

IEEE Standard Hardware Description Language Based on the Verilog[®] Hardware Description Language

Section 1

Overview

1.1 Objectives of this standard

The intent of this standard is to serve as a complete specification of the Verilog[®] Hardware Description Language (HDL). This document contains

- The formal syntax and semantics of all Verilog HDL constructs
- The formal syntax and semantics of Standard Delay Format (SDF) constructs
- Simulation system tasks and functions, such as text output display commands
- Compiler directives, such as text substitution macros and simulation time scaling
- The Programming Language Interface (PLI) binding mechanism
- The formal syntax and semantics of access routines, task/function routines, and Verilog procedural interface routines
- Informative usage examples
- Informative delay model for SDF
- Listings of header files for PLI

1.2 Conventions used in this standard

This standard is organized into sections, each of which focuses on some specific area of the language. There are sub-clauses within each section to discuss individual constructs and concepts. The discussion begins with an introduction and an optional rationale for the construct or the concept, followed by syntax and semantic descriptions, followed by some examples and notes.

The verb “shall” is used through out this standard to indicate mandatory requirements, whereas the verb “can” is used to indicate optional features. These verbs denote different meanings to different readers of this standard:

- a) To the developers of tools that process the Verilog HDL, the verb “shall” denotes a requirement that the standard imposes. The resulting implementation is required to enforce the requirements and to issue an error if the requirement is not met by the input.
- b) To the Verilog HDL model developer, the verb “shall” denotes that the characteristics of the Verilog HDL are natural consequences of the language definition. The model developer is required to adhere to the constraint implied by the characteristic. The verb “can” denotes optional features that the model developer can exercise at discretion. If used, however, the model developer is required to follow the requirements set forth by the language definition.
- c) To the Verilog HDL model user, the verb “shall” denotes that the characteristics of the models are natural consequences of the language definition. The model user can depend on the characteristics of the model implied by its Verilog HDL source text.

1.3 Syntactic description

The formal syntax of the Verilog HDL is described using Backus-Naur Form (BNF). The following conventions are used:

- a) Lowercase words, some containing embedded underscores, are used to denote syntactic categories. For example:

module_declaration

- b) Boldface words are used to denote reserved keywords, operators, and punctuation marks as a required part of the syntax. These words appear in a larger font for distinction. For example:

module => ;

- c) A vertical bar separates alternative items unless it appears in boldface, in which case it stands for itself. For example:

unary_operator ::=
+ | - | ! | ~ | **&** | ~**&** | | | ~| | ^ | ~^ | ^~

- d) Square brackets enclose optional items. For example:

input_declaration ::= **input** [range] list_of_variables ;

- e) Braces enclose a repeated item unless it appears in boldface, in which case it stands for itself. The item may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule. Thus, the following two rules are equivalent:

list_of_param_assignments ::= param_assignment { , param_assignment }
list_of_param_assignments ::=
 param_assignment
 | list_of_param_assignment , param_assignment

- f) If the name of any category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. For example, *msb_constant_expression* and *lsb_constant_expression* are equivalent to *constant_expression*.

The main text uses *italicized* font when a term is being defined, and constant-width font for examples, file names, and while referring to constants, especially 0, 1, x, and z values.

1.4 Contents of this standard

A synopsis of the sections and annexes is presented as a quick reference. There are 27 sections and 8 annexes. All the sections and annexes A, B, E, F, and G are normative parts of this standard. Annexes C, D, and H are included for informative purposes only.

- 1) **Overview**
This section discusses the conventions used in this standard and its contents.
- 2) **Lexical conventions**
This section describes how to specify and interpret the lexical tokens.
- 3) **Data types**
This section describes net and variable data types. This section also discusses the parameter data type for constant values and describes drive and charge strength of the values on nets.
- 4) **Expressions**
This section describes the operators and operands that can be used in expressions.
- 5) **Scheduling semantics**
This section describes the scheduling semantics of the Verilog HDL.
- 6) **Assignments**
This section compares the two main types of assignment statements in the Verilog HDL—continuous assignments and procedural assignments. It describes the continuous assignment statement that drives values onto nets.
- 7) **Gate and switch level modeling**
This section describes the gate and switch level primitives and logic strength modeling.
- 8) **User-defined primitives (UDPs)**
This section describes how a primitive can be defined in the Verilog HDL and how these primitives are included in Verilog HDL models.
- 9) **Behavioral modeling**
This section describes procedural assignments, procedural continuous assignments, and behavioral language statements.
- 10) **Tasks and functions**
This section describes tasks and functions—procedures that can be called from more than one place in a behavioral model. It describes how tasks can be used like subroutines and how functions can be used to define new operators.
- 11) **Disabling of named blocks and tasks**
This section describes how to disable the execution of a task and a block of statements that has a specified name.
- 12) **Hierarchical structures**
This section describes how hierarchies are created in the Verilog HDL and how parameter values declared in a module can be overridden. It describes how generated instantiations can be used to do conditional or multiple instantiations in a design.
- 13) **Configuring the contents of a design**
This section describes how to configure the contents of a design.
- 14) **Specify blocks**
This section describes how to specify timing relationships between input and output ports of a module.
- 15) **Timing checks**
This section describes how timing checks are used in specify blocks to determine if signals obey the timing constraints.
- 16) **Backannotation using the Standard Delay Format (SDF)**
This section describes syntax and semantics of Standard Delay Format (SDF) constructs.
- 17) **System tasks and functions**
This section describes the system tasks and functions.
- 18) **Value change dump (VCD) files**
This section describes the system tasks associated with Value Change Dump (VCD) file, and the format of the file.

- 19) **Compiler directives**
This section describes the compiler directives.
- 20) **PLI Overview**
This section previews the C language procedural interface standard (Programming Language Interface or PLI) and interface mechanisms that are part of the Verilog HDL.
- 21) **PLI TF and ACC interface mechanism**
This section describes the interface mechanism that provides a means for users to link PLI task/function (TF) routine and access (ACC) routine applications to Verilog software tools.
- 22) **Using ACC routines**
This section describes the ACC routines in general, including how and why to use them.
- 23) **ACC routine definitions**
This section describes the specific ACC routines, explaining their function, syntax, and usage.
- 24) **Using TF routines**
This section provides an overview of the types of operations that are done with the TF routines.
- 25) **TF routine definitions**
This section describes the specific TF routines, explaining their function, syntax, and usage.
- 26) **Using VPI routines**
This section provides an overview of the types of operations that are done with the Verilog Programming Interface (VPI) routines.
- 27) **VPI routine definitions**
This section describes the VPI routines.
- A) **Formal syntax definition**
This normative annex describes, using BNF, the syntax of the Verilog HDL.
- B) **List of keywords**
This normative annex lists the Verilog HDL keywords.
- C) **System tasks and functions**
This informative annex describes system tasks and functions that are frequently used, but that are not part of the standard.
- D) **Compiler directives**
This informative annex describes compiler directives that are frequently used, but that are not part of the standard.
- E) **acc_user.h**
This normative annex provides a listing of the contents of the `acc_user.h` file.
- F) **veriusers.h**
This normative annex provides a listing of the contents of the `vpi_user.h` file.
- G) **vpi_user.h**
This normative annex provides a listing of the contents of the `veriusers.h` file.
- H) **Bibliography**
This informative annex contains bibliographic entries pertaining to this standard.

1.5 Header file listings

The header file listings included in the annexes E, F, and G for `acc_user.h`, `veriusers.h`, and `vpi_user.h` are a normative part of this standard. All compliant software tools should use the same function declarations, constant definitions, and structure definitions contained in these header file listings.

1.6 Examples

Several small examples in the Verilog HDL and the C programming language are shown throughout this standard. These examples are *informative*—they are intended to illustrate the usage of Verilog HDL constructs and PLI functions in a simple context and do not define the full syntax.

1.7 Prerequisites

Sections 20 through 27 and annexes E through G presuppose a working knowledge of the C programming language.

Section 2

Lexical conventions

This section describes the lexical tokens used in Verilog HDL source text and their conventions.

2.1 Lexical tokens

Verilog HDL source text files shall be a stream of lexical tokens. A *lexical token* shall consist of one or more characters. The layout of tokens in a source file shall be free format—that is, spaces and newlines shall not be syntactically significant other than being token separators, except for escaped identifiers (see 2.7.1).

The types of lexical tokens in the language are as follows:

- White space
- Comment
- Operator
- Number
- String
- Identifier
- Keyword

2.2 White space

White space shall contain the characters for spaces, tabs, newlines, and formfeeds. These characters shall be ignored except when they serve to separate other lexical tokens. However, blanks and tabs shall be considered significant characters in strings (see 2.6).

2.3 Comments

The Verilog HDL has two forms to introduce comments. A *one-line comment* shall start with the two characters `//` and end with a newline. A *block comment* shall start with `/*` and end with `*/`. Block comments shall not be nested. The one-line comment token `//` shall not have any special meaning in a block comment.

2.4 Operators

Operators are single-, double-, or triple-character sequences and are used in expressions. Section 4 discusses the use of operators in expressions.

Unary operators shall appear to the left of their operand. *Binary operators* shall appear between their operands. A *conditional operator* shall have two operator characters that separate three operands.

2.5 Numbers

Constant numbers can be specified as integer constants or real constants.

```

number ::= (From Annex A - A.8.7)
    decimal_number
    | octal_number
    | binary_number
    | hex_number
    | real_number
real_number1 ::=
    unsigned_number . unsigned_number
    | unsigned_number [ . unsigned_number ] exp [ sign ] unsigned_number
exp ::= e | E
decimal_number ::=
    unsigned_number
    | [ size ] decimal_base unsigned_number
    | [ size ] decimal_base x_digit { _ }
    | [ size ] decimal_base z_digit { _ }
binary_number ::=
    [ size ] binary_base binary_value
octal_number ::=
    [ size ] octal_base octal_value
hex_number ::=
    [ size ] hex_base hex_value
sign ::= + | -
size ::= non_zero_unsigned_number
non_zero_unsigned_number1 ::= non_zero_decimal_digit { _ | decimal_digit }
unsigned_number1 ::= decimal_digit { _ | decimal_digit }
binary_value1 ::= binary_digit { _ | binary_digit }
octal_value1 ::= octal_digit { _ | octal_digit }
hex_value1 ::= hex_digit { _ | hex_digit }
decimal_base1 ::= '[s]S)d | '[s]S]D
binary_base1 ::= '[s]S]b | '[s]S]B
octal_base1 ::= '[s]S]o | '[s]S]O
hex_base1 ::= '[s]S]h | '[s]S]H
non_zero_decimal_digit ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
decimal_digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
binary_digit ::= x_digit | z_digit | 0 | 1
octal_digit ::= x_digit | z_digit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
hex_digit ::=
    x_digit | z_digit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    | a | b | c | d | e | f | A | B | C | D | E | F
x_digit ::= x | X
z_digit ::= z | Z | ?

```

¹Embedded spaces are illegal.

Syntax 2-1—Syntax for integer and real numbers

2.5.1 Integer constants

Integer constants can be specified in decimal, hexadecimal, octal, or binary format.

There are two forms to express integer constants. The first form is a simple decimal number, which shall be specified

as a sequence of digits 0 through 9, optionally starting with a plus or minus unary operator. The second form specifies a *size constant*, which shall be composed of up to three tokens—an optional size constant, a single quote followed by a base format character, and the digits representing the value of the number.

The first token, a size constant, shall specify the size of the constant in terms of its exact number of bits. It shall be specified as a non-zero unsigned decimal number. For example, the size specification for two hexadecimal digits is 8, because one hexadecimal digit requires 4 bits. Unsized unsigned constants where the high order bit is unknown (X or x) or tri-state (Z or z) are extended to the size of the expression containing the constant.

NOTE—In Verilog 1364-1995 unsized constants where the high order bit is unknown or tri-state, the x or z was only extended to 32 bits.

The second token, a base_format, shall consist of a letter specifying the base for the number, optionally preceded by the single character s (or S) to indicate a signed quantity, preceded by the single quote character ('). Legal base specifications are d, D, h, H, o, O, b, or B, for the bases decimal, hexadecimal, octal, and binary respectively.

The use of x and z in defining the value of a number is case insensitive.

The single quote and the base format character shall not be separated by any white space.

The third token, an unsigned number, shall consist of digits that are legal for the specified base format. The unsigned number token shall immediately follow the base format, optionally preceded by white space. The hexadecimal digits a to f shall be case insensitive.

Simple decimal numbers without the size and the base format shall be treated as *signed integers*, whereas the numbers specified with the base format shall be treated as signed integers if the s designator is included or as *unsigned integers* if the base format only is used. The s designator does not affect the bit pattern specified, only its interpretation.

A plus or minus operator preceding the size constant is a unary plus or minus operator. A plus or minus operator between the base format and the number is an illegal syntax.

Negative numbers shall be represented in 2's complement form.

An x represents the *unknown value* in hexadecimal, octal, and binary constants. A z represents the *high-impedance value*. See 3.1 for a discussion of the Verilog HDL value set. An x shall set 4 bits to unknown in the hexadecimal base, 3 bits in the octal base, and 1 bit in the binary base. Similarly, a z shall set 4 bits, 3 bits, and 1 bit, respectively, to the high-impedance value.

If the size of the unsigned number is smaller than the size specified for the constant, the unsigned number shall be padded to the left with zeros. If the leftmost bit in the unsigned number is an x or a z, then an x or a z shall be used to pad to the left respectively.

When used in a number, the question-mark (?) character is a Verilog HDL alternative for the z character. It sets 4 bits to the high-impedance value in hexadecimal numbers, 3 bits in octal, and 1 bit in binary. The question mark can be used to enhance readability in cases where the high-impedance value is a don't-care condition. See the discussion of **casez** and **casex** in 9.5.1. The question-mark character is also used in user-defined primitive state table. See 8.1.6, Table 8-1.

The underscore character (_) shall be legal anywhere in a number except as the first character. The underscore character is ignored. This feature can be used to break up long numbers for readability purposes.

*Examples:**Example 1—Unsize constant numbers*

```

659          // is a decimal number
'h 837FF     // is a hexadecimal number
'o7460       // is an octal number
4af          // is illegal (hexadecimal format requires 'h)

```

Example 2—Sized constant numbers

```

4'b1001     // is a 4-bit binary number
5 'D 3      // is a 5-bit decimal number
3'b01x      // is a 3-bit number with the least
              // significant bit unknown
12'hx       // is a 12-bit unknown number
16'hz       // is a 16-bit high-impedance number

```

Example 3—Using sign with constant numbers

```

8 'd -6      // this is illegal syntax
-8 'd 6      // this defines the two's complement of 6,
              // held in 8 bits—equivalent to -(8'd 6)
4 'shf       // this denotes the 4-bit number '1111', to
              // be interpreted as a 2's complement number,
              // or '-1'. This is equivalent to -4'h 1
-4 'sd15     // this is equivalent to -(-4'd 1), or '0001'.

```

Example 4—Automatic left padding

```

reg [11:0] a, b, c, d;
initial begin
    a = 'h x;      // yields xxx
    b = 'h 3x;     // yields 03x
    c = 'h z3;     // yields zz3
    d = 'h 0z3;    // yields 0z3
end
reg [84:0]      e, f, g;

    e = 'h5;       // yields {82{1'b0},3'b101}
    f = 'hx;       // yields {85{1'hx}}
    g = 'hz;       // yields {85{1'hz}}

```

Example 5—Using underscore character in numbers

```

27_195_000
16'b0011_0101_0001_1111
32 'h 12ab_f001

```

NOTES

1—Sized negative constant numbers and sized signed constant numbers are sign-extended when assigned to a reg data type, regardless of whether the reg itself is signed or not.

2—Each of the three tokens for specifying a number may be macro substituted.

3—The number of bits that make up an unsized number (which is a simple decimal number or a number without the size specification) shall be at least 32.

2.5.2 Real constants

The *real constant numbers* shall be represented as described by *IEEE Std 754-1985* [B1],¹ an IEEE standard for double-precision floating-point numbers.

Real numbers can be specified in either decimal notation (for example, 14.72) or in scientific notation (for example, 39e8, which indicates 39 multiplied by 10 to the eighth power). Real numbers expressed with a decimal point shall have at least one digit on each side of the decimal point.

Examples:

```
1.2
0.1
2394.26331
1.2E12 (the exponent symbol can be e or E)
1.30e-2
0.1e-0
23E10
29E-2
236.123_763_e-12 (underscores are ignored)
```

The following are invalid forms of real numbers because they do not have at least one digit on each side of the decimal point:

```
.12
9.
4.E3
.2e-7
```

2.5.3 Conversion

Real numbers shall be converted to integers by rounding the real number to the nearest integer, rather than by truncating it. Implicit conversion shall take place when a real number is assigned to an integer. The ties shall be rounded away from zero.

For example:

- The real numbers 35.7 and 35.5 both become 36 when converted to an integer and 35.2 becomes 35.
- Converting -1.5 to integer yields -2, converting 1.5 to integer yields 2.

2.6 Strings

A *string* is a sequence of characters enclosed by double quotes (") and contained on a single line. Strings used as operands in expressions and assignments shall be treated as unsigned integer constants represented by a sequence of 8-bit ASCII values, with one 8-bit ASCII value representing one character.

¹The numbers in brackets correspond to those of the bibliography in Annex H.

2.6.1 String variable declaration

String variables are variables of reg type (see 3.2) with width equal to the number of characters in the string multiplied by 8.

Example:

To store the twelve-character string "Hello world!" requires a reg 8 * 12, or 96 bits wide

```
reg [8*12:1] stringvar;
initial begin
    stringvar = "Hello world!";
end
```

2.6.2 String manipulation

Strings can be manipulated using the Verilog HDL operators. The value being manipulated by the operator is the sequence of 8-bit ASCII values.

Example:

```
module string_test;
reg [8*14:1] stringvar;
initial begin
    stringvar = "Hello world";
    $display("%s is stored as %h", stringvar,stringvar);
    stringvar = {stringvar,"!!!"};
    $display("%s is stored as %h", stringvar,stringvar);
end
endmodule
```

The output is:

```
Hello world is stored as 00000048656c6c6f20776f726c64
Hello world!!! is stored as 48656c6c6f20776f726c64212121
```

NOTE—When a variable is larger than required to hold a value being assigned, the contents on the left are padded with zeros after the assignment. This is consistent with the padding that occurs during assignment of nonstring values. If a string is larger than the destination string variable, the string is truncated to the left, and the leftmost characters will be lost.

2.6.3 Special characters in strings

Certain characters can only be used in strings when preceded by an introductory character called an *escape character*. Table 2-1 lists these characters in the right-hand column, with the escape sequence that represents the character in the left-hand column.

Table 2-1—Specifying special characters in string

Escape string	Character produced by escape string
\n	New line character
\t	Tab character
\\	\ character

Table 2-1—Specifying special characters in string (*continued*)

Escape string	Character produced by escape string
\"	" character
\ddd	A character specified in 1–3 octal digits ($0 \leq d \leq 7$)

2.7 Identifiers, keywords, and system names

An *identifier* is used to give an object a unique name so it can be referenced. An identifier is either a *simple identifier* or an *escaped identifier* (see 2.7.1). A *simple identifier* shall be any sequence of letters, digits, dollar signs (\$), and underscore characters (_).

The first character of a simple identifier shall not be a digit or \$; it can be a letter or an underscore. Identifiers shall be case sensitive.

Example:

```
shiftreg_a
busa_index
error_condition
merge_ab
_bus3
n$657
```

NOTE—Implementations may set a limit on the maximum length of identifiers, but they shall at least be 1024 characters. If an identifier exceeds the implementation-specified length limit, an error shall be reported.

2.7.1 Escaped identifiers

Escaped identifiers shall start with the backslash character (\) and end with white space (space, tab, newline). They provide a means of including any of the printable ASCII characters in an identifier (the decimal values 33 through 126, or 21 through 7E in hexadecimal).

Neither the leading backslash character nor the terminating white space is considered to be part of the identifier. Therefore, an escaped identifier \cpu3 is treated the same as a nonescaped identifier cpu3.

Example:

```
\busa+index
\ -clock
\***error-condition***
\net1/\net2
\{a,b}
\a*(b+c)
```

2.7.2 Generated identifiers

Generated identifiers are created by generate loops (see 12.1.3.2); and are a special case of identifiers in that they can be used in hierarchical names (see 12.4). A generated identifier is the named generate block identifier terminated with a ([digit(s)]) string. This identifier is used as a node name in hierarchical names (see 12.4).

2.7.3 Keywords

Keywords are predefined nonescaped identifiers that are used to define the language constructs. A Verilog HDL keyword preceded by an escape character is not interpreted as a keyword.

All keywords are defined in lowercase only. Annex B gives a list of all defined keywords.

2.7.4 System tasks and functions

The \$ character introduces a language construct that enables development of user-defined tasks and functions. A name following the \$ is interpreted as a *system task* or a *system function*.

The syntax for a system task or function is given in Syntax 2-2.

```

system_task_enable ::= (From Annex A - A.6.9)
    system_task_identifier [ ( expression { , expression } ) ] ;
system_function_call ::= (From Annex A - A.8.2)
    system_function_identifier [ ( expression { , expression } ) ]
system_function_identifier1 ::= (From Annex A - A.9.3)
    $[ a-zA-Z0-9_$ ] { [ a-zA-Z0-9_$ ] }
system_task_identifier1 ::=
    $[ a-zA-Z0-9_$ ] { [ a-zA-Z0-9_$ ] }

```

¹The \$ character in a `system_function_identifier` or `system_task_identifier` shall not be followed by white space. A `system_function_identifier` or `system_task_identifier` shall not be escaped.

Syntax 2-2—Syntax for system tasks and functions

The \$identifier system task or function can be defined in three places

- A standard set of \$identifier system tasks and functions, as defined in Sections 17 and 19.
- Additional \$identifier system tasks and functions defined using the PLI, as described in Section 20.
- Additional \$identifier system tasks and functions defined by software implementations.

Any valid identifier, including keywords already in use in contexts other than this construct, can be used as a system task or function name. The system tasks and functions described in Section 17 are part of this standard. Additional system tasks and functions with the \$identifier construct are not part of this standard.

Example:

```

$display ("display a message");
$finish;

```

2.7.5 Compiler directives

The ` character (the ASCII value 60, called open quote or accent grave) introduces a language construct used to implement compiler directives. The compiler behavior dictated by a compiler directive shall take effect as soon as the compiler reads the directive. The directive shall remain in effect for the rest of the compilation unless a different compiler directive specifies otherwise. A compiler directive in one description file can therefore control compilation behavior in multiple description files.

The `identifier compiler directive construct can be defined in two places

- A standard set of `identifier compiler directives defined in Section 19.
- Additional `identifier compiler directives defined by software implementations.

Any valid identifier, including keywords already in use in contexts other than this construct, can be used as a compiler directive name. The compiler directives described in Section 19 are part of this standard. Additional compiler directives with the `identifier construct are not part of this standard.

Example:

```
`define wordsize 8
```

2.8 Attributes

With the proliferation of tools other than simulators that use Verilog HDL as their source, a mechanism is included for specifying properties about objects, statements and groups of statements in the HDL source that may be used by various tools, including simulators, to control the operation or behavior of the tool. These properties shall be referred to as "attributes". This section specifies the syntactic mechanism that shall be used for specifying attributes, without standardizing on any particular attributes.

The syntax for specifying an attribute is shown in Syntax 2-3.

```
attribute_instance ::= (From Annex A - A.9.1)
    (* attr_spec { , attr_spec } ; *)
attr_spec ::=
    attr_name = constant_expression
    | attr_name
attr_name ::=
    identifier
```

Syntax 2-3—Syntax for attributes

An `attribute_instance` can appear in the Verilog description as a prefix attached to a declaration, a module item, a statement, or a port connection. It can appear as a suffix to an operator or a Verilog function name in an expression.

If a value is not specifically assigned to the attribute or a non-zero value is assigned, then its value shall be `true`. If 0 is assigned, then the attribute is `false`. For attributes that can be attached to both module definitions and module instantiations, the attribute value associated with the module instantiation shall override the attribute value associated with the module definition.

2.8.1 Examples

Example 1—The following example shows how to attach attributes to a case statement:

```
(* full_case, parallee_case *)  
case (foo)  
<rest_of_case_statement>
```

or

```
(* full_case=1, parallee_case=1 *)  
case (foo)  
<rest_of_case_statement>
```

or

```
(* full_case, // no value assigned  
   parallee_case=1 *)  
case (foo)  
<rest_of_case_statement>
```

Example 2—To attach the full_case attribute, but NOT the parallel_case attribute:

```
(* full_case *) // parallel_case not specified  
case (foo)  
<rest_of_case_statement>
```

or

```
(* full_case=1, parallel_case = 0 *)  
case (foo)  
<rest_of_case_statement>
```

Example 3—To attach an attribute to a module definition:

```
(* optimize_power *)  
module mod1 (<port_list>;
```

or

```
(* optimize_power *)  
module mod1 (<port_list>;
```

Example 4—To attach an attribute to a module instantiation:

```
(* optimize_power=0 *)  
mod1 synth1 (<port_list>;
```

Example 5—To attach an attribute to a reg declaration:

```
(* fsm_state *) reg [7:0] state1;  
(* fsm_state=1 *) reg [3:0] state2, state3;  
reg [3:0] reg1; // this reg does NOT have fsm_state set  
(* fsm_state=0 *) reg [3:0] reg2; // nor does this one
```


Example 6—To attach an attribute to an operator:

```
a = b + (* mode = "cla" *) c;
```

This sets the value for the attribute mode to be the string cla.

Example 7—To attach an attribute to a Verilog function call:

```
a = add (* mode = "cla" *) (b, c);
```

Example 8—To attach an attribute to a conditional operator:

```
a = b ? (* no_glitch *) c : d;
```

2.8.2 Syntax

The syntax for legal statements with attributes is shown in Syntax 2-4 — Syntax 2-11.

The syntax for module declaration attributes is given in Syntax 2-4.

```
module_declaration ::= (From Annex A - A.1.3)
    { attribute_instance } module_keyword module_identifier
    [ module_parameter_port_list ] [ list_of_ports ] ;
    { module_item }
endmodule
| { attribute_instance } module_keyword module_identifier
  [ module_parameter_port_list ] [ list_of_port_declarations ] ;
  { non_port_module_item }
endmodule
```

Syntax 2-4—Syntax for module declaration attributes

The syntax for port declaration attributes is given in Syntax 2-5.

```
port_declaration ::= (From Annex A - A.1.4)
    { attribute_instance } inout_declaration
    | { attribute_instance } input_declaration
    | { attribute_instance } output_declaration
```

Syntax 2-5—Syntax for port declaration attributes

The syntax for module item attributes is given in Syntax 2-6.

```

module_item ::= (From Annex A - A.1.5)
    module_or_generate_item
  | port_declaration
  | { attribute_instance } generated_instantiation
  | { attribute_instance } local_parameter_declaration
  | { attribute_instance } parameter_declaration
  | { attribute_instance } specify_block
  | { attribute_instance } specparam_declaration
module_or_generate_item ::=
    { attribute_instance } module_or_generate_item_declaration
  | { attribute_instance } parameter_override
  | { attribute_instance } continuous_assign
  | { attribute_instance } gate_instantiation
  | { attribute_instance } udp_instantiation
  | { attribute_instance } module_instantiation
  | { attribute_instance } initial_construct
  | { attribute_instance } always_construct
non_port_module_item ::=
    { attribute_instance } generated_instantiation
  | { attribute_instance } local_parameter_declaration
  | { attribute_instance } module_or_generate_item
  | { attribute_instance } parameter_declaration
  | { attribute_instance } specify_block
  | { attribute_instance } specparam_declaration

```

Syntax 2-6—Syntax for module item attributes

The syntax for function port, task, and block attributes is given in Syntax 2-7.

```

function_port_list ::= (From Annex A - A.2.6)
    { attribute_instance } input_declaration { , { attribute_instance } input_declaration }
task_item_declaration ::= (From Annex A - A.2.7)
    block_item_declaration
  | { attribute_instance } input_declaration
  | { attribute_instance } output_declaration
  | { attribute_instance } inout_declaration
task_port_item ::=
    { attribute_instance } input_declaration
  | { attribute_instance } output_declaration
  | { attribute_instance } inout_declaration
block_item_declaration ::= (From Annex A - A.2.8)
    { attribute_instance } block_reg_declaration
  | { attribute_instance } event_declaration
  | { attribute_instance } integer_declaration
  | { attribute_instance } local_parameter_declaration
  | { attribute_instance } parameter_declaration
  | { attribute_instance } real_declaration
  | { attribute_instance } realtime_declaration
  | { attribute_instance } time_declaration

```

Syntax 2-7—Syntax for function port, task, and block attributes

The syntax for port connection attributes is given in Syntax 2-8.

```
ordered_port_connection ::= (From Annex A - A.4.1)
    { attribute_instance } [ expression ]
named_port_connection ::=
    { attribute_instance } .port_identifier ( [ expression ] )
```

Syntax 2-8—Syntax for port connection attributes

The syntax for udp attributes is given in Syntax 2-9.

```
udp_declaration ::= (From Annex A - A.5.1)
    { attribute_instance } primitive udp_identifier ( udp_port_list ) ;
    udp_port_declaration { udp_port_declaration }
    udp_body
endprimitive
| { attribute_instance } primitive udp_identifier ( udp_declaration_port_list ) ;
    udp_body
endprimitive
udp_output_declaration ::= (From Annex A - A.5.2)
    { attribute_instance } output port_identifier ;
    | { attribute_instance } output reg port_identifier [ = constant_expression ] ;
udp_input_declaration ::=
    { attribute_instance } input list_of_port_identifiers ;
udp_reg_declaration ::=
    { attribute_instance } reg variable_identifier ;
```

Syntax 2-9—Syntax for udp attributes

The syntax for function and statement attributes is given in Syntax 2-10.

```
function_statement_or_null ::= (From Annex A - A.6.2)
    function_statement
    | { attribute_instance } ;
statement ::= (From Annex A - A.6.4)
    { attribute_instance } blocking_assignment ;
    | { attribute_instance } case_statement
    | { attribute_instance } conditional_statement
    | { attribute_instance } disable_statement
    | { attribute_instance } event_trigger
    | { attribute_instance } loop_statement
    | { attribute_instance } nonblocking_assignment ;
    | { attribute_instance } par_block
    | { attribute_instance } procedural_continuous_assignments ;
    | { attribute_instance } procedural_timing_control_statement
    | { attribute_instance } seq_block
    | { attribute_instance } system_task_enable
    | { attribute_instance } task_enable
    | { attribute_instance } wait_statement
statement_or_null ::=
    statement
    | { attribute_instance } ;
function_statement ::=
    { attribute_instance } function_blocking_assignment ;
    | { attribute_instance } function_case_statement
    | { attribute_instance } function_conditional_statement
    | { attribute_instance } function_loop_statement
    | { attribute_instance } function_seq_block
    | { attribute_instance } disable_statement
    | { attribute_instance } system_task_enable
```

Syntax 2-10—Syntax for function and statement attributes

The syntax for function call and expression attributes is given in Syntax 2-11.

```

constant_function_call ::= (From Annex A - A.8.2)
    function_identifier { attribute_instance }
    ( constant_expression { , constant_expression } )
function_call ::=
    hierarchical_function_identifier { attribute_instance }
    ( expression { , expression } )
genvar_function_call ::=
    genvar_function_identifier { attribute_instance }
    ( constant_expression { , constant_expression } )
conditional_expression ::= (From Annex A - A.8.3)
    expression1 ? { attribute_instance } expression2 : expression3
constant_expression ::=
    constant_primary
    | unary_operator { attribute_instance } constant_primary
    | constant_expression binary_operator { attribute_instance } constant_expression
    | constant_expression ? { attribute_instance }
      constant_expression : constant_expression
    | string
expression ::=
    primary
    | unary_operator { attribute_instance } primary
    | expression binary_operator { attribute_instance } expression
    | conditional_expression
    | string
module_path_conditional_expression ::=
    module_path_expression ? { attribute_instance }
    | module_path_expression : module_path_expression
module_path_expression ::=
    module_path_primary
    | unary_module_path_operator { attribute_instance } module_path_primary
    | module_path_expression binary_module_path_operator { attribute_instance }
      module_path_expression
    | module_path_conditional_expression

```

Syntax 2-11—Syntax for function call and expression attributes

Section 3

Data types

The set of Verilog HDL data types is designed to represent the data storage and transmission elements found in digital hardware.

3.1 Value set

The Verilog HDL value set consists of four basic values:

- 0 - represents a logic zero, or a false condition
- 1 - represents a logic one, or a true condition
- x - represents an unknown logic value
- z - represents a high-impedance state

The values 0 and 1 are logical complements of one another.

When the z value is present at the input of a gate, or when it is encountered in an expression, the effect is usually the same as an x value. Notable exceptions are the metal-oxide semiconductor (MOS) primitives, which can pass the z value.

Almost all of the data types in the Verilog HDL store all four basic values. The exception is the *event* type (see 9.7.3), which has no storage. All bits of vectors can be independently set to one of the four basic values.

The language includes *strength* information in addition to the basic value information for net variables. This is described in detail in Section 7.

3.2 Nets and variables

There are two main groups of data types: the variable data types and the net data types. These two groups differ in the way that they are assigned and hold values. They also represent different hardware structures.

3.2.1 Net declarations

The *net* data types shall represent physical connections between structural entities, such as gates. A net shall not store a value (except for the trireg net). Instead, its value shall be determined by the values of its drivers, such as a continuous assignment or a gate. See Section 6 and Section 7 for definitions of these constructs. If no driver is connected to a net, its value shall be high-impedance (z) unless the net is a trireg, in which case it shall hold the previously driven value. It is illegal to redeclare a name already declared by a net, parameter, or variable declaration.

The syntax for net declarations is given in Syntax 3-1.

```

net_declaration ::= (From Annex A - A.2.1.3)
    net_type [ signed ]
    [ delay3 ] list_of_net_identifiers ;
| net_type [ drive_strength ] [ signed ]
    [ delay3 ] list_of_net_decl_assignments ;
| net_type [ vectored | scalared ] [ signed ]
    range [ delay3 ] list_of_net_identifiers ;
| net_type [ drive_strength ] [ vectored | scalared ] [ signed ]
    range [ delay3 ] list_of_net_decl_assignments ;
| triereg [ charge_strength ] [ signed ]
    [ delay3 ] list_of_net_identifiers ;
| triereg [ drive_strength ] [ signed ]
    [ delay3 ] list_of_net_decl_assignments ;
| triereg [ charge_strength ] [ vectored | scalared ] [ signed ]
    range [ delay3 ] list_of_net_identifiers ;
| triereg [ drive_strength ] [ vectored | scalared ] [ signed ]
    range [ delay3 ] list_of_net_decl_assignments ;

net_type ::= (From Annex A - A.2.2.1)
    supply0 | supply1
    | tri | triand | trior | tri0 | tri1
    | wire | wand | wor

drive_strength ::= (From Annex A - A.2.2.2)
    ( strength0 , strength1 )
    | ( strength1 , strength0 )
    | ( strength0 , highz1 )
    | ( strength1 , highz0 )
    | ( highz0 , strength1 )
    | ( highz1 , strength0 )

strength0 ::= supply0 | strong0 | pull0 | weak0
strength1 ::= supply1 | strong1 | pull1 | weak1
charge_strength ::= ( small ) | ( medium ) | ( large )
delay3 ::= (From Annex A - A.2.2.3)
    # delay_value | # ( delay_value [ , delay_value [ , delay_value ] ] )
delay2 ::=
    # delay_value | # ( delay_value [ , delay_value ] )
delay_value ::=
    unsigned_number
    | parameter_identifier
    | specparam_identifier
    | mintypmax_expression

list_of_net_decl_assignments ::= (From Annex A - A.2.3)
    net_decl_assignment { , net_decl_assignment }

list_of_net_identifiers ::=
    net_identifier [ dimension { dimension } ]
    { , net_identifier [ dimension { dimension } ] }

net_decl_assignment ::= (From Annex A - A.2.4)
    net_identifier = expression

dimension ::= (From Annex A - A.2.5)
    [ dimension_constant_expression : dimension_constant_expression ]

range ::=
    [ msb_constant_expression : lsb_constant_expression ]

```

Syntax 3-1—Syntax for net declaration

The first two forms of net declaration are described in this section. The third form, called net assignment, is described in Section 6.

