B

Verilog Syntax

This appendix contains a syntax description of the Verilog language as supported by Synopsys HDL Compiler. It covers the following topics:

- Syntax
- Lexical Conventions
- Verilog Keywords
- Unsupported Verilog Language Constructs

Syntax

This section presents the syntax of the supported Verilog language in Backus-Naur form (BNF) and the syntax formalism.

Note:

The BNF syntax convention used in this section differs from the Synopsys syntax convention used elsewhere in this manual.

BNF Syntax Formalism

White space separates lexical tokens.

`name`

is a keyword.

`<name>`

is a syntax construct definition.

`<name>`

is a syntax construct item.

`<name>?`

is an optional item.

`<name>*`

is zero, one, or more items.

`<name>+`

is one or more items.

`<port> <,<port>>*`

is a comma-separated list of items.

`::=`

gives a syntax definition to an item.

`||=`

refers to an alternative syntax construct.

BNF Syntax

```
<source_text>
   ::= <description>*

<description>
   ::= <module>

<module>
   ::= module <name_of_module> <list_of_ports>? ;
               <module_item>*
       endmodule


<name_of_module>
   ::= <IDENTIFIER>

<list_of_ports>
   ::= ( <port> <,<port>>* )
   ||= ( )
<port>
```

```
   ::= <port_expression>?
   ||= . <name_of_port> ( <port_expression>? )

<port_expression>
   ::= <port_reference>
   ||= { <port_reference> <, <port_reference>>* }

<port_reference>
   ::= <name_of_variable>
   ||= <name_of_variable> [ <expression> ]
   ||= <name_of_variable> [ <expression> : <expression> ]

<name_of_port>
   ::= <IDENTIFIER>

<name_of_variable>
   ::= <IDENTIFIER>

<module_item>
   ::= <parameter_declaration>
   ||= <input_declaration>
   ||= <output_declaration>
   ||= <inout_declaration>
   ||= <net_declaration>
   ||= <reg_declaration>
   ||= <integer_declaration>
   ||= <gate_instantiation>
   ||= <module_instantiation>
   ||= <continuous_assign>
   ||= <function>

<function>
   ::= function <range>? <name_of_function> ;
            <func_declaration>*
            <statement_or_null>
       endfunction

<name_of_function>
   ::= <IDENTIFIER>

<func_declaration>
   ::= <parameter_declaration>
```

```
      ||= <input_declaration>
      ||= <reg_declaration>
      ||= <integer_declaration>

<always>
      ::= always @ ( <identifier> or <identifier> )
      ||= always @ ( posedge <identifier> )
      ||= always @ ( negedge <identifier> )
      ||= always @ ( <edge> or <edge> or ... )

<edge>
      ::= posedge <identifier>
      ||= negedge <identifier>

<parameter_declaration>
      ::= parameter <range>? <list_of_assignments> ;

<input_declaration>
      ::= input <range>? <list_of_variables> ;

<output_declaration>
      ::= output <range>? <list_of_variables> ;

<inout_declaration>
      ::= inout <range>? <list_of_variables> ;

<net_declaration>
     ::= <NETTYPE> <charge_strength>? <expandrange>? <delay>?
<list_of_variables> ;
     ||= <NETTYPE> <drive_strength>? <expandrange>? <delay>?
<list_of_assignments> ;

<NETTYPE>
      ::= wire
      ||= wor
      ||= wand
      ||= tri

<expandrange>
      ::= <range>
      ||= scalared <range>
      ||= vectored <range>
```

```
<reg_declaration>
   ::= reg <range>? <list_of_register_variables> ;


<integer_declaration>
   ::= integer <list_of_integer_variables> ;
<continuous_assign>
   ::= assign <drive_strength>? <delay>?
              <list_of_assignments>;

<list_of_variables>
   ::= <name_of_variable> <, <name_of_variable>>*

<name_of_variable>
   ::= <IDENTIFIER>

<list_of_register_variables>
   ::= <register_variable> <, <register_variable>>*

<register_variable>
   ::= <IDENTIFIER>

<list_of_integer_variables>
   ::= <integer_variable> <, <integer_variable>>*

<integer_variable>
   ::= <IDENTIFIER>

