signed(7 DOWNTO 0));
END divmod;

ARCHITECTURE structure OF divmod IS
   CONSTANT NOMINATOR   : signed(7 DOWNTO 0) := "11111001";
   CONSTANT DENOMINATOR : signed(7 DOWNTO 0) := "00000011";
   CONSTANT RESULT      : signed(7 DOWNTO 0) := NOMINATOR /
      DENOMINATOR;
BEGIN
   tstvec <= result;

END ARCHITECTURE structure;
```

An example where the operands are greater than 32 bits

```
LIBRARY ieee; USE ieee.std_logic_1164.ALL;
LIBRARY ieee; USE ieee.numeric_std.all;

ENTITY divmod IS
   PORT (tstvec: out unsigned(33 DOWNTO 0));
END divmod;

ARCHITECTURE structure OF divmod IS
   CONSTANT NOMINATOR   : unsigned(33 DOWNTO 0) :=
   "1000000000000000000000000000000000";
   CONSTANT DENOMINATOR : unsigned(32 DOWNTO 0) :=
   "000000000000000000000000000000011";
   CONSTANT RESULT      : unsigned(33 DOWNTO 0) := NOMINATOR /
      DENOMINATOR;
BEGIN
   tstvec <= result;
END ARCHITECTURE structure;
```

Sets and Resets

This section describes VHDL sets and resets, both asynchronous and synchronous. A set signal is an input to a flip-flop that, when activated, sets the state of the flip-flop to a logic one. A reset signal is an input to a flip-flop that, when activated, sets the state of the flip-flop to a logic zero.

The topics include:

- Asynchronous Sets and Resets, on page 257
- Synchronous Sets and Resets, on page 258

Asynchronous Sets and Resets

By definition, asynchronous sets and resets are independent of the clock and do not require an active clock edge. Therefore, you must include the set and reset signals in the sensitivity list of your process so they trigger the process to execute.

Sensitivity List

The sensitivity list is a list of signals (including ports) that, when there is an event (change in value), triggers the process to execute.

Syntax

process (*clk_name***,** *set_signal_name***,** *reset_signal_name***)**

The signals are listed in any order, separated by commas.

Example: process Template with Asynchronous, Active-high reset, set

```
process (clk, reset, set)
begin
   if reset = '1' then
-- Reset the outputs to zero.
   elsif set = '1' then
-- Set the outputs to one.
   elsif rising_edge(clk) then -- Rising clock edge clock
-- Clocked logic goes here.
   end if;
end process;
```

Example: D Flip-flop with Asynchronous, Active-high reset, set

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

entity dff1 is
   port (data, clk, reset, set : in std_logic;
         qrs: out std_logic);
end dff1;

architecture async_set_reset of dff1 is
begin
   setreset: process(clk, reset, set)
   begin
      if reset = '1' then
         qrs <= '0';
      elsif set = '1' then
         qrs <= '1';
      elsif rising_edge(clk) then
         qrs <= data;
      end if;
   end process setreset;
end async_set_reset;
```

Synchronous Sets and Resets

Synchronous sets and resets set flip-flop outputs to logic '1' or '0' respectively on an active clock edge.

Do not list the set and reset signal names in the sensitivity list of a process so they will not trigger the process to execute upon changing. Instead, trigger the process when the clock signal changes, and check the reset and set as the first statements.

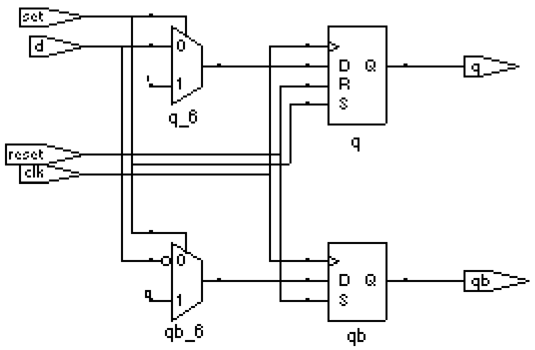
RTL View Primitives

The VHDL compiler can detect and extract the following flip-flops with synchronous sets and resets and display them in the schematic view:

- sdffr – f lip-flop with synchronous reset
- sdffs – flip-flop with synchronous set
- sdffrs – flip-flop with both synchronous set and reset
- sdffpat – vectored flip-flop with synchronous set/reset pattern

- sdffre – enabled flip-flop with synchronous reset
- sdffse – enabled flip-flop with synchronous set
- sdffpate – enabled, vectored flip-flop with synchronous set/reset pattern

You can check the name (type) of any primitive by placing the mouse pointer over it in the schematic view: a tooltip displays the name.



Sensitivity List

The sensitivity list is a list of signals (including ports) that, when there is an event (change in value), triggers the process to execute.

Syntax

process (*clk_signal_name***)**

Example: process Template with Synchronous, Active-high reset, set

```
process(clk)
begin
   if rising_edge(clk) then
      if reset = '1' then
         -- Set the outputs to '0'.
      elsif set = '1' then
         -- Set the outputs to '1'.
      else
         -- Clocked logic goes here.
      end if;
   end if;
end process;
```

Example: D Flip-flop with Synchronous, Active-high reset, set

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

entity dff2 is
   port (data, clk, reset, set : in std_logic;
         qrs: out std_logic);
end dff2;

architecture sync_set_reset of dff2 is
begin
   setreset: process (clk)
   begin
      if rising_edge(clk) then
         if reset = '1' then
            qrs <= '0';
         elsif set = '1' then
            qrs <= '1';
         else
            qrs <= data;
         end if;
      end if;
   end process setreset;
end sync_set_reset;
```

VHDL State Machines

This section describes VHDL state machines: guidelines for using them, defining state values with enumerated types, and dealing with asynchrony. The topics include:

State Machine Guidelines

A finite state machine (FSM) is hardware that advances from state to state at a clock edge.

The tool works best with synchronous state machines. You typically write a fully synchronous design, avoiding asynchronous paths such as paths through the asynchronous reset of a register. See Asynchronous State Machines in VHDL, on page 267 for information about asynchronous state machines.

The following are guidelines for coding FSMs:

- The state machine must have a synchronous or asynchronous reset, to be inferred. State machines must have an asynchronous or synchronous reset to set the hardware to a valid state after power-up, and to reset your hardware during operation (asynchronous resets are available freely in most FPGA architectures).
- The tool does not infer implicit state machines that are created using multiple wait statements in a process.
- Separate the sequential process statements from the combinational ones. Besides making it easier to read, it makes what is being registered very obvious. It also gives better control over the type of register element used.
- Represent states with defined labels or enumerated types.

- Use a case statement in a process to check the current state at the clock edge, advance to the next state, and set the output values. You can also use if-then-else statements.
- Assign default values to outputs derived from the FSM before the case statement. This helps prevent the generation of unwanted latches and makes it easier to read because there is less clutter from rarely used signals.
- If you do not have case statements for all possible combinations of the selector, use a when others assignment as the last assignment in your case statement and set the state vector to some valid state. If your state vector is not an enumerated type, set the value to X. Assign the state to X in the default clause of the case statement, to avoid mismatches between pre- and post-synthesis simulations. See Example: Default Assignment, on page 265.
- If a state machine defined in the code feeds sequential elements in a different clock domain, some encoding values can cause metastability. By default, the tool chooses the optimal encoding value based on the number of states in the state machine. This can introduce additional decode logic that could cause metastability when it feeds sequential elements in a different clock domain. To prevent this instability, use syn_encoding = "original" to guide synthesis for these cases.
- Override the default encoding style with the syn_encoding attribute. for a list of default encoding is determined by the number of state. See syn_encoding Values, on page 67 for a list of default and other encodings. When you specify a particular encoding style with syn_encoding, that value is used during the mapping stage to determine encoding style.

  ```
  attribute syn_encoding : string;
  attribute syn_encoding of <typename> : type is "sequential";
  ```

  See the *Attribute Reference* manual, for details about the syntax and values.

  One-hot implementations are not always the best choice for state machines, even in FPGAs and CPLDs. For example, one-hot state machines might result in higher speeds in CPLDs, but could cause fitting problems because of the larger number of global signals. An example of an FPGA with ineffective one-hot implementation is a state machine that drives a large decoder, generating many output signals. In a 16-state state machine, for example, the output decoder logic might reference sixteen signals in a one-hot implementation, but only four signals in an encoded representation.

In general, do not use the directive syn_enum_encoding to set the encoding style. Use syn_encoding instead. The value of syn_enum_encoding is used by the compiler to interpret the enumerated data types but is ignored by the mapper when the state machine is actually implemented.

The directive syn_enum_encoding affects the final circuit only when you have turned off the FSM Compiler. Therefore, if you are not using FSM Compiler or the syn_state_machine attribute, which use syn_encoding, you can use syn_enum_encoding to set the encoding style. See the *Attribute Reference* manual, for details about the syntax and values.

- Implement user-defined enumeration encoding, beyond the one-hot, gray, and sequential styles. Use the directive syn_enum_encoding to set the state encoding. See Example: FSM User-Defined Encoding, on page 264.

Example: FSM Coding Style

```
architecture behave of test is
   type state_value is (deflt, idle, read, write);
   signal state, next_state: state_value;
begin
-- Figure out the next state
   process (clk, rst)
   begin
      if rst = '0' then
         state <= idle;
      elsif rising_edge(clk) then
         state <= next_state;
      end if;
   end process;

   process (state, enable, data_in)
   begin
      data_out <= '0';
      -- Catch missing assignments to next_state
      next_state <= idle;
      state0 <= '0';
      state1 <= '0';
      state2 <= '0';
      case state is
         when idle =>
            if enable = '1' then
               state0 <= '1' ;data_out <= data_in(0);
               next_state <= read;
            else next_state <= idle;
            end if;
         when read =>
            if enable = '1' then
               state1 <= '1'; data_out <= data_in(1);
               next_state <= write;
            else next_state <= read;
            end if;
         when deflt =>
            if enable = '1' then
               state2 <= '1' ;data_out <= data_in(2);
               next_state <= idle;
            else next_state <= write;
            end if;
         when others => next_state <= deflt;
      end case;
   end process;
end behave;
```

Example: FSM 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;
```

Example: Default Assignment

The second others keyword in the following example pads (covers) all the bits. In this way, you need not remember the exact number of X's needed for the state variable or output signal.

```
when others =>
   state := (others => 'X');
```

Assigning X to the state variable (a "don't care" for synthesis) tells the tool that you have specified all the used states in your case statement, and any unnecessary decoding and gates related to other cases can therefore be removed. You do not have to add any special, non-VHDL directives.

