22. Compiler directives

22.1 General

This clause describes the following compiler directives (listed alphabetically):

22.2 Overview

All compiler directives are preceded by the (`) character. This character is called *grave accent* (ASCII 0x60). It is different from the character (') , which is the *apostrophe* character (ASCII 0x27). The scope of a compiler directive extends from the point where it is processed, across all files processed in the current compilation unit, to the point where another compiler directive supersedes it or the processing of the compilation unit completes. The semantics of compiler directives is defined in 3.12.1 and 5.6.4.

Unless otherwise specified below, each directive whose syntax shows a defined end to the directive may be followed by another valid language element on the same line as the end. For directives that include an indeterminate amount of text (`` `define ``, `` `ifdef ``, `` `pragma ``), the end is specified in the corresponding section below. Subsequent text on the same line is allowed as long as the end does not require a newline.

A language element is allowed on the same line before a directive as long as all other requirements are satisfied for that element and for the placement of the directive.

A compiler directive may appear within a conditional compilation block of text (see 22.6) or within the macro text in a text macro definition (see 22.5.1). Such a directive shall be processed if the block of text is not ignored or where the text macro is used. Otherwise, a compiler directive shall not appear in the middle of another directive (a macro usage is not considered a directive).

Macro expansion shall occur within a compiler directive. Directives are not recognized within comments or string literals.

Unless specified below as supporting multiple lines, compiler directives shall be all on one line.

22.3 `resetall

When the `resetall compiler directive is encountered during compilation, all compiler directives are set to the default values. This is useful for ensuring that only directives that are desired in compiling a particular source file are active.

Not all compiler directives have a default value (e.g., `define and `include). Directives that do not have a default are not affected by `resetall.

It shall be illegal for the `resetall directive to be specified within a design element.

22.4 `include

The file inclusion (`include) compiler directive is used to insert the entire contents of a source file in another file during compilation. The result is as though the contents of the included source file appear in place of the `include compiler directive.

The syntax of the `include compiler directive is given in Syntax 22-1.

```
include_compiler_directive ::=
    `include " filename "
  | `include < filename >
```

Syntax 22-1—Syntax for include compiler directive (not in Annex A)

The compiler directive `include can be specified anywhere within the SystemVerilog source description.

Only white space or a comment may appear on the same line as the `include compiler directive.

The *filename* is the name of the file to be included in the source file. The *filename* can be a full or relative path name.

The *filename* can be enclosed in either quotes or angle brackets, which affects how a tool searches for the file, as follows:

- When the *filename* is enclosed in double quotes ("*filename*"), for a relative path the compiler's current working directory, and optionally user-specified locations are searched.
- When the *filename* is enclosed in angle brackets (<*filename*>), then only an implementation-dependent location containing files defined by the language standard is searched. Relative path names are interpreted relative to that location.

When the *filename* is an absolute path, only that *filename* is included and only the double quote form of the `include can be used.

A file included in the source using the `include compiler directive may contain other `include compiler directives. The number of nesting levels for include files shall be finite. Implementations may limit the maximum number of levels to which include files can be nested, but the limit shall be at least 15.

Examples of `include compiler directives:

```
`include "parts/count.v"
`include "fileB" // including fileB
`include <List.vh>
```

22.5 `define, `undef, and `undefineall

A text macro substitution facility has been provided so that meaningful names can be used to represent commonly used pieces of text. For example, in the situation where a constant number is repetitively used throughout a description, a text macro would be useful in that only one place in the source description would need to be altered if the value of the constant needed to be changed.

The text macro facility is not affected by the compiler directive `` `resetall ``.

22.5.1 `define

The directive `` `define `` creates a macro for text substitution. This directive can be used both inside and outside design elements. After a text macro is defined, it can be used in the source description by using the (`) character, followed by the macro name. The compiler shall substitute the text of the macro for the string `` `text_macro_name `` and any actual arguments that follow it.

The macro name may be either a *simple_identifier* or an *escaped_identifier* (see 5.6). It shall be illegal to redefine a compiler directive as a macro name.

A text macro can be defined with arguments. This allows the macro to be customized for each use individually.

The syntax for text macro definitions is given in Syntax 22-2.