3.2.2 Variable declarations

A *variable* is an abstraction of a data storage element. A variable shall store a value from one assignment to the next. An assignment statement in a procedure acts as a trigger that changes the value in the data storage element. The initialization value for **reg**, **time**, and **integer** data types shall be the unknown value, x . The default initialization value for **real** and **realtime** variable datatypes shall be 0.0. If a variable declaration assignment is used (see 6.2.1), the variable shall take this value as if the assignment occurred in a blocking assignment in an initial construct. It is illegal to redeclare a name already declared by a net, parameter, or variable declaration.

NOTE—In previous versions of the Verilog standard, the term *register* was used to encompass both the **reg**, **integer**, **time**, **real** and **realtime** types; but that the term is no longer used as a Verilog data type.

The syntax for variable declarations is given in Syntax 3-2.

```
integer_declaration ::= (From Annex A - A.2.1.3)
    integer list_of_variable_identifiers ;
real_declaration ::=
    real list_of_real_identifiers ;
realtime_declaration ::=
    realtime list_of_real_identifiers ;
reg_declaration ::=
    reg [ signed ] [ range ] list_of_variable_identifiers ;
time_declaration ::=
    time list_of_variable_identifiers ;
real_type ::= (From Annex A - A.2.2.1)
    real_identifier [ = constant_expression ]
    | real_identifier dimension { dimension }
variable_type ::=
    variable_identifier [ = constant_expression ]
    | variable_identifier dimension { dimension }
list_of_real_identifiers ::= (From Annex A - A.2.3)
    real_type { , real_type }
list_of_variable_identifiers ::=
    variable_type { , variable_type }
dimension ::= (From Annex A - A.2.5)
    [ dimension_constant_expression : dimension_constant_expression ]
range ::=
    [ msb_constant_expression : lsb_constant_expression ]
```

Syntax 3-2—Syntax for variable declaration

If a set of nets or variables share the same characteristics, they can be declared in the same declaration statement.

CAUTION

Variables can be assigned negative values, but only signed regs, integer, real, and realtime variables shall retain the significance of the sign. The unsigned reg and time variables shall treat the value assigned to them as an unsigned value. Refer to 4.1.6 for a description of how signed and unsigned variables are treated by certain Verilog operators.

3.3 Vectors

A net or reg declaration without a range specification shall be considered 1 bit wide and is known as a *scalar*. Multiple bit net and reg data types shall be declared by specifying a range, which is known as a *vector*.

3.3.1 Specifying vectors

The range specification gives addresses to the individual bits in a multibit net or reg. The most significant bit specified by the *msb* constant expression is the left-hand value in the range and the least significant bit specified by the *lsb* constant expression is the right-hand value in the range.

Both *msb* constant expression and *lsb* constant expression shall be constant expressions. The *msb* and *lsb* constant expressions can be any value—positive, negative, or zero. The *lsb* constant expression can be a greater, equal, or lesser value than *msb* constant expression.

Vector nets and regs shall obey laws of arithmetic modulo 2 to the power n (2^n), where n is the number of bits in the vector. Vector nets and regs shall be treated as unsigned quantities, unless the net or reg is declared to be signed or is connected to a port that is declared to be signed (see 12.2.3).

Examples:

```
wand w;           // a scalar net of type "wand"
tri [15:0] busa;   // a tri-state 16-bit bus
triereg (small) storeit; // a charge storage node of strength small
reg a;            // a scalar reg
reg[3:0] v;       // a 4-bit vector reg made up of (from most to
                  // least significant) v[3], v[2], v[1], and v[0]
reg signed [3:0] signed_reg; // a 4-bit vector in range -8 to 7
reg [-1:4] b;     // a 6-bit vector reg
wire w1, w2;      // declares two wires
reg [4:0] x, y, z; // declares three 5-bit regs
```

NOTES

1—Implementations may set a limit on the maximum length of a vector, but they will at least be 65536 (2^{16}) bits.

2—Implementations do not have to detect overflow of integer operations.

3.3.2 Vector net accessibility

Vectored and *scalared* shall be optional advisory keywords to be used in vector net or reg declaration. If these keywords are implemented, certain operations on vectors may be restricted. If the keyword **vectored** is used, bit and part selects and strength specifications may not be permitted, and the PLI may consider the object *unexpanded*. If the key-

word **scalared** is used, bit and part selects of the object shall be permitted, and the PLI shall consider the object *expanded*.

Examples:

```
tri1 scalared [63:0] bus64; //a bus that will be expanded
tri vectored [31:0] data;  //a bus that may or may not be expanded
```

3.4 Strengths

There are two types of *strengths* that can be specified in a net declaration. They are as follows:

charge strength shall only be used when declaring a net of type **triereg**

drive strength shall only be used when placing a continuous assignment on a net in the same statement that declares the net

Gate declarations can also specify a drive strength. See Section 7 for more information on gates and for information on strengths.

3.4.1 Charge strength

The charge strength specification shall be used only with triereg nets. A triereg net shall be used to model charge storage; charge strength shall specify the relative size of the capacitance indicated by one of the following keywords:

- **small**
- **medium**
- **large**

The default charge strength of a triereg net shall be **medium**.

A triereg net can model a charge storage node whose charge decays over time. The simulation time of a charge decay shall be specified in the delay specification for the triereg net (see 7.13.2).

3.4.2 Drive strength

The drive strength specification allows a continuous assignment to be placed on a net in the same statement that declares that net. See Section 6 for more details. Net strength properties are described in detail in Section 7.

3.5 Implicit declarations

The syntax shown in 3.2 shall be used to declare nets and variables explicitly. In the absence of an explicit declaration, an implicit net of default net type shall be assumed in the following circumstances:

- If an identifier is used in a port expression declaration, then an implicit net of type **wire** shall be assumed, with the vector width of the port expression declaration. See 12.3.3 for a discussion of port expression declarations.
- If an identifier is used in the terminal list of a primitive instance or a module instance, and that identifier has not been explicitly declared previously in one of the declaration statements of the instantiating module, then an implicit scalar net of default net type shall be assumed. See Section 19 for a discussion of control of the type for implicitly declared nets with the **`default_nettype** compiler directive.

3.6 Net initialization

The default initialization value for a net shall be the value *z*. Nets with drivers shall assume the output value of their drivers. The *tri*reg net is an exception. The *tri*reg net shall default to the value *x*, with the strength specified in the net declaration (**small**, **medium**, or **large**).

3.7 Net types

There are several distinct types of nets, as shown in Table 3-1.

Table 3-1—Net types

wire	tri	tri0	supply0
wand	triand	tri1	supply1
wor	trior	tri	

3.7.1 Wire and tri nets

The *wire* and *tri* nets connect elements. The net types *wire* and *tri* shall be identical in their syntax and functions; two names are provided so that the name of a net can indicate the purpose of the net in that model. A *wire* net can be used for nets that are driven by a single gate or continuous assignment. The *tri* net type can be used where multiple drivers drive a net.

Logical conflicts from multiple sources of the same strength on a *wire* or a *tri* net result in *x* (unknown) values.

Table 3-2 is a truth table for resolving multiple drivers on *wire* and *tri* nets. Note that it assumes equal strengths for both drivers. Please refer to 7.9 for a discussion of logic strength modeling.

Table 3-2—Truth table for wire and tri nets

wire/ tri	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

3.7.2 Wired nets

Wired nets are of type *wor*, *wand*, *trior*, and *triand*, and are used to model wired logic configurations. Wired nets use different truth tables to resolve the conflicts that result when multiple drivers drive the same net. The *wor* and *trior* nets shall create *wired or* configurations, such that when any of the drivers is 1, the resulting value of the net is 1. The *wand* and *triand* nets shall create *wired and* configurations, such that if any driver is 0, the value of the net is 0.

The net types *wor* and *trior* shall be identical in their syntax and functionality. The net types *wand* and *triand* shall be identical in their syntax and functionality. Table 3-3 and Table 3-4 give the truth tables for wired nets. Note that they assume equal strengths for both drivers. See 7.9 for a discussion of logic strength modeling.

Table 3-3—Truth table for wand and triand nets

wand/ triand	0	1	x	z
0	0	0	0	0
1	0	1	x	1
x	0	x	x	x
z	0	1	x	z

Table 3-4—Truth table for wor and trior nets

wor/ trior	0	1	x	z
0	0	1	x	0
1	1	1	1	1
x	x	1	x	x
z	0	1	x	z

3.7.3 Trireg net

The *tri*reg net stores a value and is used to model charge storage nodes. A trireg net can be in one of two states:

driven state When at least one driver of a trireg net has a value of 1, 0, or x, the resolved value propagates into the trireg net and is the driven value of the trireg net.

capacitive state When all the drivers of a trireg net are at the high-impedance value (z), the trireg net retains its last driven value; the high-impedance value does not propagate from the driver to the trireg.

The strength of the value on the trireg net in the capacitive state can be **small**, **medium**, or **large**, depending on the size specified in the declaration of the trireg net. The strength of a trireg net in the driven state can be **supply**, **strong**, **pull**, or **weak**, depending on the strength of the driver.

Examples:

Figure 3-1 shows a schematic that includes a trireg net whose size is **medium**, its driver, and the simulation results.

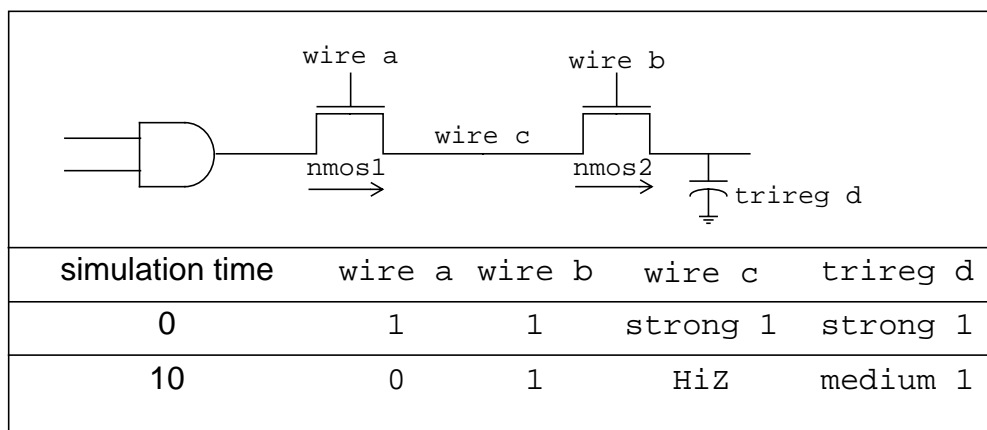


Figure 3-1—Simulation values of a trireg and its driver

- At simulation time 0, wire a and wire b have a value of 1. A value of 1 with a **strong** strength propagates from the **and** gate through the **nmos** switches connected to each other by wire c into trireg net d.
- At simulation time 10, wire a changes value to 0, disconnecting wire c from the **and** gate. When wire c is no longer connected to the **and** gate, the value of wire c changes to HiZ. The value of wire b remains 1 so wire c remains connected to trireg net d through the nmos2 switch. The HiZ value does not propagate from wire c into trireg net d. Instead, trireg net d enters the capacitive state, storing its last driven value of 1. It stores the 1 with a **medium** strength.

3.7.3.1 Capacitive networks

A capacitive network is a connection between two or more trireg nets. In a capacitive network whose trireg nets are in the capacitive state, logic and strength values can propagate between trireg nets.

Examples:

Figure 3-2 shows a capacitive network in which the logic value of some trireg nets change the logic value of other trireg nets of equal or smaller size.

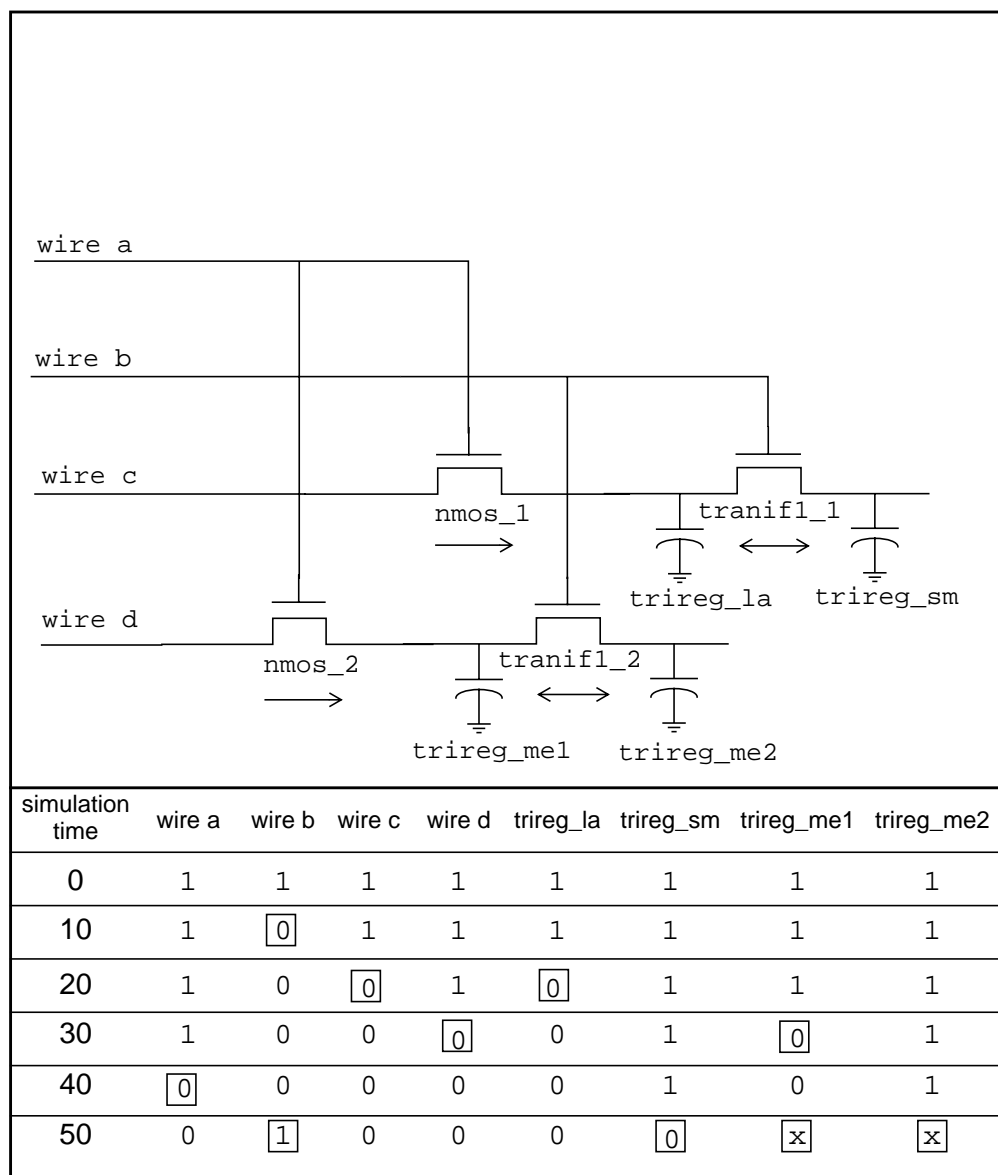


Figure 3-2—Simulation results of a capacitive network

In Figure 3-2, the capacitive strength of `trireg_la` net is **large**, `trireg_me1` and `trireg_me2` are **medium**, and `trireg_sm` is **small**. Simulation reports the following sequence of events:

- At simulation time 0, wire a and wire b have a value of 1. The wire c drives a value of 1 into `trireg_la` and `trireg_sm`; wire d drives a value of 1 into `trireg_me1` and `trireg_me2`.
- At simulation time 10, the value of wire b changes to 0, disconnecting `trireg_sm` and `trireg_me2` from their drivers. These trireg nets enter the capacitive state and store the value 1, their last driven value.
- At simulation time 20, wire c drives a value of 0 into `trireg_la`.
- At simulation time 30, wire d drives a value of 0 into `trireg_me1`.
- At simulation time 40, the value of wire a changes to 0, disconnecting `trireg_la` and `trireg_me1` from their drivers. These trireg nets enter the capacitive state and store the value 0.

- f) At simulation time 50, the value of wire b changes to 1.

This change of value in wire b connects `trireg_sm` to `trireg_la`; these `trireg` nets have different sizes and stored different values. This connection causes the smaller `trireg` net to store the value of the larger `trireg` net, and `trireg_sm` now stores a value of 0.

This change of value in wire b also connects `trireg_me1` to `trireg_me2`; these `trireg` nets have the same size and stored different values. The connection causes both `trireg_me1` and `trireg_me2` to change value to x.

In a capacitive network, charge strengths propagate from a larger `trireg` net to a smaller `trireg` net. Figure 3-3 shows a capacitive network and its simulation results.

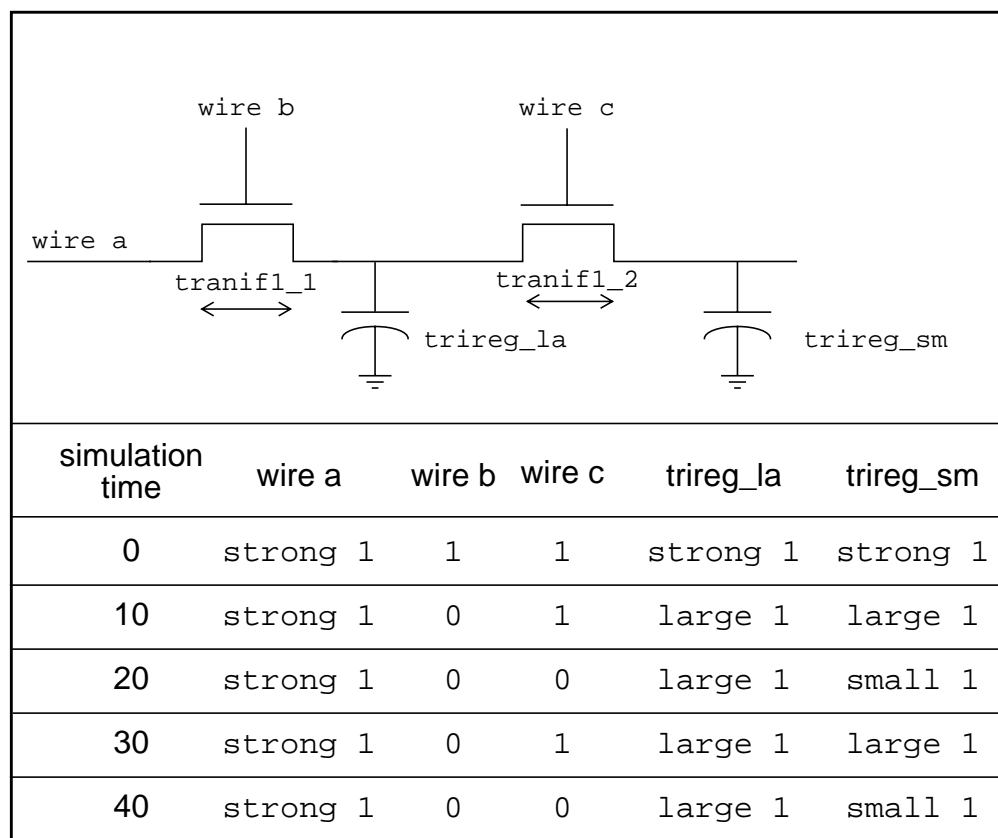


Figure 3-3—Simulation results of charge sharing

In Figure 3-3, the capacitive strength of `trireg_la` is **large** and the capacitive strength of `trireg_sm` is **small**. Simulation reports the following results:

- At simulation time 0, the values of wire a, wire b, and wire c are 1, and wire a drives a strong 1 into `trireg_la` and `trireg_sm`.
- At simulation time 10, the value of wire b changes to 0, disconnecting `trireg_la` and `trireg_sm` from wire a. The `trireg_la` and `trireg_sm` nets enter the capacitive state. Both `trireg` nets share the **large** charge of `trireg_la` because they remain connected through `tranif1_2`.
- At simulation time 20, the value of wire c changes to 0, disconnecting `trireg_sm` from `trireg_la`. The `trireg_sm` no longer shares **large** charge of `trireg_la` and now stores a **small** charge.

- d) At simulation time 30, the value of wire `c` changes to 1, connecting the two `triereg` nets. These `triereg` nets now share the same charge.
- e) At simulation time 40, the value of wire `c` changes again to 0, disconnecting `triereg_sm` from `triereg_la`. Once again, `triereg_sm` no longer shares the **large** charge of `triereg_la` and now stores a **small** charge.

3.7.3.2 Ideal capacitive state and charge decay

A *triereg* net can retain its value indefinitely or its charge can decay over time. The simulation time of charge decay is specified in the delay specification of the `triereg` net. See 7.14.2 for charge decay explanation.

3.7.4 Tri0 and tri1 nets

The *tri0* and *tri1* nets model nets with resistive *pulldown* and resistive *pullup* devices on them. When no driver drives a `tri0` net, its value is 0. When no driver drives a `tri1` net, its value is 1. The strength of this value is **pull**. See Section 7 for a description of strength modeling.

A `tri0` net is equivalent to a wire net with a continuous 0 value of pull strength driving it. A `tri1` net is equivalent to a wire net with a continuous 1 value of pull strength driving it.

A truth table for `tri0` is shown in Table 3-5. A truth table for `tri1` is shown in Table 3-6.

Table 3-5—Truth table for tri0 net

tri0	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	0

Table 3-6—Truth table for tri1 net

tri1	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	1

3.7.5 Supply nets

The *supply0* and *supply1* nets may be used to model the power supplies in a circuit. These nets shall have **supply** strengths.

3.8 regs

Assignments to a reg are made by procedural assignments (see 6.2 and 9.2). Since the reg holds a value between assignments, it can be used to model hardware registers. Edge-sensitive (i.e., flip-flops) and level sensitive (i.e., RS and transparent latches) storage elements can be modeled. A reg needs not represent a hardware storage element since it can also be used to represent combinatorial logic.

3.9 Integers, reals, times, and realtimes

In addition to modeling hardware, there are other uses for variables in an HDL model. Although reg variables can be used for general purposes such as counting the number of times a particular net changes value, the *integer* and *time* variable data types are provided for convenience and to make the description more self-documenting.

The syntax for declaring **integer**, **time**, **real**, and **realtime** variables is given in Syntax 3-3 (from Syntax 3-2).

```
integer_declaration ::= (From Annex A - A.2.1.3)
    integer list_of_variable_identifiers ;
real_declaration ::=
    real list_of_real_identifiers ;
realtime_declaration ::=
    realtime list_of_real_identifiers ;
time_declaration ::=
    time list_of_variable_identifiers ;
real_type ::= (From Annex A - A.2.2.1)
    real_identifier [ = constant_expression ]
    | real_identifier dimension { dimension }
variable_type ::=
    variable_identifier [ = constant_expression ]
    | variable_identifier dimension { dimension }
list_of_real_identifiers ::= (From Annex A- A.2.3)
    real_type { , real_type }
list_of_variable_identifiers ::=
    variable_type { , variable_type }
dimension ::= (From Annex A - A.2.5)
    [ dimension_constant_expression : dimension_constant_expression ]
```

Syntax 3-3—Syntax for integer, time, real, and realtime declarations

The syntax for list of reg variables is defined in 3.2.2.

An **integer** is a general-purpose variable used for manipulating quantities that are not regarded as hardware registers.

A **time** variable is used for storing and manipulating simulation time quantities in situations where timing checks are required and for diagnostics and debugging purposes. This data type is typically used in conjunction with the **\$time** system function (see Section 17).

The integer and time variables shall be assigned values in the same manner as reg. Procedural assignments shall be used to trigger their value changes.

The time variables shall behave the same as a reg of at least 64 bits. They shall be unsigned quantities, and unsigned arithmetic shall be performed on them. In contrast, integer variables shall be treated as signed quantities. Arithmetic

operations performed on integer variables shall produce 2's complement results.

The Verilog HDL supports *real* number constants and *real* variable data types in addition to integer and time variable data types. Except for the following restrictions, variables declared as real can be used in the same places that integer and time variables are used:

- Not all Verilog HDL operators can be used with real number values. See Table 4-9 for lists of valid and invalid operators for real numbers and real variables.
- Real variables shall not use range in the declaration
- Real variables shall default to an initial value of zero.

The *realtime* declarations shall be treated synonymously with *real* declarations and can be used interchangeably.

Examples:

```
integer a;           // integer value
time last_chng;      // time value
real float ;         // a variable to store real value
realtime rtime ;     // a variable to store time as a real value:
```

NOTE—Implementations may limit the maximum size of an **integer** variable, but they shall at least be 32 bits.

3.9.1 Operators and real numbers

The result of using logical or relational operators on real numbers and real variables is a single-bit scalar value. Not all Verilog HDL operators can be used with expressions involving real numbers and real variables. Table 4-9 lists the valid operators for use with real numbers and real variables. Real number constants and real variables are also prohibited in the following cases:

- Edge descriptors (*posedge*, *negedge*) applied to real variables
- Bit-select or part-select references of variables declared as *real*
- Real number index expressions of bit-select or part-select references of vectors
- Declaration of memories (arrays of real variables)

3.9.2 Conversion

Real numbers shall be converted to integers by rounding the real number to the nearest integer, rather than by truncating it. Implicit conversion shall take place when a real number is assigned to an integer. The ties shall be rounded away from zero.

Implicit conversion shall take place when an expression is assigned to a real. Individual bits that are x or z in the net or the variable shall be treated as zero upon conversion.

See Section 17 for a discussion of system tasks that perform explicit conversion.

3.10 Arrays

An array declaration for a net or a variable declares an element type which is either scalar or vector (see 3.3). For example:

Declaration	Element Type
reg x[11:0];	scalar reg
wire [0:7] y[5:0];	seven-bit-wide vector wire indexed from 0 to 7
reg [31:0] x [127:0];	thirty-two-bit-wide reg

NOTE—Array size does not affect the element size.

Arrays can be used to group elements of the declared element type into multi-dimensional objects. Arrays shall be declared by specifying the element address range(s) after the declared identifier. Each dimension shall be represented by an address range. See 3.2.1 and 3.2.2 for net and variable declarations. The expression(s) that specify the indices of the array shall be constant expressions. The value of the constant expression can be a positive integer, a negative integer, or zero.

One declaration statement can be used for declaring both arrays and elements of the declared data type. This ability makes it convenient to declare both arrays and elements that match the element vector width in the same declaration statement.

An element can be assigned a value in a single assignment, but complete or partial array dimensions cannot. Nor can complete or partial array dimensions be used to provide a value to an expression. To assign a value to an element of an array, an index for every dimension shall be specified. The index can be an expression. This option provides a mechanism to reference different array elements depending on the value of other variables and nets in the circuit. For example, a program counter reg can be used to index into a RAM.

3.10.1 Net arrays

Arrays of nets can be used to connect ports of generated instances. Each element of the array can be used in the same fashion as a scalar or vector net.

3.10.2 reg and variable arrays

Arrays for all variables types (**reg**, **integer**, **time**, **real**, **realtime**) shall be possible.

3.10.3 Memories

A one dimensional array with elements of type reg is also called a memory. These memories can be used to model read-only memories (ROMs), random access memories (RAMs), and reg files. Each reg in the array is known as an *element* or *word* and is addressed by a single array index.

An *n*-bit reg can be assigned a value in a single assignment, but a complete memory cannot. To assign a value to a memory word, an index shall be specified. The index can be an expression. This option provides a mechanism to reference different memory words, depending on the value of other variables and nets in the circuit. For example, a program counter reg could be used to index into a RAM.

3.10.3.1 Array examples

3.10.3.1.1 Array declarations

```

reg [7:0] mema[0:255]; // declares a memory mema of 256 8-bit
                        // registers. The indices are 0 to 255

reg arrayb[7:0][0:255]; // declare a two dimensional array of
                        // one bit registers
wire w_array[7:0][5:0]; // declare array of wires
integer inta[1:64];      // an array of 64 integer values
time chng_hist[1:1000]  // an array of 1000 time values
integer t_index;

```

3.10.3.1.2 Assignment to array elements

The assignment statements in this section assume the presence of the declarations in 3.10.3.1.1.

```

rega = 0; // Legal syntax
mema = 0; // Illegal syntax- Attempt to write to entire array
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]
mema[1] = 0; //Assigns 0 to the second element of mema
arrayb[1][0] = 0; // Assigns 0 to the bit referenced by indices
                  // [1][0]
inta[4] = 33559; // Assign decimal number to integer in array
chng_hist[t_index] = $time; // Assign current simulation time to
                           // element addressed by integer index

```

NOTE—Implementations may limit the maximum size of an array, but they shall at least be 16777216 (2²⁴).

3.10.3.1.3 Memory differences

A memory of n 1-bit regs is different from an n -bit vector reg

```

reg [1:n] rega; // An n-bit register is not the same
reg mema [1:n]; // as a memory of n 1-bit registers

```

3.11 Parameters

Verilog HDL parameters do not belong to either the variable or the net group. Parameters are not variables, they are constants. There are two types of parameters: module parameters and specify parameters. It is illegal to redeclare a name already declared by a net, parameter or variable declaration.

Both types of parameters accept a range specification. By default, *parameters* and *specparams* shall be as wide as

necessary to contain the value of the constant, except when a range specification is present.

3.11.1 Module parameters

The syntax for parameter declarations is given in Syntax 3-4.

```

local_parameter_declaration ::= (From Annex A - A.2.2.1)
    localparam [ signed ] [ range ] list_of_param_assignments ;
    | localparam integer list_of_param_assignments ;
    | localparam real list_of_param_assignments ;
    | localparam realtime list_of_param_assignments ;
    | localparam time list_of_param_assignments ;
parameter_declaration ::=
    parameter [ signed ] [ range ] list_of_param_assignments ;
    | parameter integer list_of_param_assignments ;
    | parameter real list_of_param_assignments ;
    | parameter realtime list_of_param_assignments ;
    | parameter time list_of_param_assignments ;
list_of_param_assignments ::= (From Annex A - A.2.3)
    param_assignment { , param_assignment }
param_assignment ::= (From Annex A - A.2.4)
    parameter_identifier = constant_expression
range ::= (From Annex A - A.2.5)
    [ msb_constant_expression : lsb_constant_expression ]

```

Syntax 3-4—Syntax for parameter declaration

The *list_of_param_assignments* shall be a comma-separated list of assignments, where the right hand side of the assignment shall be a constant expression; that is, an expression containing only constant numbers and previously defined parameters. (See Section 4.)

The *list_of_param_assignments* can appear in a module as a set of *module_items* or in the module declaration in the *module_parameter_port_list*. (See 12.1). If any *param_assignments* appear in a *module_parameter_port_list*, then any *param_assignments* that appear in the module become local parameters and shall not be overridden by any method.

Parameters represent constants; hence, it is illegal to modify their value at runtime. However, module parameters can be modified at compilation time to have values that are different from those specified in the declaration assignment. This allows customization of module instances. A parameter can be modified with the **defparam** statement or in the module instance statement. Typical uses of parameters are to specify delays and width of variables. See Section 12 for details on parameter value assignment.

A module parameter can have a *type* specification and a *range* specification. The type and range of module parameters shall be in accordance with the following rules:

- A parameter declaration with no type or range specification shall default to the type and range of the final value assigned to the parameter, after any value overrides have been applied.
- A parameter with a range specification, but with no type specification, shall be the range of the parameter declaration and shall be unsigned. The sign and range shall not be affected by value overrides.
- A parameter with a type specification, but with no range specification, shall be of the type specified. A signed parameter shall default to the range of the final value assigned to the parameter, after any value overrides have been applied.

- A parameter with a signed type specification and with a range specification shall be signed, and shall be the range of its declaration. The sign and range shall not be affected by value overrides.
- A parameter with no range specification, and with either a signed type specification or no type specification, shall have an implied range with an *lsb* equal to 0 and an *msb* equal to one less than the size of the final value assigned to the parameter.
- A parameter with no range specification, and with either a signed type specification or no type specification, and for which the final value assigned to it is unsized, shall have an implied range with an *lsb* equal to 0 and an *msb* equal to an implementation-dependent value of at least 31.

Examples:

```

parameter msb = 7;                // defines msb as a constant value 7
parameter e = 25, f = 9;          // defines two constant numbers
parameter r = 5.7;                // declares r as a real parameter
parameter byte_size = 8,
        byte_mask = byte_size - 1;
parameter average_delay = (r + f) / 2;

parameter signed [3:0] mux_selector = 0;
parameter real r1 = 3.5e17;
parameter p1 = 13'h7e;
parameter [31:0] dec_const = 1'b1;    // value converted to 32 bits
parameter newconst = 3'h4;            // implied range of [2:0]
parameter newconst = 4;                // implied range of at least [31:0]

parameter signed [3:0] mux_selector = 0;
parameter real r1 = 3.5e17;
parameter p1 = 13'h7e;
parameter [31:0] dec_const = 1'b1;    // valued converted to 32 bits

```

See 3.9.2 for conversion between parameter types.

3.11.2 Local parameters - **localparam**

Verilog HDL **localparam** - local parameter(s) are identical to parameters except that they can not directly be modified with the **defparam** statement or by the ordered or named parameter value assignment. Local parameters can be assigned to a constant expression containing a parameter which can be modified with the **defparam** statement or by the ordered or named parameter value assignment. See 12.1.3 for details.

The syntax for local parameter declarations is given in Syntax 3-4.