If you set the state to a used state for the when others case (for example: when others => state <= delft), the tool generates the same logic as if you assign X, but there will be pre- and post-synthesis simulation mismatches until you reset the state machine. These mismatches occur because all inputs are unknown at start up on the simulator. You therefore go immediately into the when others case, which sets the state variable to state1. When you power up the hardware, it can be in a used state, such as state2, and then advance to a state other than state1. Post-synthesis simulation behaves more like hardware with respect to initialization.

Using Enumerated Types for State Values

Generally, you represent states in VHDL with a user-defined enumerated type.

Syntax

type *type_name* **is (***state1_name***,** *state2_name***, ... ,** *stateN_name***);**

Example

```
type states is (st1, st2, st3, st4, st5, st6, st7, st8);
begin
-- The statement region of a process or subprogram.
next_state := st2;
-- Setting the next state to st2
```

Simulation Tips When Using Enumerated Types

You want initialization in simulation to mimic the behavior of hardware when it powers up. Therefore, do not initialize your state machine to a known state during simulation, because the hardware will not be in a known state when it powers up.

Creating an Extra Initialization State

If you use an enumerated type for your state vector, create an extra initialization state in your type definition (for example, stateX), and place it first in the list, as shown in the example below.

```
type state is (stateX, state1, state2, state3, state4);
```

In VHDL, the default initial value for an enumerated type is the leftmost value in the type definition (in this example, stateX). When you begin the simulation, you will be in this initial (simulation only) state.

Detecting Reset Problems

In your state machine case statement, create an entry for staying in stateX when you get in stateX. For example:

```
when stateX => next_state := stateX;
```

Look for your design entering stateX. This means that your design is not resetting properly.

Note: The tool does not create hardware to represent this initialization state (stateX). It is removed during optimization.

Detecting Forgotten Assignment to the Next State

Assign your next state value to stateX immediately before your state machine case statement.

Example

```
next_state := stateX;
case (current_state) is
...
   when state3 =>
      if (foo = '1') then
         next_state := state2;
      end if;
...
end case;
```

Asynchronous State Machines in VHDL

Avoid defining asynchronous state machines in VHDL. An asynchronous state machine has states, but no clearly defined clock, and has combinational loops. However, if you must use asynchronous state machines, you can do one of the following:

- Create a netlist of the technology primitives from the target library for your technology vendor. Any instantiated primitives that are left in the netlist are not removed during optimization.
- Use a schematic editor for the asynchronous state machine part of your design.

Do not use the tool to design asynchronous state machines; the tool might remove your hazard-suppressing logic when it performs logic optimization, causing your asynchronous state machine to work incorrectly.

The tool displays a “found combinational loop” warning message for an asynchronous FSM when it detects combinational loops in continuous assignment statements, processes and built-in gate-primitive logic.

Asynchronous State Machines that Generate Error Messages

In this example, both async1 and async2 will generate combinational loop errors, because of the recursive definition for output.

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

entity async is
-- output is a buffer mode so that it can be read
   port (output : buffer std_logic;
         g, d : in std_logic);
end async;

-- Asynchronous FSM from concurrent assignment statement
architecture async1 of async is
begin
   -- Combinational loop error, due to recursive output definition.
   output <= ((((g and d) or (not g)) and output) or d) and
      output);
end async1;

-- Asynchronous FSM created within a process
architecture async2 of async is
begin
   process(g, d, output)
   begin
-- Combinational loop error, due to recursive output definition.
      output <= ((((g and d) or (not g)) and output) or d) and
      output);
   end process;
end async2;
```

Hierarchical Design Creation in VHDL

Creating hierarchy is similar to creating a schematic. You place available parts from a library onto a schematic sheet and connect them.

To create a hierarchical design in VHDL, you instantiate one design unit inside of another. In VHDL, the design units you instantiate are called components. Before you can instantiate a component, you must declare it (step 2, below).

The basic steps for creating a hierarchical VHDL design are:

1. Write the design units (entities and architectures) for the parts you wish to instantiate.
2. Declare the components (entity interfaces) you will instantiate.
3. Instantiate the components, and connect (map) the signals (including top-level ports) to the formal ports of the components to wire them up.

Step 1 – Write Entities and Architectures

Write entities and architectures for the design units to instantiate.

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

entity muxhier is
   port (outvec: out std_logic_vector (7 downto 0);
         a_vec, b_vec: in std_logic_vector (7 downto 0);
         sel: in std_logic);
end muxhier;

architecture mux_design of muxhier is
begin
-- <mux functionality>
end mux_design;
```

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

entity reg8 is
   port (q: buffer std_logic_vector (7 downto 0);
         data: in std_logic_vector (7 downto 0);
         clk, rst: in std_logic);
end reg8;

architecture reg8_design of reg8 is -- 8-bit register
begin
-- <8-bit register functionality>
end reg8_design;

library ieee;
use ieee.std_logic_1164.all;

entity rotate is
   port (q: buffer std_logic_vector (7 downto 0);
         data: in std_logic_vector (7 downto 0);
         clk, rst, r_l: in std_logic);
end rotate;

architecture rotate_design of rotate is
begin
-- Rotates bits or loads
-- When r_l is high, it rotates; if low, it loads data
-- <Rotation functionality>
end rotate_design;
```

Step 2 – Declare the Components

Components are declared in the declarative region of the architecture with a component declaration statement.

The component declaration syntax is:

component *entity_name*
  port (*port_list***);**
end component;

The entity_name and port_list of the component must match exactly that of the entity you will be instantiating.

Example

```
architecture structural of top_level_design is
-- Component declarations are placed here in the
-- declarative region of the architecture.

component muxhier -- Component declaration for mux
   port (outvec: out std_logic_vector (7 downto 0);
         a_vec, b_vec: in std_logic_vector (7 downto 0);
         sel: in std_logic);
end component;

component reg8 -- Component declaration for reg8
   port (q: out std_logic_vector (7 downto 0);
         data: in std_logic_vector (7 downto 0);
         clk, rst: in std_logic);
end component;

component rotate -- Component declaration for rotate
   port (q: buffer std_logic_vector (7 downto 0);
         data: in std_logic_vector (7 downto 0);
         clk, rst, r_l: in std_logic);
end component;
begin
-- The structural description goes here.
end structural;
```

Step 3 – Instantiate the Components

Use the following syntax to instantiate your components:

unique_instance_name **:** *component_name*
  [**generic map (***override_generic_values***)**]
    port map (*port_connections***);**

You can connect signals either with positional mapping (the same order declared in the entity) or with named mapping (where you specify the names of the lower-level signals to connect). Connecting by name minimizes errors, and especially advantageous when the component has many ports. To use configuration specification and declaration, refer to Configuration Specification and Declaration, on page 273.

Example

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

entity top_level is
   port (q: buffer std_logic_vector (7 downto 0);
         a, b: in std_logic_vector (7 downto 0);
         sel, r_l, clk, rst: in std_logic);
end top_level;

architecture structural of top_level is
-- The component declarations shown in Step 2 go here.
-- Declare the internal signals here
signal mux_out, reg_out: std_logic_vector (7 downto 0);

begin
-- The structural description goes here.
-- Instantiate a mux, name it inst1, and wire it up.
-- Map (connect) the ports of the mux using positional association.
inst1: muxhier port map (mux_out, a, b, sel);

-- Instantiate a rotate, name it inst2, and map its ports.
inst2: rotate port map (q, reg_out, clk, r_l, rst);

-- Instantiate a reg8, name it inst3, and wire it up.
-- reg8 is connected with named association.
-- The port connections can be given in any order.
-- Notice that the actual (local) signal names are on
-- the right of the '=>' mapping operators, and the
-- formal signal names from the component
-- declaration are on the left.
inst3: reg8 port map (
   clk => clk,
   data => mux_out,
   q => reg_out,
   rst => rst);

end structural;
```

Configuration Specification and Declaration

A configuration declaration or specification can be used to define binding information of component instantiations to design entities (entity-architecture pairs) in a hierarchical design. After the structure of one level of a design has been fully described using components and component instantiations, a designer must describe the hierarchical implementation of each component.

A configuration declaration or specification can also be used to define binding information of design entities (entity-architecture pairs) that are compiled in different libraries.

This section discusses usage models of the configuration declaration statement supported by the tool. The following topics are covered:

Component declarations and component specifications are not required for a component instantiation where the component name is the same as the entity name. In this case, the entity and its last architecture denote the default binding. In direct-entity instantiations, the binding information is available as the entity is specified, and the architecture is optionally specified. Configuration declaration and/or configuration specification are required when the component name does not match the entity name. If configurations are not used in this case, VHDL simulators give error messages, and the tool creates a black box and continues synthesis.

Configuration Specification

A configuration specification associates binding information with component labels that represent instances of a given component declaration. A configuration specification is used to bind a component instance to a design entity, and to specify the mapping between the local generics and ports of the component instance and the formal generics and ports of the entity. Optionally, a configuration specification can bind an entity to one of its architectures. The tool supports a subset of configuration specification commonly used in RTL synthesis; this section discusses that support.