---

text_macro_definition ::= **`define** text_macro_name macro_text
text_macro_name ::= text_macro_identifier [**(** list_of_formal_arguments **)**]
list_of_formal_arguments ::= formal_argument { **,** formal_argument }
formal_argument ::= simple_identifier [**=** default_text]
text_macro_identifier ::= identifier

Syntax 22-2—Syntax for text macro definition (not in Annex A)

For example:

```
`define wordsize 8
logic [1:`wordsize] data;

//define a nand with variable delay
`define var_nand(dly) nand #dly

`var_nand(2) g121 (q21, n10, n11);
`var_nand(5) g122 (q22, n10, n11);
```

The macro text can be any arbitrary text specified on the same line as the text macro name. If more than one line is necessary to specify the text, the newline character shall be immediately preceded by a backslash (\ ). A newline character that is not contained in a triple-quoted string literal or in a block comment and is not preceded by a backslash shall end the macro text. The newline character preceded by a backslash shall be replaced in the expanded macro with a newline character (but without the preceding backslash character).

An exception to the previous sentence is that if the backslash-newline character sequence occurs in the middle of a string literal, both the backslash and the newline characters are omitted in the expanded macro (see 5.9).

When formal arguments are used to define a text macro, the scope of the formal argument shall extend up to the end of the macro text. A formal argument can be used in the macro text in the same manner as an identifier.

If formal arguments are used, the list of formal argument names shall be enclosed in parentheses immediately following the name of the macro. If the macro name is a *simple_identifier*, no white space shall separate the opening parenthesis from the macro name. If the macro name is an *escaped_identifier*, only the single white space character terminating the identifier name (see 5.6.1) shall separate the opening parenthesis from the macro name. The formal argument names shall be *simple_identifiers*, separated by commas and optionally white space.

A formal macro argument may have a default. A default is specified by appending an = token after the formal argument name, followed by the default text. The default text is substituted for the formal argument if no corresponding actual argument is specified.

The default text may be explicitly specified to be empty by adding an = token after the formal argument name, followed by a comma (or a right parenthesis if it is the last argument in the argument list).

If a one-line comment or block comment (see 5.4) is included in the text, then the comment shall not become part of the substituted text. If a one-line comment is followed by a backslash and newline character, the comment ends before the backslash and the macro text continues on the next line. If the macro text contains a multi-line block comment, the newline characters in the block comment are not required to be preceded by a backslash.

Example:

```
`define var_nand(dly) nand #dly // define a nand with variable delay
`var_nand(2) g121 (q21, n10, n11);

`define var_nand(dly) nand      // define a nand with variable delay \
                                /* this is a block comment
                                   embedded in a multi-line macro */ \
                           #dly // this is the end of the macro definition

`var_nand(2) g121 (q21, n10, n11);
```

Newline characters within a triple-quoted sting literal used in a macro will not terminate the macro definition. For example:

```
module main;

`define TEST """
many
many
more
lines""" // end of macro

initial $display(`TEST);

endmodule
```

will print:

```
many
many
more
lines
```

The macro text can be blank, in which case the text macro is defined to be empty and no text is substituted when the macro is used.

The syntax for using a text macro is given in Syntax 22-3.

---

text_macro_usage ::= `text_macro_identifier [(list_of_actual_arguments)]
list_of_actual_arguments ::= actual_argument { , actual_argument }
actual_argument ::= expression

Syntax 22-3—Syntax for text macro usage (not in Annex A)

For a macro without arguments, the text shall be substituted as is for every occurrence of `text_macro_identifier. However, a text macro with one or more arguments shall be expanded by substituting each formal argument with the expression used as the actual argument in the macro usage.

To use a macro defined with arguments, the name of the text macro shall be followed by a list of actual arguments in parentheses, separated by commas. Actual arguments and defaults shall not contain comma or right parenthesis characters outside matched pairs of left and right parentheses (), square brackets [], braces {}, double quotes " ", triple quotes """ """, or an escaped identifier.

White space shall be allowed between the text macro name and the left parenthesis in the macro usage. If the text macro name is an escaped identifier, then white space shall be required.

An actual argument may be empty or white space only, in which case the formal argument is substituted by the argument default if one is specified or by nothing if no default is specified.

If fewer actual arguments are specified than the number of formal arguments and all the remaining formal arguments have defaults, then the defaults are substituted for the additional formal arguments. It shall be an error if any of the remaining formal arguments does not have a default specified. For a macro with arguments, the parentheses are always required in the macro call, even if all the arguments have defaults. It shall be an error to specify more actual arguments than the number of formal arguments.