3.11.3 Specify parameters

The syntax for declaring specify parameters is shown in Syntax 3-5.

```

specparam_declaration ::= (From Annex A - A.2.2.1)
    specparam [ range ] list_of_specparam_assignments ;
list_of_specparam_assignments ::= (From Annex A- A.2.3)
    specparam_assignment { , specparam_assignment }
specparam_assignment ::= (From Annex A - A.2.4)
    specparam_identifier = constant_mintypmax_expression
    | pulse_control_specparam
pulse_control_specparam ::=
    PATHPULSE$ = ( reject_limit_value [ , error_limit_value ] ) ;
    | PATHPULSE$specify_input_terminal_descriptor$specify_output_terminal_descriptor
    = ( reject_limit_value [ , error_limit_value ] ) ;
error_limit_value ::=
    limit_value
reject_limit_value ::=
    limit_value
limit_value ::=
    constant_mintypmax_expression
range ::= (From Annex A - A.2.5)
    [ msb_constant_expression : lsb_constant_expression ]

```

Syntax 3-5—Syntax of the specparam declaration

The keyword **specparam** declares a special type of parameter which is intended only for providing timing and delay values, but can appear in any expression that is not assigned to a parameter and is not part of the range specification of a declaration. Originally permitted only in specify blocks (see Section 14), *with this revision* specify parameters (also called *specparams*) are now permitted both within the specify block and in the main module body.

A specify parameter declared outside a specify block shall be declared before it is referenced. The value assigned to a specify parameter can be any constant expression. A specify parameter can be used as part of a constant expression for a subsequent specify parameter declaration. Unlike a module parameter, a specify parameter cannot be modified from within the language, but it may be modified through SDF annotation (see Section 16).

The specify parameters and module parameters shall not be interchangeable. In addition, module parameters shall not be assigned a constant expression that includes any specify parameters. Table 3-7 summarizes the differences between the two types of parameter declarations.

Table 3-7—Differences between specparams and parameters

Specparams (specify parameter)	Parameters (module parameter)
Use keyword specparam	Use keyword parameter
Shall be declared <i>inside</i> a module or specify block	Shall be declared <i>outside</i> specify blocks
May only be used inside a module or specify block	May not be used inside specify blocks
May be assigned specparams and parameters	May not be assigned specparams
Use SDF annotation to override values	Use defparam or instance declaration parameter value passing to override values

A specify parameter can have a range specification. The range of specify parameters shall be in accordance with the following rules:

- A specparam declaration with no range specification shall default to the range of the final value assigned to the parameter, after any value overrides have been applied.
- A specparam with a range specification shall be the range of the parameter declaration. The range shall not be affected by value overrides.

Examples:

```

specify
  specparam tRise_clk_q = 150, tFall_clk_q = 200;
  specparam tRise_control = 40, tFall_control = 50;
endspecify

```

The lines between the keywords **specify** and **endspecify** declare four specify parameters. The first line declares specify parameters called `tRise_clk_q` and `tFall_clk_q` with values 150 and 200 respectively; the second line declares `tRise_control` and `tFall_control` specify parameters with values 40 and 50 respectively.

Examples:

```

module RAM16GEN (DOUT, DIN, ADR, WE, CE)
specparam dhold = 1.0;
specparam ddly = 1.0;
parameter width = 1;
parameter regsize = dhold + 1.0;    // Illegal - can't assign
                                     // specparams to parameters
endmodule

```

3.12 Name spaces

In Verilog HDL, there are six name spaces; two are global and four are local. The global name spaces are *definitions* and *text macros*. The *definitions name space* unifies all the **module** (see 12.1), **macromodule** (see 12.1), and **primitive** (see 8.1) definitions. Once a name is used to define a module, macromodule, or primitive, the name shall not be used again to declare another module, macromodule, or primitive.

The *text macro name space* is global. Since text macro names are introduced and used with a leading ``` character, they remain unambiguous with any other name space (see 19.3). The text macro names are defined in the linear order of appearance in the set of input files that make up the description of the design unit. Subsequent definitions of the same name override the previous definitions for the balance of the input files.

There are four local name spaces: *block*, *module*, *port*, and *specify block*. Once a name is defined within one of the four name spaces, it shall not be defined again with the same type or another type.

The *block name space* is introduced by the named block (see 9.8), function (see 10.3), and task (see 10.2) constructs. It unifies the definitions of the named blocks, functions, tasks, parameters, named events and the variable type of declaration (see 3.2.2). The variable type of declaration includes the **reg**, **integer**, **time**, **real**, and **realtime** declarations.

The *module name space* is introduced by the **module**, **macromodule**, and **primitive** constructs. It unifies the definition of functions, tasks, named blocks, instance names, parameters, named events, net type of declaration, and variable type of declaration. The net type of declaration includes **wire**, **wor**, **wand**, **tri**, **trior**, **triand**, **tri0**, **tri1**, **triereg**, **supply0**, and **supply1** (see 3.7).

The *port name space* is introduced by the **module**, **macromodule**, **primitive**, **function**, and **task** constructs. It provides a means of structurally defining connections between two objects that are in two different name spaces. The connection can be unidirectional (either **input** or **output**) or bidirectional (**inout**). The port name space overlaps the module and the block name spaces. Essentially, the port name space specifies the type of connection between names in different name spaces. The port type of declarations include **input**, **output**, and **inout** (see 12.3). A port name introduced in the port name space may be reintroduced in the module name space by declaring a variable or a wire

with the same name as the port name.

The *specify block name space* is introduced by the **specify** construct (see 14.2). A **specparam** name can be defined and used only in the specify block name space. Any other type of name cannot be defined in this name space.

Section 4

Expressions

This section describes the operators and operands available in the Verilog HDL and how to use them to form expressions.

An *expression* is a construct that combines *operands* with *operators* to produce a result that is a function of the values of the operands and the semantic meaning of the operator. Any legal operand, such as a net bit-select, without any operator is considered an expression. Wherever a value is needed in a Verilog HDL statement, an expression can be used.

Some statement constructs require an expression to be a *constant expression*. The operands of a constant expression consist of constant numbers, parameter names, constant bit-selects of parameters, constant part-selects of parameters, and *constant function calls* (see 10.3.5) only, but they can use any of the operators defined in Table 4-1.

A *scalar expression* is an expression that evaluates to a scalar (single-bit) result. If the expression evaluates to a vector (multibit) result, then the least significant bit of the result is used as the scalar result.

The data types **reg**, **integer**, **time**, **real**, and **realtime** are all variable data types. Descriptions pertaining to variable usage apply to all of these data types.

An *operand* can be one of the following:

- Constant number (including real)
- Net
- Variables of type reg, integer, time, real, and realtime
- Net bit-select
- Bit-select of type reg, integer, and time
- Net part-select
- Part-select of type reg, integer, and time
- Array element
- A call to a user-defined function or system-defined function that returns any of the above

4.1 Operators

The symbols for the Verilog HDL operators are similar to those in the C programming language. Table 4-1 lists these operators.

Table 4-1—Operators in the Verilog HDL

{ } { }	Concatenation, replication
+ - * / **	Arithmetic
%	Modulus
> >= < <=	Relational
!	Logical negation

Table 4-1—Operators in the Verilog HDL (*continued*)

&&	Logical and
	Logical or
==	Logical equality
!=	Logical inequality
===	Case equality
!==	Case inequality
~	Bit-wise negation
&	Bit-wise and
	Bit-wise inclusive or
^	Bit-wise exclusive or
^~ or ~^	Bit-wise equivalence
&	Reduction and
~&	Reduction nand
	Reduction or
~	Reduction nor
^	Reduction xor
~^ or ^~	Reduction xnor
<<	Logical left shift
>>	Logical right shift
<<<	Arithmetic left shift
>>>	Arithmetic right shift
? :	Conditional
or	Event or

4.1.1 Operators with real operands

The operators shown in Table 4-2 shall be legal when applied to real operands. All other operators shall be considered illegal when used with real operands.

Table 4-2—Legal operators for use in real expressions

unary + unary -	Unary operators
+ - * / **	Arithmetic
> >= < <=	Relational
! &&	Logical
== !=	Logical equality

Table 4-2—Legal operators for use in real expressions (*continued*)

?:	Conditional
or	Event or

The result of using logical or relational operators on real numbers is a single-bit scalar value.

Table 4-3 lists operators that shall not be used to operate on real numbers.

Table 4-3—Operators not allowed for real expressions


{ } { { } }	Concatenate, replicate
%	Modulus
=== !=	Case equality
~ & ^ ^~ ^~^	Bit-wise
^ ^~ ^~^ & ~& ~	Reduction
<< >> <<< >>>	Shift

See 3.9.1 for more information on use of real numbers.

4.1.2 Binary operator precedence

The precedence order of *binary operators* and the *conditional operator* (?:) is shown in Table 4-4. The Verilog HDL has two equality operators. They are discussed in 4.1.8.

Table 4-4—Precedence rules for operators

+ - ! ~ (unary)	Highest precedence
**	
* / %	
+ - (binary)	
<< >> <<< >>>	
< <= > >=	
== != === !==	
& ~&	
^ ^~ ^~^	
~	
&&	
?: (conditional operator)	Lowest precedence

Operators shown on the same row in Table 4-4 shall have the same precedence. Rows are arranged in order of

decreasing precedence for the operators. For example, `*`, `/`, and `%` all have the same precedence, which is higher than that of the binary `+` and `-` operators.

All operators shall associate left to right with the exception of the conditional operator, which shall associate right to left. Associativity refers to the order in which the operators having the same precedence are evaluated. Thus, in the following example B is added to A and then C is subtracted from the result of A+B.

```
A + B - C
```

When operators differ in precedence, the operators with higher precedence shall associate first. In the following example, B is divided by C (division has higher precedence than addition) and then the result is added to A.

```
A + B / C
```

Parentheses can be used to change the operator precedence.

```
(A + B) / C    // not the same as A + B / C
```

4.1.3 Using integer numbers in expressions

Integer numbers can be used as operands in expressions. An integer number can be expressed as

- An unsized, unbased integer (e.g., 12)
- An unsized, based integer (e.g., `'d12`, `'sd12`)
- A sized, based integer (e.g., `16'd12`, `16'sd12`)

A negative value for an integer with no base specifier shall be interpreted differently than for an integer with a base specifier. An integer with no base specifier shall be interpreted as a signed value in 2's complement form. An integer with an unsigned base specifier shall be interpreted as an unsigned value.

Example:

This example shows four ways to write the expression “minus 12 divided by 3.” Note that `-12` and `- 'd12` both evaluate to the same 2's complement bit pattern, but, in an expression, the `- 'd12` loses its identity as a signed negative number.

```
integer IntA;
IntA = -12 / 3;           // The result is -4.

IntA = -'d 12 / 3;       // The result is 1431655761.

IntA = -'sd 12 / 3;      // The result is -4.

IntA = -4'sd 12 / 3;     // -4'sd12 is the negative of the 4-bit
                        // quantity 1100, which is -4. -(-4) = 4.
```

4.1.4 Expression evaluation order

The operators shall follow the associativity rules while evaluating an expression as described in 4.1.2. However, if the final result of an expression can be determined early, the entire expression need not be evaluated. This is called *short-circuiting* an expression evaluation.

Example:

```
reg regA, regB, regC, result ;
result = regA & (regB | regC) ;
```

If regA is known to be zero, the result of the expression can be determined as zero without evaluating the sub-expression `regB | regC`.

4.1.5 Arithmetic operators

The binary arithmetic operators are given in Table 4-5.

Table 4-5—Arithmetic operators defined

<code>a + b</code>	a plus b
<code>a - b</code>	a minus b
<code>a * b</code>	a multiplied by b (or a times b)
<code>a / b</code>	a divided by b
<code>a % b</code>	a modulo b
<code>a ** b</code>	a to the power of b

The integer division shall truncate any fractional part toward zero. For the division or modulus operators, if the second operand is a zero, then the entire result value shall be `x`. The modulus operator, for example `y % z`, gives the remainder when the first operand is divided by the second, and thus is zero when `z` divides `y` exactly. The result of a modulus operation shall take the sign of the first operand.

The result of the power operator shall be real if either operand is a real, integer, or signed. If both operands are unsigned then the result shall be unsigned. The result of the power operator is unspecified if the first operand is zero and the second operand is non-positive, or if the first operand is negative and the second operand is not an integral value.

The unary arithmetic operators shall take precedence over the binary operators. The unary operators are given in Table 4-6.

Table 4-6—Unary operators defined

<code>+m</code>	Unary plus m (same as m)
<code>-m</code>	Unary minus m

For the arithmetic operators, if any operand bit value is the unknown value `x` or the high-impedance value `z`, then the entire result value shall be `x`.

Example:

Table 4-7 gives examples of modulus operations.

Table 4-7—Examples of modulus operators

Modulus expression	Result	Comments
10 % 3	1	10/3 yields a remainder of 1
11 % 3	2	11/3 yields a remainder of 2
12 % 3	0	12/3 yields no remainder
-10 % 3	-1	The result takes the sign of the first operand
11 % -3	2	The result takes the sign of the first operand
-4'd12 % 3	1	-4'd12 is seen as a large, positive number that leaves a remainder of 1 when divided by 3

4.1.6 Arithmetic expressions with regs and integers

An arithmetic operation on a reg type variable shall be treated differently than an arithmetic operation on an integer data type. A reg data type shall be treated as an *unsigned* value and an integer data type shall be treated as a *signed* value. Thus, if a sized constant with a negative value is stored in a reg type variable, a positive constant, which is a 2's complement of the sized constant, shall be the value stored in the reg type variable. When this reg is used in an arithmetic expression, the positive constant shall be used as the value of the reg. In contrast, if a sized constant with a negative value is stored in an integer type variable and used in an arithmetic expression, the expression shall evaluate using signed arithmetic.

Table 4-8 lists how arithmetic operators interpret each data type.

Table 4-8—Data type interpretation by arithmetic operators

Data type	Interpretation
unsigned net	Unsigned
signed net	Signed, 2's complement
unsigned reg	Unsigned
signed reg	Signed, 2's complement
integer	Signed, 2's complement
time	Unsigned
real, realtime	Signed, floating point

Example:

The following example shows various ways to divide “minus twelve by three”— using **integer** and **reg** data types in expressions.


```

integer intA;
reg [15:0] regA;
reg signed [15:0] regS;

intA = -4'd12;
regA = intA / 3; // expression result is -4,
                // intA is an integer data type, regA is 65532

regA = -4'd12; // regA is 65524
intA = regA / 3; // expression result is 21841,
                // regA is a reg data type

intA = -4'd12 / 3; // expression result is 1431655761.
                // -4'd12 is effectively a 32-bit reg data type

regA = -12 / 3; // expression result is -4, -12 is effectively
               // an integer data type. regA is 65532

regS = -12 / 3; // expression result is -4. regS is a signed
               // reg

regS = -4'sd12 / 3; // expression result is 1. -4'sd12 is actually
                  // 4. The rules for integer division yield 4/3==1

```

4.1.7 Relational operators

Table 4-9 lists and defines the relational operators.

Table 4-9—Definitions of the relational operators

$a < b$	a less than b
$a > b$	a greater than b
$a \leq b$	a less than or equal to b
$a \geq b$	a greater than or equal to b

An expression using these *relational operators* shall yield the scalar value 0 if the specified relation is *false* or the value 1 if it is *true*. If either operand of a relational operator contains an unknown (x) or high impedance (z) value, then the result shall be a 1-bit unknown value (x).

When two operands of unequal bit lengths are used, the smaller operand shall be zero filled on the most significant bit side to extend to the size of the larger operand.

All the relational operators shall have the same precedence. Relational operators shall have lower precedence than arithmetic operators.

Examples:

The following examples illustrate the implications of this precedence rule:

```

a < foo - 1           // this expression is the same as
a < (foo - 1)         // this expression, but . . .
foo - (1 < a)         // this one is not the same as
foo - 1 < a           // this expression

```

When `foo - (1 < a)` evaluates, the relational expression evaluates first and then either zero or one is subtracted from `foo`. When `foo - 1 < a` evaluates, the value of `foo` operand is reduced by one and then compared with `a`.

When both operands of a relational expression are signed integral operands (an integer, or a unsized, unbased integer) then the expression shall be interpreted as a comparison between signed values. When either operand of a relational expression is a real operand then the other operand shall be converted to an equivalent real value, and the expression shall be interpreted as a comparison between two real values.

Otherwise the expression shall be interpreted as a comparison between unsigned values.

4.1.8 Equality operators

The *equality operators* shall rank lower in precedence than the relational operators. Table 4-10 lists and defines the equality operators.

Table 4-10—Definitions of the equality operators

<code>a === b</code>	a equal to b, including x and z
<code>a !== b</code>	a not equal to b, including x and z
<code>a == b</code>	a equal to b, result may be unknown
<code>a != b</code>	a not equal to b, result may be unknown

All four equality operators shall have the same precedence. These four operators compare operands bit for bit, with zero filling if the two operands are of unequal bit length. As with the relational operators, the result shall be 0 if comparison fails, 1 if it succeeds.

For the *logical equality* and *logical inequality* operators (`==` and `!=`), if, due to unknown or high-impedance bits in the operands, the relation is ambiguous, then the result shall be a one bit unknown value (x).

For the *case equality* and *case inequality* operators (`===` and `!==`), the comparison shall be done just as it is in the procedural case statement (see 9.5). Bits that are x or z shall be included in the comparison and shall match for the result to be considered equal. The result of these operators shall always be a known value, either 1 or 0.

4.1.9 Logical operators

The operators *logical and* (`&&`) and *logical or* (`||`) are logical connectives. The result of the evaluation of a logical comparison shall be 1 (defined as *true*), 0 (defined as *false*), or, if the result is ambiguous, the unknown value (x). The precedence of `&&` is greater than that of `||`, and both are lower than relational and equality operators.

A third logical operator is the unary *logical negation* operator (`!`). The negation operator converts a nonzero or true operand into 0 and a zero or false operand into 1. An ambiguous truth value remains as x.

Examples:

Example 1—If reg alpha holds the integer value 237 and beta holds the value zero, then the following examples perform as described:

```
regA = alpha && beta;    // regA is set to 0
regB = alpha || beta;    // regB is set to 1
```

Example 2—The following expression performs a logical and of three subexpressions without needing any parentheses:

```
a < size-1 && b != c && index != lastone
```

However, it is recommended for readability purposes that parentheses be used to show very clearly the precedence intended, as in the following rewrite of this example:

```
(a < size-1) && (b != c) && (index != lastone)
```

Example 3—A common use of ! is in constructions like the following:

```
if (!inword)
```

In some cases, the preceding construct makes more sense to someone reading the code than this equivalent construct:

```
if (inword == 0)
```

4.1.10 Bit-wise operators

The *bit-wise operators* shall perform bit-wise manipulations on the operands—that is, the operator shall combine a bit in one operand with its corresponding bit in the other operand to calculate one bit for the result. Logic Tables 4-11 through 4-15 show the results for each possible calculation.

Table 4-11—Bit-wise binary and operator

&	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

Table 4-12—Bit-wise binary or operator

	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

Table 4-13—Bit-wise binary exclusive or operator

^	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

Table 4-14—Bit-wise binary exclusive nor operator

[^] _~ _~ [^]	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

Table 4-15—Bit-wise unary negation operator

~	
0	1
1	0
x	x
z	x

When the operands are of unequal bit length, the shorter operand is zero-filled in the most significant bit positions.

4.1.11 Reduction operators

The *unary reduction operators* shall perform a bit-wise operation on a single operand to produce a single bit result. For *reduction and*, *reduction or*, and *reduction xor* operators, the first step of the operation shall apply the operator between the first bit of the operand and the second using logic Tables 4-16 through 4-18. The second and subsequent steps shall apply the operator between the 1-bit result of the prior step and the next bit of the operand using the same logic table. For *reduction nand*, *reduction nor*, and *reduction xnor* operators, the result shall be computed by inverting the result of the reduction and, reduction or, and reduction xor operation respectively.

Table 4-16—Reduction unary and operator

&	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

Table 4-17—Reduction unary or operator

	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

Table 4-18—Reduction unary exclusive or operator

^	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

Example:

Table 4-19 shows the results of applying reduction operators on different operands.

Table 4-19—Results of unary reduction operations

Operand	&	~&		~	^	~^	Comments
4'b0000	0	1	0	1	0	1	No bits set
4'b1111	1	0	1	0	0	1	All bits set
4'b0110	0	1	1	0	0	1	Even number of bits set
4'b1000	0	1	1	0	1	0	Odd number of bits set

4.1.12 Shift operators

There are two types of *shift operators*, the logical shift operators, << and >>, and the arithmetic shift operators, <<< and >>>. The left shift operators, << and <<<, shall shift their left operand to the left by the number of bit positions given by the right operand. In both cases, the vacated bit positions shall be filled with zeroes. The right shift operators, >> and >>>, shall shift their left operand to the right by the number of bit positions given by the right operand. The logical right shift shall fill the vacated bit positions with zeroes. The arithmetic right shift shall fill the vacated bit positions with zeroes if the result type is unsigned. It shall fill the vacated bit positions with the value of the most-significant (i.e., *sign*) bit of the left operand if the result type is signed. If the right operand has an unknown or high impedance value, then the result shall be unknown. The right operand is always treated as an unsigned number and has no effect on the signedness of the result. The result signedness is determined by the left-hand operand and the remainder of the expression, as outlined in section 4.5.1.

Examples:

Example 1—In this example, the reg `result` is assigned the binary value 0100, which is 0001 shifted to the left two positions and zero-filled.

```

module shift;
reg [3:0] start, result;
initial begin
    start = 1;
    result = (start << 2);
end
endmodule

```

Example 2—In this example, the reg `result` is assigned the binary value 1110, which is 1000 shifted to the right two positions and sign-filled.

```

module ashift;
reg [3:0] start, result;
initial begin
    start = 4'b1000;
    result = (start >>> 2);
end
endmodule

```

4.1.13 Conditional operator

The *conditional operator*, also known as *ternary operator*, shall be right associative and shall be constructed using three operands separated by two operators in the format given in Syntax 4-1.

```
conditional_expression ::= (From Annex A - A.8.3)
    expression1 ? { attribute_instance } expression2 : expression3
expression1 ::=
    expression
expression2 ::=
    expression
expression3 ::=
    expression
```

Syntax 4-1—Syntax for conditional operator

The evaluation of a conditional operator shall begin with the evaluation of expression1. If expression1 evaluates to false (0), then expression3 shall be evaluated and used as the result of the conditional expression. If expression1 evaluates to true (known value other than 0), then expression2 is evaluated and used as the result. If expression1 evaluates to ambiguous value (x or z), then both expression2 and expression3 shall be evaluated and their results shall be combined, bit by bit, using Table 4-20 to calculate the final result unless expression2 or expression3 is real, in which case the result shall be 0. If the lengths of expression2 and expression3 are different, the shorter operand shall be lengthened to match the longer and zero-filled from the left (the high-order end).

Table 4-20—Ambiguous condition results for conditional operator

?:	0	1	x	z
0	0	x	x	x
1	x	1	x	x
x	x	x	x	x
z	x	x	x	x

Example:

The following example of a tri-state output bus illustrates a common use of the conditional operator.

```
wire [15:0] busa = drive_busa ? data : 16'bz;
```

The bus called data is driven onto busa when drive_busa is 1. If drive_busa is unknown, then an unknown value is driven onto busa. Otherwise, busa is not driven.

4.1.14 Concatenations

A concatenation is the joining together of bits resulting from two or more expressions. The concatenation shall be expressed using the brace characters { and }, with commas separating the expressions within.

Unsize constant numbers shall not be allowed in concatenations. This is because the size of each operand in the concatenation is needed to calculate the complete size of the concatenation.

Examples:

This example concatenates four expressions:

```
{a, b[3:0], w, 3'b101}
```

and it is equivalent to the following example:

```
{a, b[3], b[2], b[1], b[0], w, 1'b1, 1'b0, 1'b1}
```

Another form of concatenation is the replication operation. The first expression shall be a non-zero, non-X and non-Z constant expression, the second expression follows the rules for concatenations. This example replicates "w" 4 times.

```
{4{w}} // This is equivalent to {w, w, w, w}
a[31:0] = {1'b1, {0{1'b0}}} ; //illegal. RHS becomes {1'b1, ;
a[31:0] = {1'b1, {1'bz{1'b0}}} ; //illegal. RHS becomes {1'b1, ;
a[31:0] = {1'b1, {1'bx{1'b0}}} ; //illegal. RHS becomes {1'b1, ;
```

If the replication operator is used on a function call operand, the function need not be evaluated multiple times. For example:

```
result = {4{func(w)}}
```

may be computed as

```
result = {func(w), func(w), func(w), func(w)}
```

or

```
y = func(w) ;
result = {y, y, y, y}
```

This is another form of expression evaluation short-circuiting.

The next example illustrates nested concatenations:

```
{b, {3{a, b}}} // This is equivalent to {b, a, b, a, b, a, b}
```

4.1.15 Event or

The event **or** operator shall perform an or of events. The **,** operator does the same thing. See 9.7 for events and triggering of events.

Example:

The following example shows both ways to make an assignment to rega when an event (change) occurs on trig or enable.

```
@(trig or enable) rega = regb ;
@(trig , enable) rega = regb ;
```

4.2 Operands

There are several types of operands that can be specified in expressions. The simplest type is a reference to a net or variable in its complete form—that is, just the name of the net or variable is given. In this case, all of the bits making

up the net or variable value shall be used as the operand.

If a single bit of a vector net, reg variable, integer variable, or time variable is required, then a bit-select operand shall be used. A part-select operand shall be used to reference a group of adjacent bits in a vector net, vector reg, integer variable, or time variable.

A memory word can be referenced as an operand. A concatenation of other operands (including nested concatenations) can be specified as an operand. A function call is an operand.

4.2.1 Vector bit-select and part-select addressing

Bit-selects extract a particular bit from a vector net, vector reg, integer variable, or time variable. The bit can be addressed using an expression. If the bit-select is out of the address bounds or the bit-select is *x* or *z*, then the value returned by the reference shall be *x*. The bit-select or part-select of a variable declared as **real** or **realtime** shall be considered illegal.

Several contiguous bits in a vector net, vector reg, integer variable, or time variable can be addressed and are known as *part-selects*. There are two types of part-selects, a constant part-select and an indexed part-select. A constant part-select of a vector reg or net is given with the following syntax.:

```
vect[msb_expr:lsb_expr]
```

Both expressions shall be constant expressions. The first expression has to address a more significant bit than the second expression. If the part-select is out of the address bounds or the part-select is *x* or *z*, then the value returned by the reference shall be *x*.

An indexed part select of a vector net, vector reg, integer variable, or time variable is given with the following syntax:

```
reg [15:0] big_vect;
reg [0:15] little_vect;

big_vect[lsb_base_expr +: width_expr]
little_vect[msb_base_expr +: width_expr]

big_vect[msb_base_expr -: width_expr]
little_vect[lsb_base_expr -: width_expr]
```

The *width_expr* shall be a constant expression. It also shall not be affected by run-time parameter assignments. The *lsb_base_expr* and *msb_base_expr* can vary at run-time. The first two examples select bits starting at the base and ascending the bit range. The number of bits selected is equal to the width expression. The second two examples select bits starting at the base and descending the bit range. Part-selects that address a range of bits that are completely out of the address bounds of the net, reg, integer, or time, or when the part-select is *x* or *z*, shall yield the value *x* when read, and shall have no effect on the data stored when written. Part-selects that are partially out of range shall when read return *x* for the bits that are out of range, and when written shall only affect the bits that are in range.

Examples:

```
reg [31:0] big_vect;
reg [0:31] little_vect;
reg [63:0] dword;
integer sel;
```

The first four *if* statements show the identity between the two part select constructs. The last one shows an indexable nature.


```

initial begin
  if ( big_vect[0 +:8] == big_vect[7 : 0]) begin end
  if (little_vect[0 +:8] == little_vect[0 : 7]) begin end
  if ( big_vect[15 -:8] == big_vect[15 : 8]) begin end
  if (little_vect[15 -:8] == little_vect[8 :15]) begin end
  if (sel >0 && sel < 8)
    dword[8*sel +:8] = big_vect[7:0]; // Replace the byte selected.

```

Examples:

Example 1—The following example specifies the single bit of `acc` vector that is addressed by the operand index.

```
acc[index]
```

The actual bit that is accessed by an address is, in part, determined by the declaration of `acc`. For instance, each of the declarations of `acc` shown in the next example causes a particular value of `index` to access a *different* bit:

```

reg [15:0] acc;
reg [2:17] acc

```

Example 2—The next example and the bullet items that follow it illustrate the principles of bit addressing. The code declares an 8-bit `reg` called `vect` and initializes it to a value of 4. The list describes how the separate bits of that vector can be addressed.

```

reg [7:0] vect;
vect = 4; // fills vect with the pattern 00000100
          // msb is bit 7, lsb is bit 0

```

- If the value of `addr` is 2, then `vect[addr]` returns 1.
- If the value of `addr` is out of bounds, then `vect[addr]` returns `x`.
- If `addr` is 0, 1, or 3 through 7, `vect[addr]` returns 0.
- `vect[3:0]` returns the bits 0100.
- `vect[5:1]` returns the bits 00010.
- `vect[expression that returns x]` returns `x`.
- `vect[expression that returns z]` returns `x`.
- If any bit of `addr` is `x` or `z`, then the value of `addr` is `x`.

NOTES

1—Part-select indices that evaluate to `x` or `z` may be flagged as a compile time error.

2—Bit-select or part-select indices that are outside of the declared range may be flagged as a compile time error.

4.2.2 Array and memory addressing

Declaration of arrays and memories (one dimensional arrays of `reg`) are discussed in 3.10. This subclause discusses array addressing.

Examples:

The next example declares a memory of 1024 8-bit words:

```
reg [7:0] mem_name[0:1023];
```

The syntax for a memory address shall consist of the name of the memory and an expression for the address, specified with the following format:

```
mem_name[addr_expr]
```

The `addr_expr` can be any expression; therefore, memory indirections can be specified in a single expression. The next example illustrates memory indirection:

```
mem_name[mem_name[3]]
```

In this example, `mem_name[3]` addresses word three of the memory called `mem_name`. The value at word three is the index into `mem_name` that is used by the memory address `mem_name[mem_name[3]]`. As with bit-selects, the address bounds given in the declaration of the memory determine the effect of the address expression. If the index is out of the address bounds or if any bit in the address is `x` or `z`, then the value of the reference shall be `x`.

Examples:

The next example declares an array of 256 by 256 8-bit elements and an array 256 by 256 by 8 1-bit elements:

```
reg [7:0] twod_array[0:255][0:255];
wire threed_array[0:255][0:255][0:7];
```

The syntax for access to the array shall consist of the name of the memory or array and an expression for each addressed dimension:

```
twod_array[addr_expr][addr_expr]
threed_array[addr_expr][addr_expr][addr_expr]
```

As before, the `addr_expr` can be any expression. The array `twod_array` accesses a whole 8-bit vector, while the array `threed_array` accesses a single bit of the three dimensional array.

To express bit selects or part selects of array elements, the desired word shall first be selected by supplying an address for each dimension. Once selected, bit and part selects shall be addressed in the same manner as net and reg bit and part selects (see 4.2.1).

Examples:

```
twod_array[14][1][3:0]      // access lower 4 bits of word
twod_array[1][3][6]         // access bit 6 of word
twod_array[1][3][sel]       // use variable bit select
threed_array[14][1][3:0]    // Illegal
```

4.2.3 Strings

String operands shall be treated as constant numbers consisting of a sequence of 8-bit ASCII codes, one per character. Any Verilog HDL operator can manipulate string operands. The operator shall behave as though the entire string were a single numeric value.

When a variable is larger than required to hold the value being assigned, the contents after the assignment shall be padded on the left with zeros. This is consistent with the padding that occurs during assignment of nonstring values.

Example:

The following example declares a string variable large enough to hold 14 characters and assigns a value to it. The example then manipulates the string using the concatenation operator.

```

module string_test;
reg [8*14:1] stringvar;

initial begin
    stringvar = "Hello world";
    $display("%s is stored as %h", stringvar, stringvar);
    stringvar = {stringvar, "!!!"};
    $display("%s is stored as %h", stringvar, stringvar);
end
endmodule

```

The result of simulating the above description is

```

    Hello world is stored as 00000048656c6c6f20776f726c64
    Hello world!!! is stored as 48656c6c6f20776f726c64212121

```

4.2.3.1 String operations

The common string operations *copy*, *concatenate*, and *compare* are supported by Verilog HDL operators. Copy is provided by simple assignment. Concatenation is provided by the concatenation operator. Comparison is provided by the equality operators.

When manipulating string values in vector regs, the regs should be at least 8*n bits (where n is the number of ASCII characters) in order to preserve the 8-bit ASCII code.

4.2.3.2 String value padding and potential problems

When strings are assigned to variables, the values stored shall be padded on the left with zeros. Padding can affect the results of comparison and concatenation operations. The comparison and concatenation operators shall not distinguish between zeros resulting from padding and the original string characters (`\0`, ASCII NULL).

Examples:

The following example illustrates the potential problem.

```

reg [8*10:1] s1, s2;
initial begin
    s1 = "Hello";
    s2 = " world!";
    if ({s1,s2} == "Hello world!")
        $display("strings are equal");
end

```

The comparison in this example fails because during the assignment the string variables are padded as illustrated in the next example:

```

s1 = 000000000048656c6c6f
s2 = 00000020776f726c6421

```

The concatenation of `s1` and `s2` includes the zero padding, resulting in the following value:

```
000000000048656c6c6f00000020776f726c6421
```

Since the string “Hello world!” contains no zero padding, the comparison fails, as shown in the following example:

```

      s1      s2
000000000048656c6c6f00000020776f726c6421
48656c6c6f20776f726c6421
"Hello"  " world!"

```

This comparison yields a result of zero, which is equivalent to false.

4.2.3.3 Null string handling

The null string (“”) shall be considered equivalent to the ASCII NULL (“\0”) which has a value zero (0), which is different from a string “0”.

4.3 Minimum, typical, and maximum delay expressions

Verilog HDL delay expressions can be specified as three expressions separated by colons and enclosed by parentheses. This is intended to represent minimum, typical, and maximum values—in that order. The syntax is given in Syntax 4-2.

```

constant_expression ::= (From Annex A - A.8.3)
    constant_primary
    | unary_operator { attribute_instance } constant_primary
    | constant_expression binary_operator { attribute_instance } constant_expression
    | constant_expression ? { attribute_instance } constant_expression
    | string
constant_mintypmax_expression ::=
    constant_expression
    | constant_expression : constant_expression : constant_expression
expression ::=
    primary
    | unary_operator { attribute_instance } primary
    | expression binary_operator { attribute_instance } expression
    | conditional_expression
    | string
mintypmax_expression ::=
    expression
    | expression : expression : expression
constant_primary ::= (From Annex A - A.8.4)
    constant_concatenation
    | constant_function_call
    | ( constant_mintypmax_expression )
    | constant_multiple_concatenation
    | genvar_identifier
    | number
    | parameter_identifier
    | specparam_identifier
primary ::=
    number
    | hierarchical_identifier
    | hierarchical_identifier [ expression ] { [ expression ] }
    | hierarchical_identifier [ expression ] { [ expression ] } [ range_expression ]
    | hierarchical_identifier [ range_expression ]
    | concatenation
    | multiple_concatenation
    | function_call
    | system_function_call
    | constant_function_call
    | ( mintypmax_expression )

```

Syntax 4-2—Syntax for mintypmax expression

Verilog HDL models typically specify three values for delay expressions. The three values allow a design to be tested with minimum, typical, or maximum delay values.