<charge_strength>
   ::= ( small )
   ||= ( medium )
   ||= ( large )

<drive_strength>
   ::= ( <STRENGTH0> , <STRENGTH1> )
   ||= ( <STRENGTH1> , <STRENGTH0> )

<STRENGTH0>
   ::= supply0
   ||= strong0
   ||= pull0
```

```
   ||= weak0
   ||= highz0

<STRENGTH1>
   ::= supply1
   ||= strong1
   ||= pull1
   ||= weak1
   ||= highz1

<range>
   ::= [ <expression> : <expression> ]

<list_of_assignments>
   ::= <assignment> <, <assignment>>*

<gate_instantiation>
   ::= <GATETYPE> <drive_strength>? <delay>?
            <gate_instance> <, <gate_instance>>* ;

<GATETYPE>
   ::= and
   ||= nand
   ||= or
   ||= nor
   ||= xor
   ||= xnor
   ||= buf
   ||= not

<gate_instance>
   ::= <name_of_gate_instance>? ( <terminal>
                        <, <terminal>>* )

<name_of_gate_instance>
   ::= <IDENTIFIER>

<terminal>
   ::= <identifier>
   ||= <expression>

<module_instantiation>
```

```
   ::= <name_of_module> <parameter_value_assignment>?
       <module_instance> <, <module_instance>>* ;

<name_of_module>
   ::= <IDENTIFIER>

<parameter_value_assignment>
   ::= #( <expression> <,<expression>>*)

<module_instance>
   ::= <name_of_module_instance>
       ( <list_of_module_terminals>? )

<name_of_module_instance>
   ::= <IDENTIFIER>

<list_of_module_terminals>
   ::= <module_terminal>? <,<module_terminal>>*
   ||= <named_port_connection> <,<named_port_connection>>*

<module_terminal>
   ::= <identifier>
   ||= <expression>

<named_port_connection>
   ::= . IDENTIFIER ( <identifier> )
   ||= . IDENTIFIER ( <expression> )

<statement>
   ::= <assignment>
   ||= if ( <expression> )
          <statement_or_null>
   ||= if ( <expression> )
          <statement_or_null>
       else
          <statement_or_null>
   ||= case ( <expression> )
          <case_item>+
       endcase
   ||= casex ( <expression> )
          <case_item>+
       endcase
```

```
   ||= casez ( <expression> )
          <case_item>+
       endcase
   ||= for ( <assignment> ; <expression> ; <assignment> )
          <statement>
   ||= <seq_block>
   ||= disable <IDENTIFIER> ;
   ||= forever <statement>
   ||= while ( <expression> ) <statement>

<statement_or_null>
   ::= statement
   ||= ;

<assignment>
   ::= <lvalue> = <expression>
<case_item>
   ::= <expression> <,<expression>>* :
<statement_or_null>
   ||= default : <statement_or_null>
   ||= default <statement_or_null>

<seq_block>
   ::= begin
           <statement>*
       end
   ||= begin : <name_of_block>
           <block_declaration>*
           <statement>*
       end

<name_of_block>
   ::= <IDENTIFIER>

<block_declaration>
   ::= <parameter_declaration>
   ||= <reg_declaration>
   ||= <integer_declaration>

<lvalue>
   ::= <IDENTIFIER>
   ||= <IDENTIFIER> [ <expression> ]
```

```
      ||= <concatenation>

<expression>
   ::= <primary>
   ||= <UNARY_OPERATOR> <primary>
   ||= <expression> <BINARY_OPERATOR>
   ||= <expression> ? <expression> : <expression>

<UNARY_OPERATOR>
   ::= !
   ||= ~
   ||= &
   ||= ~&
   ||= |
   ||= ~|
   ||= ^
   ||= ~^
   ||= -
   ||= +

<BINARY_OPERATOR>
   ::=  +
   ||=  -
   ||=  *
   ||=  /
   ||=  %
   ||=  ==
   ||=  !=
   ||=  &&
   ||=  ||
   ||=  <
   ||=  <=
   ||=  >
   ||=  >=
   ||=  &
   ||=  |
   ||=  <<
   ||=  >>

<primary>
   ::= <number>
   ||= <identifier>
```

```
   ||= <identifier> [ <expression> ]
   ||= <identifier> [ <expression> : <expression> ]
   ||= <concatenation>
   ||= <multiple_concatenation>
   ||= <function_call>
   ||= ( <expression> )

<number>
   ::= <NUMBER>
   ||= <BASE> <NUMBER>
   ||= <SIZE> <BASE> <NUMBER>

<NUMBER>
```

A number can have any of these characters: 0123456789abcdefxzABCDEFXZ.

```
<SIZE>
   ::= 'b
   ||= 'B
   ||= 'o
   ||= 'O
   ||= 'd
   ||= 'D
   ||= 'h
   ||= 'H

<SIZE>
```