Example macro without defaults:

```
`define D(x,y) initial $display("start", x , y, "end");

`D( "msg1" , "msg2" )
   // expands to 'initial $display("start", "msg1" , "msg2", "end");'
`D( " msg1", )
   // expands to 'initial $display("start", " msg1" , , "end");'
`D(, "msg2 ")
   // expands to 'initial $display("start",  , "msg2 ", "end");'
`D(,)
   // expands to 'initial $display("start",  , , "end");'
`D(  ,  )
   // expands to 'initial $display("start",  , , "end");'
`D("msg1")
   // illegal, only one argument
```

```
`D()
   // illegal, only one empty argument
`D(,,)
   // illegal, more actual than formal arguments
```

Example macros with defaults:

```
`define MACRO1(a=5,b="B",c) $display(a,,b,,c);

`MACRO1 ( , 2, 3 )  // argument a omitted, replaced by default
                    // expands to '$display(5,,2,,3);'
`MACRO1 ( 1 , , 3 ) // argument b omitted, replaced by default
                    // expands to '$display(1,,"B",,3);'
`MACRO1 ( , 2, )    // argument c omitted, replaced by nothing
                    // expands to '$display(5,,2,,);'
`MACRO1 ( 1 )       // ILLEGAL: b and c omitted, no default for c


`define MACRO2(a=5, b, c="C") $display(a,,b,,c);

`MACRO2 (1, , 3)    // argument b omitted, replaced by nothing
                    // expands to '$display(1,,,,3);'
`MACRO2 (, 2, )     // a and c omitted, replaced by defaults
                    // expands to '$display(5,,2,,"C");'
`MACRO2 (, 2)       // a and c omitted, replaced by defaults
                    // expands to '$display(5,,2,,"C");'

// Example of escaped identifier as macro name. Single white space
// required between macro name and left parenthesis in macro definition.

`define \M@CRO3 (a=5, b=0, c="C") $display(a,,b,,c);

`\M@CRO3 ( 1 )       // b and c omitted, replaced by defaults
                    // expands to '$display(1,,0,,"C");'
`\M@CRO3 ( )         // all arguments replaced by defaults
                    // expands to '$display(5,,0,,"C");'
`\M@CRO3             // ILLEGAL: parentheses required
```

Once a text macro name has been defined, it can be used anywhere in the compilation unit where it is defined. There are no other scope restrictions once inside the compilation unit.

The text specified for macro text shall not be split across the following lexical tokens:

- Comments
- Numbers
- String literals
- Identifiers
- Keywords
- Operators

The following is illegal syntax because it is split across a string:

```
`define first_half "start of string
$display(`first_half end of string");
```

Each actual argument is substituted for the corresponding formal argument literally. Therefore, when an expression is used as an actual argument, the expression will be substituted in its entirety. This may cause an expression to be evaluated more than once if the formal argument was used more than once in the macro text. For example:

```
`define max(a,b)((a) > (b) ? (a) : (b))
n = `max(p+q, r+s);
```

will expand as

```
n = ((p+q) > (r+s) ? (p+q) : (r+s));
```

Here, the larger of the two expressions `p + q` and `r + s` will be evaluated twice.

The word `define` is known as a *compiler directive keyword,* and it is not part of the normal set of keywords. Thus, normal identifiers in a SystemVerilog source description can be the same as compiler directive keywords. The following problems should be considered:

a) Text macro names shall not be the same as compiler directive keywords.
b) Text macro names can reuse names being used as ordinary identifiers. For example, `signal_name` and `` `signal_name `` are different.
c) Redefinition of text macros is allowed; the latest definition of a particular text macro read by the compiler prevails when the macro name is encountered in the source text.

The macro text and argument defaults may contain usages of other text macros. Such usages shall be substituted after the outer macro is substituted, not when it is defined. If an actual argument or an argument default contains a macro usage, the macro usage shall be expanded only after substitution into the outer macro text.

Any compiler directives appearing within the macro text shall be ignored until the macro is used.

If a formal argument has a nonempty default and one wants to replace the formal argument with an empty actual argument, one cannot simply omit the actual argument, as then the default will be used. However, one can define an empty text macro, say `` `EMPTY ``, and use that as the actual argument. This will be substituted in place of the formal argument and will be replaced by empty text after expansion of the empty text macro.

When a macro usage is passed as an actual argument or a default to another macro, the argument expansion does not introduce new uses of the formal arguments to the top-level macro.

Example:

```
`define TOP(a,b) a + b
`TOP( `TOP(b,1), `TOP(42,a) )
```

expands to: `b + 1 + 42 + a`
not into: `42 + a + 1 + 42 + a`
nor into: `b + 1 + 42 + b + 1`

It shall be an error for a macro to expand directly or indirectly to text containing another usage of itself (a recursive macro). However, an actual argument to a macro or a default may contain a usage of itself, as in the previous example.

Macro substitution and argument substitution shall not occur within string literals. For example,

```
module main;
```

```
    `define HI Hello
    `define LO "`HI, world"
    `define H(x) "Hello, x"
    initial begin
       $display("`HI, world");
       $display(`LO);
       $display(`H(world));
    end
endmodule
```

will print:

```
`HI, world
`HI, world
Hello, x
```

The `` `define `` macro text can also include `` `" ``, `` `\`" ``, and `` `` ``.