The following Backus-Naur Form (BNF) grammar is supported (VHDL-93 LRM pp.73-79):

configuration_specification ::=

for component_specification binding_indication**;**

component_specification ::=

instantiation_list : component_name

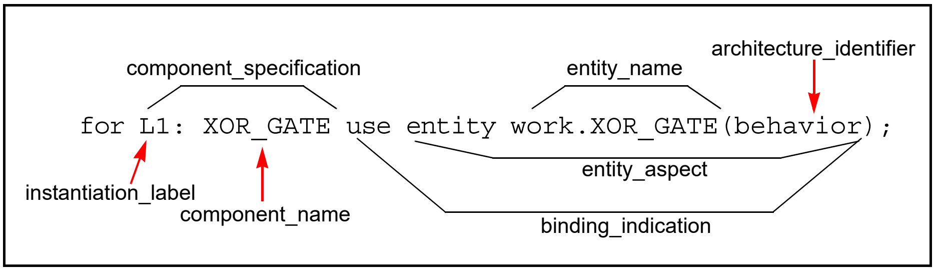
instantiation_list ::=

instantiation_label {, instantiation_label } | **others** | **all**

binding_indication ::= [**use** entity_aspect]
[generic_map_aspect]
[port_map_aspect]

entity_aspect ::=

entity entity_name [(architecture_identifier)] |
configuration configuration_name



```
for others: AND_GATE use entity work.AND_GATE(structure);
for all: XOR_GATE use entity work.XOR_GATE;
```

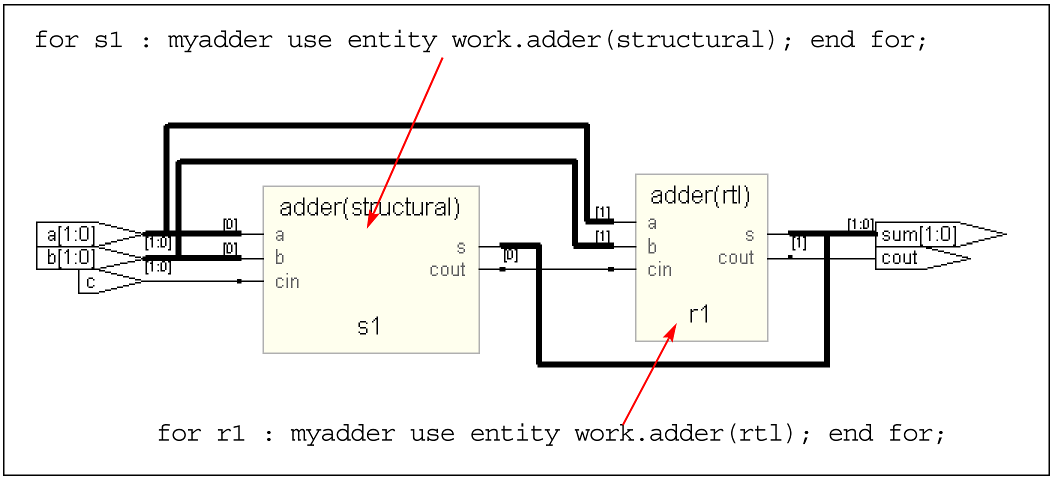
Example: Configuration Specification

In the following example, two architectures (RTL and structural) are defined for an adder. There are two instantiations of an adder in design top. A configuration statement defines the adder architecture to use for each instantiation.

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

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

library IEEE;
use IEEE.std_logic_unsigned.all;

architecture rtl of adder is
signal tmp : std_logic_vector(1 downto 0);
begin
   tmp <= ('0' & a) - b - cin;
   s <= tmp(0);
   cout <= tmp(1);
end rtl;

architecture structural of adder is
begin
   s <= a xor b xor cin;
   cout <= ((not a) and b and cin) or ( a and (not b) and cin)
      or (a and b and (not cin)) or (a and b and cin);
end structural;

library IEEE;
use IEEE.std_logic_1164.all;

entity top is
   port (a : in std_logic_vector(1 downto 0);
         b : in std_logic_vector(1 downto 0);
         c : in std_logic;
         cout : out std_logic;
         sum : out std_logic_vector(1 downto 0));
end top;

architecture top_a of top is
component myadder
   port (a : in std_logic;
         b : in std_logic;
         cin : in std_logic;
         s : out std_logic;
         cout : out std_logic);
end component;
```

```
signal carry : std_logic;
for s1 : myadder use entity work.adder(structural);
for r1 : myadder use entity work.adder(rtl);
begin
   s1 : myadder port map (a(0), b(0), c, sum(0), carry);
   r1 : myadder port map (a(1), b(1), carry, sum(1), cout);
end top_a;
```

Results


for s1 : myadder use entity work.adder(structural); end for;
adder(structural)
a
b
cin
s
cout
s1
adder(rtl)
a
b
cin
s
cout
r1
a[1:0]
b[1:0]
c
[1:0]
[1:0]
[0]
[0]
[0]
[1]
[1]
[1]
[1:0]
sum[1:0]
cout
for r1 : myadder use entity work.adder(rtl); end for;

Unsupported Constructs for Configuration Specification

The following configuration specification construct is *not* supported by the tool. An appropriate message is issued in the log file when this construct is used.

- The VHDL-93 LRM defines entity_aspect in the binding indication as:

  entity_aspect ::=

  > **entity** entity_name [(architecture_identifier)] |
  > **configuration** configuration_name **| open**

  The tool supports entity_name and configuration_name in the entity_aspect of a binding indication. The tool does not yet support the open construct.

Configuration Declaration

Configuration declaration specifies binding information of component instantiations to design entities (entity-architecture pairs) in a hierarchical design. Configuration declaration can bind component instantiations in an architecture, in either a block statement, a for…generate statement, or an if…generate statement. It is also possible to bind different entity-architecture pairs to different indices of a for…generate statement.

The tool supports a subset of configuration declaration commonly used in RTL synthesis. The following Backus-Naur Form (BNF) grammar is supported (VHDL-93 LRM pp.11-17):

configuration_declaration ::=

configuration identifier **of** entity_name **is**

block_configuration

end [**configuration**] [configuration_simple_name] **;**

block_configuration ::=

for block_specification

{ configuration_item }

end for ;

block_specification ::=

achitecture_name | block_statement_label |
generate_statement_label [(index_specification)]

index_specification ::=

discrete_range | static_expression

configuration_item ::=

block_configuration | component_configuration

component_configuration ::=

```
for component_specification
    [binding_indication ;]
    [block_configuration]
end for;
```

The BNF grammar for component_specification and binding_indication is described in Configuration Specification, on page 273.

Configuration Declaration within a Hierarchy

The following example shows a configuration declaration describing the binding in a 3-level hierarchy, for…generate statement labeled label1, within block statement blk1 in architecture arch_gen3. Each architecture implementation of an instance of my_and1 is determined in the configuration declaration and depends on the index value of the instance in the for…generate statement.

```
entity and1 is
   port(a,b: in bit ; o: out bit);
end;

architecture and_arch1 of and1 is
begin
   o <= a and b;
end;

architecture and_arch2 of and1 is
begin
   o <= a and b;
end;

architecture and_arch3 of and1 is
begin
   o <= a and b;
end;

library WORK; use WORK.all;
entity gen3_config is
   port(a,b: in bit_vector(0 to 7);
       res: out bit_vector(0 to 7));
end;
```

```
library WORK; use WORK.all;
architecture arch_gen3 of gen3_config is
component my_and1 port(a,b: in bit ; o: out bit); end component;
begin
   label1: for i in 0 to 7 generate
      blk1: block
      begin
         a1: my_and1 port map(a(i),b(i),res(i));
      end block;
   end generate;
end;

library work;
configuration config_gen3_config of gen3_config is
   for arch_gen3 -- ARCHITECTURE block_configuration "for
         -- block_specification"
      for label1 (0 to 3) --GENERATE block_config "for
         -- block_specification"
         for blk1 -- BLOCK block_configuration "for
         -- block_specification"
         -- {configuration_item}
            for a1 : my_and1 -- component_configuration
            -- Component_specification "for idList : compName"
               use entity work.and1(and_arch1); --
binding_indication
            end for; -- a1 component_configuration
         end for; -- blk1 BLOCK block_configuration
      end for; -- label1 GENERATE block_configuration
      for label1 (4) -- GENERATE block_configuration "for
         -- block_specification"
         for blk1
            for a1 : my_and1
               use entity work.and1(and_arch3);
            end for;
         end for;
      end for;