Values expressed in min:typ:max format can be used in expressions. The min:typ:max format can be used wherever expressions can appear.

Examples:

Example 1—This example shows an expression that defines a single triplet of delay values. The minimum value is the sum of a+d; the typical value is b+e; the maximum value is c+f, as follows:

```
(a:b:c) + (d:e:f)
```

Example 2—The next example shows some typical expressions that are used to specify `min:typ:max` format values:

```
val = (32'd 50: 32'd 75: 32'd 100)
```

4.4 Expression bit lengths

Controlling the number of bits that are used in expression evaluations is important if consistent results are to be achieved. Some situations have a simple solution; for example, if a bit-wise and operation is specified on two 16-bit regs, then the result is a 16-bit value. However, in some situations it is not obvious how many bits are used to evaluate an expression, or what size the result should be.

For example, should an arithmetic add of two 16-bit values perform the evaluation using 16 bits, or should the evaluation use 17 bits in order to allow for a possible carry overflow? The answer depends on the type of device being modeled, and whether that device handles carry overflow. The Verilog HDL uses the bit length of the operands to determine how many bits to use while evaluating an expression. The bit length rules are given in 4.4.1. In the case of the addition operator, the bit length of the largest operand, including the left-hand side of an assignment, shall be used.

Examples:

```
reg [15:0] a, b; // 16-bit regs
reg [15:0] sumA; // 16-bit reg
reg [16:0] sumB; // 17-bit reg

sumA = a + b; // expression evaluates using 16 bits
sumB = a + b; // expression evaluates using 17 bits
```

4.4.1 Rules for expression bit lengths

The rules governing the expression bit lengths have been formulated so that most practical situations have a natural solution.

The number of bits of an expression (known as the size of the expression) shall be determined by the operands involved in the expression and the context in which the expression is given.

A *self-determined expression* is one where the bit length of the expression is solely determined by the expression itself—for example, an expression representing a delay value.

A *context-determined expression* is one where the bit length of the expression is determined by the bit length of the expression *and* by the fact that it is part of another expression. For example, the bit size of the right-hand side expression of an assignment depends on itself and the size of the left-hand side.

Table 4-21 shows how the form of an expression shall determine the bit lengths of the results of the expression. In Table 4-21, *i*, *j*, and *k* represent expressions of an operand, and $L(i)$ represents the bit length of the operand represented by *i*.

Table 4-21—Bit lengths resulting from self-determined expressions

Expression	Bit length	Comments
Unsigned constant number ¹	Same as integer	
Sized constant number	As given	
i op j, where op is: + - * / % & ^ ^~ ~^	max(L(i),L(j))	
op i, where op is: + - ~	L(i)	
i op j, where op is: === != == != && > >= < <=	1 bit	Operands are sized to max(L(i),L(j))
op i, where op is: & ~& ~ ^ ^~ ^~ !	1 bit	All operands are self-determined
i op j, where op is: >> << **	L(i)	j is self-determined
i ? j : k	max(L(j),L(k))	i is self-determined
{i,...,j}	L(i)+...+L(j)	All operands are self-determined
{i{j,...,k}}	i * (L(j)+...+L(k))	All operands are self-determined

¹If an unsigned constant is part of an expression that is longer than 32 bits, then if the most significant bit is unknown (X or x) or tri-state (Z or z) the most significant bit is extended up to the size of the expression, otherwise signed constants are sign extended and unsigned constants are zero extended.

NOTE—Multiplication without losing any overflow bits is still possible simply by assigning the result to something wide enough to hold it.

4.4.2 An example of an expression bit-length problem

During the evaluation of an expression, interim results shall take the size of the largest operand (in case of an assignment, this also includes the left-hand side). Care has to be taken to prevent loss of a significant bit during expression evaluation. The example below describes how the bit lengths of the operands could result in the loss of a significant bit.

Given the following declarations

```
reg [15:0] a, b, answer; // 16-bit regs
```

The intent is to evaluate the expression

```
answer = (a + b) >> 1; //will not work properly
```

where a and b are to be added, which may result in an overflow, and then shifted right by 1 bit to preserve the carry bit in the 16-bit answer.

A problem arises, however, because all operands in the expression are of a 16-bit width. Therefore, the expression (a + b) produces an interim result that is only 16 bits wide, thus losing the carry bit before the evaluation performs the 1-bit right shift operation.

The solution is to force the expression (a + b) to evaluate using at least 17 bits. For example, adding an integer

value of 0 to the expression will cause the evaluation to be performed using the bit size of integers. The following example will produce the intended result:

```
answer = (a + b + 0) >> 1; //will work correctly
```

In the following example:

```
module bitlength();
  reg [3:0] a,b,c;
  reg [4:0] d;

  initial begin
    a = 9;
    b = 8;
    c = 1;
    $display("answer = %b", c ? (a&b) : d);
  end
endmodule
```

the \$display statement will display:

```
answer = 01000
```

By itself, the expression `a&b` would have the bit length 4, but since it is in the context of the conditional expression, which uses the maximum bit-length, the expression `a&b` actually has length 5, the length of `d`.

4.4.3 Example of self-determined expressions

```
reg [3:0] a;
reg [5:0] b;
reg [15:0] c;

initial begin
  a = 4'hF;
  b = 6'ha;
  $display("a*b=%x",
    a*b);           // expression size is self determined
  c = {a**b};       // expression a**b is self determined
  $display("a**b=%x", c); // due to {}
  c = a**b;         // expression size is determined by c
  $display("c=%x", c);
end
```

Simulator output for this example:

```
a*b=16           // 96 was truncated since expression size is 6
a**b=1           // expression size is 4 bits (size of a)
c=21             // expression size is 6 bits (size of c)
```

4.5 Signed expressions

Controlling the sign of an expression is important if consistent results are to be achieved. In addition to the rules outlined in the following sections, two system functions shall be used to handle type casting on expressions: `$signed()` and `$unsigned()`. These functions shall evaluate the input expression and return a value with the same size and value of the input expression and the type defined by the function:

\$signed - returned value is signed

\$unsigned - returned value is unsigned

Example:

```
reg [7:0] regA;
reg signed [7:0] regS;

regA = $unsigned(-4); // regA = 4'b1100
regS = $signed(4'b1100); // regS = -4
```

4.5.1 Rules for expression types

The following are the rules for determining the resulting type of an expression:

- Expression type depends only on the operands. It does not depend on the LHS (if any).
- Decimal numbers are signed.
- Based_numbers are unsigned, except where the *s* notation is used in the base specifier (as in "4'sd12").
- Bit-select results are unsigned, regardless of the operands.
- Part-select results are unsigned, regardless of the operands.

NOTE—This is true even if the part-select specifies the entire vector.

```
reg [15:0] a;
reg signed [7:0] b;

initial
    a = b[7:0]; // b[7:0] is unsigned and therefore zero-extended
```

- Concatenate results are unsigned, regardless of the operands.
- Comparison results (1, 0) are unsigned, regardless of the operands.
- Reals converted to integers by type coercion are signed
- The sign and size of any self-determined operand is determined by the operand itself and independent of the remainder of the expression.
- For non-self-determined operands the following rules apply:
 - if any operand is real, the result is real;
 - if any operand is unsigned, the result is unsigned, regardless of the operator;
 - if all operands are signed, the result will be signed, regardless of operator, except as noted.

4.5.2 Steps for evaluating an expression

- Determine the expression size based upon the standard rules of expression size determination.
- Determine the sign of the expression using the rules outlined in 4.5.1.
- Coerce the type of each operand of the expression (excepting those which are self-determined) to the type of the expression.
- Extend the size of each operand (excepting those which are self-determined) to the size of the expression. Perform sign extension if and only if the operand type (after type coercion) is signed.

4.5.3 Steps for evaluating an assignment

- Determine the size of the RHS by the standard assignment size determination rules (see 4.4)
- If needed, extend the size of the RHS, performing sign extension if and only if the type of the RHS is signed.

4.5.4 Handling X and Z in signed expressions

If a signed operand is to be resized to a larger signed width and the value of the sign bit is X, the resulting value shall be bit-filled with Xs. If the sign bit of the value is Z, then the resulting value shall be bit-filled with Zs. If any bit of a signed value is X or Z, then any non logical operation involving the value shall result in the entire resultant value being an X and the type consistent with the expression's type.

Section 5

Scheduling semantics

5.1 Execution of a model

The balance of the sections of this standard describe the behavior of each of the elements of the language. This section gives an overview of the interactions between these elements, especially with respect to the scheduling and execution of events.

The elements that make up the Verilog HDL can be used to describe the behavior, at varying levels of abstraction, of electronic hardware. An HDL has to be a parallel programming language. The execution of certain language constructs is defined by parallel execution of blocks or processes. It is important to understand what execution order is guaranteed to the user, and what execution order is indeterminate.

Although the Verilog HDL is used for more than simulation, the semantics of the language are defined for simulation, and everything else is abstracted from this base definition.

5.2 Event simulation

The Verilog HDL is defined in terms of a discrete event execution model. The discrete event simulation is described in more detail in this section to provide a context to describe the meaning and valid interpretation of Verilog HDL constructs. These resulting definitions provide the standard Verilog reference model for simulation, which all compliant simulators shall implement. Note, though, that there is a great deal of choice in the definitions that follow, and differences in some details of execution are to be expected between different simulators. In addition, Verilog HDL simulators are free to use different algorithms than those described in this section, provided the user-visible effect is consistent with the reference model.

A design consists of connected threads of execution or processes. Processes are objects that can be evaluated, that may have state, and that can respond to changes on their inputs to produce outputs. Processes include primitives, modules, initial and always procedural blocks, continuous assignments, asynchronous tasks, and procedural assignment statements.

Every change in value of a net or variable in the circuit being simulated, as well as the named event, is considered an *update event*.

Processes are sensitive to update events. When an update event is executed, all the processes that are sensitive to that event are evaluated in an arbitrary order. The evaluation of a process is also an event, known as an *evaluation event*.

In addition to events, another key aspect of a simulator is time. The term *simulation time* is used to refer to the time value maintained by the simulator to model the actual time it would take for the circuit being simulated. The term *time* is used interchangeably with simulation time in this section.

Events can occur at different times. In order to keep track of the events and to make sure they are processed in the correct order, the events are kept on an *event queue*, ordered by simulation time. Putting an event on the queue is called *scheduling an event*.

5.3 The stratified event queue

The Verilog event queue is logically segmented into five different regions. Events are added to any of the five regions but are only removed from the *active* region.

- 1) Events that occur at the current simulation time and can be processed in any order. These are the *active* events.
- 2) Events that occur at the current simulation time, but that shall be processed after all the active events are processed. These are the *inactive* events.
- 3) Events that have been evaluated during some previous simulation time, but that shall be assigned at this simulation time after all the active and inactive events are processed. These are the *nonblocking assign update* events.
- 4) Events that shall be processed after all the active, inactive, and nonblocking assign update events are processed. These are the *monitor* events.
- 5) Events that occur at some future simulation time. These are the *future* events. Future events are divided into *future inactive events*, and *future nonblocking assignment update events*.

The processing of all the active events is called a *simulation cycle*.

The freedom to choose any active event for immediate processing is an essential source of nondeterminism in the Verilog HDL.

An *explicit zero delay* (#0) requires that the process be suspended and added as an inactive event for the current time so that the process is resumed in the next simulation cycle in the current time.

A nonblocking assignment (see 9.2.2) creates a nonblocking assign update event, scheduled for current or a later simulation time.

The **\$monitor** and **\$strobe** system tasks (see 17.1) create monitor events for their arguments. These events are continuously re-enabled in every successive time step. The monitor events are unique in that they cannot create any other events.

The call back procedures scheduled with PLI routines such as `tf_synchronize()` (see 25.58) or `vpi_register_cb(cb_readwrite)` (see 27.33) shall be treated as inactive events.

5.4 The Verilog simulation reference model

In all the examples that follow, T refers to the current simulation time, and all events are held in the event queue, ordered by simulation time.

```
while (there are events) {
    if (no active events) {
        if (there are inactive events) {
            activate all inactive events;
        } else if (there are nonblocking assign update events) {
            activate all nonblocking assign update events;
        } else if (there are monitor events) {
            activate all monitor events;
        } else {
            advance T to the next event time;
            activate all inactive events for time T;
        }
    }
    E = any active event;
    if (E is an update event) {
        update the modified object;
        add evaluation events for sensitive processes to event queue;
    }
}
```

```

    } else { /* shall be an evaluation event */
        evaluate the process;
        add update events to the event queue;
    }
}

```

5.4.1 Determinism

This standard guarantees a certain scheduling order.

- 1) Statements within a `begin-end` block shall be executed in the order in which they appear in that `begin-end` block. Execution of statements in a particular `begin-end` block can be suspended in favor of other processes in the model; however, in no case shall the statements in a `begin-end` block be executed in any order other than that in which they appear in the source.
- 2) Nonblocking assignments shall be performed in the order the statements were executed. Consider the following example:

```

initial begin
    a <= 0;
    a <= 1;
end

```

When this block is executed, there will be two events added to the nonblocking assign update queue. The previous rule requires that they be entered on the queue in source order; this rule requires that they be taken from the queue and performed in source order as well. Hence, at the end of time step 1, the variable `a` will be assigned 0 and then 1.

5.4.2 Nondeterminism

One source of nondeterminism is the fact that active events can be taken off the queue and processed in any order. Another source of nondeterminism is that statements without time-control constructs in behavioral blocks do not have to be executed as one event. Time control statements are the `#` expression and `@` expression constructs (see 9.7). At any time while evaluating a behavioral statement, the simulator may suspend execution and place the partially completed event as a pending active event on the event queue. The effect of this is to allow the interleaving of process execution. Note that the order of interleaved execution is nondeterministic and not under control of the user.

5.5 Race conditions

Because the execution of expression evaluation and net update events may be intermingled, race conditions are possible:

```

assign p = q;
initial begin
    q = 1;
    #1 q = 0;
    $display(p);
end

```

The simulator is correct in displaying either a 1 or a 0. The assignment of 0 to `q` enables an update event for `p`. The simulator may either continue and execute the `$display` task or execute the update for `p`, followed by the `$display` task.

5.6 Scheduling implication of assignments

Assignments are translated into processes and events as follows.

5.6.1 Continuous assignment

A continuous assignment statement (Section 6) corresponds to a process, sensitive to the source elements in the expression. When the value of the expression changes, it causes an active update event to be added to the event queue, using current values to determine the target.

5.6.2 Procedural continuous assignment

A procedural continuous assignment (which are the **assign** or **force** statement; see 9.3) corresponds to a process that is sensitive to the source elements in the expression. When the value of the expression changes, it causes an active update event to be added to the event queue, using current values to determine the target.

A **deassign** or a **release** statement deactivates any corresponding **assign** or **force** statement(s).

5.6.3 Blocking assignment

A blocking assignment statement with a delay computes the right-hand side value using the current values, then causes the executing process to be suspended and scheduled as a future event. If the delay is 0, the process is scheduled as an inactive event for the current time.

When the process is returned (or if it returns immediately if no delay is specified), the process performs the assignment to the left-hand side and enables any events based upon the update of the left-hand side. The values at the time the process resumes are used to determine the target(s). Execution may then continue with the next sequential statement or with other active events.

5.6.4 Nonblocking assignment

A nonblocking assignment statement always computes the updated value and schedules the update as a nonblocking assign update event, either in this time step if the delay is zero or as a future event if the delay is nonzero. The values in effect when the update is placed on the event queue are used to compute both the right-hand value and the left-hand target.

5.6.5 Switch (transistor) processing

The event-driven simulation algorithm described in 5.4 depends on unidirectional signal flow and can process each event independently. The inputs are read, the result is computed, and the update is scheduled.

The Verilog HDL provides switch-level modeling in addition to behavioral and gate-level modeling. Switches provide bi-directional signal flow and require coordinated processing of nodes connected by switches.

The Verilog HDL source elements that model switches are various forms of transistors, called **tran**, **tranif0**, **tranif1**, **rtran**, **rtranif0**, and **rtranif1**.

Switch processing shall consider all the devices in a bidirectional switch-connected net before it can determine the appropriate value for any node on the net, because the inputs and outputs interact. A simulator can do this using a relaxation technique. The simulator can process tran at any time. It can process a subset of tran-connected events at a particular time, intermingled with the execution of other active events.

Further refinement is required when some transistors have gate value x. A conceptually simple technique is to solve the network repeatedly with these transistors set to all possible combinations of fully conducting and nonconducting transistors. Any node that has a unique logic level in all cases has steady-state response equal to this level. All other nodes have steady-state response x.

5.6.6 Port connections

Ports connect processes through implicit continuous assignment statements or implicit bidirectional connections. Bidirectional connections are analogous to an always-enabled tran connection between the two nets, but without any strength reduction. Port connection rules require that a value receiver be a net or a structural net expression.

Ports can always be represented as declared objects connected as follows:

- If an input port, then a continuous assignment from an outside expression to a local (input) net
- If an output port, then a continuous assignment from a local output expression to an outside net
- If an inout, then a nonstrength-reducing transistor connecting the local net to an outside net

5.6.7 Functions and tasks

Task and function parameter passing is by value, and it copies in on invocation and copies out on return. The copy out on the return function behaves in the same manner as does any blocking assignment.

Section 6

Assignments

The assignment is the basic mechanism for placing values into nets and variables. There are two basic forms of assignments:

- The *continuous assignment*, which assigns values to *nets*
- The *procedural assignment*, which assigns values to *variables*

There are two additional forms of assignments, assign / deassign and force / release which are called *procedural continuous assignments*, described in 9.3.

An assignment consists of two parts, a left-hand side and a right-hand side, separated by the equals (=) character; or, in the case of nonblocking procedural assignment, the less-than-equals (<=) character pair. The right-hand side can be any expression that evaluates to a value. The left-hand side indicates the variable to which the right-hand side value is to be assigned. The left-hand side can take one of the forms given in Table 6-1, depending on whether the assignment is a continuous assignment or a procedural assignment.

Table 6-1—Legal left-hand side forms in assignment statements

Statement type	Left-hand side (LHS)
Continuous assignment	Net (vector or scalar) Constant bit select of a vector net Constant part select of a vector net Constant indexed part select of a vector net Concatenation of any of the above four LHS
Procedural assignment	Variables (vector or scalar) Bit-select of a vector reg, integer, or time variable Constant part select of a vector reg, integer, or time variable Memory word Indexed part select of a vector reg, integer, or time variable Concatenation of regs; bit or part selects of regs

6.1 Continuous assignments

Continuous assignments shall drive values onto nets, both vector and scalar. This assignment shall occur whenever the value of the right-hand side changes. Continuous assignments provide a way to model combinational logic without specifying an interconnection of gates. Instead, the model specifies the logical expression that drives the net.

The syntax for continuous assignments is given in Syntax 6-1.

```

net_declaration ::= (From Annex A - A.2.1.3)
    net_type [ signed ]
        [ delay3 ] list_of_net_identifiers ;
| net_type [ drive_strength ] [ signed ]
    [ delay3 ] list_of_net_decl_assignments ;
| net_type [ vector | scalared ] [ signed ]
    range [ delay3 ] list_of_net_identifiers ;
| net_type [ drive_strength ] [ vector | scalared ] [ signed ]
    range [ delay3 ] list_of_net_decl_assignments ;
| trireg [ charge_strength ] [ signed ]
    [ delay3 ] list_of_net_identifiers ;
| trireg [ drive_strength ] [ signed ]
    [ delay3 ] list_of_net_decl_assignments ;
| trireg [ charge_strength ] [ vector | scalared ] [ signed ]
    range [ delay3 ] list_of_net_identifiers ;
| trireg [ drive_strength ] [ vector | scalared ] [ signed ]
    range [ delay3 ] list_of_net_decl_assignments ;
list_of_net_decl_assignments ::= (From Annex A - A.2.3)
    net_decl_assignment { , net_decl_assignment }
net_decl_assignment ::= (From Annex A - A.2.4)
    net_identifier = expression
continuous_assign ::= (From Annex A - A.6.1)
    assign [ drive_strength ] [ delay3 ] list_of_net_assignments ;
list_of_net_assignments ::=
    net_assignment { , net_assignment }
net_assignment ::=
    net_lvalue = expression

```

Syntax 6-1—Syntax for continuous assignment

6.1.1 The net declaration assignment

The first two alternatives in the net declaration are discussed in see 3.2. The third alternative, the net declaration assignment, allows a continuous assignment to be placed on a net in the same statement that declares the net.

Example:

The following is an example of the net declaration form of a continuous assignment:

```
wire (strong1, pull0) mynet = enable ;
```

NOTE—Because a net can be declared only once, only one net declaration assignment can be made for a particular net. This contrasts with the continuous assignment statement; one net can receive multiple assignments of the continuous assignment form.

6.1.2 The continuous assignment statement

The continuous assignment statement shall place a continuous assignment on a net data type. The net may be explicitly declared, or may inherit an implicit declaration in accordance with the implicit declarations rules defined in 3.5.

Assignments on nets shall be continuous and automatic. This means that whenever an operand in the right-hand side expression changes value, the whole right-hand side shall be evaluated and if the new value is different from the previous value, then the new value shall be assigned to the left-hand side.

Examples:

Example 1—The following is an example of a continuous assignment to a net that has been previously declared:

```
wire mynet ;
assign (strong1, pull0) mynet = enable ;
```

Example 2—The following is an example of the use of a continuous assignment to model a 4-bit adder with carry. The assignment could not be specified directly in the declaration of the nets because it requires a concatenation on the left-hand side.

```
module adder (sum_out, carry_out, carry_in, ina, inb);
output [3:0] sum_out;
output carry_out;
input [3:0] ina, inb;
input carry_in;
wire carry_out, carry_in;
wire [3:0] sum_out, ina, inb;
assign {carry_out, sum_out} = ina + inb + carry_in;
endmodule
```

Example 3—The following example describes a module with one 16-bit output bus. It selects between one of four input busses and connects the selected bus to the output bus.

```
module select_bus(busout, bus0, bus1, bus2, bus3, enable, s);
parameter n = 16;
parameter Zee = 16'bz;
output [1:n] busout;
input [1:n] bus0, bus1, bus2, bus3;
input enable;
input [1:2] s;
tri [1:n] data;           // net declaration
// net declaration with continuous assignment
tri [1:n] busout = enable ? data : Zee;
// assignment statement with four continuous assignments
assign
    data = (s == 0) ? bus0 : Zee,
    data = (s == 1) ? bus1 : Zee,
    data = (s == 2) ? bus2 : Zee,
    data = (s == 3) ? bus3 : Zee;
endmodule
```

The following sequence of events is experienced during simulation of this example:

- a) The value of *s*, a bus selector input variable, is checked in the assign statement. Based on the value of *s*, the net data receives the data from one of the four input busses.
- b) The setting of data net triggers the continuous assignment in the net declaration for busout. If enable is set, the contents of data are assigned to busout; if enable is 0, the contents of Zee are assigned to busout.

6.1.3 Delays

A delay given to a continuous assignment shall specify the time duration between a right-hand side operand value change and the assignment made to the left-hand side. If the left-hand side references a scalar net, then the delay shall be treated in the same way as for gate delays—that is, different delays can be given for the output rising, falling, and changing to high impedance (see Section 7).

If the left-hand side references a vector net, then up to three delays can be applied. The following rules determine which delay controls the assignment:

- If the right-hand side makes a transition from nonzero to zero, then the falling delay shall be used.
- If the right-hand side makes a transition to z, then the turn-off delay shall be used.
- For all other cases, the rising delay shall be used.

Specifying the delay in a continuous assignment that is part of the net declaration shall be treated differently from specifying a net delay and then making a continuous assignment to the net. A delay value can be applied to a net in a net declaration, as in the following example:

```
wire #10 wireA;
```

This syntax, called a *net delay*, means that any value change that is to be applied to `wireA` by some other statement shall be delayed for ten time units before it takes effect. When there is a continuous assignment in a declaration, the delay is part of the continuous assignment and is *not* a net delay. Thus, it shall not be added to the delay of other drivers on the net. Furthermore, if the assignment is to a vector net, then the rising and falling delays shall not be applied to the individual bits if the assignment is included in the declaration.

In situations where a right-hand side operand changes before a previous change has had time to propagate to the left-hand side, then the following steps are taken:

- a) The value of the right-hand side expression is evaluated.
- b) If this RHS value differs from the value currently scheduled to propagate to the left-hand side, then the currently scheduled propagation event is descheduled.
- c) If the new RHS value equals the current left-hand side value, no event is scheduled.
- d) If the new RHS value differs from the current LHS value, a delay is calculated in the standard way using the current value of the left-hand side, the newly calculated value of the right-hand side, and the delays indicated on the statement; a new propagation event is then scheduled to occur delay time units in the future.

6.1.4 Strength

The driving strength of a continuous assignment can be specified by the user. This applies only to assignments to scalar nets of the following types:

wire	tri	triereg
wand	triand	tri0
wor	trior	tri1

Continuous assignments driving strengths can be specified in either a net declaration or in a stand-alone assignment, using the **assign** keyword. The strength specification, if provided, shall immediately follow the keyword (either the keyword for the net type or **assign**) and precede any delay specified. Whenever the continuous assignment drives the net, the strength of the value shall be simulated as specified.

A drive strength specification shall contain one strength value that applies when the value being assigned to the net is 1 and a second strength value that applies when the assigned value is 0. The following keywords shall specify the strength value for an assignment of 1:

supply1 strong1 pull1 weak1 highz1

The following keywords shall specify the strength value for an assignment of 0:

supply0 strong0 pull0 weak0 highz0

The order of the two strength specifications shall be arbitrary. The following two rules shall constrain the use of drive strength specifications:

- The strength specifications (**highz1, highz0**) and (**highz0, highz1**) shall be treated as illegal constructs.
- If drive strength is not specified, it shall default to (**strong1, strong0**).

6.2 Procedural assignments

The primary discussion of procedural assignments is in 9.2. However, a description of the basic ideas in this clause highlights the differences between continuous assignments and procedural assignments.

As stated in 6.1, continuous assignments drive nets in a manner similar to the way gates drive nets. The expression on the right-hand side can be thought of as a combinatorial circuit that drives the net continuously. In contrast, procedural assignments put values in variables. The assignment does not have duration; instead, the variable holds the value of the assignment until the next procedural assignment to that variable.

Procedural assignments occur within procedures such as **always**, **initial** (see Section 9), **task**, and **function** (see Section 10) and can be thought of as “triggered” assignments. The trigger occurs when the flow of execution in the simulation reaches an assignment within a procedure. Reaching the assignment can be controlled by conditional statements. Event controls, delay controls, **if** statements, **case** statements, and looping statements can all be used to control whether assignments are evaluated. Section 9 gives details and examples.

6.2.1 Variable declaration assignment

The variable declaration assignment is a special case of procedural assignment as it assigns a value to a variable. It allows an initial value to be placed in a variable in the same statement that declares the variable. The assignment shall be to a constant expression. The assignment does not have duration; instead, the variable holds the value until the next assignment to that variable. Variable declaration assignments to an array are not allowed. Variable declaration assignments are only allowed at the module level.

Examples:

Example 1—Declare a 4 bit reg and assign it the value 4.

```
reg[3:0] a = 4'h4;
```

This is equivalent to writing:

```
reg[3:0] a;  
initial a = 4'h4;
```

Example 2—The following example is not legal.

```
reg [3:0] array [3:0] = 0;
```

Example 3—Declare two integers, the first is assigned the value of 0.

```
integer i = 0, j;
```

Example 4—Declare two real variables, assigned to the values 2.5 and 300,000.

```
real r1 = 2.5, n300k = 3E6;
```

Example 5—Declare a time variable and realtime variable with initial values.

```
time t1 = 25;  
realtime rt1 = 2.5;
```

NOTE—If the same variable is assigned different values both in an initial block and in a variable declaration assignment, the order of the evaluation is undefined.

6.2.2 Variable declaration syntax

The syntax for variable declaration assignments is given in Syntax 6-2.

```
integer_declaration ::= (From Annex A - A.2.1.3)  
    integer list_of_variable_identifiers ;  
real_declaration ::=  
    real list_of_real_identifiers ;  
realtime_declaration ::=  
    realtime list_of_real_identifiers ;  
reg_declaration ::=  
    reg [ signed ] [ range ] list_of_variable_identifiers ;  
time_declaration ::=  
    time list_of_variable_identifiers ;  
real_type ::= (From Annex A - A.2.2.1)  
    real_identifier [ = constant_expression ]  
    | real_identifier dimension { dimension }  
variable_type ::=  
    variable_identifier [ = constant_expression ]  
    | variable_identifier dimension { dimension }  
list_of_real_identifiers ::= (From Annex A - A.2.3)  
    real_type { , real_type }  
list_of_variable_identifiers ::=  
    variable_type { , variable_type }
```

Syntax 6-2—Syntax for reg declaration assignment

Section 7

Gate and switch level modeling

This section describes the syntax and semantics of these built-in primitives and how a hardware design can be described using these primitives.

There are 14 logic gates and 12 switches predefined in the Verilog HDL to provide the *gate* and *switch* level modeling facility. Modeling with logic gates and switches has the following advantages:

- Gates provide a much closer one-to-one mapping between the actual circuit and the model.
- There is no continuous assignment equivalent to the bidirectional transfer gate.

7.1 Gate and switch declaration syntax

Syntax 7-1 shows the gate and switch declaration syntax.

A gate or a switch instance declaration shall have the following specifications:

- The keyword that names the type of gate or switch primitive
- An optional *drive strength*
- An optional *propagation delay*
- An optional identifier that names each gate or switch instance
- An optional range for *array of instances*
- The terminal connection list

Multiple instances of the one type of gate or switch primitive can be declared as a comma-separated list. All such instances shall have the same drive strength and delay specification.

```

gate_instantiation ::= (From Annex A - A.3.1)
    cmos_switchtype [delay3] cmos_switch_instance { , cmos_switch_instance } ;
    | enable_gatetype [drive_strength] [delay3] enable_gate_instance { , enable_gate_instance } ;
    | mos_switchtype [delay3] mos_switch_instance { , mos_switch_instance } ;
    | n_input_gatetype [drive_strength] [delay2] n_input_gate_instance { , n_input_gate_instance } ;
    | n_output_gatetype [drive_strength] [delay2] n_output_gate_instance
      { , n_output_gate_instance } ;
    | pass_en_switchtype [delay2] pass_enable_switch_instance { , pass_enable_switch_instance } ;
    | pass_switchtype pass_switch_instance { , pass_switch_instance } ;
    | pulldown [pulldown_strength] pull_gate_instance { , pull_gate_instance } ;
    | pullup [pullup_strength] pull_gate_instance { , pull_gate_instance } ;

cmos_switch_instance ::= [ name_of_gate_instance ]
    ( output_terminal , input_terminal , ncontrol_terminal , pcontrol_terminal )

enable_gate_instance ::= [ name_of_gate_instance ]
    ( output_terminal , input_terminal , enable_terminal )

mos_switch_instance ::= [ name_of_gate_instance ]
    ( output_terminal , input_terminal , enable_terminal )

n_input_gate_instance ::= [ name_of_gate_instance ]
    ( output_terminal , input_terminal { , input_terminal } )

n_output_gate_instance ::= [ name_of_gate_instance ]
    ( output_terminal { , output_terminal } , input_terminal )

pass_switch_instance ::= [ name_of_gate_instance ] ( inout_terminal , inout_terminal )

pass_enable_switch_instance ::= [ name_of_gate_instance ]
    ( inout_terminal , inout_terminal , enable_terminal )

pull_gate_instance ::= [ name_of_gate_instance ] ( output_terminal )

name_of_gate_instance ::= gate_instance_identifier [ range ]

pulldown_strength ::= (From Annex A - A.3.2)
    ( strength0 , strength1 )
    | ( strength1 , strength0 )
    | ( strength0 )

pullup_strength ::= ( strength0 , strength1 )
    | ( strength1 , strength0 )
    | ( strength1 )

enable_terminal ::= (From Annex A - A.3.3)
    expression

inout_terminal ::= net_lvalue

input_terminal ::= expression

ncontrol_terminal ::= expression

output_terminal ::= net_lvalue

pcontrol_terminal ::= expression

cmos_switchtype ::= (From Annex A - A.3.4)
    cmos | rcmos

enable_gatetype ::= bufif0 | bufif1 | notif0 | notif1

mos_switchtype ::= nmos | pmos | rnmos | rpmos

n_input_gatetype ::= and | nand | or | nor | xor | xnor

n_output_gatetype ::= buf | not

pass_en_switchtype ::= tranif0 | tranif1 | rtranif1 | rtranif0

pass_switchtype ::= tran | rtran

```

Syntax 7-1—Syntax for gate instantiation

7.1.1 The gate type specification

A gate or switch instance declaration shall begin with the keyword that specifies the gate or switch primitive being used by the instances that follow in the declaration. Table 7-1 lists the keywords that shall begin a gate or a switch instance declaration.

Table 7-1—Built-in gates and switches

n_input gates	n_output gates	tristate gates	pull gates	MOS switches	bidirectional switches
and	buf	bufif0	pulldown	cmos	rtran
nand	not	bufif1	pullup	nmos	rtranif0
nor		notif0		pmos	rtranif1
or		notif1		rcmos	tran
xnor				rnmos	tranif0
xor				rpmos	tranif1

Explanations of the built-in gates and switches shown in Table 7-1 begin in 7.2.

7.1.2 The drive strength specification

An optional drive strength specification shall specify the *strength* of the logic values on the output terminals of the gate instance. Only the instances of the gate primitives shown in Table 7-2 can have the drive strength specification.

Table 7-2—Valid gate types for strength specifications

and	nand	buf	not	pulldown
or	nor	bufif0	notif0	pullup
xor	xnor	bufif1	notif1	

The drive strength specification for a gate instance, with the exception of **pullup** and **pulldown**, shall have a *strength1* specification and a *strength0* specification. The *strength1* specification shall specify the strength of signals with a logic value 1, and the *strength0* specification shall specify the strength of signals with a logic value 0. The strength specification shall follow the gate type keyword and precede any delay specification. The *strength0* specification can precede or follow the *strength1* specification. The *strength1* and *strength0* specifications shall be separated by a comma and enclosed within a pair of parentheses.

The **pullup** gate can have only *strength1* specification; *strength0* specification shall be optional. The **pulldown** gate can have only *strength0* specification; *strength1* specification shall be optional.

The *strength1* specification shall be one of the following keywords:

supply1 strong1 pull1 weak1

The *strength0* specification shall be one of the following keywords:

supply0 strong0 pull0 weak0

Specifying **highz1** as *strength1* shall cause the gate or switch to output a logic value z in place of a 1. Specifying **highz0** shall cause the gate to output a logic value z in place of a 0. The strength specifications (**highz0**, **highz1**) and (**highz1**, **highz0**) shall be considered invalid.

In the absence of a strength specification, the instances shall have the default strengths **strong1** and **strong0**.

Example:

The following example shows a drive strength specification in a declaration of an open collector **nor** gate:

```
nor (highz1,strong0) n1(out1,in1,in2);
```

In this example, the **nor** gate outputs a z in place of a 1.