A `` `" `` overrides the usual lexical meaning of `"` and indicates that the expansion shall include the quotation mark, substitution of actual arguments, and expansions of embedded macros. This allows string literals to be constructed from macro arguments.

Similar to a `` `" ``, a `` `""" `` overrides the usual meaning of `"""`, indicating that the expansion shall include the triple quotation mark, substitution of actual arguments, and expansions of embedded macros. Unlike triple-quoted string literals, the newline characters within the expansion of a `` `""" `` will terminate the macro definition. An exception to the previous sentence is that if the backslash-newline character sequence occurs in the middle of a `` `""" `` expansion, the backslash is omitted in the expanded macro, while the newline is not. For example:

```
`define MACRO4(VAL) `"""line1\
line2 - VAL\
line3 - backslash\\\
line4`""" // end of macro

$display(`TEST(FOO));
```

will print

```
line1
line2 - FOO
line3 - backslash\
line4
```

Note that the first two backslashes on `line3` in the example are unaffected during macro expansion; only the third backslash is omitted. The remaining two backslashes are an escaped backslash within the triple-quoted string literal in the expanded text. The result is a single backslash displayed in the final output.

A mixture of `` `" `` and `"` or `` `""" `` and `"""` is allowed in the macro text, but the use of `"` or `"""` always starts a string literal and shall have a corresponding terminating delimiter. Any characters embedded inside this string literal, including `` ` ``, become part of the string in the replaced macro text. Thus, if `"` is followed by `` `" ``, the `"` starts a string literal whose last character is `` ` `` and is terminated by the `"` of `` `" ``.

A `` `\`" `` indicates that the expansion should include the escape sequence `\"`. For example:

```
`define msg(x,y) `"x: `\`"y`\`"`"
```

An example of using this `msg macro is:

```
$display(`msg(left side,right side));
```

The preceding example expands to:

```
$display("left side: \"right side\"");
```

A `` delimits lexical tokens without introducing white space, allowing identifiers to be constructed from arguments. For example:

```
`define append(f) f``_primary
```

An example of using this `append macro is:

```
`append(clock)
```

This example expands to:

```
clock_primary
```

Here is an example of the `include directive being followed by a macro, instead of by a string literal:

```
`define home(filename) `"/home/mydir/filename`"
`include `home(myfile)
```

22.5.2 `undef

The directive `undef shall undefine the specified text macro if previously defined by a `define compiler directive within the compilation unit. An attempt to undefine a text macro that was not previously defined using a `define compiler directive can issue a warning. The syntax for the `undef compiler directive is given in Syntax 22-4.

---

undefine_compiler_directive ::=
    `undef text_macro_identifier

Syntax 22-4—Syntax for undef compiler directive (not in Annex A)

An undefined text macro has no value, just as if it had never been defined.

22.5.3 `undefineall

The `undefineall directive shall undefine all text macros previously defined by `define compiler directives within the compilation unit. This directive takes no arguments and may appear anywhere in the source description.

22.6 `ifdef, `else, `elsif, `endif, `ifndef

These conditional compilation compiler directives are used to include optionally blocks of text of SystemVerilog source description during compilation. These directives may appear anywhere in the source description.

Situations where the conditional compilation compiler directives may be useful include the following:

— Selecting different representations of a design element such as behavioral, structural, or switch level
— Choosing different timing or structural information
— Selecting different stimulus for a given run

The conditional compilation compiler directives have the syntax shown in Syntax 22-5.

---

conditional_compilation_directive ::=
    ifdef_or_ifndef ifdef_condition block_of_text
    { `**elsif** ifdef_condition block_of_text }
    [`**else** block_of_text]
    `**endif**

ifdef_or_ifndef ::= **`ifdef** | **`ifndef**

ifdef_condition ::=
    text_macro_identifier
  | **(** ifdef_macro_expression **)**

ifdef_macro_expression ::=
    text_macro_identifier
  | ifdef_macro_expression binary_logical_operator ifdef_macro_expression
  | **!** ifdef_macro_expression
  | **(** ifdef_macro_expression **)**

binary_logical_operator ::= **&&** | **||** | **->** | **<->**

Syntax 22-5—Syntax for conditional compilation directives (not in Annex A)

The *text_macro_identifier* is a SystemVerilog *identifier*. The blocks of text are parts of a SystemVerilog source description. The `else and `elsif compiler directives and all of the blocks of text are optional.

The `ifndef *text_macro_identifier* directive is treated as `ifdef (!*text_macro_identifier*), and the `ifndef (*ifdef_macro_expression*) directive is treated as `ifdef (!(*ifdef_macro_expression*)). Consequently, all of the description of `ifdef below also applies to `ifndef.