      for label1 (5 to 7)
         for blk1
            for a1 : my_and1
               use entity work.and1(and_arch2);
            end for;
         end for;
      end for;
   end for; -- ARCHITECTURE block_configuration
end config_gen3_config;
```

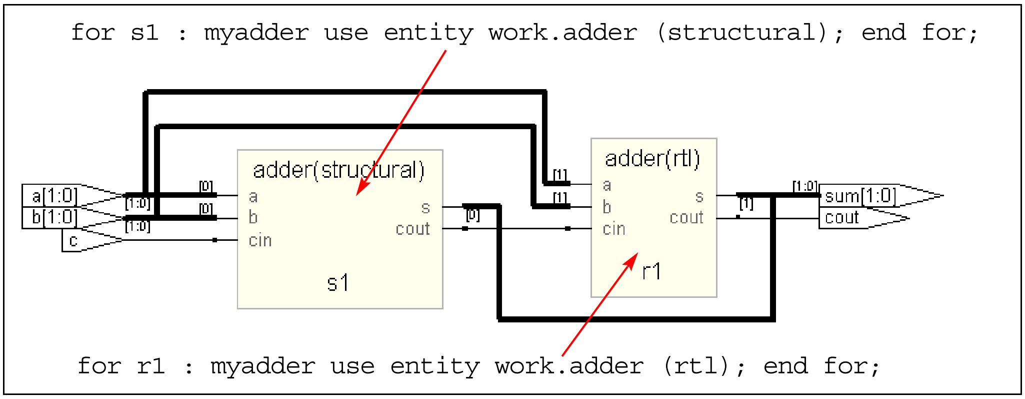
Selection with Configuration Declaration

In the following example, two architectures (RTL and structural) are defined for an adder. There are two instantiations of an adder in design top. A configuration declaration defines the adder architecture to use for each instantiation. This example is similar to the configuration specification example.

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

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

library IEEE;
use IEEE.std_logic_unsigned.all;

architecture rtl of adder is
signal tmp : std_logic_vector(1 downto 0);
begin
   tmp <= ('0' & a) - b - cin;
   s <= tmp(0);
   cout <= tmp(1);
end rtl;

architecture structural of adder is
begin
   s <= a xor b xor cin;
   cout <= ((not a) and b and cin) or (a and (not b) and cin) or
      (a and b and (not cin)) or (a and b and cin);
end structural;

library IEEE;
use IEEE.std_logic_1164.all;

entity top is
   port (a : in std_logic_vector(1 downto 0);
         b : in std_logic_vector(1 downto 0);
         c : in std_logic;
         cout : out std_logic;
         sum : out std_logic_vector(1 downto 0));
end top;
```

```
architecture top_a of top is
component myadder
   port (a : in std_logic;
         b : in std_logic;
         cin : in std_logic;
         s : out std_logic;
         cout : out std_logic);
end component;

signal carry : std_logic;
begin
   s1 : myadder port map (a(0), b(0), c, sum(0), carry);
   r1 : myadder port map (a(1), b(1), carry, sum(1), cout);
end top_a;

library work;
configuration config_top of top is -- configuration_declaration
   for top_a -- block_configuration "for block_specification"
   -- component_configuration
      for s1: myadder -- component_specification
         use entity work.adder (structural); -- binding_indication
      end for; -- component_configuration
   -- component_configuration
      for r1: myadder -- component_specification
         use entity work.adder (rtl); -- binding_indication
      end for; -- component_configuration
   end for; -- block_configuration
end config_top;
```

Results



```
for s1 : myadder use entity work.adder (structural); end for;
```

```
for r1 : myadder use entity work.adder (rtl); end for;
```

Direct Instantiation of Entities Using Configuration

In this method, a configured entity (i.e., an entity with a configuration declaration) is directly instantiated by writing a component instantiation statement that directly names the configuration.

Syntax

label **: configuration** configurationName
    [**generic map (**actualGeneric1**,** actualGeneric2**,** ... **)**]
    [**port map (** port1**,** port2**,** ... **)**] **;**

Unsupported Constructs for Configuration Declaration

The following are the configuration declaration constructs that are *not* supported by the tool. Appropriate messages are displayed in the log file if these constructs are used.

1. The VHDL-93 LRM defines the configuration declaration as:

   configuration_declaration ::=

   configuration identifier **of** entity_name **is**
       configuration_declarative_part
       block_configuration
   end [**configuration**] [configuration_simple_name] **;**

   configuration_declarative_part ::= { configuration_declarative_item }

   configuration_declarative_item ::=

   use_clause | attribute_specification | group_declaration

   The tool does not support the configuration_declarative_part. It parses the use_clause and attribute_specification without any warning message. The group_declaration is not supported and an error message is issued.

2. VHDL-93 LRM defines entity aspect in the binding indication as:

   entity_aspect ::=

   entity entity_name [(architecture_identifier)] |
   configuration configuration_name **|** **open**

   block_configuration ::=

```
for block_specification
    { use_clause }
    { configuration_item }
end for ;
```

The tool does not support use_clause in block_configuration. This construct is parsed and ignored.

VHDL Configuration Statement Enhancement

This section highlights the VHDL configuration statement support and handling component declarations with corresponding entity descriptions. Topics include:

Generic mapping

Generics and ports can have different names and sizes at the entity and component levels. You use the configuration statement to bind them together with a configuration specification or a configuration declaration. The binding priority follows this order:

- Configuration specification
- Component specification
- Component declaration

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

```
entity test is
generic (range1 : integer := 11);
   port (a, a1 : in std_logic_vector(range1 - 1 downto 0);
         b, b1 : in std_logic_vector(range1 - 1 downto 0);
         c, c1 : out std_logic_vector(range1 - 1 downto 0));
end test;

architecture test_a of test is
component submodule1 is
generic (size : integer := 6);
   port (a : in std_logic_vector(size -1 downto 0);
         b : in std_logic_vector(size -1 downto 0);
         c : out std_logic_vector(size -1 downto 0));
end component;

for all : submodule1
use entity work.sub1(rtl)
generic map (size => range1);
begin
   UUT1 : submodule1 generic map (size => 4)
   port map (a => a,b => b,c => c);

end test_a;
```

If you define the following generic map for sub1, it takes priority:

```
entity sub1 is
generic(size: integer:=1);
   port (a: in std_logic_vector(size -1 downto 0);
         b : in std_logic_vector(size -1 downto 0);
         c : out std_logic_vector(size -1 downto 0);
end sub1;
```

Port Mapping

See Generic mapping, on page 283 for information about using the configuration statement and binding priority.

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

entity test is
generic (range1 : integer := 1);
   port (ta, ta1 : in std_logic_vector(range1 - 1 downto 0);
         tb, tb1 : in std_logic_vector(range1 - 1 downto 0);
         tc, tc1 : out std_logic_vector(range1 - 1 downto 0));
end test;
```

```
architecture test_a of test is
component submodule1
generic (my_size1 : integer := 6; my_size2 : integer := 6);
   port (d : in std_logic_vector(my_size1 -1 downto 0);
         e : in std_logic_vector(my_size1 -1 downto 0);
         f : out std_logic_vector(my_size2 -1 downto 0));
end component;

for UUT1 : submodule1
use entity work1.sub1(rtl)
generic map (size1 => my_size1, size2 => my_size2)
port map (a => d, b => e, c => f);

   begin
   UUT1 : submodule1 generic map (my_size1 => 1, my_size2 => 1)
   port map (d => ta, e => tb,f => tc);
   end test_a;
```

If you define the following port map for sub1, it overrides the previous definition:

```
entity sub1 is
generic(size1: integer:=6; size2:integer:=6);
port (a: in std_logic_vector (size1 -1 downto 0);
      b : in std_logic_vector (size1 -1 downto 0);
      c : out std_logic_vector (size2 -1 downto 0);
end sub1:
```

Mapping Multiple Entity Names to the Same Component

When a single component has multiple entities, you can use the configuration statement and the for label clause to bind them. The following is an example:

```
entity test is
generic (range1 : integer := 1);
   port (ta, ta1 : in std_logic_vector(range1 - 1 downto 0);
         tb, tb1 : in std_logic_vector(range1 - 1 downto 0);
         tc, tc1 : out std_logic_vector(range1 - 1 downto 0));
end test;

architecture test_a of test is
component submodule
generic (my_size1 : integer := 6; my_size2 : integer := 6);
   port (d,e : in std_logic_vector(my_size1 -1 downto 0);
         f : out std_logic_vector(my_size2 -1 downto 0));
end component;
```

```
begin
UUT1 : submodule generic map (1, 1)
   port map (d => ta, e => tb, f => tc);
UUT2 : submodule generic map (1, 1) port map
   (d => ta1, e => tb1, f => tc1)
end test_a;

configuration my_config of test is
for test_a
   for UUT1 : submodule
      use entity work.sub1(rtl)
      generic map (my_size1, my_size2)
      port map (d, e, f);
   end for;
   for others : submodule
      use entity work.sub2(rtl)
      generic map (my_size1, my_size2)
      port map (d, e, f);
   end for;
end for;
end my_config;
```

You can map multiple entities to the same component, as shown here:

```
entity sub1 is
generic(size1: integer:=6; size2:integer:=6);
port (a: in std_logic_vector (size1 -1 downto 0);
      b : in std_logic_vector (size1 -1 downto 0);
      c : out std_logic_vector (size2 -1 downto 0);
end sub1:

entity sub2 is
generic(width1: integer; width2:integer);
port (a1: in std_logic_vector(width1 -1 downto 0);
      b1 : in std_logic_vector (width1 -1 downto 0);
      c1 : out std_logic_vector (width2 -1 downto 0);
end sub1:
```

Generics Assigned to Configurations

Generics can be assigned to configurations instead of entities.

Entities can contain more generics than their associated component declarations. Any additional generics on the entities must have default values to be able to synthesize.

Entities can also contain fewer generics than their associated component declarations. The extra generics on the component have no affect on the implementation of the entity.

Following are some examples.

Example 1

Configuration conf_module1 contains a generic map on configuration conf_c. The component declaration for submodule1 does not have the generic use_extraSYN_ff, however, the entity has it.

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

entity submodule1 is
generic (width : integer := 16;
use_extraSYN_ff : boolean := false);
   port (clk : in std_logic;
         b : in std_logic_vector(width - 1 downto 0);
         c : out std_logic_vector(width - 1 downto 0));
end submodule1;

architecture rtl of submodule1 is
signal d : std_logic_vector(width - 1 downto 0);
begin
no_resynch : if use_extraSYN_ff = false generate
   d <= b;
end generate no_resynch;

resynch : if use_extraSYN_ff = true generate
   process (clk)
   begin
      if falling_edge(clk) then
         d <= b;
      end if;
   end process;
end generate resynch;

   process (clk)
   begin
      if rising_edge(clk) then
         c <= d;
      end if;
   end process;
end rtl;
```

```
configuration conf_c of submodule1 is
   for rtl
   end for;
end configuration conf_c;

library ieee;
use ieee.std_logic_1164.all;

entity module1 is
generic (width: integer := 16);
   port (clk : in std_logic;
         b : in std_logic_vector(width - 1 downto 0);
         c : out std_logic_vector(width - 1 downto 0));
end module1;

architecture rtl of module1 is
component submodule1
generic (width: integer := 8);
   port (clk : in std_logic;
         b : in std_logic_vector(width - 1 downto 0);
         c : out std_logic_vector(width - 1 downto 0));
end component;

begin
UUT2 : submodule1 port map (clk => clk,
   b => b,
   c => c);
end rtl;

library ieee;
configuration conf_module1 of module1 is
   for rtl
      for UUT2 : submodule1
         use configuration conf_c generic map(width => 16,
         use_extraSYN_ff => true);
      end for;
   end for;
end configuration conf_module1;
```

Example 2

The component declaration for mod1 has the generic size, which is not in the entity. A component declaration can have more generics than the entity, however, extra component generics have no affect on the entity's implementation.