Logic strength modeling is discussed in more detail in 7.9 through 7.13.

7.1.3 The delay specification

An optional delay specification shall specify the propagation delay through the gates and switches in a declaration. Gates and switches in declarations with no delay specification shall have no propagation delay. A delay specification can contain up to three delay values, depending on the gate type. The **pullup** and **pulldown** instance declarations shall not include delay specifications. Delays are discussed in more detail in 7.14.

7.1.4 The primitive instance identifier

An optional name can be given to a gate or switch instance. If multiple instances are declared as an array of instances, an identifier shall be used to name the instances.

7.1.5 The range specification

There are many situations when repetitive instances are required. These instances shall differ from each other only by the index of the vector to which they are connected.

In order to specify an array of instances, the instance name shall be followed by the range specification. The range shall be specified by two constant expressions, left-hand index (*lhi*) and right-hand index (*rhi*), separated by a colon and enclosed within a pair of square brackets. A [*lhi*:*rhi*] range specification shall represent an array of $\text{abs}(\text{lhi}-\text{rhi})+1$ instances. Neither of the two constant expressions are required to be zero, and *lhi* is not required to be larger than *rhi*. If both constant expressions are equal, only one instance shall be generated.

An array of instances shall have a continuous range. One instance identifier shall be associated with only one range to declare an array of instances.

The range specification shall be optional. If no range specification is given, a single instance shall be created.

Example:

A declaration shown below is illegal:

```
nand #2 t_nand[0:3] ( ... ), t_nand[4:7] ( ... );
```

It could be declared correctly as one array of eight instances, or two arrays with unique names of four elements each:

```
nand #2 t_nand[0:7]( ... );  
nand #2 x_nand[0:3] ( ... ), y_nand[4:7] ( ... );
```

7.1.6 Primitive instance connection list

The terminal list describes how the gate or switch connects to the rest of the model. The gate or switch type can limit these expressions. The connection list shall be enclosed in a pair of parentheses, and the terminals shall be separated by commas. The output or bidirectional terminals shall always come first in the terminal list, followed by the input

terminals.

The terminal connections for an array of instances shall follow these rules:

- The bit length of each port expression in the declared instance-array shall be compared with the bit length of each single-instance port or terminal in the instantiated module or primitive.
- For each port or terminal where the bit length of the instance-array port expression is the same as the bit length of the single-instance port, the instance-array port expression shall be connected to each single-instance port.
- If bit lengths are different, each instance shall get a part-select of the port expression as specified in the range, starting with the right-hand index.
- Too many or too few bits to connect to all the instances shall be considered an error.

An individual instance from an array of instances shall be referenced in the same manner as referencing an element of an array of regs.

Examples:

Example 1—The following declaration of `nand_array` declares four instances that can be referenced by `nand_array[1]`, `nand_array[2]`, `nand_array[3]`, and `nand_array[4]` respectively.

```
nand #2 nand_array[1:4]( ... ) ;
```

Example 2—The two module descriptions that follow are equivalent except for indexed instance names, and they demonstrate the range specification and connection rules for declaring an array of instances:

```
module driver (in, out, en);
input [3:0] in;
output [3:0] out;
input en;

bufif0 ar[3:0] (out, in, en); // array of tri-state buffers

endmodule

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

bufif0 ar3 (out[3], in[3], en); // each buffer declared separately
bufif0 ar2 (out[2], in[2], en);
bufif0 ar1 (out[1], in[1], en);
bufif0 ar0 (out[0], in[0], en);

endmodule
```

Example 3—The two module descriptions that follow are equivalent except for indexed instance names, and they demonstrate how different instances within an array of instances are connected when the port sizes do not match.

```

module busdriver (busin, bushigh, buslow, enh, enl);
input [15:0] in;
output [7:0] bushigh, buslow;
input enh, enl;

driver busar3 (busin[15:12], bushigh[7:4], enh);
driver busar2 (busin[11:8], bushigh[3:0], enh);
driver busar1 (busin[7:4], buslow[7:4], enl);
driver busar0 (busin[3:0], buslow[3:0], enl);

endmodule

module busdriver_equiv (busin, bushigh, buslow, enh, enl);
input [15:0] busin;
output [7:0] bushigh, buslow;
input enh, enl;

driver busar[3:0] (.out({bushigh, buslow}), .in(busin),
                  .en({enh, enh, enl, enl}));

endmodule

```

Example 4—This example demonstrates how a series of modules can be chained together. Figure 7-1 shows an equivalent schematic interconnection of DFF instances.

```

module dffn (q, d, clk);
parameter bits = 1;
input [bits-1:0] d;
output [bits-1:0] q;
input clk ;

DFF dff[bits-1:0] (q, d, clk); // create a row of D flip-flops

endmodule

module MxN_pipeline (in, out, clk);
parameter M = 3, N = 4; // M=width,N=depth
input [M-1:0] in;
output [M-1:0] out;
input clk;
wire [M*(N-1):1] t;

// #(M) redefines the bits parameter for dffn
// create p[1:N] columns of dffn rows (pipeline)

dffn #(M) p[1:N] ({out, t}, {t, in}, clk);

endmodule

```

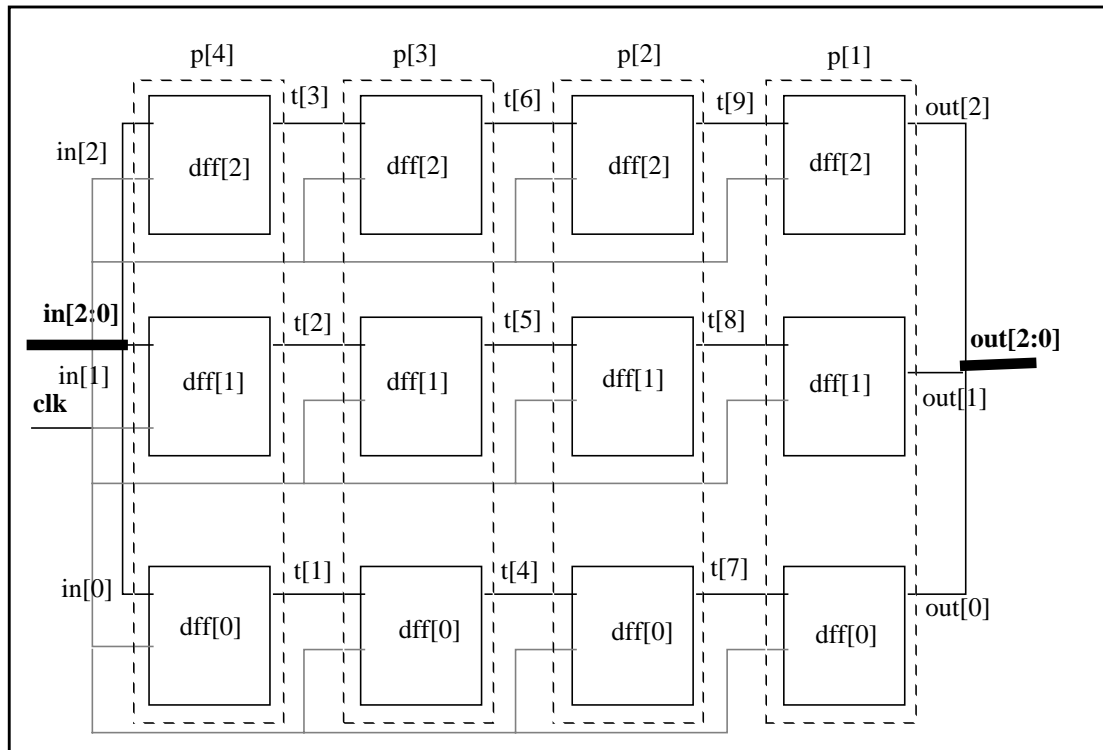


Figure 7-1—Schematic diagram of interconnections in array of instances

7.2 and, nand, nor, or, xor, and xnor gates

The instance declaration of a multiple input logic gate shall begin with one of the following keywords:

and nand nor or xor xnor

The delay specification shall be zero, one, or two delays. If the specification contains two delays, the first delay shall determine the output rise delay, the second delay shall determine the output fall delay, and the smaller of the two delays shall apply to output transitions to x. If only one delay is specified, it shall specify both the rise delay and the fall delay. If there is no delay specification, there shall be no propagation delay through the gate.

These six logic gates shall have one output and one or more inputs. The first terminal in the terminal list shall connect to the output of the gate and all other terminals connect to its inputs.

The truth tables for these gates, showing the result of two input values, appear in Table 7-3.

Table 7-3—Truth tables for multiple input logic gates

and	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

or	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

xor	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

nand	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x

nor	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

xnor	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

Versions of these six logic gates having more than two inputs shall have a natural extension, but the number of inputs shall not alter propagation delays.

Example:

The following example declares a two input **and** gate:

```
and a1 (out, in1, in2);
```

The inputs are in1 and in2. The output is out. The instance name is a1.

7.3 buf and not gates

The instance declaration of a multiple output logic gate shall begin with one of the following keywords:

buf **not**

The delay specification shall be zero, one, or two delays. If the specification contains two delays, the first delay shall determine the output rise delay, the second delay shall determine the output fall delay, and the smaller of the two delays shall apply to output transitions to x. If only one delay is specified, it shall specify both the rise delay and the fall delay. If there is no delay specification, there shall be no propagation delay through the gate.

These two logic gates shall have one input and one or more outputs. The last terminal in the terminal list shall connect to the input of the logic gate, and the other terminals shall connect to the outputs of the logic gate.

Truth tables for these logic gates with one input and one output are shown in Table 7-4.

Table 7-4—Truth tables for multiple output logic gates

buf		not	
input	output	input	output
0	0	0	1
1	1	1	0
x	x	x	x
z	x	z	x

Example:

The following example declares a two output **buf**:

```
buf b1 (out1, out2, in);
```

The input is **in**. The outputs are **out1** and **out2**. The instance name is **b1**.

7.4 bufif1, bufif0, notif1, and notif0 gates

The instance declaration of a tri-state logic gate shall begin with one of the following keywords:

bufif0 **bufif1** **notif1** **notif0**

These four logic gates model three-state drivers. In addition to logic values 1 and 0, these gates can output **z**.

The delay specification shall be zero, one, two, or three delays. If the delay specification contains three delays, the first delay shall determine the rise delay, the second delay shall determine the fall delay, the third delay shall determine the delay of transitions to **z**, and the smallest of the three delays shall determine the delay of transitions to **x**. If the specification contains two delays, the first delay shall determine the output rise delay, the second delay shall determine the output fall delay, and the smaller of the two delays shall apply to output transitions to **x** and **z**. If only one delay is specified, it shall specify the delay for all output transitions. If there is no delay specification, there shall be no propagation delay through the gate.

Some combinations of data input values and control input values can cause these gates to output either of two values, without a preference for either value (see 7.10.2). These logic tables for these gates include two symbols representing such unknown results. The symbol **L** shall represent a result that has a value 0 or **z**. The symbol **H** shall represent a result that has a value 1 or **z**. Delays on transitions to **H** or **L** shall be treated the same as delays on transitions to **x**.

These four logic gates shall have one output, one data input, and one control input. The first terminal in the terminal list shall connect to the output, the second terminal shall connect to the data input, and the third terminal shall connect to the control input.

Table 7-5 presents the logic tables for these gates.

Table 7-5—Truth tables for tristate logic gates

bufif0	CONTROL				
		0	1	x	z
D	0	0	z	L	L
A	1	1	z	H	H
T	x	x	z	x	x
A	z	x	z	x	x

bufif1	CONTROL				
		0	1	x	z
D	0	z	0	L	L
A	1	z	1	H	H
T	x	z	x	x	x
A	z	z	x	x	x

notif0	CONTROL				
		0	1	x	z
D	0	1	z	H	H
A	1	0	z	L	L
T	x	x	z	x	x
A	z	x	z	x	x

notif1	CONTROL				
		0	1	x	z
D	0	z	1	H	H
A	1	z	0	L	L
T	x	z	x	x	x
A	z	z	x	x	x

Example:

The following example declares an instance of **bufif1**:

```
bufif1 bf1 (outw, inw, controlw);
```

The output is outw, the input is inw, and the control is controlw. The instance name is bf1.

7.5 MOS switches

The instance declaration of a MOS switch shall begin with one of the following keywords:

cmos **nmos** **pmos** **rcmos** **rnmos** **rpmos**

The **cmos** and **rcmos** switches are described in 7.7.

The **pmos** keyword stands for the P-type metal-oxide semiconductor (PMOS) transistor and the **nmos** keyword stands for the N-type metal-oxide semiconductor (NMOS) transistor. PMOS and NMOS transistors have relatively low impedance between their sources and drains when they conduct. The **rpmos** keyword stands for resistive PMOS transistor and the **rnmos** keyword stands for resistive NMOS transistor. Resistive PMOS and resistive NMOS transistors have significantly higher impedance between their sources and drains when they conduct than PMOS and NMOS transistors have. The load devices in static MOS networks are examples of **rpmos** and **rnmos** transistors. These four switches are *unidirectional channels* for data similar to the **bufif** gates.

The delay specification shall be zero, one, two, or three delays. If the delay specification contains three delays, the first delay shall determine the rise delay, the second delay shall determine the fall delay, the third delay shall determine the delay of transitions to z, and the smallest of the three delays shall determine the delay of transitions to x. If the specification contains two delays, the first delay shall determine the output rise delay, the second delay shall determine the output fall delay, and the smaller of the two delays shall apply to output transitions to x and z. If only one delay is specified, it shall specify the delay for all output transitions. If there is no delay specification, there shall be no propagation delay through the switch.

Some combinations of data input values and control input values can cause these switches to output either of two values, without a preference for either value. The logic tables for these switches include two symbols representing such unknown results. The symbol L represents a result that has a value 0 or z. The symbol H represents a result that has a value 1 or z. Delays on transitions to H and L shall be the same as delays on transitions to x.

These four switches shall have one output, one data input, and one control input. The first terminal in the terminal list shall connect to the output, the second terminal shall connect to the data input, and the third terminal shall connect to the control input.

The **nmos** and **pmos** switches shall pass signals from their inputs and through their outputs with a change in the strength of the signal in only one case, as discussed in 7.11. The **rnmos** and **rpmos** switches shall reduce the strength of signals that propagate through them, as discussed in 7.12.

Table 7-6 presents the logic tables for these switches.

Table 7-6—Truth tables for MOS switches

pmos rpmos	CONTROL				
		0	1	x	z
D	0	0	z	L	L
A	1	1	z	H	H
T	x	x	z	x	x
A	z	z	z	z	z

nmos rnmos	CONTROL				
		0	1	x	z
D	0	z	0	L	L
A	1	z	1	H	H
T	x	z	x	x	x
A	z	z	z	z	z

Example:

The following example declares a **pmos** switch:

```
pmos p1 (out, data, control);
```

The output is out, the data input is data, and the control input is control. The instance name is p1.

7.6 Bidirectional pass switches

The instance declaration of a bidirectional pass switch shall begin with one of the following keywords:

tran	tranif1	tranif0
rtran	rtranif1	rtranif0

The bidirectional pass switches shall not delay signals propagating through them. When **tranif0**, **tranif1**, **rtranif0**, or **rtranif1** devices are turned off they shall block signals, and when they are turned on they shall pass signals. The **tran**

and **rtran** devices cannot be turned off, and they shall always pass signals.

The delay specifications for **tranif1**, **tranif0**, **rtranif1**, and **rtranif0** devices shall be zero, one, or two delays. If the specification contains two delays, the first delay shall determine the *turn-on delay*, and the second delay shall determine the *turn-off delay*, and the smaller of the two delays shall apply to output transitions to *x* and *z*. If only one delay is specified, it shall specify both the turn-on and the turn-off delays. If there is no delay specification, there shall be no turn-on or turn-off delay for the bidirectional pass switch.

The bidirectional pass switches **tran** and **rtran** shall not accept delay specification.

The **tranif1**, **tranif0**, **rtranif1**, and **rtranif0** devices shall have three items in their terminal lists. The first two shall be bidirectional terminals that conduct signals to and from the devices, and the third terminal shall connect to a control input. The **tran** and **rtran** devices shall have terminal lists containing two bidirectional terminals. Both bidirectional terminals shall unconditionally conduct signals to and from the devices, allowing signals to pass in either direction through the devices. The bidirectional terminals of all six devices shall be connected only to scalar nets or bit-selects of vector nets.

The **tran**, **tranif0**, and **tranif1** devices shall pass signals with an alteration in their strength in only one case, as discussed in 7.11. The **rtran**, **rtranif0**, and **rtranif1** devices shall reduce the strength of the signals passing through them according to rules discussed in 7.12.

Example:

The following example declares an instance of **tranif1**:

```
tranif1 t1 (inout1,inout2,control);
```

The bidirectional terminals are *inout1* and *inout2*. The control input is *control*. The instance name is *t1*.

7.7 CMOS switches

The instance declaration of a CMOS switch shall begin with one of the following keywords:

cmos

rcmos

The delay specification shall be zero, one, two, or three delays. If the delay specification contains three delays, the first delay shall determine the rise delay, the second delay shall determine the fall delay, the third delay shall determine the delay of transitions to *z*, and the smallest of the three delays shall determine the delay of transitions to *x*. Delays in transitions to *H* or *L* are the same as delays in transitions to *x*. If the specification contains two delays, the first delay shall determine the output rise delay, the second delay shall determine the output fall delay, and the smaller of the two delays shall apply to output transitions to *x* and *z*. If only one delay is specified, it shall specify the delay for all output transitions. If there is no delay specification, there shall be no propagation delay through the switch.

The **cmos** and **rcmos** switches shall have a data input, a data output, and two control inputs. In the terminal list, the first terminal shall connect to the data output, the second terminal shall connect to the data input, the third terminal shall connect to the n-channel control input, and the last terminal shall connect to the p-channel control input.

The **cmos** gate shall pass signals with an alteration in their strength in only one case, as discussed in 7.11. The **rcmos** gate shall reduce the strength of signals passing through it according to rules described in 7.12.

The **cmos** switch shall be treated as the combination of a **pmos** switch and an **nmos** switch. The **rcmos** switch shall be treated as the combination of an **rpmos** switch and an **rnmos** switch. The combined switches in these configurations shall share data input and data output terminals, but they shall have separate control inputs.

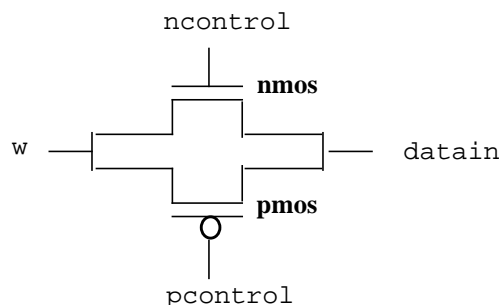
Example:

The equivalence of the **cmos** gate to the pairing of an **nmos** gate and a **pmos** gate is shown in the following example:

```
cmos (w, datain, ncontrol, pcontrol);
```

is equivalent to:

```
nmos (w, datain, ncontrol);  
pmos (w, datain, pcontrol);
```



7.8 pullup and pulldown sources

The instance declaration of a pullup or a pulldown source shall begin with one of the following keywords:

pullup

pulldown

A **pullup** source shall place a logic value 1 on the nets connected in its terminal list. A **pulldown** source shall place a logic value 0 on the nets connected in its terminal list. The signals that these sources place on nets shall have **pull** strength in the absence of a strength specification. If conflicting strength specification is declared, it shall be ignored. There shall be no delay specifications for these sources.

Example:

The following example declares two **pullup** instances:

```
pullup (strong1) p1 (neta), p2 (netb);
```

In this example, the p1 instance drives neta and the p2 instance drives netb.

7.9 Logic strength modeling

The Verilog HDL provides for accurate modeling of signal contention, bidirectional pass gates, resistive MOS devices, dynamic MOS, charge sharing, and other technology-dependent network configurations by allowing scalar net signal values to have a full range of unknown values and different levels of strength or combinations of levels of strength. This multiple-level logic strength modeling resolves combinations of signals into known or unknown values to represent the behavior of hardware with improved accuracy.

A strength specification shall have two components

- a) The strength of the 0 portion of the net value, called strength0, designated as one of the following:

supply0 strong0 pull0 weak0 highz0

- b) The strength of the 1 portion of the net value, called strength1, designated as one of the following:

supply1 strong1 pull1 weak1 highz1

The combinations (**highz0**, **highz1**) and (**highz1**, **highz0**) shall be considered illegal.

Despite this division of the strength specification, it is helpful to consider strength as a property occupying regions of a continuum in order to predict the results of combinations of signals.

Table 7-7 demonstrates the continuum of strengths. The left column lists the keywords used in specifying strengths. The right column gives correlated strength levels.

Table 7-7—Strength levels for scalar net signal values

Strength name	Strength level
supply0	7
strong0	6
pull0	5
large0	4
weak0	3
medium0	2
small0	1
highz0	0
highz1	0
small1	1
medium1	2
weak1	3
large1	4
pull1	5
strong1	6
supply1	7

In Table 7-7, there are four *driving strengths*:

supply strong pull weak

Signals with driving strengths shall propagate from gate outputs and continuous assignment outputs.

In Table 7-7, there are three *charge storage strengths*:

large medium small

Signals with the charge storage strengths shall originate in the **triereg** net type.

It is possible to think of the strengths of signals in the preceding table as locations on the scale in Figure 7-2.

strength0								strength1							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

Figure 7-2—Scale of strengths

Discussions of signal combinations later in this section employs graphics similar to those used in Figure 7-2.

If the signal value of a net is known, all of its strength levels shall be in either the strength0 part of the scale represented by Figure 7-2, or all strength levels shall be in its strength1 part. If the signal value of a net is unknown, it shall have strength levels in both the strength0 and the strength1 parts. A net with a signal value *z* shall have a strength level only in one of the 0 subdivisions of the parts of the scale.

7.10 Strengths and values of combined signals

In addition to a signal value, a net shall have either a single unambiguous strength level or an ambiguous strength consisting of more than one level. When signals combine, their strengths and values shall determine the strength and value of the resulting signal in accordance with the principles in 7.10.1 through 7.10.4.

7.10.1 Combined signals of unambiguous strength

This subclause deals with combinations of signals in which each signal has a known value and a single strength level.

If two or more signals of unequal strength combine in a wired net configuration, the stronger signal shall dominate all the weaker drivers and determine the result. The combination of two or more signals of like value shall result in the same value with the greater of all the strengths. The combination of signals identical in strength and value shall result in the same signal.

The combination of signals with unlike values and the same strength can have three possible results. Two of the results occur in the presence of wired logic and the third occurs in its absence. Wired logic is discussed in 7.10.4. The result in the absence of wired logic is the subject of Figure 7-4.

Example:

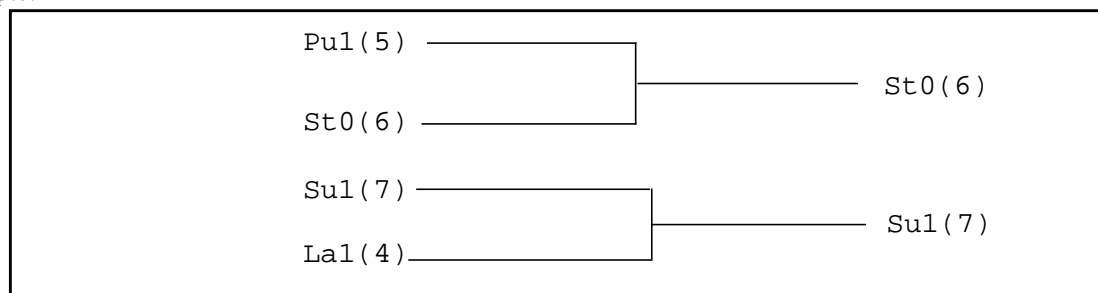


Figure 7-3—Combining unequal strengths

In Figure 7-3, the numbers in parentheses indicate the relative strengths of the signals. The combination of a **pull** 1 and a **strong** 0 results in a **strong** 0, which is the stronger of the two signals.

7.10.2 Ambiguous strengths: sources and combinations

There are several classifications of signals possessing ambiguous strengths

- Signals with known values and multiple strength levels
- Signals with a value x , which have strength levels consisting of subdivisions of both the strength1 and the strength0 parts of the scale of strengths in Figure 7-2
- Signals with a value L , which have strength levels that consist of high impedance joined with strength levels in the strength0 part of the scale of strengths in Figure 7-2
- Signals with a value H , which have strength levels that consist of high impedance joined with strength levels in the strength1 part of the scale of strengths in Figure 7-2

Many configurations can produce signals of ambiguous strength. When two signals of equal strength and opposite value combine, the result shall be a value x , along with the strength levels of both signals and all the smaller strength levels.

Examples:

Figure 7-4 shows the combination of a **weak** signal with a value 1 and a **weak** signal with a value 0 yielding a signal with **weak** strength and a value x .

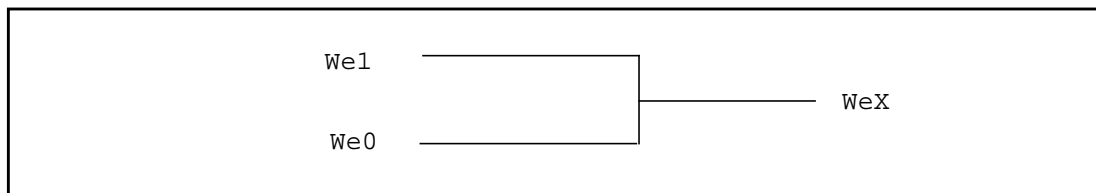


Figure 7-4—Combination of signals of equal strength and opposite values

This output signal is described in Figure 7-5.

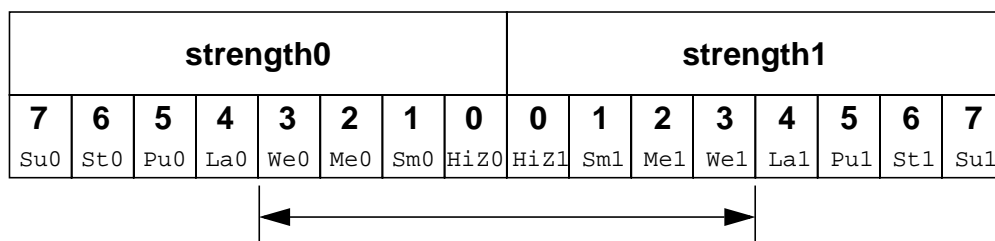


Figure 7-5—Weak x signal strength

An ambiguous signal strength can be a range of possible values. An example is the strength of the output from the tri-state drivers with unknown control inputs as shown in Figure 7-6.

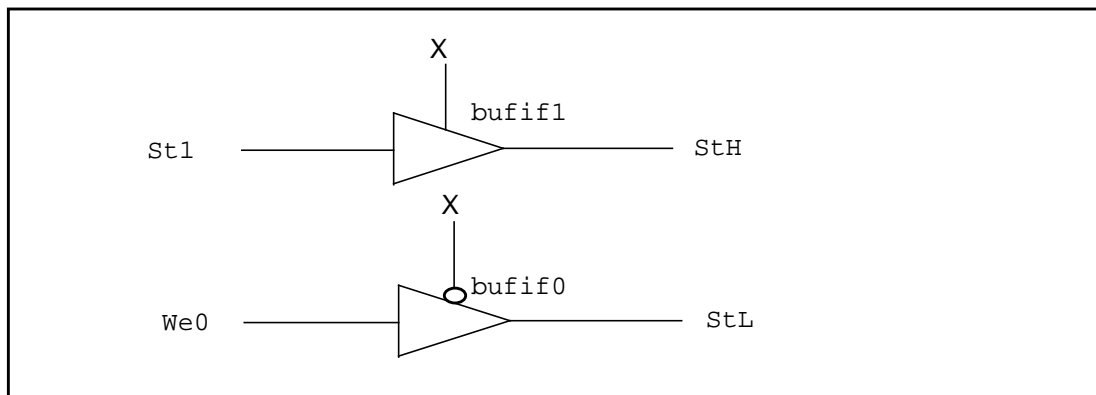


Figure 7-6—Bufifs with control inputs of x

The output of the **bufif1** in Figure 7-6 is a **strong H**, composed of the range of values described in Figure 7-7.

strength0								strength1							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

Figure 7-7—Strong H range of values

The output of the **bufif0** in Figure 7-6 is a **strong L**, composed of the range of values described in Figure 7-8.

strength0								strength1							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

Figure 7-8—Strong L range of values

The combination of two signals of ambiguous strength shall result in a signal of ambiguous strength. The resulting signal shall have a range of strength levels that includes the strength levels in its component signals. The combination of outputs from two tri-state drivers with unknown control inputs, shown in Figure 7-9, is an example.

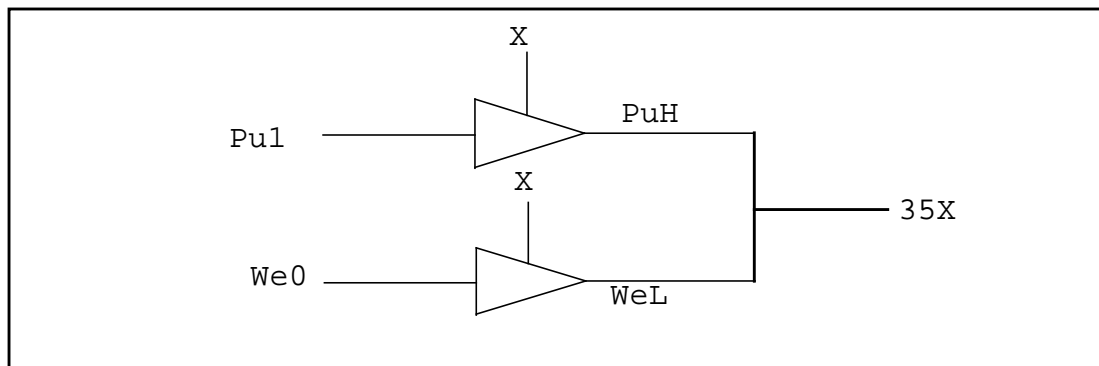


Figure 7-9—Combined signals of ambiguous strength

In Figure 7-9, the combination of signals of ambiguous strengths produces a range that includes the extremes of the signals and all the strengths between them, as described in Figure 7-10.

strength0								strength1							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

Figure 7-10—Range of strengths for an unknown signal

The result is a value x because its range includes the values 1 and 0. The number 35, which precedes the x , is a concatenation of two digits. The first is the digit 3, which corresponds to the highest strength0 level for the result. The second digit, 5, corresponds to the highest strength1 level for the result.

Switch networks can produce a ranges of strengths of the same value, such as the signals from the upper and lower configurations in Figure 7-11.

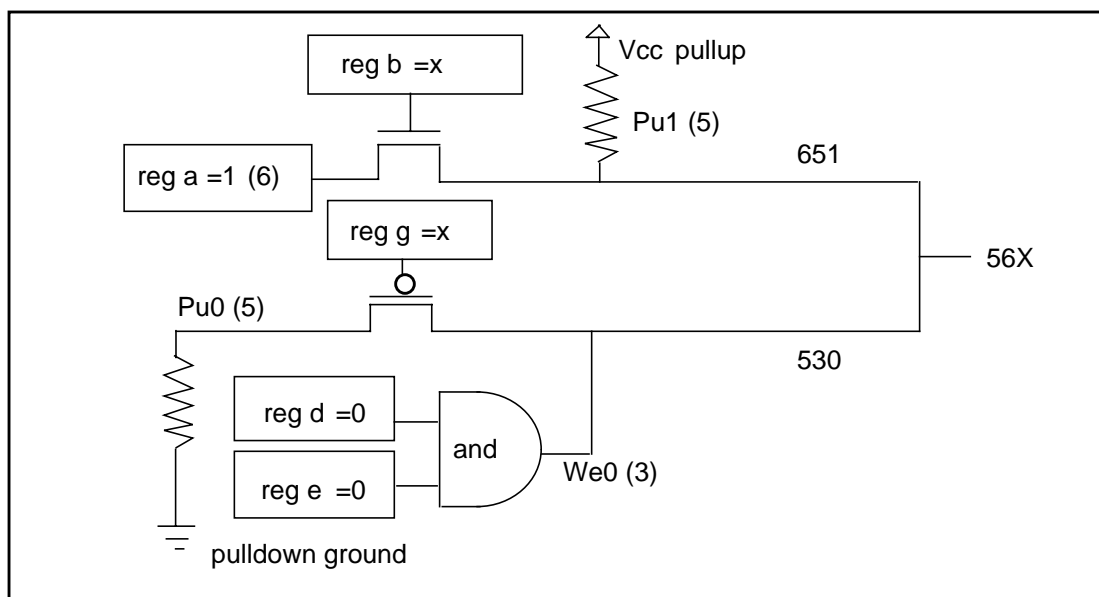


Figure 7-11—Ambiguous strengths from switch networks

In Figure 7-11, the upper combination of a reg, a gate controlled by a reg of unspecified value, and a pullup produces a signal with a value of 1 and a range of strengths (651) described in Figure 7-12.

strength0								strength1							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

↔

Figure 7-12—Range of two strengths of a defined value

In Figure 7-11, the lower combination of a **pulldown**, a gate controlled by a reg of unspecified value, and an **and** gate produces a signal with a value 0 and a range of strengths (530) described in Figure 7-13.

strength0								strength1							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

↔

Figure 7-13—Range of three strengths of a defined value

When the signals from the upper and lower configurations in Figure 7-11 combine, the result is an unknown with a range (56x) determined by the extremes of the two signals shown in Figure 7-14.

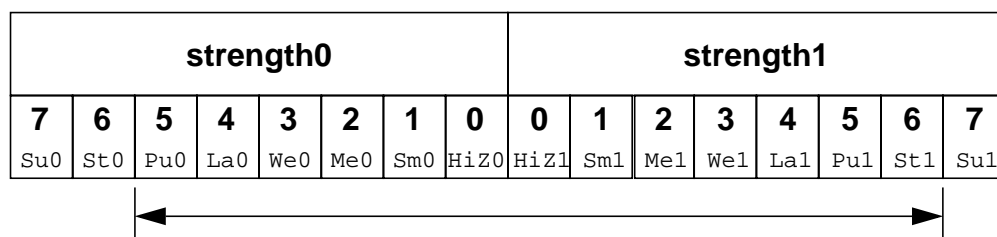


Figure 7-14—Unknown value with a range of strengths

In Figure 7-11, replacing the **pulldown** in the lower configuration with a **supply0** would change the range of the result to the range (StX) described in Figure 7-15.

The range in Figure 7-15 is **strong x**, because it is unknown and the extremes of both its components are **strong**. The extreme of the output of the lower configuration is **strong** because the lower **pmos** reduces the strength of the **supply0** signal. This modeling feature is discussed in 7.11.