The blocks of text consist of all text up to but not including a subsequent directive from this category as follows:

— The *block_of_text* after `ifdef or `elsif terminates when `elsif, `else, or `endif is seen.
— The *block_of_text* after `else terminates when `endif is seen.
— A directive within a comment, string literal, or text macro definition is hidden and shall not terminate a *block_of_text*.

White space is not considered when determining termination, and there are no extra white space requirements around the directives beyond serving as token separators (see 5.2).

The *ifdef_macro_expression* may contain the following:

— identifiers (see 5.6)
— logical operators (see 11.4.7)
— parentheses

When the *ifdef_macro_expression* is evaluated, all identifiers are replaced with 1 if that identifier represents a currently defined text macro (see 22.5) or with 0 otherwise; then the expression is resolved to 1 or 0 according to the rules in 11.8.

Nesting of conditional compilation constructs shall be permitted. After an `` `ifdef `` directive, if another `` `ifdef `` appears before `` `endif ``, an inner, nested, conditional compilation construct is started. Until the inner conditional compilation construct is terminated with `` `endif ``, all compiler directives apply only to the inner construct and shall not terminate the *block_of_text* in the outer construct.

The conditional compiler directives work together in the following manner:

— If the `` `ifdef `` is followed by a *text_macro_identifier* without parentheses, it succeeds if that identifier is currently defined as a text macro name (see 22.5). If the `` `ifdef `` is followed by an *ifdef_macro_expression* in parentheses, it succeeds if that expression resolves to `1`.
If the evaluation succeeds, the associated *block_of_text* (if specified) is processed as part of the description, and if there are subsequent `` `elsif `` or `` `else `` directives before the next `` `endif ``, these directives and their blocks of text are ignored.
If the evaluation does not succeed, the associated *block_of_text* (if specified) is ignored, and any subsequent `` `elsif `` and `` `else `` directives up to the next `` `endif `` are evaluated as specified below, in the order they appear in the source description.

— If the `` `elsif `` is followed by a *text_macro_identifier* without parentheses, it succeeds if that identifier is currently defined as a text macro name (see 22.5). If the `` `elsif `` is followed by an *ifdef_macro_expression* in parentheses, it succeeds if that expression resolves to `1`.
If the evaluation succeeds, the associated *block_of_text* (if specified) is processed as part of the description, and if there are subsequent `` `elsif `` or `` `else `` directives before the next `` `endif ``, these directives and their blocks of text are ignored.
If the evaluation does not succeed, the associated *block_of_text* (if specified) is ignored, and this step repeats on any subsequent `` `elsif `` directive.

— If there is an `` `else `` compiler directive and none of the previous evaluations succeeded, the `` `else `` *block_of_text* (if specified) is processed as part of the description.

Although the names of compiler directives are contained in the same name space as text macro names, the names of compiler directives are considered not to be defined by `` `ifdef `` and `` `elsif ``.

Any *block_of_text* that the compiler ignores shall still follow the SystemVerilog lexical conventions as specified in Clause 5.

Example 1: The following example shows a simple usage of an `` `ifdef `` directive for conditional compilation. If the identifier `behavioral` is defined, a continuous net assignment will be compiled in; otherwise, an **and** gate will be instantiated.

```
module and_op (a, b, c);
   output a;
   input b, c;

   `ifdef behavioral
      wire a = b & c;
   `else
      and a1 (a,b,c);
   `endif
endmodule
```

Example 2: The following example shows usage of nested conditional compilation directives:

```
module test(out);
   output out;
   `define wow
   `define nest_one
   `define second_nest
```

```
   `define nest_two
   `ifdef wow
         initial $display("wow is defined");
         `ifdef nest_one
            initial $display("nest_one is defined");
            `ifdef nest_two
               initial $display("nest_two is defined");
            `else
               initial $display("nest_two is not defined");
            `endif
         `else
            initial $display("nest_one is not defined");
         `endif
   `else
         initial $display("wow is not defined");
         `ifdef second_nest
            initial $display("second_nest is defined");
         `else
            initial $display("second_nest is not defined");
         `endif
   `endif
endmodule
```

Example 3: The following example shows usage of chained nested conditional compilation directives:

```
module test;
   `ifdef first_block
         `ifndef second_nest
            initial $display("first_block is defined");
         `else
            initial $display("first_block and second_nest defined");
         `endif
   `elsif second_block
         initial $display("second_block defined, first_block is not");
   `else
         `ifndef last_result
            initial $display("first_block, second_block,",
               " last_result not defined.");
         `elsif real_last
            initial $display("first_block, second_block not defined,",
               " last_result and real_last defined.");
         `else
            initial $display("Only last_result defined!");
         `endif
   `endif
endmodule
```