A size can have any number of these digits: 0123456789

```
<concatenation>
   ::= { <expression> <,<expression>>* }

<multiple_concatenation>
   ::= { <expression> { <expression> <,<expression>>* } }

<function_call>
   ::= <name_of_function> ( <expression> <,<expression>>*)

<name_of_function>
```

```
   ::= <IDENTIFIER>

<identifier>
```

An identifier is any sequence of letters, digits, and the underscore character (_), where the first character is a letter or an underscore. Uppercase and lowercase letters are treated as different characters. Identifiers can be any size, and all characters are significant. Escaped identifiers start with the backslash character (\) and end with a space. The leading backslash character (\) is not part of the identifier. Use escaped identifiers to include any printable ASCII characters in an identifier.

```
<delay>
   ::= # <NUMBER>
   ||= # <identifier>
   ||= # ( <expression> <,<expression>>* )
```

Lexical Conventions

The lexical conventions HDL Compiler uses are nearly identical to those of the Verilog language. The types of lexical tokens HDL Compiler uses are described in the following subsections:

- White Space
- Comments
- Numbers
- Identifiers
- Operators
- Macro Substitution
- include Construct
- Simulation Directives
- Verilog System Functions

White Space

White space separates words in the input description and can contain spaces, tabs, new lines, and form feeds. You can place white space anywhere in the description. HDL Compiler ignores white space.

Comments

You can enter comments anywhere in a Verilog description, in two forms:

- Beginning with two slashes //

  HDL Compiler ignores all text between these characters and the end of the current line.

- Beginning with the two characters /* and ending with */

  HDL Compiler ignores all text between these characters, so you can continue comments over more than one line.

  Note:

  You cannot nest comments.

Numbers

You can declare numbers in several different radices and bit-widths. A radix is the base number on which a numbering system is built. For example, the binary numbering system has a radix of 2, octal has a radix of 8, and decimal has a radix of 10.

You can use these three number formats:

- A simple decimal number that is a sequence of digits in the range of 0 to 9. All constants declared this way are assumed to be 32-bit numbers.
- A number that specifies the bit-width as well as the radix. These numbers are the same as those in the previous format, except that they are preceded by a decimal number that specifies the bit-width.

- A number followed by a two-character sequence prefix that specifies the number's size and radix. The radix determines which symbols you can include in the number. Constants declared this way are assumed to be 32-bit numbers. Any of these numbers can include underscores (_), which improve readability and do not affect the value of the number. Table B-1 summarizes the available radices and valid characters for the number.

Table B-1 Verilog Radices

Name	Character prefix	Valid characters
Binary	'b	0 1 x X z Z _ ?
Octal	'o	0–7 x X z Z _ ?
Decimal	'd	0–9 _
Hexadecimal	'h	0–9 a–f A–F x X z Z _ ?

Example B-1 shows some valid number declarations.

Example B-1 Valid Verilog Number Declarations

```
391               //  32-bit decimal number
'h3a13            //  32-bit hexadecimal number
10'o1567          //  10-bit octal number
3'b010            //  3-bit binary number
4'd9              //  4-bit decimal number
40'hFF_FFFF_FFFF  //  40-bit hexadecimal number
2'bxx             //  2-bits don't care
3'bzzz            //  3-bits high-impedance
```

Identifiers

Identifiers are user-defined words for variables, function names, module names, and instance names. Identifiers can be composed of letters, digits, and the underscore character (_). The first character of an identifier cannot be a number. Identifiers can be any length. Identifiers are case-sensitive, and all characters are significant.

Identifiers that contain special characters, begin with numbers, or have the same name as a keyword can be specified as an escaped identifier. An escaped identifier starts with the backslash character (\), followed by a sequence of characters, followed by white space.

Some escaped identifiers are shown in Example B-2.

Example B-2 Sample Escaped Identifiers

```
\a+b                    \3state
\module                 \(a&b)|c
```

The Verilog language supports the concept of hierarchical names, which can be used to access variables of submodules directly from a higher-level module. These are partially supported by HDL Compiler. (For more information, see “Unsupported Verilog Language Constructs” on page B-21.)

Operators

Operators are one- or two-character sequences that perform operations on variables. Some examples of operators are +, ~^, <=, and >>. Operators are described in detail in “Operators” on page 4-3.

Macro Substitution

Macro substitution assigns a string of text to a macro variable. The string of text is inserted into the code where the macro is encountered. The definition begins with the back quotation mark (‘), followed by the keyword `define`, followed by the name of the macro variable. All text from the macro variable until the end of the line is assigned to the macro variable.