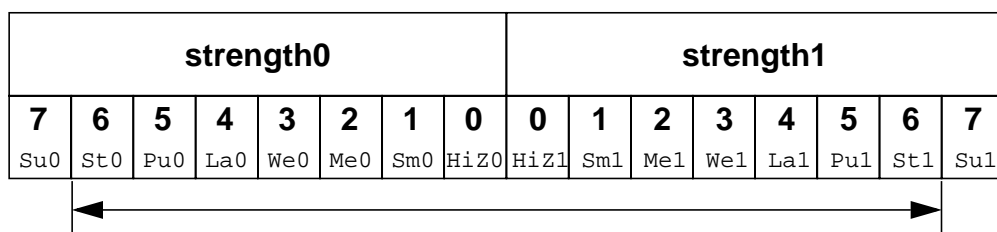


Figure 7-15—Strong X range

Logic gates produce results with ambiguous strengths as well as tri-state drivers. Such a case appears in Figure 7-16. The **and** gate N1 is declared with **highz0** strength, and N2 is declared with **weak0** strength.

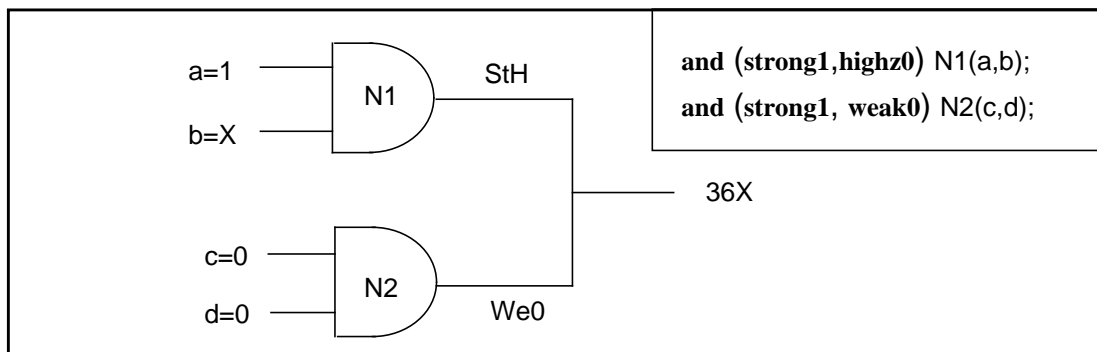


Figure 7-16—Ambiguous strength from gates

In Figure 7-16, reg b has an unspecified value, so input to the upper **and** gate is **strong x**. The upper **and** gate has a strength specification including **highz0**. The signal from the upper **and** gate is a **strong H** composed of the values as described in Figure 7-17.

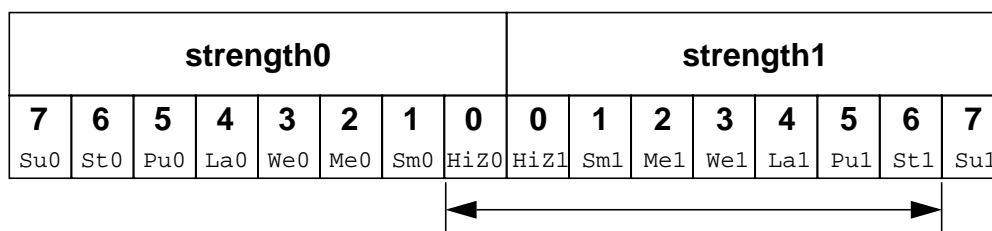


Figure 7-17—Ambiguous strength signal from a gate

HiZ0 is part of the result, because the strength specification for the gate in question specified that strength for an output with a value 0. A strength specification other than high impedance for the 0 value output results in a gate output value x. The output of the lower **and** gate is a **weak** 0 as described in Figure 7-18.

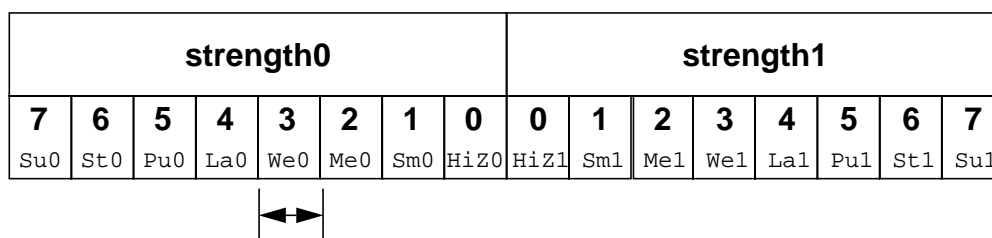


Figure 7-18—Weak 0

When the signals combine, the result is the range (36x) as described in Figure 7-19.

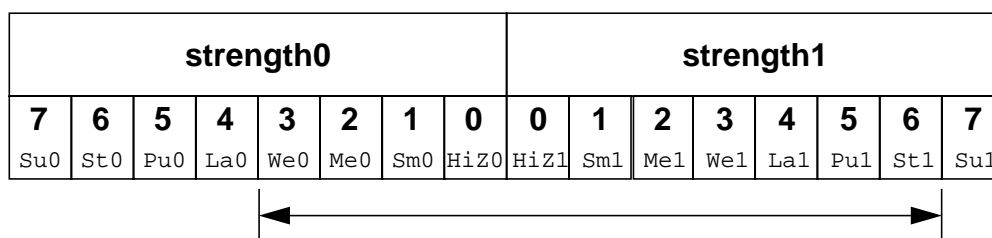


Figure 7-19—Ambiguous strength in combined gate signals

Figure 7-19 presents the combination of an ambiguous signal and an unambiguous signal. Such combinations are the topic of 7.10.3.

7.10.3 Ambiguous strength signals and unambiguous signals

The combination of a signal with unambiguous strength and known value with another signal of ambiguous strength presents several possible cases. To understand a set of rules governing this type of combination, it is necessary to consider the strength levels of the ambiguous strength signal separately from each other and relative to the unambiguous strength signal. When a signal of known value and unambiguous strength combines with a component of a signal of ambiguous strength, these shall be the effects

- The strength levels of the ambiguous strength signal that are greater than the strength level of the unambiguous signal shall remain in the result.
- The strength levels of the ambiguous strength signal that are smaller than or equal to the strength level of the unambiguous signal shall disappear from the result, subject to rule c.
- If the operation of rule a and rule b results in a gap in strength levels because the signals are of opposite value, the signals in the gap shall be part of the result.

The following figures show some applications of the rules.

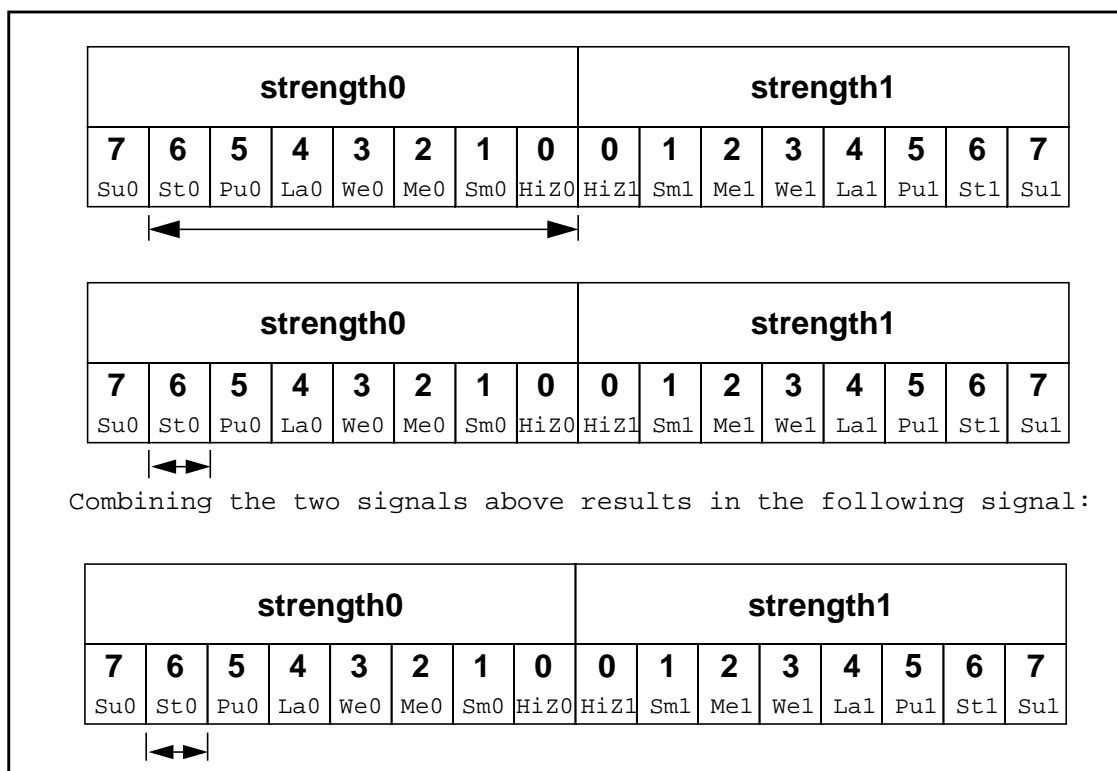


Figure 7-20—Elimination of strength levels

In Figure 7-20, the strength levels in the ambiguous strength signal that are smaller than or equal to the strength level of the unambiguous strength signal disappear from the result, demonstrating rule b.

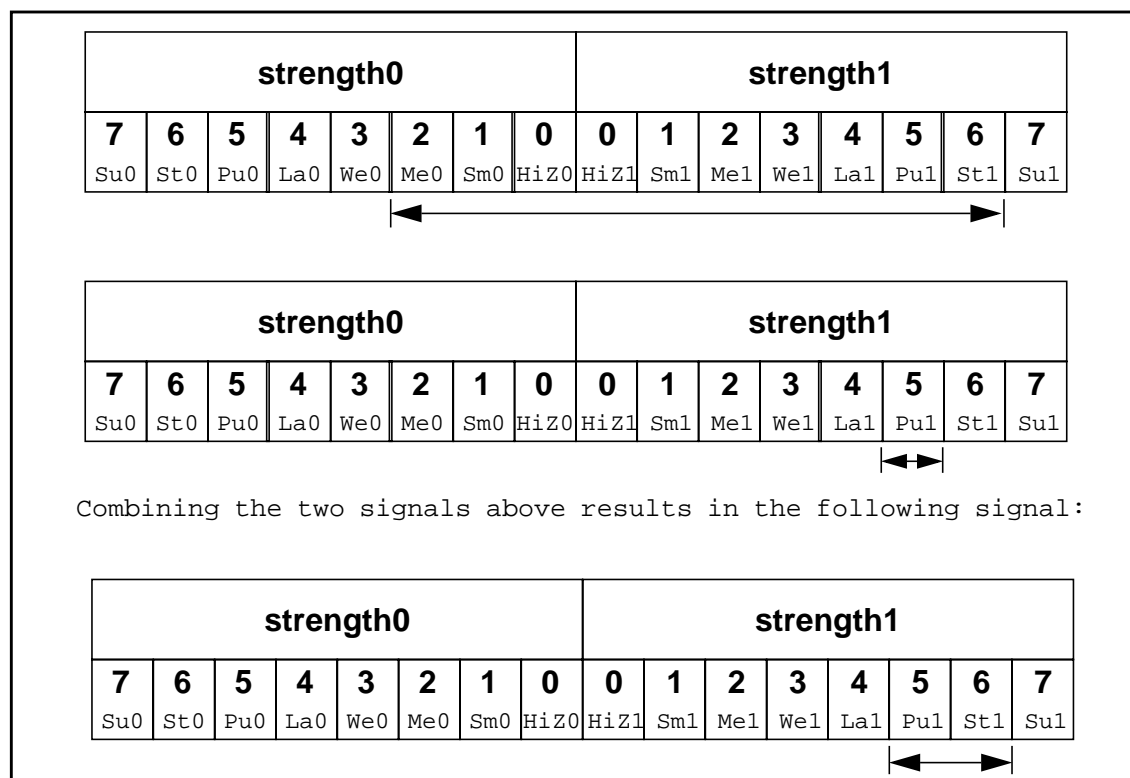


Figure 7-21—Result demonstrating a range and the elimination of strength levels of two values

In Figure 7-21, rules a, b, and c apply. The strength levels of the ambiguous strength signal that are of opposite value and lesser strength than the unambiguous strength signal disappear from the result. The strength levels in the ambiguous strength signal that are less than the strength level of the unambiguous strength signal, and of the same value, disappear from the result. The strength level of the unambiguous strength signal and the greater extreme of the ambiguous strength signal define a range in the result.

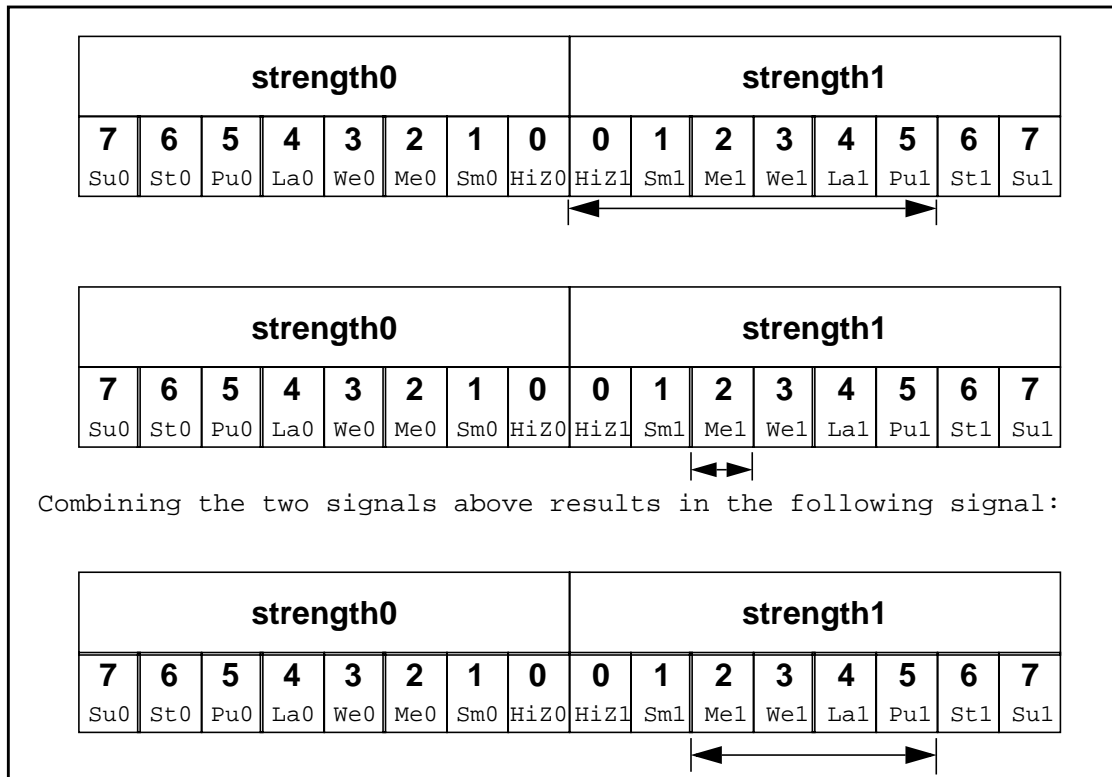


Figure 7-22—Result demonstrating a range and the elimination of strength levels of one value

In Figure 7-22, rules a and b apply. The strength levels in the ambiguous strength signal that are less than the strength level of the unambiguous strength signal disappear from the result. The strength level of the unambiguous strength signal and the strength level at the greater extreme of the ambiguous strength signal define a range in the result.

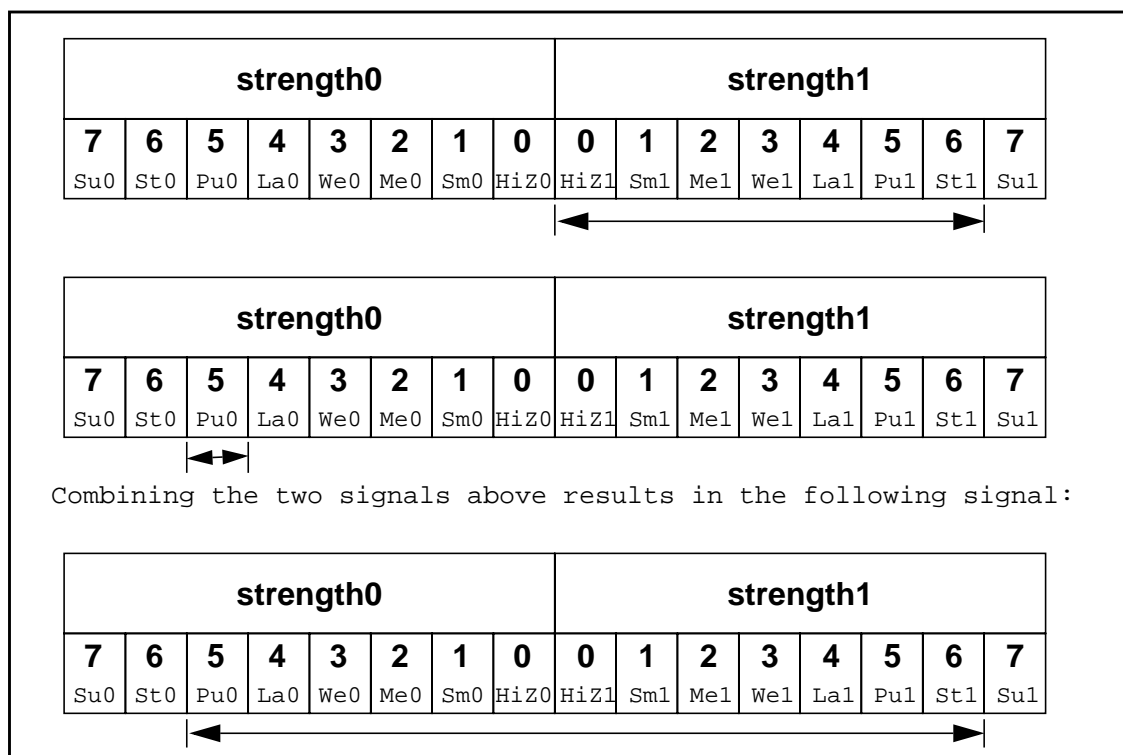


Figure 7-23—A range of both values

In Figure 7-23, rules a, b, and c apply. The greater extreme of the range of strengths for the ambiguous strength signal is larger than the strength level of the unambiguous strength signal. The result is a range defined by the greatest strength in the range of the ambiguous strength signal and by the strength level of the unambiguous strength signal.

7.10.4 Wired logic net types

The net types **triand**, **wand**, **trior**, and **wor** shall resolve conflicts when multiple drivers have the same strength. These net types shall resolve signal values by treating signals as inputs of logic functions.

Examples:

Consider the combination of two signals of unambiguous strength in Figure 7-24.

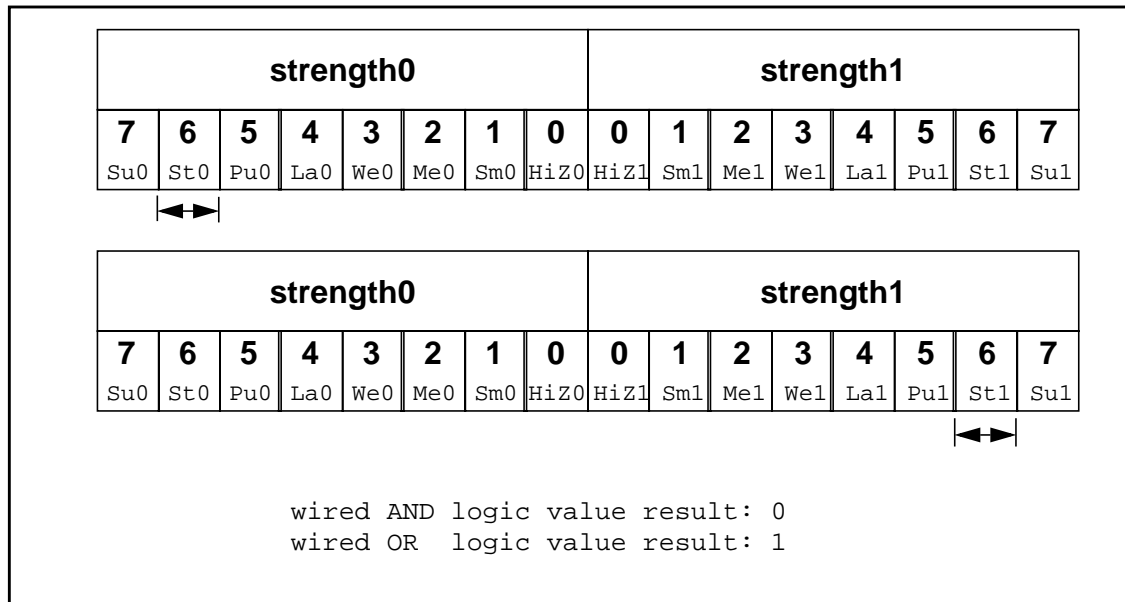


Figure 7-24—Wired logic with unambiguous strength signals

The combination of the signals in Figure 7-24, using *wired and* logic, produces a result with the same value as the result produced by an **and** gate with the value of the two signals as its inputs. The combination of signals using *wired or* logic produces a result with the same value as the result produced by an **or** gate with the values of the two signals as its inputs. The strength of the result is the same as the strength of the combined signals in both cases. If the value of the upper signal changes so that both signals in Figure 7-24 possess a value 1, then the results of both types of logic have a value 1.

When ambiguous strength signals combine in wired logic, it is necessary to consider the results of all combinations of each of the strength levels in the first signal with each of the strength levels in the second signal, as shown in Figure 7-25.

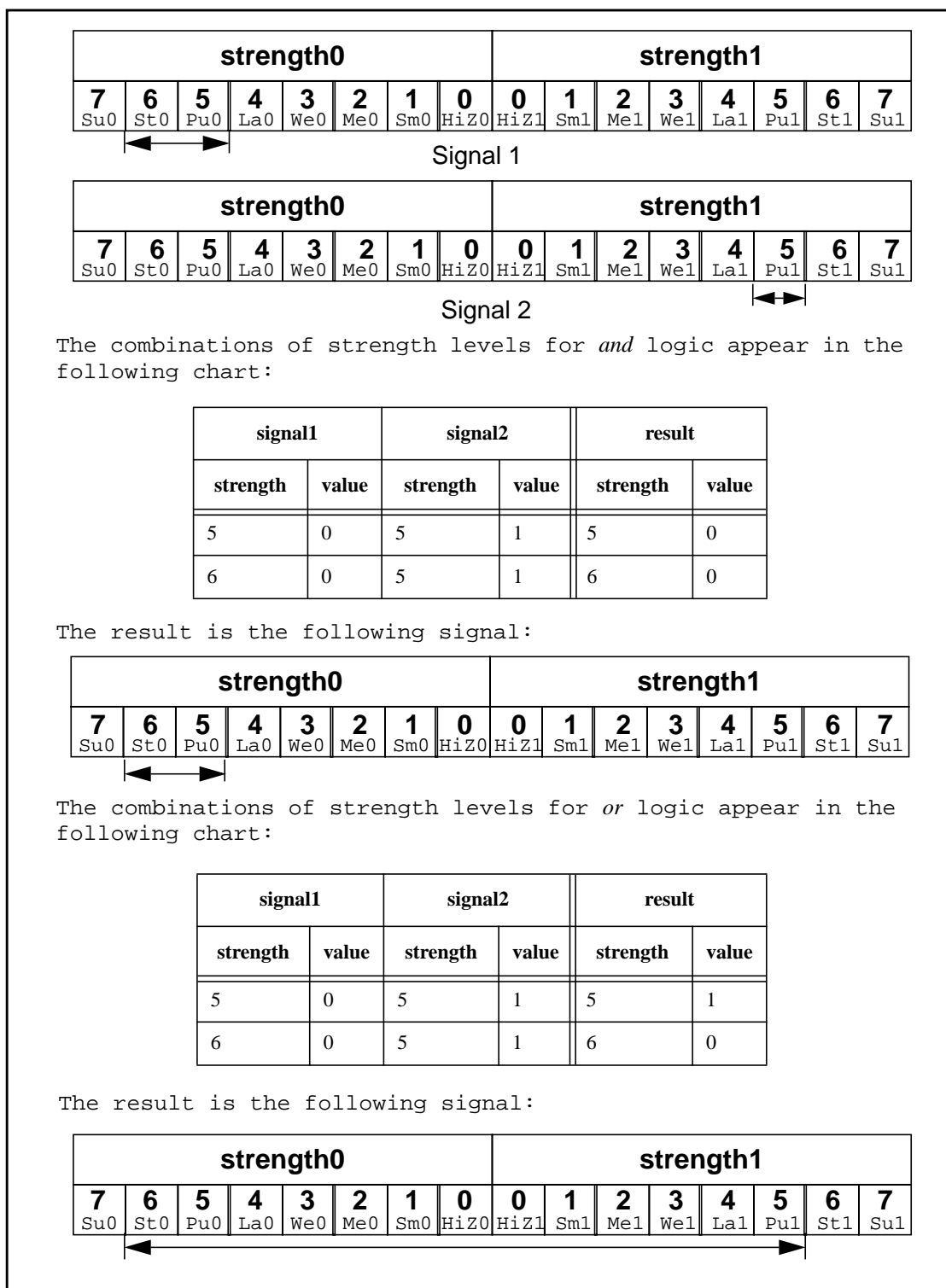


Figure 7-25—Wired logic and ambiguous strengths

7.11 Strength reduction by nonresistive devices

The **nmos**, **pmos**, and **cmos** switches shall pass the strength from the data input to the output, except that a **supply** strength shall be reduced to a **strong** strength.

The **tran**, **tranif0**, and **tranif1** switches shall not affect signal strength across the bidirectional terminals, except that a **supply** strength shall be reduced to a **strong** strength.

7.12 Strength reduction by resistive devices

The **rnmos**, **rpmos**, **rcmos**, **rtran**, **rtranif1**, and **rtranif0** devices shall reduce the strength of signals that pass through them according to Table 7-8.

Table 7-8—Strength reduction rules

Input strength	Reduced strength
Supply drive	Pull drive
Strong drive	Pull drive
Pull drive	Weak drive
Large capacitor	Medium capacitor
Weak drive	Medium capacitor
Medium capacitor	Small capacitor
Small capacitor	Small capacitor
High impedance	High impedance

7.13 Strengths of net types

The **tri0**, **tri1**, **supply0**, and **supply1** net types shall generate signals with specific strength levels. The **triereg** declaration can specify either of two signal strength levels other than a default strength level.

7.13.1 tri0 and tri1 net strengths

The **tri0** net type models a net connected to a resistive **pulldown** device. In the absence of an overriding source, such a signal shall have a value 0 and a **pull** strength. The **tri1** net type models a net connected to a resistive **pullup** device. In the absence of an overriding source, such a signal shall have a value 1 and a **pull** strength.

7.13.2 triereg strength

The **triereg** net type models charge storage nodes. The strength of the drive resulting from a **triereg** net that is in the charge storage state (that is, a driver charged the net and then went to high impedance) shall be one of these three strengths: **large**, **medium**, or **small**. The specific strength associated with a particular **triereg** net shall be specified by the user in the net declaration. The default shall be **medium**. The syntax of this specification is described in 3.4.1.

7.13.3 supply0 and supply1 net strengths

The **supply0** net type models ground connections. The **supply1** net type models connections to power supplies. The **supply0** and **supply1** net types shall have **supply** driving strengths.

7.14 Gate and net delays

Gate and net delays provide a means of more accurately describing delays through a circuit. The *gate delays* specify the signal propagation delay from any gate input to the gate output. Up to three values per output representing rise, fall, and turn-off delays can be specified (see 7.2 through 7.8).

Net delays refer to the time it takes from any driver on the net changing value to the time when the net value is updated and propagated further. Up to three delay values per net can be specified.

For both gates and nets, the *default delay* shall be zero when no delay specification is given. When one delay value is given, then this value shall be used for all propagation delays associated with the gate or the net. When two delays are given, the first delay shall specify the rise delay and the second delay shall specify the fall delay. The delay when the signal changes to high impedance or to unknown shall be the lesser of the two delay values.

For a three-delay specification

- The first delay refers to the transition to the 1 value (rise delay).
- The second delay refers to the transition to the 0 value (fall delay).
- The third delay refers to the transition to the high-impedance value.

When a value changes to the unknown (x) value, the delay is the smallest of the three delays. The strength of the input signal shall not affect the propagation delay from an input to an output.

Table 7-9 summarizes the from-to propagation delay choice for the two- and three-delay specifications.

Table 7-9—Rules for propagation delays

From value:	To value:	Delay used if there are	
		2 delays	3 delays
0	1	d1	d1
0	x	min(d1, d2)	min(d1, d2, d3)
0	z	min(d1, d2)	d3
1	0	d2	d2
1	x	min(d1, d2)	min(d1, d2, d3)
1	z	min(d1, d2)	d3
x	0	d2	d2
x	1	d1	d1
x	z	min(d1, d2)	d3
z	0	d2	d2
z	1	d1	d1
z	x	min(d1, d2)	min(d1, d2, d3)

Examples:

Example 1—The following is an example of a delay specification with one, two, and three delays:

```

and #(10) a1 (out, in1, in2);           // only one delay
and #(10,12) a2 (out, in1, in2);       // rise and fall delays
bufif0 #(10,12,11) b3 (out, in, ctrl); // rise, fall, and turn-off delays

```

Example 2—The following example specifies a simple latch module with tri-state outputs, where individual delays are given to the gates. The propagation delay from the primary inputs to the outputs of the module will be cumulative, and it depends on the signal path through the network.

```

module tri_latch (qout, nqout, clock, data, enable);
output qout, nqout;
input clock, data, enable;
tri qout, nqout;

not    #5          n1 (ndata, data);
nand   #(3,5)      n2 (wa, data, clock),
          n3 (wb, ndata, clock);
nand   #(12,15)    n4 (q, nq, wa),
          n5 (nq, q, wb);
bufif1 #(3,7,13)   q_drive (qout, q, enable),
          nq_drive (nqout, nq, enable);

endmodule

```

7.14.1 min:typ:max delays

The syntax for delays on gate primitives (including user-defined primitives; see Section 8), nets, and continuous assignments shall allow three values each for the rising, falling, and turn-off delays. The minimum, typical, and maximum values for each delay shall be specified as constant expressions separated by colons. There shall be no required relation (e.g., $\text{min} \leq \text{typ} \leq \text{max}$) between the expressions for minimum, typical, and maximum delays. These can be any three constant expressions.

Examples:

The following example shows min:typ:max values for rising, falling, and turn-off delays:

```

module iobuf (io1, io2, dir);
    ...
bufif0 #(5:7:9, 8:10:12, 15:18:21) b1 (io1, io2, dir);
bufif1 #(6:8:10, 5:7:9, 13:17:19) b2 (io2, io1, dir);
    ...
endmodule

```

The syntax for delay controls in procedural statements (see 9.7) also allows minimum, typical, and maximum values. These are specified by expressions separated by colons. The following example illustrates this concept.

```

parameter min_hi = 97, typ_hi = 100, max_hi = 107;
reg clk;

always begin
    #(95:100:105) clk = 1;
    #(min_hi:typ_hi:max_hi) clk = 0;
end

```

7.14.2 trireg net charge decay

Like all nets, the delay specification in a **trireg** net declaration can contain up to three delays. The first two delays shall specify the delay for transition to the 1 and 0 logic states when the **trireg** net is driven to these states by a driver. The third delay shall specify the *charge decay time* instead of the delay in a transition to the z logic state. The charge decay time specifies the delay between when the drivers of a **trireg** net turn off and when its stored charge can no longer be determined.

A **trireg** net does not need a turn-off delay specification because a **trireg** net never makes a transition to the z logic state. When the drivers of a **trireg** net make transitions from the 1, 0, or x logic states to off, the **trireg** net shall retain the previous 1, 0, or x logic state that was on its drivers. The z value shall not propagate from the drivers of a **trireg** net to a **trireg** net. A **trireg** net can only hold a z logic state when z is the initial logic state of the **trireg** net or when the **trireg** net is forced to the z state with a **force** statement (see 9.3.2).

A delay specification for charge decay models a charge storage node that is not ideal—a charge storage node whose charge leaks out through its surrounding devices and connections.

The following subclauses describe the charge decay process and the delay specification for charge decay.

7.14.2.1 The charge decay process

Charge decay is the cause of transition of a 1 or 0 that is stored in a **trireg** net to an unknown value (x) after a specified delay. The charge decay process shall begin when the drivers of the **trireg** net turn off and the **trireg** net starts to hold charge. The charge decay process shall end under the following two conditions:

- a) The delay specified by charge decay time elapses and the **trireg** net makes a transition from 1 or 0 to x.
- b) The drivers of **trireg** net turn on and propagate a 1, 0, or x into the **trireg** net.

7.14.2.2 The delay specification for charge decay time

The third delay in a **trireg** net declaration shall specify the charge decay time. A three-valued delay specification in a **trireg** net declaration shall have the following form:

```

#(d1, d2, d3)          // (rise_delay, fall_delay, charge_decay_time)

```

The charge decay time specification in a **trireg** net declaration shall be preceded by a rise and a fall delay specification.

Examples:

Example 1—The following example shows a specification of the charge decay time in a **trireg** net declaration:

```

trireg (large) #(0,0,50) cap1;

```

This example declares a **trireg** net named **cap1**. This **trireg** net stores a **large** charge. The delay specifications for the rise delay is 0, the fall delay is 0, and the charge decay time specification is 50 time units.

Example 2—The next example presents a source description file that contains a **triereg** net declaration with a charge decay time specification. Figure 7-26 shows an equivalent schematic for the source description.

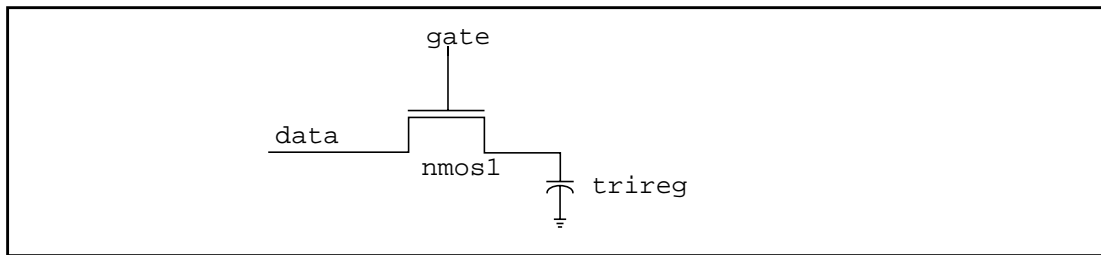


Figure 7-26—Triereg net with capacitance

```
module capacitor;
reg data, gate;

// triereg declaration with a charge decay time of 50 time units
triereg (large) #(0,0,50) cap1;

nmos nmos1 (cap1, data, gate); // nmos that drives the triereg

initial begin
    $monitor("%0d data=%v gate=%v cap1=%v", $time, data, gate, cap1);
    data = 1;
    // Toggle the driver of the control input to the nmos switch
        gate = 1;
    #10 gate = 0;
    #30 gate = 1;
    #10 gate = 0;
    #100 $finish;
end
endmodule
```

Section 8

User-defined primitives (UDPs)

This section describes a modeling technique to augment the set of predefined gate primitives by designing and specifying new primitive elements called user-defined primitives (UDPs). Instances of these new UDPs can be used in exactly the same manner as the gate primitives to represent the circuit being modeled.