Example 4: The following example shows usage of `` `ifdef `` and `` `elsif `` expressions:

```
module test;
   // define two macros
   // note that the macro text value is irrelevant to the uses in this example
   `define example_def0 0
   `define example_def1

   `ifdef (!example_def0)
      initial $display("this will not print, example_def0 is defined");
   `endif
```

```
    `ifdef (example_def0 && example_def1)
       initial $display("this will print, both && terms are defined");
    `endif
    `ifdef (example_def0 && example_def2)
       initial $display("this will not print, example_def2 is not defined");
    `elsif (!example_def0 || !example_def1)
       initial $display("this will not print, both || terms are defined");
    `else
       initial $display("this will print, preceding evaluations are false");
    `endif
    initial if (`ifdef example_def1 1 `else 0 `endif)
       $display("this will print, example_def1 is defined");
endmodule
```

22.7 `timescale

This directive specifies the time unit and time precision of the design elements that follow it. The time unit is the unit of measurement for time values such as the simulation time and delay values.

It shall be illegal for the `timescale directive to be specified within a design element.

To use design elements with different time units in the same design, the following timescale constructs are useful:

- The **timeunit** and **timeprecision** keywords to specify the unit of measurement for time and precision of time in specific design elements (see 3.14.2.2)
- The `timescale compiler directive to specify the unit of measurement for time and precision of time in the design elements that follow the directive
- The $printtimescale system task to display the time unit and precision of a design element
- The $time and $realtime system functions, the $timeformat system task, and the %t format specification to specify how time information is reported

The `timescale compiler directive specifies the default unit of measurement for time and delay values and the degree of accuracy for delays in all design elements that follow this directive, and that do not have **timeunit** and **timeprecision** constructs specified within the design element, until another `timescale compiler directive is read.

See 3.14.2.3 for the precedence rules of the **timeunit** and **timeprecision** constructs versus the `timescale directive.

If there is no `timescale specified or it has been reset by a `resetall directive, the default time unit and precision are tool-specific.

The syntax for the `timescale directive is given in Syntax 22-6.

```
timescale_compiler_directive ::=
    `timescale time_unit / time_precision
```

Syntax 22-6—Syntax for timescale compiler directive (not in Annex A)

The *time_unit* argument specifies the unit of measurement for times and delays.

The *time_precision* argument specifies how delay values are rounded before being used in simulation.

The *time_precision* argument shall be at least as precise as the *time_unit* argument; it cannot specify a longer unit of time than *time_unit*.

The integers in these arguments specify an order of magnitude for the size of the value; the valid integers are 1, 10, and 100. The character strings represent units of measurement; the valid character strings are `s`, `ms`, `us`, `ns`, `ps`, and `fs`.

See 3.14 for the semantics and effects of *time_unit* and *time_precision*.

The following example shows how this directive is used:

```
`timescale 1 ns / 1 ps
```

Here, all time values in the design elements that follow the directive are multiples of 1 ns because the `time_unit` argument is "1 ns." Delays are rounded to real numbers with three decimal places—or precise to within one thousandth of a nanosecond—because the `time_precision` argument is "1 ps," or one thousandth of a nanosecond.

Consider the following example:

```
`timescale 10 us / 100 ns
```

The time values in the design elements that follow this directive are multiples of `10 us` because the `time_unit` argument is "10 us." Delays are rounded to within one tenth of a microsecond because the `time_precision` argument is "100 ns," or one tenth of a microsecond.

The following example shows a `` `timescale `` directive in the context of a module:

```
`timescale 10 ns / 1 ns
module test;
   logic set;
   parameter d = 1.55;

   initial begin
      #d set = 0;
      #d set = 1;
   end
endmodule
```

The `` `timescale 10 ns / 1 ns `` compiler directive specifies that the time unit for module `test` is 10 ns. As a result, the time values in the module are multiples of 10 ns, rounded to the nearest 1 ns; therefore, the value stored in parameter `d` is scaled to a delay of 16 ns. In other words, the value 0 is assigned to variable `set` at simulation time 16 ns (1.6 × 10 ns), and the value 1 at simulation time 32 ns.

Parameter `d` retains its value no matter what timescale is in effect.

These simulation times are determined as follows:

a) The value of parameter `d` is rounded from 1.55 to 1.6 according to the time precision.

b) The time unit of the module is 10 ns, and the precision is 1 ns; therefore, the delay of parameter `d` is scaled from 1.6 to 16.

c) The assignment of 0 to variable `set` is scheduled at simulation time 16 ns, and the assignment of 1 at simulation time 32 ns. The time values are not rounded when the assignments are scheduled.