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

entity module1 is
generic (width: integer := 16;
use_extraSYN_ff : boolean := false);
   port (clk : in std_logic;
         b : in std_logic_vector (width - 1 downto 0);
         c : out std_logic_vector(width - 1 downto 0));
end module1;

architecture rtl of module1 is
signal d : std_logic_vector(width - 1 downto 0);
begin
   no_resynch : if use_extraSYN_ff = false generate
      d <= b;
end generate no_resynch;

resynch : if use_extraSYN_ff = true generate -- insert pipeline
   -- registers
   process (clk)
   begin
      if falling_edge(clk) then
         d <= b;
      end if;
   end process;
end generate resynch;

   process (clk)
   begin
      if rising_edge(clk) then
         c <= d;
      end if;
   end process;
end rtl;

configuration module1_c of module1 is
   for rtl
   end for;
end module1_c;
```

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

entity test is
   port (clk : in std_logic;
         tb : in std_logic_vector(7 downto 0);
         tc : out std_logic_vector(7 downto 0));
end test;

architecture test_a of test is
component mod1
generic (width: integer := 16;
use_extraSYN_ff: boolean := false;
size : integer := 8);
   port (clk : in std_logic;
         b : in std_logic_vector(width - 1 downto 0);
         c : out std_logic_vector(width - 1 downto 0));
end component;

begin
UUT1 : mod1 generic map (width => 18)
   port map (clk => clk,
      b => tb,
      c => tc);
end test_a;

Configuration test_c of test is
for test_a
   for UUT1 : mod1
      use configuration module1_c
      generic map (width => 8, use_extraSYN_ff => true);
   end for;
end for;
end test_c;
```

Arithmetic Operators and Functions in Generic Maps

Arithmetic operators and functions can be used in generic maps. Following is an example.

Example

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

entity sub is
generic (width : integer:= 16);
   port (clk : in std_logic;
         a : in std_logic_vector (width - 1 downto 0);
         y : out std_logic_vector (width - 1 downto 0));
end sub;

architecture rtl1 of sub is
begin
   process (clk, a)
   begin
      if (clk = '1' and clk'event) then
         y <= a;
      end if;
   end process;
end rtl1;

architecture rtl2 of sub is
begin y <= a;
end rtl2;

configuration sub_c of sub is
for rtl1 end for;
end sub_c;

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

```
entity test is
generic (mcu_depth : integer:=1;
mcu_width : integer:=16);
   port (clk : in std_logic;
         a : in std_logic_vector
            ((mcu_depth*mcu_width)-1 downto 0);
         y : out std_logic_vector
            ((mcu_depth*mcu_width)-1downto 0));
end test;

architecture RTL of test is
constant CWIDTH : integer := 2;
constant size : unsigned := "100";
component sub generic (width : integer);
   port (clk : in std_logic;
         a : in std_logic_vector (CWIDTH - 1 downto 0);
         y : out std_logic_vector (CWIDTH - 1 downto 0));
end component;

begin i_sub : sub
generic map (width => CWIDTH) port map (clk => clk,
   a => a,
   y => y );
end RTL;

library ieee;
use ieee.std_logic_arith.all;

configuration test_c of test is
   for RTL
      for i_sub : sub use
         configuration sub_c
         generic map(width => (CWIDTH ** (conv_integer (size))));
      end for;
   end for;
end test_c;
```

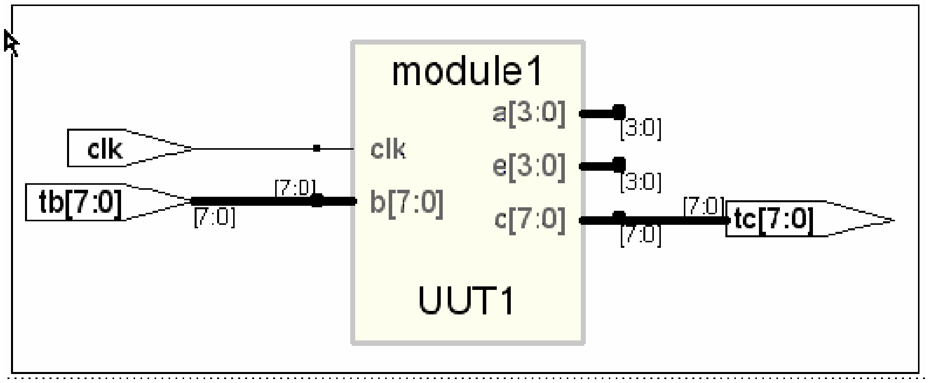
Ports in Component Declarations

Entities can contain more or fewer ports than their associated component declarations. Following are some examples.

Example 1

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

entity module1 is
generic (width: integer := 16; use_extraSYN_ff : boolean := false);
   port (clk : in std_logic;
         b : in std_logic_vector (width - 1 downto 0);
         a : out integer range 0 to 15; --extra output port
            on entity
         e : out integer range 0 to 15; -- extra output port
            on entity
         c : out std_logic_vector(width - 1 downto 0));
end module1;

architecture rtl of module1 is
signal d : std_logic_vector(width - 1 downto 0);
begin
e <= width;
a <= width;
no_resynch : if use_extraSYN_ff = false generate
   d <= b;
end generate no_resynch;

resynch : if use_extraSYN_ff = true generate
   process (clk)
   begin
      if falling_edge(clk) then
         d <= b;
      end if;
   end process;
end generate resynch;

   process (clk)
   begin
      if rising_edge(clk) then
         c <= d;
      end if;
   end process;
end rtl;
```

```
configuration module1_c of module1 is
for rtl
end for;
end module1_c;

library ieee;
use ieee.std_logic_1164.all;

entity test is
   port (clk : in std_logic;
         tb : in std_logic_vector(7 downto 0);
         tc : out std_logic_vector(7 downto 0));
end test;

architecture test_a of test is
component mod1
generic (width: integer := 16);
   port (clk : in std_logic;
         b : in std_logic_vector(width - 1 downto 0);
         c : out std_logic_vector(width - 1 downto 0));
end component;

begin
UUT1 : mod1 generic map (width => 18)
port map (clk => clk,
   b => tb,
   c => tc);
end test_a;

Configuration test_c of test is
for test_a
   for UUT1 : mod1
      use configuration module1_c
      generic map (width => 8, use_extraSYN_ff => true);
   end for;
end for;
end test_c;
```

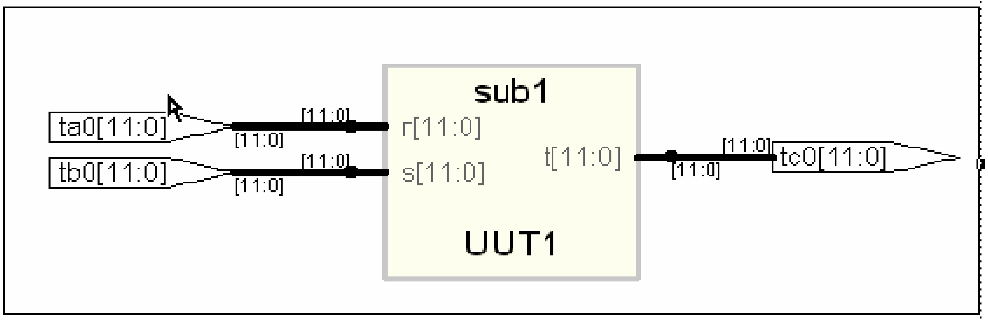


In the figure above, the entity module1 has extra ports a and e that are not defined in the corresponding component declaration mod1. The additional ports are not connected during synthesis.

Example 2

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY sub1 IS
GENERIC(
size1 : integer := 11;
size2 : integer := 12);
   PORT (r : IN std_logic_vector(size1 -1 DOWNTO 0);
         s : IN std_logic_vector(size1 -1 DOWNTO 0);
         t : OUT std_logic_vector(size2 -1 DOWNTO 0));
END sub1;

ARCHITECTURE rtl OF sub1 IS
BEGIN
   t <= r AND s;
END ARCHITECTURE rtl;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY test IS
GENERIC (range1 : integer := 12);
   PORT (ta0 : IN std_logic_vector(range1 - 1 DOWNTO 0);
         tb0 : IN std_logic_vector(range1 - 1 DOWNTO 0);
         tc0 : OUT std_logic_vector(range1 - 1 DOWNTO 0));
END test;
```

```
ARCHITECTURE test_a OF test IS
COMPONENT submodule
GENERIC (
my_size1 : integer := 4;
my_size2 : integer := 5);
   PORT (d : IN std_logic_vector(my_size1 -1 DOWNTO 0);
         e : IN std_logic_vector(my_size1 -1 DOWNTO 0);
         ext_1 : OUT std_logic_vector(my_size1 -1 DOWNTO 0);
         ext_2 : OUT std_logic_vector(my_size1 -1 DOWNTO 0);
         f : OUT std_logic_vector(my_size2 -1 DOWNTO 0));
END COMPONENT;

BEGIN
UUT1 : submodule
GENERIC MAP (
my_size1 => range1,
my_size2 => range1)
   PORT MAP (ext_1 => open,
      ext_2 => open,
      d => ta0,
      e => tb0,
      f => tc0);
END test_a;

CONFIGURATION my_config OF test IS
   FOR test_a
      FOR UUT1 : submodule
      USE ENTITY work.sub1(rtl)
      GENERIC MAP (
         size1 => my_size1,
         size2 => my_size2)
      PORT MAP (r => d,
         s => e,
         t => f );
      END FOR;
   END FOR; -- test_a
END my_config;
```



In the figure above, the component declaration has more ports (ext_1 ext_2) than entity sub1. The component is synthesized based on the number of ports on the entity.