The following two types of behavior can be represented in a user-defined primitive:

- a) Combinational—modeled by a combinational UDP
- b) Sequential—modeled by a sequential UDP

A combinational UDP uses the value of its inputs to determine the next value of its output. A sequential UDP uses the value of its inputs and the current value of its output to determine the value of its output. Sequential UDPs provide a way to model sequential circuits such as flip-flops and latches. A sequential UDP can model both level-sensitive and edge-sensitive behavior.

Each UDP has exactly one output, which can be in one of three states: 0, 1, or x. The tri-state value z is not supported. In sequential UDPs, the output always has the same value as the internal state.

The z values passed to UDP inputs shall be treated the same as x values.

8.1 UDP definition

UDP definitions are independent of modules; they are at the same level as module definitions in the syntax hierarchy. They can appear anywhere in the source text, either before or after they are instantiated inside a module. They shall not appear between the keywords **module** and **endmodule**.

NOTE—Implementations may limit the maximum number of UDP definitions in a model, but they shall allow at least 256.

The formal syntax of the UDP definition is given in Syntax 8-1.

```

udp_declaration ::= (From Annex A - A.5.1)
    { attribute_instance } primitive udp_identifier ( udp_port_list );
    udp_port_declaration { udp_port_declaration }
    udp_body
    endprimitive
| { attribute_instance } primitive udp_identifier ( udp_declaration_port_list );
    udp_body
    endprimitive

udp_port_list ::= (From Annex A - A.5.2)
    output_port_identifier , input_port_identifier { , input_port_identifier }

udp_declaration_port_list ::=
    udp_output_declaration , udp_input_declaration { , udp_input_declaration }

udp_port_declaration ::=
    udp_output_declaration
| udp_input_declaration
| udp_reg_declaration

udp_output_declaration ::=
    { attribute_instance } output port_identifier ;
| { attribute_instance } output reg port_identifier [ = constant_expression ] ;

udp_input_declaration ::=
    { attribute_instance } input list_of_port_identifiers ;

udp_reg_declaration ::=
    { attribute_instance } reg variable_identifier ;

udp_body ::= (From Annex A - A.5.3)
    combinational_body | sequential_body

combinational_body ::=
    table combinational_entry { combinational_entry } endtable

combinational_entry ::=
    level_input_list : output_symbol ;

sequential_body ::=
    [ udp_initial_statement ] table sequential_entry { sequential_entry } endtable

udp_initial_statement ::=
    initial output_port_identifier = init_val ;

init_val ::= 1'b0 | 1'b1 | 1'bx | 1'bX | 1'B0 | 1'B1 | 1'Bx | 1'BX | 1 | 0

sequential_entry ::=
    seq_input_list : current_state : next_state ;

seq_input_list ::=
    level_input_list | edge_input_list

level_input_list ::=
    level_symbol { level_symbol }

edge_input_list ::=
    { level_symbol } edge_indicator { level_symbol }

edge_indicator ::=
    ( level_symbol level_symbol ) | edge_symbol

current_state ::= level_symbol

next_state ::= output_symbol | -

output_symbol ::= 0 | 1 | x | X

level_symbol ::= 0 | 1 | x | X | ? | b | B

edge_symbol ::= r | R | f | F | p | P | n | N | *

```

Syntax 8-1—Syntax for user-defined primitives

8.1.1 UDP header

A UDP definition shall have one of two alternate forms. The first form shall begin with the keyword **primitive**, followed by an identifier, which is the name of the UDP. This in turn is followed by a comma-separated list of ports enclosed in parentheses, which is followed by a semicolon. The UDP definition header is followed by port declarations and a state table. The UDP definition shall be terminated by the keyword **endprimitive**.

The second form shall begin with the keyword **primitive**, followed by an identifier, which is the name of the UDP. This in turn is followed by a comma separated list of ports declarations enclosed in parenthesis, followed by a semicolon. The UDP definition header is followed by a state table. The UDP definition shall be terminated by the keyword **endprimitive**.

UDPs have multiple input ports and exactly one output port; bidirectional inout ports are not permitted on UDPs. All ports of a UDP shall be scalar; vector ports are not permitted.

The output port shall be the first port in the port list.

8.1.2 UDP port declarations

UDPs shall contain input and output port declarations. The output port declaration begins with the keyword **output**, followed by one output port name. The input port declaration begins with the keyword **input**, followed by one or more input port names.

Sequential UDPs shall contain a **reg** declaration for the output port, either in addition to the output declaration, when the UDP is declared using the first form of a UDP Header, or as part of the output_declaration, in either case. Combinational UDPs cannot contain a **reg** declaration. The initial value of the output port can be specified in an **initial** statement in a sequential UDP (see 8.1.3).

NOTE—Implementations may limit the maximum number of inputs to a UDP, but they shall allow at least 9 inputs for sequential UDPs and 10 inputs for combinational UDPs.

8.1.3 Sequential UDP initial statement

The sequential UDP initial statement specifies the value of the output port when simulation begins. This statement begins with the keyword **initial**. The statement that follows shall be an assignment statement that assigns a single-bit literal value to the output port.

8.1.4 UDP state table

The state table defines the behavior of a UDP. It begins with the keyword **table** and is terminated with the keyword **endtable**. Each row of the table is terminated by a semicolon.

Each row of the table is created using a variety of characters (see Table 8-1), which indicate input values and output state. Three states—0, 1, and x—are supported. The z state is explicitly excluded from consideration in user-defined primitives. A number of special characters are defined to represent certain combinations of state possibilities. These are described in Table 8-1.

The order of the input state fields of each row of the state table is taken directly from the port list in the UDP definition header. It is not related to the order of the input port declarations.

Combinational UDPs have one field per input and one field for the output. The input fields are separated from the output field by a colon (:). Each row defines the output for a particular combination of the input values (see 8.2).

Sequential UDPs have an additional field inserted between the input fields and the output field. This additional field represents the current state of the UDP and is considered equivalent to the current output value. It is delimited by colons. Each row defines the output based on the current state, particular combinations of input values, and at most one input transition (see 8.4). A row such as the one shown below is illegal:

(01) (10) 0 : 0 : 1 ;

If all input values are specified as x, then the output state shall be specified as x.

It is not necessary to explicitly specify every possible input combination. All combinations of input values that are not explicitly specified result in a default output state of x.

It is illegal to have the same combination of inputs, including edges, specified for different outputs.

8.1.5 Z values in UDP

The z value in a table entry is not supported and it is considered illegal. The z values passed to UDP inputs shall be treated the same as x values.

8.1.6 Summary of symbols

To improve the readability and to ease writing of the state table, several special symbols are provided. Table 8-1 summarizes the meaning of all the value symbols that are valid in the table part of a UDP definition.

Table 8-1—UDP table symbols

Symbol	Interpretation	Comments
0	Logic 0	
1	Logic 1	
x	Unknown	Permitted in the input fields of all UDPs and in the current state field of sequential UDPs. Not permitted in the output field.
?	Iteration of 0, 1, and x	Not permitted in output field.
b	Iteration of 0 and 1	Permitted in the input fields of all UDPs and in the current state field of sequential UDPs. Not permitted in the output field.
-	No change	Permitted only in the output field of a sequential UDP.
(vw)	Value change from v to w	v and w can be any one of 0, 1, x, ?, or b, and are only permitted in the input field.
*	Same as (??)	Any value change on input.
r	Same as (01)	Rising edge on input.
f	Same as (10)	Falling edge on input.
p	Iteration of (01), (0 x) and (x1)	Potential positive edge on the input.
n	Iteration of (10), (1x) and (x0)	Potential negative edge on the input.

8.2 Combinational UDPs

In combinational UDPs, the output state is determined solely as a function of the current input states. Whenever an input state changes, the UDP is evaluated and the output state is set to the value indicated by the row in the state table that matches all the input states. All combinations of the inputs that are not explicitly specified will drive the output state to the unknown value *x*.

Examples:

The following example defines a multiplexer with two data inputs and a control input.

```

primitive multiplexer (mux, control, dataA, dataB);
output mux;
input control, dataA, dataB;
table
// control dataA dataB mux
    0      1      0 : 1 ;
    0      1      1 : 1 ;
    0      1      x : 1 ;
    0      0      0 : 0 ;
    0      0      1 : 0 ;
    0      0      x : 0 ;
    1      0      1 : 1 ;
    1      1      1 : 1 ;
    1      x      1 : 1 ;
    1      0      0 : 0 ;
    1      1      0 : 0 ;
    1      x      0 : 0 ;
    x      0      0 : 0 ;
    x      1      1 : 1 ;
endtable
endprimitive

```

The first entry in this example can be explained as follows: when *control* equals 0, and *dataA* equals 1, and *dataB* equals 0, then output *mux* equals 1.

The input combination 0xx (*control*=0, *dataA*=x, *dataB*=x) is not specified. If this combination occurs during simulation, the value of output port *mux* will become x.

Using *?*, the description of a multiplexer can be abbreviated as

```

primitive multiplexer (mux, control, dataA, dataB);
output mux;
input control, dataA, dataB;
table
// control dataA dataB mux
    0      1      ? : 1 ;    // ? = 0 1 x
    0      0      ? : 0 ;
    1      ?      1 : 1 ;
    1      ?      0 : 0 ;
    x      0      0 : 0 ;
    x      1      1 : 1 ;
endtable
endprimitive

```

8.3 Level-sensitive sequential UDPs

Level-sensitive sequential behavior is represented the same way as combinational behavior, except that the output is declared to be of type **reg**, and there is an additional field in each table entry. This new field represents the current state of the UDP. The output field in a sequential UDP represents the next state.

Example:

Consider the example of a latch:

```

primitive latch (q, clock, data);
output q; reg q;
input clock, data;
table
// clock data q q+
    0      1 : ? : 1 ;
    0      0 : ? : 0 ;
    1      ? : ? : - ;    // - = no change
endtable
endprimitive

```

This description differs from a combinational UDP model in two ways. First, the output identifier *q* has an additional **reg** declaration to indicate that there is an internal state *q*. The output value of the UDP is always the same as the internal state. Second, a field for the current state, which is separated by colons from the inputs and the output, has been added.

8.4 Edge-sensitive sequential UDPs

In level-sensitive behavior, the values of the inputs and the current state are sufficient to determine the output value. Edge-sensitive behavior differs in that changes in the output are triggered by specific transitions of the inputs. This makes the state table a transition table.

Each table entry can have a transition specification on at most one input. A transition is specified by a pair of values in parenthesis such as (01) or a transition symbol such as *x*. Entries such as the following are illegal:

```
(01)(01)0 : 0 : 1 ;
```

All transitions that do not affect the output shall be explicitly specified. Otherwise, such transitions cause the value of the output to change to *x*. All unspecified transitions default to the output value *x*.

If the behavior of the UDP is sensitive to edges of any input, the desired output state shall be specified for *all* edges of *all* inputs.

Example:

The following example describes a rising edge D flip-flop:

```
primitive d_edge_ff (q, clock, data);
output q; reg q;
input clock, data;
table
// clock data      q      q+
// obtain output on rising edge of clock
(01)  0      :  ?  :  0  ;
(01)  1      :  ?  :  1  ;
(0?)  1      :  1  :  1  ;
(0?)  0      :  0  :  0  ;
// ignore negative edge of clock
(?0)  ?      :  ?  :  -  ;
// ignore data changes on steady clock
?      (??)  :  ?  :  -  ;
endtable
endprimitive
```

The terms such as (01) represent transitions of the input values. Specifically, (01) represents a transition from 0 to 1. The first line in the table of the preceding UDP definition is interpreted as follows: when clock changes value from 0 to 1, and data equals 0, the output goes to 0 no matter what the current state

The transition of clock from 0 to *x* with data equal to 0 and current state equal to 1 will result in the output *q* going to *x*.

8.5 Sequential UDP initialization

The initial value on the output port of a sequential UDP can be specified with an initial statement that provides a procedural assignment. The initial statement is optional.

Like initial statements in modules, the initial statement in UDPs begin with the keyword **initial**. The valid contents of initial statements in UDPs and the valid left-hand and right-hand sides of their procedural assignment statements differ from initial statements in modules. A partial list of differences between these two types of initial statements is described in Table 8-2.

Table 8-2—Initial statements in UDPs and modules

Initial statements in UDPs	Initial statements in modules
Contents limited to one procedural assignment statement	Contents can be one procedural statement of any type or a block statement that contains more than one procedural statement

Table 8-2—Initial statements in UDPs and modules (*continued*)

Initial statements in UDPs	Initial statements in modules
The procedural assignment statement shall assign a value to a reg whose identifier matches the identifier of an output terminal	Procedural assignment statements in initial statements can assign values to a reg whose identifier does not match the identifier of an output terminal
The procedural assignment statement shall assign one of the following values: 1'b1, 1'b0, 1'bx, 1, 0	Procedural assignment statements can assign values of any size, radix, and value

Examples:

Example 1—The following example shows a sequential UDP that contains an initial statement.

```

primitive srff (q, s, r);
output q; reg q;
input s, r;
initial q = 1'b1;
table
//  s  r    q    q+
   1  0  :  ?  :  1  ;
   f  0  :  1  :  -  ;
   0  r  :  ?  :  0  ;
   0  f  :  0  :  -  ;
   1  1  :  ?  :  0  ;
endtable
endprimitive

```

The output *q* has an initial value of 1 at the start of the simulation; a delay specification on an instantiated UDP does not delay the simulation time of the assignment of this initial value to the output. When simulation starts, this value is the current state in the state table. Delays are not permitted in a UDP initial statement.

Example 2—The following example and figure show how values are applied in a module that instantiates a sequential UDP with an initial statement.

```

primitive dff1 (q, clk, d);
input clk, d;
output q; reg q;
initial q = 1'b1;
table
// clk    d      q      q+
   r      0    :    ?    :    0    ;
   r      1    :    ?    :    1    ;
   f      ?    :    ?    :    -    ;
   ?      *    :    ?    :    -    ;
endtable
endprimitive

module dff (q, qb, clk, d);
input clk, d;
output q, qb;
    dff1 g1 (qi, clk, d);
    buf #3 g2 (q, qi);
    not #5 g3 (qb, qi);
endmodule

```

The UDP `dff1` contains an initial statement that sets the initial value of its output to 1. The module `dff` contains an instance of UDP `dff1`.

Figure 8-1 shows the schematic of the preceding module and the simulation propagation times of the initial value of the UDP output.

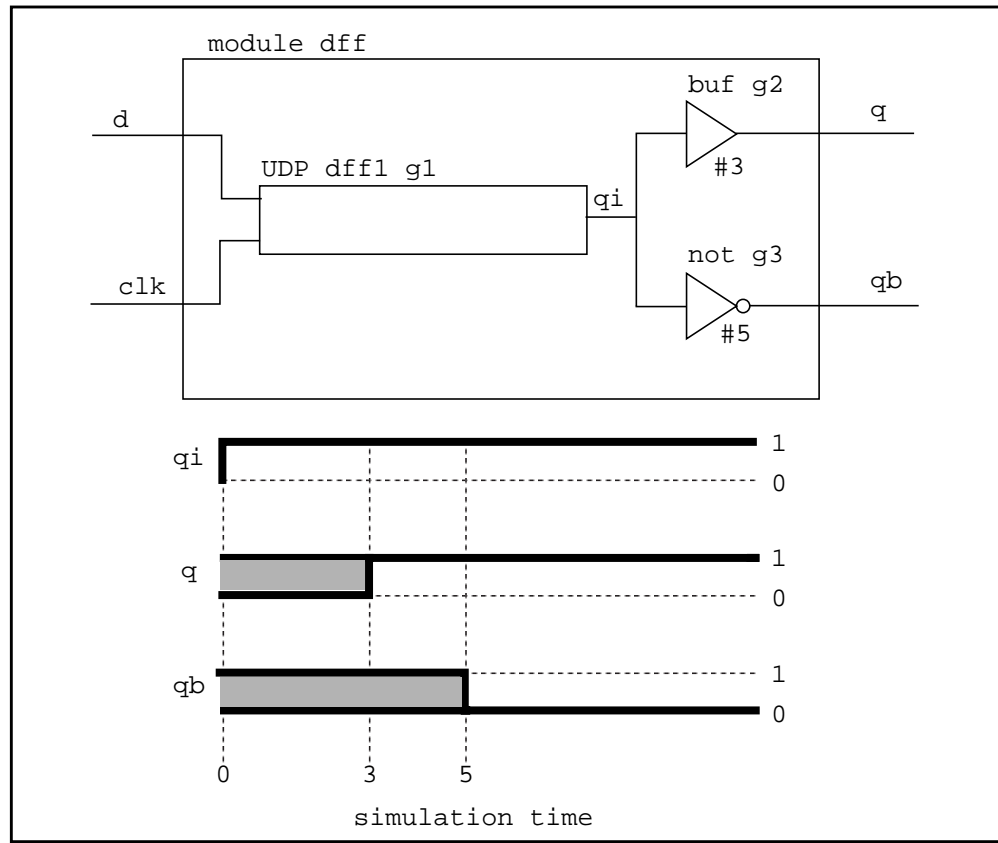


Figure 8-1—Module schematic and simulation times of initial value propagation

In Figure 8-1, the fanout from the UDP output *qi* includes nets *q* and *qb*. At simulation time 0, *qi* changes value to 1. That initial value of *qi* does not propagate to net *q* until simulation time 3, and it does not propagate to net *qb* until simulation time 5.

8.6 UDP instances

The syntax for creating a UDP instance is shown in Syntax 8-2.

```

udp_instantiation ::= (From Annex A- A.5.4)
    udp_identifier [ drive_strength ] [ delay2 ]
    [ attribute_instance ] udp_instance { , udp_instance } ;
udp_instance ::=
    [ name_of_udp_instance ] ( output_terminal , input_terminal
    { , input_terminal } )
name_of_udp_instance ::=
    udp_instance_identifier [ range ]

```

Syntax 8-2—Syntax for UDP instances

Instances of user-defined primitives are specified inside modules in the same manner as gates (see 7.1). The instance name is optional, just as for gates. The port connection order is as specified in the UDP definition. Only two delays

can be specified because *z* is not supported for UDPs. An optional range may be specified for an array of UDP instances. The port connection rules remain the same as outlined in 7.1.

Example:

The following example creates an instance of the D-type flip-flop `d_edge_ff` (defined in 8.4).

```

module flip;
reg clock, data;
parameter p1 = 10;
parameter p2 = 33;
parameter p3 = 12;

d_edge_ff #p3 d_inst (q, clock, data);

initial begin
    data = 1;
    clock = 1;
    #(20 * p1) $finish;
end
always #p1 clock = ~clock;
always #p2 data = ~data;
endmodule

```

8.7 Mixing level-sensitive and edge-sensitive descriptions

UDP definitions allow a mixing of the level-sensitive and the edge-sensitive constructs in the same table. When the input changes, the edge-sensitive cases are processed first, followed by level-sensitive cases. Thus, when level-sensitive and edge-sensitive cases specify different output values, the result is specified by the level-sensitive case.

Example:

```

primitive jk_edge_ff (q, clock, j, k, preset, clear);
output q; reg q;
input clock, j, k, preset, clear;
table
// clock jk pc state output/next state
?    ?? 01 : ? : 1 ; // preset logic
?    ?? *1 : 1 : 1 ;
?    ?? 10 : ? : 0 ; // clear logic
?    ?? 1* : 0 : 0 ;
r    00 00 : 0 : 1 ; // normal clocking cases
r    00 11 : ? : - ;
r    01 11 : ? : 0 ;
r    10 11 : ? : 1 ;
r    11 11 : 0 : 1 ;
r    11 11 : 1 : 0 ;
f    ?? ?? : ? : - ;
b    *? ?? : ? : - ; // j and k transition cases
b    ?* ?? : ? : - ;
endtable
endprimitive

```

In this example, the preset and clear logic is level-sensitive. Whenever the preset and clear combination is 01, the output has value 1. Similarly, whenever the preset and clear combination has value 10, the output has value 0.

The remaining logic is sensitive to edges of the clock. In the normal clocking cases, the flip-flop is sensitive to the rising clock edge, as indicated by an *r* in the clock field in those entries. The insensitivity to the falling edge of clock is indicated by a hyphen (-) in the output field (see Table 8-1) for the entry with an *f* as the value of clock. Remember that the desired output for this input transition shall be specified to avoid unwanted *x* values at the output. The last two entries show that the transitions in *j* and *k* inputs do not change the output on a steady low or high *clock*.

8.8 Level-sensitive dominance

Table 8-3 shows level-sensitive and edge-sensitive entries in the example from 8.7, their level-sensitive or edge-sensitive behavior, and a case of input values that each includes.

Table 8-3—Mixing of level-sensitive and edge-sensitive entries

Entry	Included case	Behavior
? ?? 01: ?; 1;	0 00 01: 0: 1;	Level-sensitive
f ?? ??: ?; -;	f 00 01: 0: 0;	Edge-sensitive

The included cases specify opposite next state values for the same input and current state combination. The level-sensitive included case specifies that when the inputs *clock*, *jk*, and *pc* values are 0, 00, and 01 and the current state is 0, the output changes to 1. The edge-sensitive included case specifies that when *clock* falls from 1 to 0, the other inputs *jk* and *pc* are 00 and 01, and the current state is 0, then the output changes to 0.

When the edge-sensitive case is processed first, followed by the level-sensitive case, the output changes to 1.

Section 9

Behavioral modeling

The language constructs introduced so far allow hardware to be described at a relatively detailed level. Modeling a circuit with logic gates and continuous assignments reflects quite closely the logic structure of the circuit being modeled; however, these constructs do not provide the power of abstraction necessary for describing complex high-level aspects of a system. The procedural constructs described in this section are well suited to tackling problems such as describing a microprocessor or implementing complex timing checks.

This section starts with a brief overview of a behavioral model to provide a context for many types of behavioral statements in the Verilog HDL.

9.1 Behavioral model overview

Verilog *behavioral models* contain *procedural statements* that control the simulation and manipulate variables of the data types previously described. These statements are contained within procedures. Each procedure has an activity flow associated with it.

The activity starts at the control constructs **initial** and **always**. Each *initial* construct and each *always* construct starts a separate activity flow. All of the activity flows are concurrent to model the inherent concurrence of hardware. These constructs are formally described in 9.9.

The following example shows a complete Verilog behavioral model.

```
module behave;
  reg [1:0] a, b;

  initial begin
    a = 'b1;
    b = 'b0;
  end
  always begin
    #50 a = ~a;
  end
  always begin
    #100 b = ~b;
  end
endmodule
```

During simulation of this model, all of the flows defined by the initial and always constructs start together at simulation time zero. The initial constructs execute once, and the always constructs execute repetitively.

In this model, the reg variables a and b initialize to 1 and 0 respectively at simulation time zero. The initial construct is then complete and does not execute again during this simulation run. This initial construct contains a *begin-end block* (also called a *sequential block*) of statements. In this begin-end block a is initialized first, followed by b.

The always constructs also start at time zero, but the values of the variables do not change until the times specified by the delay controls (introduced by #) have elapsed. Thus, reg a inverts after 50 time units and reg b inverts after 100

time units. Since the `always` constructs repeat, this model will produce two square waves. The `reg a` toggles with a period of 100 time units, and `reg b` toggles with a period of 200 time units. The two `always` constructs proceed concurrently throughout the entire simulation run.

9.2 Procedural assignments

As described in Section 6, procedural assignments are used for updating **reg**, **integer**, **time**, **real**, **realtime**, and memory data types. There is a significant difference between procedural assignments and continuous assignments:

- *Continuous assignments* drive nets and are evaluated and updated whenever an input operand changes value.
- *Procedural assignments* update the value of variables under the control of the procedural flow constructs that surround them.

The right-hand side of a procedural assignment can be any expression that evaluates to a value. The left-hand side shall be a variable that receives the assignment from the right-hand side. The left-hand side of a procedural assignment can take one of the following forms:

- **reg**, **integer**, **real**, **realtime**, or **time** data type: an assignment to the name reference of one of these data types.
- Bit-select of a **reg**, **integer**, or **time** data type: an assignment to a single bit that leaves the other bits untouched.
- Part-select of a **reg**, **integer**, or **time** data type: a part-select of one or more contiguous bits that leaves the rest of the bits untouched.
- Memory word: a single word of a memory.
- Concatenation of any of the above: a concatenation of any of the previous four forms can be specified, which effectively partitions the result of the right-hand side expression and assigns the partition parts, in order, to the various parts of the concatenation.

NOTE—When the right-hand side evaluates to fewer bits than the left-hand side, then if the right-hand side is signed (see 4.5), it shall be sign-extended to the size of the left-hand side.

The Verilog HDL contains two types of procedural assignment statements:

- Blocking procedural assignment statements
- Nonblocking procedural assignment statements

Blocking and nonblocking procedural assignment statements specify different procedural flows in sequential blocks.

9.2.1 Blocking procedural assignments

A *blocking procedural assignment* statement shall be executed before the execution of the statements that follow it in a sequential block (see 9.8.1). A blocking procedural assignment statement shall not prevent the execution of statements that follow it in a parallel block (see 9.8.2).

The syntax for a blocking procedural assignment is given in Syntax 9-1.

```

blocking_assignment ::= (From Annex A - A.6.2)
    variable_lvalue = [ delay_or_event_control ] expression
delay_control ::= (From Annex A - A.6.5)
    # delay_value
    | # ( mintypmax_expression )
delay_or_event_control ::=
    delay_control
    | event_control
    | repeat ( expression ) event_control
event_control ::=
    @ event_identifier
    | @ ( event_expression )
    | @*
    | @ ( * )
event_expression ::=
    expression
    | hierarchical_identifier
    | posedge expression
    | negedge expression
    | event_expression or event_expression
    | event_expression , event_expression
variable_lvalue ::= (From Annex A - A.8.5)
    hierarchical_variable_identifier
    | hierarchical_variable_identifier [ expression ] { [ expression ] }
    | hierarchical_variable_identifier [ expression ] { [ expression ] }
      [ range_expression ]
    | hierarchical_variable_identifier [ range_expression ]
    | variable_concatenation

```

Syntax 9-1—Syntax for blocking assignments

In this syntax, `reg_lvalue` is a data type that is valid for a procedural assignment statement, `=` is the assignment operator, and `delay_or_event_control` is the optional intra-assignment timing control. The control can be either a delay control (e.g., `#6`) or an event_control (e.g., `@(posedge clk)`). The expression is the right-hand side value that shall be assigned to the left-hand side. If `reg_lvalue` requires an evaluation, it shall be evaluated at the time specified by the intra-assignment timing control.

The `=` assignment operator used by blocking procedural assignments is also used by procedural continuous assignments and continuous assignments.

Example:

The following examples show blocking procedural assignments.

```

rega = 0;
rega[3] = 1;           // a bit-select
rega[3:5] = 7;         // a part-select
mema[address] = 8'hff; // assignment to a mem element
{carry, acc} = rega + regb; // a concatenation

```

9.2.2 The nonblocking procedural assignment

The *nonblocking procedural assignment* allows assignment scheduling without blocking the procedural flow. The nonblocking procedural assignment statement can be used whenever several variable assignments within the same time step can be made without regard to order or dependence upon each other.

The syntax for a nonblocking procedural assignment is given in Syntax 9-2.

```

nonblocking_assignment ::= (From Annex A - A.6.2)
    variable_lvalue <= [ delay_or_event_control ] expression
delay_control ::= (From Annex A - A.6.5)
    # delay_value
    | # ( mintypmax_expression )
delay_or_event_control ::=
    delay_control
    | event_control
    | repeat ( expression ) event_control
event_control ::=
    @ event_identifier
    | @ ( event_expression )
    | @*
    | @ ( * )
event_expression ::=
    expression
    | hierarchical_identifier
    | posedge expression
    | negedge expression
    | event_expression or event_expression
    | event_expression , event_expression
variable_lvalue ::= (From Annex A - A.8.5)
    hierarchical_variable_identifier
    | hierarchical_variable_identifier [ expression ] { [ expression ] }
    | hierarchical_variable_identifier [ expression ] { [ expression ] }
    | [ range_expression ]
    | hierarchical_variable_identifier [ range_expression ]
    | variable_concatenation

```

Syntax 9-2—Syntax for nonblocking assignments

In this syntax, *reg_lvalue* is a data type that is valid for a procedural assignment statement, *<=* is the nonblocking assignment operator, and *delay_or_event_control* is the optional intra-assignment timing control. If *reg_lvalue* requires an evaluation, it shall be evaluated at the same time as the expression on the right-hand side. The order of evaluation of the *reg_lvalue* and the expression on the right-hand side is undefined if timing control is not specified.

The nonblocking assignment operator is the same operator as the less-than-or-equal-to relational operator. The interpretation shall be decided from the context in which *<=* appears. When *<=* is used in an expression, it shall be interpreted as a relational operator, and when it is used in a nonblocking procedural assignment, it shall be interpreted as an assignment operator.

The nonblocking procedural assignments shall be evaluated in two steps as discussed in Section 5. These two steps are shown in the following example.

Example 1:

```

module evaluates2 (out);
output out;
reg a, b, c;

initial begin
    a = 0;
    b = 1;
    c = 0;
end

always c = #5 ~c;

always @(posedge c) begin
    a <= b; // evaluates, schedules,
    b <= a; // and executes in two steps
end
endmodule

```

Step 1:

At posedge c, the simulator evaluates the right-hand sides of the nonblocking assignments and schedules the assignments of the new values at the end of the *nonblocking assign update* events (see 5.4).

Step 2:

When the simulator activates the *nonblocking assign update* events, the simulator updates the left-hand side of each nonblocking assignment statement.

Nonblocking
assignment
schedules
changes at
time 5

a = 0

b = 1

Assignment
values are:

a = 1

b = 0

At the end of the time step means that the nonblocking assignments are the last assignments executed in a time step—with one exception. Nonblocking assignment events can create blocking assignment events. These blocking assignment events shall be processed after the scheduled nonblocking events.

Unlike an event or delay control for blocking assignments, the nonblocking assignment does not block the procedural flow. The nonblocking assignment evaluates and schedules the assignment, but it does not block the execution of subsequent statements in a **begin-end** block.

Example 2:

```

//non_block1.v
module non_block1;
reg a, b, c, d, e, f;

//blocking assignments
initial begin
    a = #10 1; // a will be assigned 1 at time 10
    b = #2 0;  // b will be assigned 0 at time 12
    c = #4 1;  // c will be assigned 1 at time 16
end
//non-blocking assignments
initial begin
    d <= #10 1; // d will be assigned 1 at time 10
    e <= #2 0;  // e will be assigned 0 at time 2
    f <= #4 1;  // f will be assigned 1 at time 4
end
endmodule

```

*scheduled
changes at
time 2*

e = 0

*scheduled
changes at
time 4*

f = 1

*scheduled
changes at
time 10*

d = 1

As shown in the previous example, the simulator evaluates and schedules assignments for the end of the current time step and can perform swapping operations with the nonblocking procedural assignments.

Example 3:

```
//non_block1.v
module non_block1;
reg a, b;
initial begin
    a = 0;
    b = 1;
    a <= b; // evaluates, schedules, and
    b <= a; // executes in two steps
end
initial begin
    $monitor ($time, , "a = %b b = %b", a, b);
    #100 $finish;
end
endmodule
```

Step 1: The simulator evaluates the right-hand side of the nonblocking assignments and schedules the assignments for the end of the current time step.

Step 2:

At the end of the current time step, the simulator updates the left-hand side of each nonblocking assignment statement.

assignment values are:

<p><i>a = 1</i></p> <p><i>b = 0</i></p>

When multiple nonblocking assignments are scheduled to occur in the same variable in a particular time slot, the order in which the assignments are evaluated is not guaranteed—the final value of the variable is indeterminate. As shown in the following example, the value of reg a is not known until the end of time step 4.

Example 4:

```
module multiple2 (out);
output out;
reg a;

initial a = 1;
// The assigned value of the reg is indeterminate
initial begin
    a <= #4 0; // schedules a = 0 at time 4
    a <= #4 1; // schedules a = 1 at time 4
end // At time 4, a = ??
endmodule
```

If the simulator executes two procedural blocks concurrently, and if these procedural blocks contain nonblocking assignment operators to the same variable, the final value of that variable is indeterminate. For example, the value of reg a is indeterminate in the following example.

Example 5:

```
module multiple3 ;
reg a;

initial a = 1;
initial a <= #4 0; // schedules 0 at time 4
initial a <= #4 1; // schedules 1 at time 4

// At time 4, a = ??
endmodule
```


When multiple nonblocking assignments with timing controls are made to the same variable, the assignments are made without cancelling nonblocking assignments scheduled at other times. Scheduling an assignment to a variable at the same time as a previously scheduled assignment to the same variable shall result in an arbitrary order of assignment to that variable, and, hence, the final value of that variable cannot be predicted.

The following example shows how the value of `i[0]` is assigned to `r1` and how the assignments are scheduled to occur after each time delay.

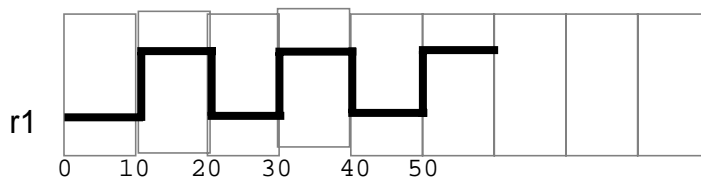
Example 6:

```

module multiple;
reg r1;
reg [2:0] i;

initial begin
  // starts at time 0, does not hold the block
  r1 = 0;
  // makes assignments to r1 without cancelling previous assignments
  for (i = 0; i <= 5; i = i+1)
    r1 <= # (i*10) i[0];
end
endmodule

```



9.3 Procedural continuous assignments

The *procedural continuous assignments* (using keywords **assign** and **force**) are procedural statements that allow expressions to be driven continuously onto variables or nets. The syntax for these statements is given in Syntax 9-3.

```

net_assignment ::= (From Annex A - A.6.1)
    net_lvalue = expression
procedural_continuous_assignments ::= (From Annex A - A.6.2)
    assign variable_assignment
    | deassign variable_lvalue
    | force variable_assignment
    | force net_assignment
    | release variable_lvalue
    | release net_lvalue
variable_assignment ::= (From Annex A - A.6.3)
    variable_lvalue = expression
net_lvalue ::= (From Annex A - A.8.5)
    hierarchical_net_identifier
    | hierarchical_net_identifier [ constant_expression ] { [ constant_expression ] }
    | hierarchical_net_identifier [ constant_expression ] { [ constant_expression ] }
      [ constant_range_expression ]
    | hierarchical_net_identifier [ constant_range_expression ]
    | net_concatenation
variable_lvalue ::=
    hierarchical_variable_identifier
    | hierarchical_variable_identifier [ expression ] { [ expression ] }
    | hierarchical_variable_identifier [ expression ] { [ expression ] }
      [ range_expression ]
    | hierarchical_variable_identifier [ range_expression ]
    | variable_concatenation

```

Syntax 9-3—Syntax for procedural continuous assignments

The left-hand side of the assignment in the *assign statement* shall be a variable reference or a concatenation of variables. It shall not be a memory word (array reference) or a bit-select or a part-select of a variable.