22.8 `default_nettype

The directive `default_nettype controls the net type created for implicit net declarations (see 6.10). It can be used only outside design elements. Multiple `default_nettype directives are allowed. The latest occurrence of this directive in the source controls the type of nets that will be implicitly declared. Syntax 22-7 contains the syntax of the directive.

---

default_nettype_compiler_directive ::=
    `default_nettype default_nettype_value
default_nettype_value ::= **wire** | **tri** | **tri0** | **tri1** | **wand** | **triand** | **wor** | **trior** | **trireg** | **uwire** | **none**

Syntax 22-7—Syntax for default_nettype compiler directive (not in Annex A)

When no `default_nettype directive is present or if the `resetall directive is specified, implicit nets are of type **wire**. When the `default_nettype is set to none, all nets shall be explicitly declared. If a net is not explicitly declared, an error is generated.

22.9 `unconnected_drive and `nounconnected_drive

All unconnected input ports of a module, program or interface appearing between the directives `unconnected_drive and `nounconnected_drive are pulled up or pulled down instead of the normal default.

The directive `unconnected_drive takes one of two arguments—**pull1** or **pull0**. When **pull1** is specified, all unconnected input ports are automatically pulled up. When **pull0** is specified, unconnected ports are pulled down. It is advisable to pair each `unconnected_drive with a `nounconnected_drive, but it is not required; these are two independent directives. The most recent occurrence of either directive in the source controls what happens to unconnected ports. These directives shall be specified outside the design element declarations.

The `resetall directive includes the effects of a `nounconnected_drive directive.

22.10 `celldefine and `endcelldefine

The directives `celldefine and `endcelldefine tag modules as *cell modules*. Cells are used by certain PLI routines and may be useful for applications such as delay calculations. It is advisable to pair each `celldefine with an `endcelldefine, but it is not required; these are two independent directives. The most recent occurrence of either directive in the source controls whether modules are tagged as cell modules. More than one of these pairs may appear in a single source description.

These directives may appear anywhere in the source description, but it is recommended that the directives be specified outside any design elements.

The `resetall directive includes the effects of a `endcelldefine directive.

22.11 `pragma

The `pragma directive is a structured specification that alters interpretation of the SystemVerilog source. The specification introduced by this directive is referred to as a *pragma*. The effect of pragmas other than those specified in this standard is implementation-specific. The syntax for the `pragma directive is given in Syntax 22-8.

```
pragma ::=
    `pragma pragma_name [ pragma_expression { , pragma_expression } ]
pragma_name ::= simple_identifier
pragma_expression ::=
      pragma_keyword
    | pragma_keyword = pragma_value
    | pragma_value
pragma_value ::=
      ( pragma_expression { , pragma_expression } )
    | number
    | string
    | identifier
pragma_keyword ::= simple_identifier
```

Syntax 22-8—Syntax for pragma compiler directive (not in Annex A)

The pragma specification is identified by the *pragma_name*, which follows the `pragma directive. The *pragma_name* is followed by an optional list of *pragma_expressions*, which qualify the altered interpretation indicated by the *pragma_name*. Unless otherwise specified, pragma directives for *pragma_names* that are not recognized by an implementation shall have no effect on interpretation of the SystemVerilog source text.

22.11.1 Standard pragmas

The `reset` and `resetall` pragmas shall restore the default values and state of *pragma_keywords* associated with the affected pragmas. These default values shall be the values that the tool defines before any SystemVerilog text has been processed. The `reset` pragma shall reset the state for all *pragma_names* that appear as *pragma_keywords* in the directive. The `resetall` pragma shall reset the state of all *pragma_names* recognized by the implementation.

The **`protect`** pragma is used to specify protected envelopes, as described in Clause 34.

22.12 `line

It is important for SystemVerilog tools to keep track of the file names of the SystemVerilog source files and the line numbers in the files. This information can be used for error messages or source code debugging and can be accessed by the Programming Language Interface (PLI) (see Clause 36).

In many cases, however, the SystemVerilog source is preprocessed by some other tool, and the line and file information of the original source file can be lost because the preprocessor might add additional lines to the source code file, combine multiple source code lines into one line, concatenate multiple source files, and so on.

The `line` compiler directive can be used to specify the original source code line number and file name. This allows the location in the original file to be maintained if another process modifies the source. After the new line number and file name are specified, the compiler can correctly refer to the original source location. However, a tool is not required to produce `line` directives. These directives are not intended to be inserted manually into the code, although they can be.