Scalable Designs

This section describes creating and using scalable VHDL designs. You can create a VHDL design that is scalable, meaning that it can handle a user-specified number of bits or components.

Creating a Scalable Design Using Unconstrained Vector Ports

Do not size (constrain) the ports until you need them. This first example is coding the adder using the - operator, and gives much better synthesized results than the second and third scalable design examples, which code the adders as random logic.

Example

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity addn is
-- Notice that a, b, and result ports are not constrained.
-- In VHDL, they automatically size to whatever is connected
-- to them.
   port (result : out std_logic_vector;
         cout : out std_logic;
         a, b : in std_logic_vector;
         cin : in std_logic);
end addn;

architecture stretch of addn is
   signal tmp : std_logic_vector (a'length downto 0);
begin
-- The next line works because "-" sizes to the largest operand
-- (also, you need only pad one argument).
tmp <= ('0' & a) - b - cin;
```

```
result <= tmp(a'length - 1 downto 0);
cout <= tmp(a'length);
assert result'length = a'length;
assert result'length = b'length;
end stretch;

-- Top level design
-- Here is where you specify the size for a, b,
-- and result. It is illegal to leave your top
-- level design ports unconstrained.

library ieee;
use ieee.std_logic_1164.all;

entity addntest is
   port (result : out std_logic_vector (7 downto 0);
         cout : out std_logic;
         a, b : in std_logic_vector (7 downto 0);
         cin : in std_logic);
end addntest;

architecture top of addntest is
component addn
   port (result : std_logic_vector;
         cout : std_logic;
         a, b : std_logic_vector;
         cin : std_logic);
end component;

begin
test : addn port map (result => result,
   cout => cout,
   a => a,
   b => b,
   cin => cin );
end;
```

Creating a Scalable Design Using VHDL Generics

Create a VHDL generic with default value. The generic is used to represent bus sizes inside a architecture, or a number of components. You can define more than one generic per declaration by separating the generic definitions with semicolons (;).

Syntax

generic (*generic_1_name* **:** *type* [**:=** *default_value*]**) ;**

Examples

```
generic (num : integer := 8) ;
generic (top : integer := 16; num_bits : integer := 32);
```

Using a Scalable Architecture with VHDL Generics

Instantiate the scalable architecture, and override the default generic value with the generic map statement.

Syntax

generic map (*list_of_overriding_values* **)**

Examples

Generic map construct

```
generic map (16)
-- These values will get mapped in order given.
generic map (8, 16)
```

Creating a scalable adder

```
library ieee;
use ieee.std_logic_1164.all;
entity adder is
   generic(num_bits : integer := 4); -- Default adder
      -- size is 4 bits
   port (a : in std_logic_vector (num_bits downto 1);
         b : in std_logic_vector (num_bits downto 1);
         cin : in std_logic;
         sum : out std_logic_vector (num_bits downto 1);
         cout : out std_logic );
end adder;
```

```
architecture behave of adder is
begin
   process (a, b, cin)
   variable vsum : std_logic_vector (num_bits downto 1);
   variable carry : std_logic;
   begin
   carry := cin;
      for i in 1 to num_bits loop
         vsum(i) := (a(i) xor b(i)) xor carry;
         carry := (a(i) and b(i)) or (carry and (a(i) or b(i)));
      end loop;
   sum <= vsum;
   cout <= carry;
   end process;
end behave;
```

Scaling the Adder by Overriding the generic Statement

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

entity adder16 is
   port (a : in std_logic_vector (16 downto 1);
         b : in std_logic_vector (16 downto 1);
         cin : in std_logic;
         sum : out std_logic_vector (16 downto 1);
         cout : out std_logic );
end adder16;

architecture behave of adder16 is
-- The component declaration goes here.
-- This allows you to instantiate the adder.
component adder
-- The default adder size is 4 bits.
generic(num_bits : integer := 4);
   port (a : in std_logic_vector ;
         b : in std_logic_vector;
         cin : in std_logic;
         sum : out std_logic_vector;
         cout : out std_logic );
end component;

begin
my_adder : adder
   generic map (16) -- Use a 16 bit adder
   port map(a, b, cin, sum, cout);
end behave;
```

Creating a Scalable Design Using Generate Statements

A VHDL generate statement allows you to repeat logic blocks in your design without having to write the code to instantiate each one individually.

Creating a 1-bit Adder

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

Instantiating the 1-bit Adder Many Times with a Generate Statement

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

entity addern is
generic(n : integer := 8);
   port (a, b : in std_logic_vector (n downto 1);
         cin : in std_logic;
         sum : out std_logic_vector (n downto 1);
         cout : out std_logic);
end addern;

architecture structural of addern is
-- The adder component declaration goes here.
component adder
   port (a, b, cin : in std_logic;
         sum, cout : out std_logic);
end component;
```

```
signal carry : std_logic_vector (0 to n);
begin
-- Generate instances of the single-bit adder n times.
-- You need not declare the index 'i' because
-- indices are implicitly declared for all FOR
-- generate statements.

gen: for i in 1 to n generate
   add: adder port map(
      a => a(i),
      b => b(i),
      cin => carry(i - 1),
      sum => sum(i),
      cout => carry(i));
end generate;

carry(0) <= cin;
cout <= carry(n);

end structural;
```

Instantiating Black Boxes in VHDL

Black boxes are design units with just the interface specified; internal information is ignored by the tool. Black boxes can be used to directly instantiate:

- Technology-vendor primitives and macros (including I/Os).
- User-defined macros whose functionality was defined in a schematic editor, or another input source (when the place-and-route tool can merge design netlists from different sources).

Black boxes are specified with the syn_black_box synthesis directive, in conjunction with other directives. If the black box is a technology-vendor I/O pad, use the black_box_pad_pin directive instead.

Here is a list of the directives that you can use to specify modules as black boxes, and to define design objects on the black box for consideration during synthesis:

- syn_black_box
- black_box_pad_pin
- black_box_tri_pins
- syn_isclock
- syn_tco<*n*>
- syn_tpd<*n*>
- syn_tsu<*n*>

For descriptions of the black-box attributes and directives, see the *Attribute Reference* manual.

For information on how to instantiate black boxes and technology-vendor I/Os, see *Defining Black Boxes for Synthesis*, on page 398 of the *User Guide*.

Black-Box Timing Constraints

You can provide timing information for your individual black box instances. The following are the three predefined timing constraints available for black boxes.

- syn_tpd<*n*> – Timing propagation for combinational delay through the black box.
- syn_tsu<*n*> – Timing setup delay required for input pins (relative to the clock).
- syn_tco<*n*>– Timing clock to output delay through the black box.

Here, *n* is an integer from 1 through 10, inclusive. See syn_black_box, on page 44, for details about constraint syntax.

VHDL Attribute and Directive Syntax

Synthesis attributes and directives can be defined in the VHDL source code to control the way the design is analyzed, compiled, and mapped. *Attributes* direct the way your design is optimized and mapped during synthesis. *Directives* control the way your design is analyzed prior to compilation. Because of this distinction, directives must be included in your VHDL source code while attributes can be specified either in the source code or in a constraint file.

The directives and attributes are predefined in the attributes package in the synthesis library for the tool. This library package contains the built-in attributes, along with declarations for timing constraints (including black-box timing constraints) and vendor-specific attributes. The file is located here:

installDirectory/lib/vhd/synattr.vhd

There are two ways to specify VHDL attributes and directives:

- Using the attributes Package, on page 306
- Declaring Attributes, on page 307

You can either use the attributes package or redeclare the types of directives and attributes each time you use them. You typically use the attributes package.

Using the attributes Package

This is the most typical way to specify the attributes, because you only need to specify the package once. You specify the attributes package, using the following code:

library synplify;
use synplify.attributes.all;
-- *design_unit_declarations*
attribute *productname_attribute* **of** *object* **:** *object_type* **is** *value***;**

The following is an example using syn_noclockbuf from the attributes package:

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

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

// No explicit type declaration is necessary
attribute syn_noclockbuf of clk : signal is true;

-- Other code
```

Declaring Attributes

The alternative method is to declare the attributes to explicitly define them. You must do this each time you use an attribute. Here is the syntax for declaring directives and attributes in your code, without referencing the attributes package:

-- *design_unit_declarations*
attribute *attribute_name* **:** *data_type* **;**
attribute *attribute_name* **of** *object* **:** *object_type* **is** *value***;**

Here is an example using the syn_noclockbuf attribute:

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

// Explicit type declaration
attribute syn_noclockbuf : boolean;
attribute syn_noclockbuf of clk : signal is true;

-- Other code
```

Case Sensitivity

Although VHDL is case-insensitive, directives, attributes, and their values are case sensitive and must be declared in the code using the correct case. This rule applies especially for port names in directives.

For example, if a port in VHDL is defined as GIN, the following code does not work:

```
attribute black_box_tri_pin : string;
attribute black_box_tri_pin of BBDLHS : component is "gin";
```

The following code is correct because the case of the port name is correct:

```
attribute black_box_tri_pin : string;
attribute black_box_tri_pin of BBDLHS : component is "GIN";
```

VHDL Synthesis Examples

This section describes the VHDL examples that are provided with the tool. The topics include:

- Combinational Logic Examples, on page 308
- Sequential Logic Examples, on page 309

Combinational Logic Examples

The following combinational logic synthesis examples are included in the *installDirectory*/examples/vhdl/common_rtl/combinat directory:

- Adders
- ALU
- Bus Sorter (illustrates using procedures in VHDL)
- 3-to-8 Decoders
- 8-to-3 Priority Encoders
- Comparator
- Interrupt Handler (coded with an if-then-else statement for the desired priority encoding)
- Multiplexers (concurrent signal assignments, case statements, or if-then-else statements can be used to create multiplexers; the tool automatically creates parallel multiplexers when the conditions in the branches are mutually exclusive)
- Parity Generator
- Tristate Drivers

Sequential Logic Examples

The following sequential logic synthesis examples are included in the *installDirectory*/examples/vhdl/common_rtl/sequentl directory:

- Flip-flops and level-sensitive latches
- Counters (up, down, and up/down)
- Register file
- Shift register
- State machines

For additional information on synthesizing flip-flops and latches, see:

- Creating Flip-flops and Registers Using VHDL Processes, on page 247
- Level-sensitive Latches Using Concurrent Signal Assignments, on page 251
- Level-sensitive Latches Using VHDL Processes, on page 252
- Asynchronous Sets and Resets, on page 257
- Synchronous Sets and Resets, on page 258

PREP VHDL Benchmarks

PREP (Programmable Electronics Performance) Corporation distributes benchmark results that show how FPGA vendors compare with each other in terms of device performance and area.

The following PREP benchmarks are included in the *installDirectory*/examples/vhdl/common_rtl/prep directory:

- PREP Benchmark 1, Data Path (prep1.vhd)
- PREP Benchmark 2, Timer/Counter (prep2.vhd)
- PREP Benchmark 3, Small State Machine (prep3.vhd)
- PREP Benchmark 4, Large State Machine (prep4.vhd)
- PREP Benchmark 5, Arithmetic Circuit (prep5.vhd)
- PREP Benchmark 6, 16-Bit Accumulator (prep6.vhd)
- PREP Benchmark 7, 16-Bit Counter (prep7.vhd)
- PREP Benchmark 8, 16-Bit Pre-scaled Counter (prep8.vhd)
- PREP Benchmark 9, Memory Map (prep9.vhd)

The source code for the benchmarks can be used for design examples for synthesis or for doing your own FPGA vendor comparisons.

CHAPTER 4

VHDL 2008 Language Support

This chapter describes support for the VHDL 2008 standard for the Synopsys tool. For information on the VHDL standard, see Chapter 3, *VHDL Language Support* and the IEEE 1076™-2008 standard. The following sections describe the current level of VHDL 2008 support.

Operators and Expressions

VHDL 2008 includes support for the following operators:

- Logical Reduction operators – the logic operators: and, or, nand, nor, xor, and xnor can now be used as unary operators
- Condition operator (??) – converts a bit or std_ulogic value to a boolean value
- Matching Relational operators (?=, ?/=, ?<, ?<=, ?>, ?>=) – similar to the normal relational operators, but return bit or std_ulogic values in place of Boolean values
- Bit-string literals – bit-string characters other than 0 and 1 and string formats including signed/unsigned and string length
- Aggregates (aggregate pattern assignments) are used to group values in an array or structured expression.

Logical Reduction Operators

The logical operators and, or, nand, nor, xor, and xnor can be used as unary operators.

Example – Logical Operators

Condition Operator

The condition operator (??) converts a bit or std_ulogic value to a boolean value. The operator is implicitly applied in a condition where the expression would normally be interpreted as a boolean value as shown in the if statement in the two examples below.

Example – VHDL 2008 Style Conditional Operator

Example – VHDL 1993 Style Conditional Operator

In the VHDL 2008 example, the statement

```
if sel then
```

is equivalent to:

```
if (?? sel) then
```

The implicit use of the ?? operator occurs in the following conditional expressions:

- after if or elsif in an if statement
- after if in an if-generate statement
- after until in a wait statement
- after while in a while loop
- after when in a conditional signal statement
- after assert in an assertion statement
- after when in a next statement or an exit statement

Matching Relational Operators

The matching relational operators return a bit or std_ulogic result in place of a Boolean.

Example – Relational Operators

Bit-string Literals

Bit-string literal support in VHDL 2008 includes:

- Support for characters other than 0 and 1 in the bit string, such as X or Z.

  For example:

  `X"Z45X"` is equivalent to `"ZZZZ01000101XXXX"`

  `B"0001-"` is equivalent to `"0001-"`

  `O"75X"` is equivalent to `"111101XXX"`

- Optional support for a length specifier that determines the length of the string to be assigned.

  Syntax: [*length*] *baseSpecifier* **"***bitStringvalue***"**

  For example:

  `12X"45"` is equivalent to `"000001000101"`

  `5O"17"` is equivalent to `"01111"`

- Optional support for a signed (S) or unsigned (U) qualifier that determines how the bit-string value is expanded/truncated when a length specifier is used.

  Syntax: [*length*] **S|U** *baseSpecifier* **"***bitStringvalue***"**

For example:

`12UB"101"` is equivalent to `"000000000101"`

`12SB"101"` is equivalent to `"111111111101"`

`12UX"96"` is equivalent to `"000010010110"`

`12SX"96"` is equivalent to `"111110010110"`

- Additional support for a base specifier for decimal numbers (D). The number of characters in the bit string can be determined by using the expression $(\log_2 n)+1$; where n is the decimal integer.

  Syntax: [*length*] **D "***bitStringvalue***"**

  For example:

  `D"10"` is equivalent to `"1010"`

  `10D"35"` is equivalent to `"0000100011"`

For complete descriptions of bit-string literal requirements, see the VHDL 2008 LRM.

Array Aggregates

Aggregates (aggregate pattern assignments) are used to group values in an array or structured expression. Earlier versions of VHDL required that an array aggregate be comprised of only individual elements. VHDL 2008 extends the rules, allowing aggregates to use a mixture of individual elements and slices of the array.

Example 1: LHS Slices in an Array Aggregate

This example of an array aggregate contains LHS slices of the array.

Example 2: RHS Slices in an Array Aggregate

This example of an array aggregate contains RHS slices of the array.

Unconstrained Data Types

VHDL 2008 allows the element types for arrays and the field types for records to be unconstrained. In addition, VHDL 2008 includes support for partially constrained subtypes in which some elements of the subtype are constrained, while others elements are unconstrained. Specifically, VHDL 2008:

- Supports unconstrained arrays of unconstrained arrays (i.e., element types of arrays can be unconstrained)
- Supports the VHDL 2008 syntax that allows a new subtype to be declared that constrains any element of an existing type that is not yet constrained
- Supports the ‘element attribute that returns the element subtype of an array object
- Supports the new ‘subtype attribute that returns the subtype of an object

Example – Unconstrained Element Types

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

package myTypes is
   type memUnc is array (natural range <>) of std_logic_vector;
   function summation(varx: memUnc) return std_logic_vector;
end package myTypes;

package body myTypes is
   function summation(varx: memUnc) return std_logic_vector is
      variable sum: varx’element;
   begin
      sum := (others => '0');
         for I in 0 to varx’length - 1 loop
            sum := sum + varx(I);
         end loop;
      return sum;
   end function summation;
end package body myTypes;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use work.myTypes.all;
```

```
entity sum is
   port (in1: memUnc(0 to 2)(3 downto 0);
         out1: out std_logic_vector(3 downto 0));
end sum;

architecture uncbehv of sum is
begin
   out1 <= summation(in1);
end uncbehv;
```

Example – Unconstrained Elements within Nested Arrays

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

package myTypes is
   type t1 is array (0 to 1) of std_logic_vector;
   type memUnc is array (natural range <>) of t1;
   function doSum(varx: memUnc) return std_logic_vector;
end package myTypes;

package body myTypes is
   function addVector(vec: t1) return std_logic_vector is
      variable vecres: vec’element := (others => '0');
   begin
      for I in vec’Range loop
         vecres := vecres + vec(I);
      end loop;
      return vecres;
   end function addVector;
   function doSum(varx: memUnc) return std_logic_vector is
      variable sumres: varx’element’element;
   begin
      if (varx’length = 1) then
         return addVector(varx(varx’low));
      end if;
      if (varx’Ascending) then
         sumres := addVector(varx(varx’high)) +
            doSum(varx(varx’low to varx’high-1));
      else
         sumres := addVector(varx(varx’low)) +
            doSum(varx(varx’high downto varx’low+1));
      end if;
      return sumres;
   end function doSum;
end package body myTypes;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use work.myTypes.all;

entity uncfunc is
   port (in1: in memUnc(1 downto 0)(open)(0 to 3);
         in2: in memUnc(0 to 2)(open)(5 downto 0);
         in3: in memUnc(3 downto 0)(open)(2 downto 0);
         out1: out std_logic_vector(5 downto 0);
         out2: out std_logic_vector(0 to 3);
         out3: out std_logic_vector(2 downto 0));
end uncfunc;

architecture uncbehv of uncfunc is
begin
   out1 <= doSum(in2);
   out2 <= doSum(in1);
   out3 <= doSum(in3);
end uncbehv;
```

Unconstrained Record Elements

VHDL 2008 allows element types for records to be unconstrained (earlier versions of VHDL required that the element types for records be fully constrained). In addition, VHDL 2008 supports the concept of partially constrained subtypes in which some parts of the subtype are constrained, while others remain unconstrained.

Example – Unconstrained Record Elements

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

entity unctest is
   port (in1: in std_logic_vector (2 downto 0);
         in2: in std_logic_vector (3 downto 0);
         out1: out std_logic_vector(2 downto 0));
end unctest;
```

```
architecture uncbehv of unctest is
   type zRec is record
      f1: std_logic_vector;
      f2: std_logic_vector;
   end record zRec;
subtype zCnstrRec is zRec(f1(open), f2(3 downto 0));
subtype zCnstrRec2 is zCnstrRec(f1(2 downto 0), f2(open));
signal mem: zCnstrRec2;
begin
   mem.f1 <= in1;
   mem.f2 <= in2;
   out1 <= mem.f1 and mem.f2(2 downto 0);
end uncbehv;
```

Predefined Functions

VHDL 2008 adds the minimum and maximum predefined functions. The behavior of these functions is defined in terms of the **“<”** operator for the operand type. The functions can be binary to compare two elements, or unary when the operand is an array type.