You can declare and use macro variables anywhere in the description. The definitions can carry across several files that are read into Design Compiler at the same time. To make a macro substitution, type a back quotation mark (‘) followed by the macro variable name.

Some sample macro variable declarations are shown in Example B-3.

Example B-3 Macro Variable Declarations

```
‘define highbits      31:29
‘define bitlist       {first, second, third}
wire [31:0] bus;
‘bitlist = bus[‘highbits];
```

Text macros are not supported when used with sized constants, as shown in Example B-4.

Example B-4 Macro With Sized Constants

```
‘define SIZE 4

module test (in,out);
output [3:0] out;
input [3:0] in;

assign out = ‘SIZE’b0101; //text macro from ‘define statement
                   //cannot be used with a sized constant

endmodule
```

include Construct

The `include` construct in Verilog is similar to the `#include` directive in C. You can use this construct to include Verilog code, such as type declarations and functions, from one module in another module. Example B-5 shows an application of the `include` construct.

Example B-5 Including a File Within a File

```
Contents of file1.v

‘define WORDSIZE 8
function [WORDSIZE-1:0] fastadder;
.
.
endfunction
Contents of secondfile

module secondfile (in1,in2,out)
‘include ”file1.v”
wire [WORDSIZE-1:0] temp;
assign temp = fastadder (in1,in2);
.
.
endmodule
```

Included files can include other files, with up to 24 levels of nesting. You cannot use the `include` construct recursively. Set the include directory with the `search_path` variable in dc_shell.

Simulation Directives

Simulation directives refer to special commands that affect the operation of the Verilog HDL Simulator. You can include these directives in your design description, because HDL Compiler parses and ignores them:

```
‘accelerate
‘celldefine
‘default_nettype
‘endcelldefine
‘endprotect
‘expand_vectornets
‘noaccelerate
‘noexpand_vectornets
‘noremove_netnames
‘nounconnected_drive
‘protect
‘remove_netnames
‘resetall
‘timescale
‘unconnected_drive
```

Verilog System Functions

Verilog system functions are special functions Verilog HDL Simulators implement to generate input or output during simulation. Their names start with a dollar sign ($). These functions are parsed and ignored by HDL Compiler.

Verilog Keywords

Verilog uses keywords, shown in Table B-2, to interpret an input file. You cannot use these words as user variable names unless you use an escaped identifier. For more information, see “Identifiers” on page B-16.

Table B-2 Verilog Keywords

always	force	or	trireg
and	forever	output	table
assign	fork	parameter	task
begin	function	pmos	time
buf	highz0	posedge	tran
bufif0	highz1	primitive	tranif0
bufif1	if	pull0	tranif1
case	initial	pull1	tri
casex	inout	rcmos	triand
casez	input	reg	tri0
cmos	integer	release	tri1
deassign	join	repeat	vectored
default	large	rnmos	wait
defparam	medium	rpmos	wand
disable	module	rtran	weak0
end	nand	rtranif0	weak1
endcase	negedge	rtranif1	while

Table B-2 Verilog Keywords (continued)

endfunction	nmos	scalared	wire
endmodule	nor	small	wor
endprimitive	not	strong0	xnor
endtable	notif0	strong1	xor
endtask	notif1	supply0	
event	pulldown	supply1	
for	pullup	trior	

Unsupported Verilog Language Constructs

HDL Compiler does not support the following Verilog constructs:

- Unsupported definitions and declarations
  - primitive definition
  - `time` declaration
  - `event` declaration
  - `triand`, `trior`, `tri1`, `tri0`, and `trireg` net types
  - Ranges and arrays for integers
- Unsupported statements
  - `defparam` statement
  - `initial` statement
  - `repeat` statement

  - delay control
  - event control
  - `wait` statement
  - `fork` statement
  - `deassign` statement
  - `force` statement
  - `release` statement
- Unsupported operators
  - Case equality and inequality operators (=== and !==)
  - Division and modulus operators for variables
- Unsupported gate-level constructs
  - nmos, pmos, cmos, rnmos, rpmos, rcmos
  - pullup, pulldown, tranif0, tranif1, rtran, rtrainf0, and rtrainf1 gate types
- Unsupported miscellaneous constructs, such as hierarchical names within a module

Constructs added to the Verilog Simulator in versions after Verilog 1.6 might not be supported.

If you use an unsupported construct in a Verilog description, HDL Compiler issues a syntax error such as

```
event is not supported
```