The compiler shall maintain the current line number and file name of the file being compiled. The `line directive shall set the line number and file name of the following line to those specified in the directive. The directive can be specified anywhere within the SystemVerilog source description. However, only white space may appear on the same line as the `line directive. Comments are not allowed on the same line as a `line directive. All parameters in the `line directive are required. The results of this directive are not affected by the `resetall directive.

The syntax for the `line compiler directive is given in Syntax 22-9.

---

line_compiler_directive ::=
    `**line** number " filename " level

Syntax 22-9—Syntax for line compiler directive (not in Annex A)

The *number* parameter shall be a positive integer that specifies the new line number of the following text line. The *filename* parameter shall be a string literal that is treated as the new name of the file. The *filename* can also be a full or relative path name. The *level* parameter shall be 0, 1, or 2. The value 1 indicates that the following line is the first line after an include file has been entered. The value 2 indicates that the following line is the first line after an include file has been exited. The value 0 indicates any other line.

For example:

```
`line 3 "orig.v" 2
// This line is line 3 of orig.v after exiting include file
```

As the compiler processes the remainder of the file and new files, the line number shall be incremented as each line is read, and the name shall be updated to the new current file being processed. The line number shall be reset to 1 at the beginning of each file. When beginning to read include files, the current line and file name shall be stored for restoration at the termination of the include file. The updated line number and file name information shall be available for PLI access. The mechanism of library searching is not affected by the effects of the `line compiler directive.

22.13 `__FILE__ and `__LINE__

`__FILE__ expands to the name of the current input file, in the form of a string literal. This is the path by which a tool opened the file, not the short name specified in `include or as a tool's input file name argument. The format of this path name may be implementation dependent.

`__LINE__ expands to the current input line number, in the form of a simple decimal number.

`__FILE__ and `__LINE__ are useful in generating an error message to report a problem; the message can state the source line at which the problem was detected.

For example:

```
$display("Internal error: null handle at %s, line %d.",
`__FILE__, `__LINE__);
```

An `include directive changes the expansions of `__FILE__ and `__LINE__ to correspond to the included file. At the end of that file, when processing resumes on the input file that contained the `include directive, the expansions of `__FILE__ and `__LINE__ revert to the values they had before the `include (but `__LINE__ is then incremented by one as processing moves to the line after the `include).

A `line directive changes `__LINE__ and may change `__FILE__ as well.

22.14 `begin_keywords, `end_keywords

A pair of directives, `begin_keywords and `end_keywords, can be used to specify what identifiers are reserved as keywords within a block of source code, based on a specific version of IEEE Std 1364 or IEEE Std 1800. These are two distinct directives, but they shall be used in a pair with `begin_keywords appearing first and `end_keywords appearing sometime later.

The syntax of the `begin_keywords and `end_keywords directives is given in Syntax 22-10.

---

keywords_directive ::= **`begin_keywords** "version_specifier"

version_specifier ::=
  1800-2023
| 1800-2017
| 1800-2012
| 1800-2009
| 1800-2005
| 1364-2005
| 1364-2001
| 1364-2001-noconfig
| 1364-1995

endkeywords_directive ::= **`end_keywords**

Syntax 22-10—Syntax for begin_keywords and end_keywords compiler directives (not in Annex A)

The *version_specifier* specifies that only the identifiers listed as reserved keywords in the specified version are considered to be reserved words. The `begin_keywords and `end_keywords directives only specify the set of identifiers that are reserved as keywords. The directives do not affect the semantics, tokens, and other aspects of the SystemVerilog language.

Implementations and other standards are permitted to extend the `begin_keywords directive with custom version specifiers. It shall be an error if an implementation does not recognize the *version_specifier* used with the `begin_keywords directive.

The `begin_keywords and `end_keywords directives can only be specified outside a design element (see 3.2). The `begin_keywords directive affects all source code that follows the directive, even across source code file boundaries, until the matching `end_keywords directive. The results of these directives are not affected by the `resetall directive.

The `begin_keywords...`end_keywords directive pair can be nested. Each nested pair is stacked so that when an `end_keywords directive is encountered, the implementation returns to using the *version_specifier* that was in effect prior to the matching `begin_keywords directive.

If no `begin_keywords directive is specified, then the reserved keyword list shall be the implementation's default set of keywords. The default set of reserved keywords used by an implementation shall be implementation dependent. For example, an implementation based on IEEE Std 1800-2005 would most likely use the IEEE Std 1800-2005 set of reserved keywords as its default, whereas an implementation based on IEEE Std 1364-2001 would most likely use the IEEE Std 1364-2001 set of reserved keywords as its default. Implementations may provide other mechanisms for specifying the set of reserved keywords to be used as the default. One possible use model might be for an implementation to use invocation options to