Example – Minimum/Maximum Predefined Functions

```
entity minmaxTest is
   port (ary1, ary2: in bit_vector(3 downto 0);
         minout, maxout: out bit_vector(3 downto 0);
         unaryres: out bit);
end minmaxTest;

architecture rtlArch of minmaxTest is
begin
   maxout <= maximum(ary1, ary2);
   minout <= minimum(ary1, ary2);
   unaryres <= maximum(ary1);
end rtlArch;
```

Generic Types

VHDL 2008 introduces several types of generics that are not present in VHDL IEEE Std 1076-1993. These types include generic types, generic packages, and generic subprograms.

Generic Types

Generic types allow logic descriptions that are independent of type. These descriptions can be declared as a generic parameter in both packages and entities. The actual type must be provided when instantiating a component or package.

Example of a generic type declaration:

```
entity mux is
   generic (type dataType);
   port (sel: in bit; za, zb: in dataType; res: out dataType);
end mux;
```

Example of instantiating an entity with a type generic:

```
inst1:  mux generic map (bit_vector(3 downto 0))
   port map (selval,in1,in2,out1);
```

Generic Packages

Generic packages allow descriptions based on a formal package. These descriptions can be declared as a generic parameter in both packages and entities. An actual package (an instance of the formal package) must be provided when instantiating a component or package.

Example of a generic package declaration:

```
entity mux is generic(
   package argpkg is new dataPkg generic map (<>);
);
   port (sel: in bit; za, zb: in bit_vector(3 downto 0);
      res: out bit_vector(3 downto 0));
end mux;
```

Example of instantiating a component with a package generic:

```
package memoryPkg is new dataPkg generic map (4, 16);

...

inst1: entity work.mux generic map (4, 16, argPkg => memoryPkg)
```

Generic Subprograms

Generic subprograms allow descriptions based on a formal subprogram that provides the function prototype. These descriptions can be declared as a generic parameter in both packages and entities. An actual function must be provided when instantiating a component or package.

Example of a generic subprogram declaration:

```
entity mux is
   generic (type dataType; function filter(datain: dataType)
      return dataType);
   port (sel: in bit; za, zb: in dataType; res: out dataType);
end mux;
```

Example of instantiating a component with a subprogram generic:

```
architecture myarch2 of myTopDesign is
   function intfilter(din: integer) return integer is
   begin
      return din + 1;
   end function intfilter;

...

begin
   inst1: mux generic map (integer, intfilter)
      port map (selval,intin1,intin2,intout);
```

Packages

VHDL 2008 includes several new packages and modifies some of the existing packages. The new and modified packages are located in the $LIB/vhd2008 folder instead of $LIB/vhd.

New Packages

The following packages are supported in VHDL 2008:

- fixed_pkg.vhd, float_pkg.vhd, fixed_generic_pkg.vhd, float_generic_pkg.vhd, fixed_-float_types.vhd – IEEE fixed and floating point packages
- numeric_bit_unsigned.vhd – Overloads for bit_vector to have all operators defined for ieee.numeric_bit.unsigned
- numeric_std_unsigned.vhd – Overloads for std_ulogic_vector to have all operators defined for ieee.numeric_std.unsigned

String and text I/O functions in the above packages are not to be supported. These functions include read(), write().

Modified Packages

The following modified IEEE packages are supported with the exception of the new string and text I/O functions (the previously supported string and text I/O functions are unchanged):

- std.vhd – new overloads
- std_logic_1164.vhd – std_logic_vector is now a subtype of std_ulogic_vector; new overloads
- numeric_std.vhd – new overloads
- numeric_bit.vhd – new overloads

Supported Package Functions

VHDL 2008 supports the following functions in the numeric_std.vhd, numeric_bit.vhd, and std_logic_1164.vhd packages:

- to_01
- to_string/to_ostring/to_hstring

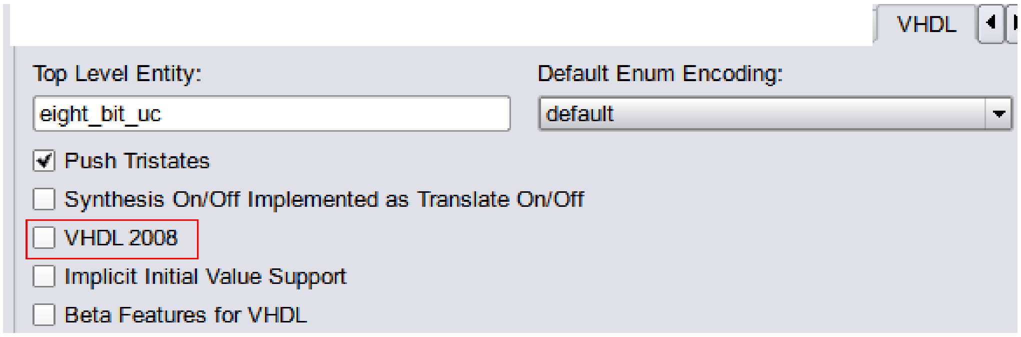
Unsupported Packages/Functions

The following packages and functions are not currently supported:

- string and text I/O functions in the new packages
- The fixed_pkg_params.vhd or float_pkg_params.vhd packages, which were temporarily supported to allow the default parameters to be changed for fixed_pkg.vhd and float_pkg.vhd packages, have been obsoleted by the inclusion of the fixed_generic_pkg.vhd or float_generic_pkg.vhd packages.

Using the Packages

A switch for VHDL 2008 is located in the GUI on the VHDL panel (Implementation Options dialog box) to enable use of these packages and the ?? operator.



Top Level Entity: eight_bit_uc

Default Enum Encoding: default

[✓] Push Tristates

[ ] Synthesis On/Off Implemented as Translate On/Off

[ ] VHDL 2008

[ ] Implicit Initial Value Support

[ ] Beta Features for VHDL

You can also enable the VHDL 2008 packages by including the following command in the compiler options section of your design file:

```
set_option -vhdl2008 1
```

Generics in Packages

In VHDL 2008, packages can include generic clauses. These generic packages can then be instantiated by providing values for the generics as shown in the following example.

Example – Including Generics in Packages

Context Declarations

VHDL 2008 provides a new type of design unit called a context declaration. A context is a collection of library and use clauses. Both context declarations and context references are supported as shown in the following example.

Example – Context Declaration

In VHDL 2008, a context clause cannot precede a context declaration. The following code segment results in a compiler error.

```
library ieee; -- Illegal context clause before a
               -- context declaration
context zcontext is
   use ieee.std_logic_1164.all;
   use ieee.numeric_std.all;
end context zcontext;
```

Similarly, VHDL 2008 does not allow reference to the library name work in a context declaration. The following code segment also results in a compiler error.

```
context zcontext is
   use work.zpkg.all; -- Illegal reference to library work
                -- in a context declaration
   library ieee;
   use ieee.numeric_std.all;
end context zcontext;
```

VHDL 2008 supports the following two, standard context declarations in the IEEE package:

- IEEE_BIT_CONTEXT
- IEEE_STD_CONTEXT

Case-generate Statements

The case-generate statement is a new type of generate statement incorporated into VHDL 2008. Within the statement, alternatives are specified similar to a case statement. A static (computable at elaboration) select statement is compared against a set of choices as shown in the following syntax:

```
caseLabel:  case expression generate
             when choice1 =>
                    -- statement list
              when choice2 =>
                    -- statement list
              ...
              end generate caseLabel;
```

To allow for configuration of alternatives in case-generate statements, each alternative can include a label preceding the choice value (e.g., labels L1 and L2 in the syntax below):

```
caseLabel:  case expression generate
             when L1: choice1 =>
                    -- statement list
              when L2: choice2 =>
                    -- statement list
              ...
              end generate caseLabel;
```

Example – Case-generate Statement with Alternatives

Example – Case-generate Statement with Labels for Configuration

Matching case and select Statements

Matching case and matching select statements are supported – case? (matching case statement) and select? (matching select statement). The statements use the ?= operator to compare the case selector against the case options.

Example – Use of case? Statement

Example – Use of select? Statement

Else/elsif Clauses

In VHDL 2008, else and elsif clauses can be included in if-generate statements. You can configure specific if/else/elsif clauses using configurations by adding a label before each condition. In the code example below, the labels on the branches of the if-generate statement are spec1, spec2, and spec3. These labels are later referenced in the configuration myconfig to specify the appropriate entity/architecture pair. This form of labeling allows statements to be referenced in configurations.

Example – Else/elsif Clauses in If-Generate Statements

Sequential Signal Assignments

Earlier versions of VHDL allowed when-else and with-select assignments to be used only as concurrent statements. VHDL 2008 supports that these assignments can also be used in a sequential context, such as, inside a process block.

Using When-Else and With-Select Assignments

Here are examples of when-else and with-select assignments inside a process block.

Example: When-else in a process block

Example: With-select in a process block

Using Output Ports in a Sensitivity List

VHDL 2008 supports the use of output ports in the sensitivity list of a process block.

Syntax Conventions

The following syntax conventions are supported in VHDL 2008:

- All keyword
- Comment delimiters
- Extended character set

All Keyword

VHDL 2008 supports the use of an all keyword in place of the list of input signals to a process in the sensitivity list.

Example – All Keyword in Sensitivity List

Comment Delimiters

VHDL 2008 supports the /* and */ comment-delimiter characters. All text enclosed between the beginning /* and the ending */ is treated as a comment, and the commented text can span multiple lines. The standard VHDL “--” comment-introduction character string is also supported.

Extended Character Set

The extended ASCII character literals (ASCII values from 128 to 255) are supported.

Example – Extended Character Set

Index