In contrast, the left-hand side of the assignment in the *force statement* can be a variable reference or a net reference. It can be a concatenation of any of the above. Bit-selects and part-selects of vector variables or vector nets are not allowed.

9.3.1 The assign and deassign procedural statements

The *assign* procedural continuous assignment statement shall override all procedural assignments to a variable. The *deassign* procedural statement shall end a procedural continuous assignment to a variable. The value of the variable shall remain the same until the reg is assigned a new value through a procedural assignment or a procedural continuous assignment. The assign and deassign procedural statements allow, for example, modeling of asynchronous clear/preset on a D-type edge-triggered flip-flop, where the clock is inhibited when the clear or preset is active.

If the keyword **assign** is applied to a variable for which there is already a procedural continuous assignment, then this new procedural continuous assignment shall deassign the variable before making the new procedural continuous assignment.

Example:

The following example shows a use of the assign and deassign procedural statements in a behavioral description of a D-type flip-flop with preset and clear inputs.

```

module dff (q, d, clear, preset, clock);
output q;
input d, clear, preset, clock;
reg q;

always @(clear or preset)
    if (!clear)
        assign q = 0;
    else if (!preset)
        assign q = 1;
    else
        deassign q;

always @(posedge clock)
    q = d;
endmodule

```

If either `clear` or `preset` is low, then the output `q` will be held continuously to the appropriate constant value and a positive edge on the `clock` will not affect `q`. When both the `clear` and `preset` are high, then `q` is deassigned.

If either operand to an arithmetic operator is real, the resulting expression is of type real.

9.3.2 The force and release procedural statements

Another form of procedural continuous assignment is provided by the *force* and *release* procedural statements. These statements have a similar effect to the assign-deassign pair, but a force can be applied to nets as well as to variables. The left-hand side of the assignment can be a variable, a net, a constant bit-select of a vector net, a part-select of a vector net, or a concatenation. It cannot be a memory word (array reference) or a bit-select or a part-select of a vector variable.

A *force* statement to a variable shall override a procedural assignment or procedural continuous assignment that takes place on the variable until a release procedural statement is executed on the variable. After the *release* procedural statement is executed, the variable shall not immediately change value (as would a net that is forced). The value specified in the force statement shall be maintained in the variable until the next procedural assignment takes place, except in the case where a procedural continuous assignment is active on the variable.

A force procedural statement on a net overrides all drivers of the net—gate outputs, module outputs, and continuous assignments—until a release procedural statement is executed on the net.

Releasing a variable that currently has an active procedural continuous assignment shall re-establish that assignment.

Example:

```

module test;
reg a, b, c, d;
wire e;

and and1 (e, a, b, c);

initial begin
    $monitor("%d d=%b,e=%b", $stime, d, e);
    assign d = a & b & c;
    a = 1;
    b = 0;
    c = 1;
    #10;
    force d = (a | b | c);
    force e = (a | b | c);
    #10 $stop;
    release d;
    release e;
    #10 $finish;
end
endmodule

```

```

Results:
    0 d=0,e=0
   10 d=1,e=1
   20 d=0,e=0

```

In this example, an **and** gate instance and1 is “patched” as an **or** gate by a force procedural statement that forces its output to the value of its logical or inputs, and an assign procedural statement of logical and values is “patched” as an assign procedural statement of logical or values.

The right-hand side of a procedural continuous assignment or a force statement can be an expression. This shall be treated just as a continuous assignment; that is, if any variable on the right-hand side of the assignment changes, the assignment shall be re-evaluated while the assign or force is in effect. For example:

```

force a = b + f(c) ;

```

Here, if b changes or c changes, a will be forced to the new value of the expression $b + f(c)$.

9.4 Conditional statement

The *conditional statement* (or *if-else* statement) is used to make a decision as to whether a statement is executed or not. Formally, the syntax is given in Syntax 9-4.

```

conditional_statement ::= (From Annex A - A.6.6)
    if ( expression )
        statement_or_null [ else statement_or_null ]
    | if_else_if_statement
function_conditional_statement ::= (From Annex A - A.6.6)
    if ( expression ) function_statement_or_null
    [ else function_statement_or_null ]
    | function_if_else_if_statement

```

Syntax 9-4—Syntax of if statement

If the expression evaluates to true (that is, has a nonzero known value), the first statement shall be executed. If it evaluates to false (has a zero value or the value is x or z), the first statement shall not execute. If there is an else statement and expression is false, the else statement shall be executed.

Since the numeric value of the if expression is tested for being zero, certain shortcuts are possible. For example, the following two statements express the same logic:

```

if (expression)
if (expression != 0)

```

Because the else part of an if-else is optional, there can be confusion when an else is omitted from a nested if sequence. This is resolved by always associating the else with the closest previous if that lacks an else. In the example below, the else goes with the inner if, as shown by indentation.

```

if (index > 0)
    if (rega > regb)
        result = rega;
    else // else applies to preceding if
        result = regb;

```

If that association is not desired, a begin-end block statement shall be used to force the proper association, as shown below.

```

if (index > 0) begin
    if (rega > regb)
        result = rega;
end
else result = regb;

```

9.4.1 If-else-if construct

The following construction occurs so often that it is worth a brief separate discussion:

```

if_else_if_statement ::= (From Annex A - A.6.6)
    if ( expression ) statement_or_null
    { else if ( expression ) statement_or_null }
    [ else statement_or_null ]

function_if_else_if_statement ::= (From Annex A - A.6.6)
    if ( expression ) function_statement_or_null
    { else if ( expression ) function_statement_or_null }
    [ else function_statement_or_null ]

```

Syntax 9-5—Syntax of if-else-if construct

This sequence of if statements (known as an *if-else-if* construct) is the most general way of writing a multiway decision. The expressions shall be evaluated in order; if any expression is true, the statement associated with it shall be executed, and this shall terminate the whole chain. Each statement is either a single statement or a block of statements.

The last else part of the if-else-if construct handles the none-of-the-above or default case where none of the other conditions were satisfied. Sometimes there is no explicit action for the default; in that case, the trailing else statement can be omitted or it can be used for error checking to catch an impossible condition.

Example:

The following module fragment uses the if-else statement to test the variable `index` to decide whether one of three `modify_seg` regs has to be added to the memory address, and which increment is to be added to the `index` reg. The first ten lines declare the regs and parameters.

```

// declare regs and parameters
reg [31:0] instruction, segment_area[255:0];
reg [7:0] index;
reg [5:0] modify_seg1,
    modify_seg2,
    modify_seg3;
parameter
    segment1 = 0, inc_seg1 = 1,
    segment2 = 20, inc_seg2 = 2,
    segment3 = 64, inc_seg3 = 4,
    data = 128;

// test the index variable
if (index < segment2) begin
    instruction = segment_area [index + modify_seg1];
    index = index + inc_seg1;
end
else if (index < segment3) begin
    instruction = segment_area [index + modify_seg2];
    index = index + inc_seg2;
end
else if (index < data) begin
    instruction = segment_area [index + modify_seg3];
    index = index + inc_seg3;
end
else
    instruction = segment_area [index];

```

9.5 Case statement

The *case* statement is a multiway decision statement that tests whether an expression matches one of a number of other expressions and branches accordingly. The case statement has the syntax shown in Syntax 9-6.

```

case_statement ::= (From Annex A - A.6.7)
    case ( expression )
        case_item { case_item } endcase
    | casez ( expression )
        case_item { case_item } endcase
    | casex ( expression )
        case_item { case_item } endcase
case_item ::=
    expression { , expression } : statement_or_null
    | default [ : ] statement_or_null
function_case_statement ::=
    case ( expression )
        function_case_item { function_case_item } endcase
    | casez ( expression )
        function_case_item { function_case_item } endcase
    | casex ( expression )
        function_case_item { function_case_item } endcase
function_case_item ::=
    expression { , expression } : function_statement_or_null
    | default [ : ] function_statement_or_null

```

Syntax 9-6—Syntax for case statement

The *default* statement shall be optional. Use of multiple default statements in one case statement shall be illegal.

The case expression and the case item expression can be computed at runtime; neither expression is required to be a constant expression.

Examples:

A simple example of the use of the case statement is the decoding of reg `rega` to produce a value for `result` as follows:

```
reg [15:0] rega;  
reg [9:0] result;  
  
case (rega)  
  16'd0: result = 10'b0111111111;  
  16'd1: result = 10'b1011111111;  
  16'd2: result = 10'b1101111111;  
  16'd3: result = 10'b1110111111;  
  16'd4: result = 10'b1111011111;  
  16'd5: result = 10'b1111101111;  
  16'd6: result = 10'b1111110111;  
  16'd7: result = 10'b1111111011;  
  16'd8: result = 10'b1111111101;  
  16'd9: result = 10'b1111111110;  
  default result = 'bx;  
endcase
```

The *case item expressions* shall be evaluated and compared in the exact order in which they are given. During the linear search, if one of the *case item expressions* matches the case expression given in parentheses, then the statement associated with that case item shall be executed. If all comparisons fail and the default item is given, then the default item statement shall be executed. If the default statement is not given and all of the comparisons fail, then none of the case item statements shall be executed.

Apart from syntax, the *case* statement differs from the multiway if-else-if construct in two important ways:

- a) The conditional expressions in the if-else-if construct are more general than comparing one expression with several others, as in the case statement.
- b) The case statement provides a definitive result when there are *x* and *z* values in an expression.

In a case expression comparison, the comparison only succeeds when each bit matches exactly with respect to the values 0, 1, *x*, and *z*. As a consequence, care is needed in specifying the expressions in the *case* statement. The bit length of all the expressions shall be equal so that exact bit-wise matching can be performed. The length of all the *case item expressions*, as well as the case expression in the parentheses, shall be made equal to the length of the longest case expression and case item expression.

NOTE—The default length of *x* and *z* is same as the default length of an integer.

The reason for providing a case expression comparison that handles the *x* and *z* values is that it provides a mechanism for detecting such values and reducing the pessimism that can be generated by their presence.

Examples:

Example 1—The following example illustrates the use of a case statement to handle *x* and *z* values properly.


```

case (select[1:2])
  2'b00: result = 0;
  2'b01: result = flaga;
  2'b0x,
  2'b0z: result = flaga ? 'bx : 0;
  2'b10: result = flagb;
  2'b1x,
  2'b1z: result = flagb ? 'bx : 0;
  default result = 'bx;
endcase

```

In this example, if `select[1]` is 0 and `flaga` is 0, then whether the value of `select[2]` is x or z, `result` should be 0—which is resolved by the third case.

Example 2—The following example shows another way to use a case statement to detect x and z values.

```

case (sig)
  1'bz: $display("signal is floating");
  1'bx: $display("signal is unknown");
  default: $display("signal is %b", sig);
endcase

```

9.5.1 Case statement with don't-cares

Two other types of case statements are provided to allow handling of don't-care conditions in the case comparisons. One of these treats high-impedance values (z) as don't-cares, and the other treats both high-impedance and unknown (x) values as don't-cares.

These case statements can be used in the same way as the traditional case statement, but they begin with keywords **casez** and **casex** respectively.

Don't-care values (z values for **casez**, z and x values for **casex**) in any bit of either the case expression or the case items shall be treated as don't-care conditions during the comparison, and that bit position shall not be considered. The don't-care conditions in case expression can be used to control dynamically which bits should be compared at any time.

The syntax of literal numbers allows the use of the question mark (?) in place of z in these case statements. This provides a convenient format for specification of don't-care bits in case statements.

Examples:

Example 1—The following is an example of the **casez** statement. It demonstrates an instruction decode, where values of the most significant bits select which task should be called. If the most significant bit of `ir` is a 1, then the task `instruction1` is called, regardless of the values of the other bits of `ir`.

```
reg [7:0] ir;

casez (ir)
  8'b1??????? : instruction1(ir);
  8'b01??????? : instruction2(ir);
  8'b00010??? : instruction3(ir);
  8'b000001?? : instruction4(ir);
endcase
```

Example 2—The following is an example of the casex statement. It demonstrates an extreme case of how don't-care conditions can be dynamically controlled during simulation. In this case, if $r = 8'b01100110$, then the task stat2 is called.

```
reg [7:0] r, mask;

mask = 8'bxx0x0x0x0;
casex (r ^ mask)
  8'b001100xx : stat1;
  8'b1100xx00 : stat2;
  8'b00xx0011 : stat3;
  8'bx010100 : stat4;
endcase
```

9.5.2 Constant expression in case statement

A constant expression can be used for case expression. The value of the constant expression shall be compared against case item expressions.

Example:

The following example demonstrates the usage by modeling a 3-bit priority encoder.

```
reg [2:0] encode ;

case (1)
  encode[2] : $display("Select Line 2") ;
  encode[1] : $display("Select Line 1") ;
  encode[0] : $display("Select Line 0") ;
  default $display("Error: One of the bits expected ON");
endcase
```

Note that the case expression is a constant expression (1). The case items are expressions (bit-selects) and are compared against the constant expression for a match.

9.6 Looping statements

There are four types of looping statements. These statements provide a means of controlling the execution of a statement zero, one, or more times.

<i>forever</i>	Continuously executes a statement.
<i>repeat</i>	Executes a statement a fixed number of times. If the expression evaluates to unknown or high impedance, it shall be treated as zero, and no statement shall be executed.
<i>while</i>	Executes a statement until an expression becomes false. If the expression starts out false, the statement shall not be executed at all.
<i>for</i>	Controls execution of its associated statement(s) by a three-step process, as follows: <ol style="list-style-type: none"> Executes an assignment normally used to initialize a variable that controls the number of loops executed. Evaluates an expression—if the result is zero, the for-loop shall exit, and if it is not zero, the for-loop shall execute its associated statement(s) and then perform step c. If the expression evaluates to an unknown or high-impedance value, it shall be treated as zero. Executes an assignment normally used to modify the value of the loop-control variable, then repeats step b.

Syntax 9-7 shows the syntax for various looping statements.

```

function_loop_statement ::= (From Annex A - A.6.8)
    forever function_statement
    | repeat ( expression ) function_statement
    | while ( expression ) function_statement
    | for ( variable_assignment ; expression ; variable_assignment )
      function_statement
loop_statement ::=
    forever statement
    | repeat ( expression ) statement
    | while ( expression ) statement
    | for ( variable_assignment ; expression ; variable_assignment )
      statement

```

Syntax 9-7—Syntax for the looping statements

The rest of this clause presents examples for three of the looping statements. The forever loop should only be used in conjunction with the timing controls or the disable statement, therefore, this example is presented in 9.7.2.

Examples:

Example 1—Repeat statement: In the following example of a repeat loop, add and shift operators implement a multiplier.

```

parameter size = 8, longsize = 16;
reg [size:1] opa, opb;
reg [longsize:1] result;

begin : mult
    reg [longsize:1] shift_opa, shift_opb;
    shift_opa = opa;
    shift_opb = opb;
    result = 0;
    repeat (size) begin
        if (shift_opb[1])
            result = result + shift_opa;
            shift_opa = shift_opa << 1;
            shift_opb = shift_opb >> 1;
        end
    end

```

Example 2—While statement: The following example counts the number of logic 1 values in rega.

```

begin : count1s
    reg [7:0] tempreg;
    count = 0;
    tempreg = rega;
    while (tempreg) begin
        if (tempreg[0])
            count = count + 1;
            tempreg = tempreg >> 1;
        end
    end

```

Example 3—For statement: The for statement accomplishes the same results as the following pseudo-code that is based on the while loop:

```

begin
    initial_assignment;
    while (condition) begin
        statement
        step_assignment;
    end
end

```

The for loop implements this logic while using only two lines, as shown in the pseudo-code below.

```

for (initial_assignment; condition; step_assignment)
    statement

```

9.7 Procedural timing controls

The Verilog HDL has two types of explicit timing control over when procedural statements can occur. The first type is a *delay control*, in which an expression specifies the time duration between initially encountering the statement and when the statement actually executes. The delay expression can be a dynamic function of the state of the circuit, but it can be a simple number that separates statement executions in time. The delay control is an important feature when specifying stimulus waveform descriptions. It is described in 9.7.1 and 9.7.7.

The second type of timing control is the *event expression*, which allows statement execution to be delayed until the occurrence of some simulation event occurring in a procedure executing concurrently with this procedure. A simulation event can be a change of value on a net or variable (an implicit event) or the occurrence of an explicitly named event that is triggered from other procedures (an explicit event). Most often, an event control is a positive or negative edge on a clock signal. Event control is discussed in 9.7.2 through 9.7.7.

The procedural statements encountered so far all execute without advancing simulation time. Simulation time can advance by one of the following three methods:

- A **delay** control, which is introduced by the symbol #
- An **event** control, which is introduced by the symbol @
- The **wait** statement, which operates like a combination of the event control and the while loop

Syntax 9-8 describes timing control in procedural statements.

```

delay_control ::= (From Annex A - A.6.5)
    # delay_value
    | # ( mintypmax_expression )
delay_or_event_control ::=
    delay_control
    | event_control
    | repeat ( expression ) event_control
event_control ::=
    @ event_identifier
    | @ ( event_expression )
    | @*
    | @ ( * )
event_expression ::=
    expression
    | hierarchical_identifier
    | posedge expression
    | negedge expression
    | event_expression or event_expression
    | event_expression , event_expression

```

Syntax 9-8—Syntax for procedural timing control

The gate and net delays also advance simulation time, as discussed in Section 6. The next subclauses discuss the three procedural timing control methods.

9.7.1 Delay control

A procedural statement following the delay control shall be delayed in its execution with respect to the procedural statement preceding the delay control by the specified delay. If the delay expression evaluates to an unknown or high-impedance value, it shall be interpreted as zero delay. If the delay expression evaluates to a negative value, it shall be

interpreted as a 2's complement unsigned integer of the same size as a time variable. Specify parameters are permitted in the delay expression. They may be overridden by SDF annotation, in which case the expression is reevaluated.

Examples:

Example 1—The following example delays the execution of the assignment by 10 time units:

```
#10 rega = regb;
```

Example 2—The next three examples provide an expression following the number sign (#). Execution of the assignment is delayed by the amount of simulation time specified by the value of the expression.

```
#d rega = regb;           // d is defined as a parameter
#((d+e)/2) rega = regb; // delay is average of d and e
#regr regr = regr + 1;    // delay is the value in regr
```

9.7.2 Event control

The execution of a procedural statement can be synchronized with a value change on a net or variable or the occurrence of a declared event. The value changes on nets and variable can be used as events to trigger the execution of a statement. This is known as detecting *an implicit event*. The event can also be based on the direction of the change—that is, towards the value 1 (**posedge**) or towards the value 0 (**negedge**). The behavior of posedge and negedge event is shown in Table 9-1 and can be described as follows:

- A *negedge* shall be detected on the transition from 1 to x, z, or 0, and from x or z to 0
- A *posedge* shall be detected on the transition from 0 to x, z, or 1, and from x or z to 1

Table 9-1—Detecting posedge and negedge

To From	0	1	x	z
0	No edge	posedge	posedge	posedge
1	negedge	No edge	negedge	negedge
x	negedge	posedge	No edge	No edge
z	negedge	posedge	No edge	No edge

If the expression evaluates to more than a 1-bit result, the edge transition shall be detected on the least significant bit of the result. The change of value in any of the operands without a change in the value of the least significant bit of the expression result shall not be detected as an edge.

Example:

The following example shows illustrations of edge-controlled statements.

```

@r rega = regb; // controlled by any value change in the reg r
@(posedge clock) rega = regb; // controlled by posedge on clock
forever @(negedge clock) rega = regb; // controlled by negative edge

```

9.7.3 Named events

A new data type, in addition to nets and variables, called “event” can be declared. An identifier declared as an event data type is called a *named event*. A named event can be triggered explicitly. It can be used in an event expression to control the execution of procedural statements in the same manner as event control described in 9.7.1. Named events can be made to occur from a procedure. This allows control over the enabling of multiple actions in other procedures.

An event name shall be declared explicitly before it is used. Syntax 9-9 gives the syntax for declaring events.

```

event_declaration ::= (From Annex A - A.2.1.3)
event list_of_event_identifiers ;
list_of_event_identifiers ::= (From Annex A - A.2.3)
    event_identifier [ dimension { dimension } ]
    { , event_identifier [ dimension { dimension } ] }
dimension ::= (From Annex A - A.2.5)
    [ dimension_constant_expression : dimension_constant_expression ]

```

Syntax 9-9—Syntax for event declaration

An event shall not hold any data. The following are the characteristics of a named event:

- It can be made to occur at any particular time
- It has no time duration
- Its occurrence can be recognized by using the event control syntax described in

A declared event is made to occur by the activation of an event triggering statement with the syntax given in Syntax 9-10.

```

event_trigger ::= (From Annex A - A.6.5)
    -> hierarchical_event_identifier ;

```

Syntax 9-10—Syntax for event trigger

An event-controlled statement (for example, @trig rega = regb;) shall cause simulation of its containing procedure to wait until some other procedure executes the appropriate event-triggering statement (for example, -> trig).

Named events and event control give a powerful and efficient means of describing the communication between, and synchronization of, two or more concurrently active processes. A basic example of this is a small waveform clock generator that synchronizes control of a synchronous circuit by signalling the occurrence of an explicit event periodically while the circuit waits for the event to occur.

9.7.4 Event or operator

The logical or of any number of events can be expressed such that the occurrence of any one of the events triggers the execution of the procedural statement that follows it. The keyword **or** or a comma character (,) is used as an event logical or operator. A combination of these can be used in the same event expression. Comma-separated sensitivity lists shall be synonymous to **or**-separated sensitivity lists.

Examples:

The next two examples show the logical or of two and three events respectively.

```
@(trig or enable) rega = regb;           // controlled by trig or enable
@(posedge clk_a or posedge clk_b or trig) rega = regb;
```

The following examples show the use of the comma (,) as an event logical or operator.

```
always @(a, b, c, d, e)
always @(posedge clk, negedge rstn)
always @(a or b, c, d or e)
```

9.7.5 Implicit event_expression list

The event_expression list of an event control is a common source of bugs in RTL simulations. Users tend to forget to add some of the nets or variables read in the timing control statement. This is often found when comparing RTL and gate level versions of a design. The implicit event_expression, @*, is a convenient shorthand that eliminates these problems by adding all nets and variables which are read by the statement (which can be a statement group) of a procedural_timing_control_statement to the event_expression.

All net and variable identifiers which appear in the statement will be automatically added to the event expression with these exceptions:

- Identifiers which only appear in wait or event expressions.
- Identifiers which only appear as a *hierarchical_reg_identifier* in the reg_lvalue of the left hand side of assignments.

Nets and variables which appear on the right hand side of assignments, in function and task calls, or case and conditional expressions shall all be included by these rules.

Examples:

Example 1

```
always @(*) // equivalent to @(a or b or c or d or f)
y = (a & b) | (c & d) | myfunction(f);
```

Example 2

```
always @* begin // equivalent to @(a or b or c or d or tmp1 or tmp2)
    tmp1 = a & b;
    tmp2 = c & d;
    y = tmp1 | tmp2;
end
```


Example 3

```

always @* begin // equivalent to @(b)
    @(i) kid = b; // i is not added to @*
end

```

Example 4

```

always @* begin // equivalent to @(a or b or c or d)
    x = a ^ b;
    @* // equivalent to @(c or d)
    x = c ^ d;
end

```

9.7.6 Level-sensitive event control

The execution of a procedural statement can also be delayed until a condition becomes true. This is accomplished using the *wait* statement, which is a special form of event control. The nature of the wait statement is level-sensitive, as opposed to basic event control (specified by the @ character), which is edge-sensitive.

The wait statement shall evaluate a condition, and, if it is false, the procedural statements following the wait statement shall remain blocked until that condition becomes true before continuing. The wait statement has the form given in Syntax 9-11.

wait_statement ::= (From Annex A - A.6.5)
wait (expression) statement_or_null

*Syntax 9-11—Syntax for wait statement**Example:*

The following example shows the use of the wait statement to accomplish level-sensitive event control.

```

begin
    wait (!enable) #10 a = b;
    #10 c = d;
end

```

If the value of *enable* is 1 when the block is entered, the wait statement will delay the evaluation of the next statement (*#10 a = b;*) until the value of *enable* changes to 0. If *enable* is already 0 when the *begin-end* block is entered, then the assignment “*a = b;*” is evaluated after a delay of 10 and no additional delay occurs.

9.7.7 Intra-assignment timing controls

The delay and event control constructs previously described precede a statement and delay its execution. In contrast, the *intra-assignment delay and event controls* are contained within an assignment statement and modify the flow of activity in a different way. This subclause describes the purpose of intra-assignment timing controls and the repeat timing control that can be used in intra-assignment delays.

An intra-assignment delay or event control shall delay the assignment of the new value to the left-hand side, but the right-hand side expression shall be evaluated before the delay, instead of after the delay. The syntax for intra-assign-

ment delay and event control is given in Syntax 9-12.

```

blocking_assignment ::= (From Annex A - A.6.2)
    variable_lvalue = [ delay_or_event_control ] expression
nonblocking_assignment ::=
    variable_lvalue <= [ delay_or_event_control ] expression
delay_control ::= (From Annex A - A.6.5)
    # delay_value
    | # ( mintypmax_expression )
delay_or_event_control ::=
    delay_control
    | event_control
    | repeat ( expression ) event_control
event_control ::=
    @ event_identifier
    | @ ( event_expression )
    | @*
    | @ ( * )
event_expression ::=
    expression
    | hierarchical_identifier
    | posedge expression
    | negedge expression
    | event_expression or event_expression
    | event_expression , event_expression

```

Syntax 9-12—Syntax for intra-assignment delay and event control

The intra-assignment delay and event control can be applied to both blocking assignments and nonblocking assignments. The event expression shall be resolved to a 1-bit value. The *repeat* event control shall specify an intra-assignment delay of a specified number of occurrences of an event. If the *repeat* count literal, or signed reg holding the *repeat* count, is less than or equal to 0 at the time of evaluation, the assignment occurs as if there is no *repeat* construct.

Examples:

```

repeat (-3) @ (event_expression)
    // will not execute event_expression.

repeat (a) @ (event_expression)
    // if a is assigned -3 it will execute the event_expression
    // if a is declared as an unsigned reg but not if it is signed.

```

This construct is convenient when events have to be synchronized with counts of clock signals.

Examples:

Table 9-2 illustrates the philosophy of intra-assignment timing controls by showing the code that could accomplish

the same timing effect without using intra-assignment.

Table 9-2—Intra-assignment timing control equivalence

Intra-assignment timing control	
With intra-assignment construct	Without intra-assignment construct
<code>a = #5 b;</code>	<code>begin temp = b; #5 a = temp; end</code>
<code>a = @(posedge clk) b;</code>	<code>begin temp = b; @(posedge clk) a = temp; end</code>
<code>a = repeat(3) @(posedge clk) b;</code>	<code>begin temp = b; @(posedge clk); @(posedge clk); @(posedge clk) a = temp; end</code>

The next three examples use the fork-join behavioral construct. All statements between the keywords **fork** and **join** execute concurrently. This construct is described in more detail in 9.8.2.

The following example shows a race condition that could be prevented by using intra-assignment timing control:

```
fork
    #5 a = b;
    #5 b = a;
join
```

The code in this example samples and sets the values of both *a* and *b* at the same simulation time, thereby creating a race condition. The intra-assignment form of timing control used in the next example prevents this race condition.

```
fork                // data swap
    a = #5 b;
    b = #5 a;
join
```

Intra-assignment timing control works because the intra-assignment delay causes the values of *a* and *b* to be evaluated *before* the delay, and the assignments to be made *after* the delay. Some existing tools that implement intra-assignment timing control use temporary storage in evaluating each expression on the right-hand side.

Intra-assignment waiting for *events* is also effective. In the following example, the right-hand side expressions are evaluated when the assignment statements are encountered, but the assignments are delayed until the rising edge of the clock signal.

```
fork                // data shift
    a = @(posedge clk) b;
    b = @(posedge clk) c;
join
```

The following is an example of a repeat event control as the intra-assignment delay of a nonblocking assignment:

```
a <= repeat(5) @(posedge clk) data;
```

Figure 9-1 illustrates the activities that result from this repeat event control.

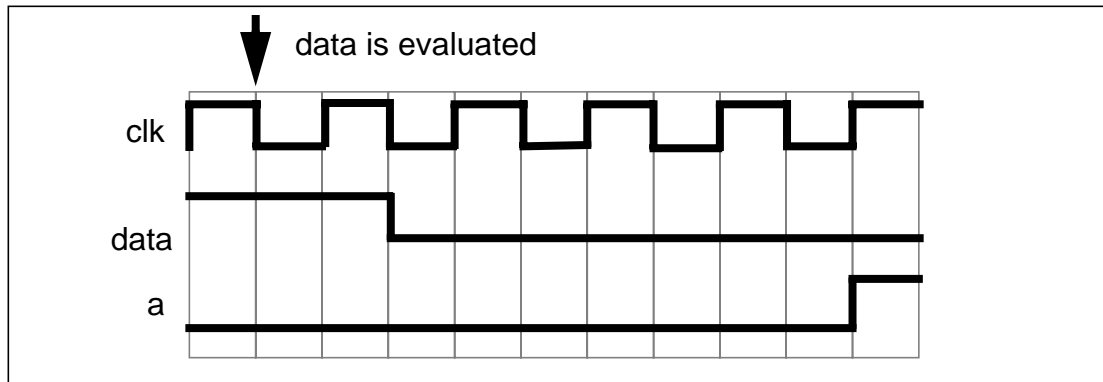


Figure 9-1—Repeat event control utilizing a clock edge

In this example, the value of `data` is evaluated when the assignment is encountered. After five occurrences of **posedge** `clk`, `a` is assigned the value of `data`.

The following is an example of a repeat event control as the intra-assignment delay of a procedural assignment:

```
a = repeat(num) @(clk) data;
```

In this example, the value of `data` is evaluated when the assignment is encountered. After the number of transitions of `clk` equals the value of `num`, `a` is assigned the value of `data`.

The following is an example of a repeat event control with expressions containing operations to specify both the number of event occurrences and the event that is counted:

```
a <= repeat(a+b) @(posedge phi1 or negedge phi2) data;
```

In this example, the value of `data` is evaluated when the assignment is encountered. After the sum of the positive edges of `phi1` and the negative edges of `phi2` equals the sum of `a` and `b`, `a` is assigned the value of `data`. Even if **posedge** `phi1` and **negedge** `phi2` occurred at the same simulation time, each will be detected separately.

9.8 Block statements

The *block statements* are a means of grouping two or more statements together so that they act syntactically like a single statement. There are two types of blocks in the Verilog HDL:

- *Sequential block*, also called *begin-end block*
- *Parallel block*, also called *fork-join block*

The sequential block shall be delimited by the keywords **begin** and **end**. The procedural statements in sequential block shall be executed sequentially in the given order.

The parallel block shall be delimited by the keywords **fork** and **join**. The procedural statements in parallel block shall be executed concurrently.

9.8.1 Sequential blocks

A *sequential block* shall have the following characteristics:

- Statements shall be executed in sequence, one after another
- Delay values for each statement shall be treated relative to the simulation time of the execution of the previous statement
- Control shall pass out of the block after the last statement executes

Syntax 9-13 gives the formal syntax for a sequential block.

```

function_seq_block ::= (From Annex A - A.6.3)
    begin [ : block_identifier
        { block_item_declaration } ] { function_statement } end
seq_block ::=
    begin [ : block_identifier
        { block_item_declaration } ] { statement } end
block_item_declaration ::= (From Annex A - A.2.8)
    { attribute_instance } block_reg_declaration
    | { attribute_instance } event_declaration
    | { attribute_instance } integer_declaration
    | { attribute_instance } local_parameter_declaration
    | { attribute_instance } parameter_declaration
    | { attribute_instance } real_declaration
    | { attribute_instance } realtime_declaration
    | { attribute_instance } time_declaration

```

Syntax 9-13—Syntax for the sequential block

Examples:

Example 1—A sequential block enables the following two assignments to have a deterministic result:

```

begin
    areg = breg;
    creg = areg;      // creg stores the value of breg
end

```

The first assignment is performed and areg is updated before control passes to the second assignment.

Example 2—Delay control can be used in a sequential block to separate the two assignments in time.

```
begin
    areg = breg;
    @(posedge clock) creg = areg; // assignment delayed until
end                                // posedge on clock
```

Example 3—The following example shows how the combination of the sequential block and delay control can be used to specify a time-sequenced waveform.

```
parameter d = 50;    // d declared as a parameter and
reg [7:0] r;         // r declared as an 8-bit reg

begin    // a waveform controlled by sequential delay
    #d r = 'h35;
    #d r = 'hE2;
    #d r = 'h00;
    #d r = 'hF7;
    #d -> end_wave; // trigger an event called end_wave
end
```

9.8.2 Parallel blocks

A *parallel block* shall have the following characteristics:

- Statements shall execute concurrently
- Delay values for each statement shall be considered relative to the simulation time of entering the block
- Delay control can be used to provide time-ordering for assignments
- Control shall pass out of the block when the last time-ordered statement executes

Syntax 9-14 gives the formal syntax for a parallel block.

```
par_block ::= (From Annex A - A.6.3)
    fork [ : block_identifier
        { block_item_declaration } ] { statement } join
block_item_declaration ::= (From Annex A - A.2.8)
    { attribute_instance } block_reg_declaration
    | { attribute_instance } event_declaration
    | { attribute_instance } integer_declaration
    | { attribute_instance } local_parameter_declaration
    | { attribute_instance } parameter_declaration
    | { attribute_instance } real_declaration
    | { attribute_instance } realtime_declaration
    | { attribute_instance } time_declaration
```

Syntax 9-14—Syntax for the parallel block

The timing controls in a fork-join block do not have to be ordered sequentially in time.

Example:

The following example codes the waveform description shown in example 3 of 9.8.1 by using a parallel block instead of a sequential block. The waveform produced on the reg is exactly the same for both implementations.

```
fork
    #50 r = 'h35;
    #100 r = 'hE2;
    #150 r = 'h00;
    #200 r = 'hF7;
    #250 -> end_wave;
join
```

9.8.3 Block names

Both sequential and parallel blocks can be named by adding : name_of_block after the keywords **begin** or **fork**. The naming of blocks serves several purposes:

- It allows local variables, parameters, and named events to be declared for the block.
- It allows the block to be referenced in statements such as the disable statement (Section 11).

All variables shall be static—that is, a unique location exists for all variables and leaving or entering blocks shall not affect the values stored in them.

The block names give a means of uniquely identifying all variables at any simulation time.

9.8.4 Start and finish times

Both sequential and procedural blocks have the notion of a start and finish time. For sequential blocks, the start time is when the first statement is executed, and the finish time is when the last statement has been executed. For parallel blocks, the start time is the same for all the statements, and the finish time is when the last time-ordered statement has been executed.

Sequential and parallel blocks can be embedded within each other, allowing complex control structures to be expressed easily and with a high degree of structure. When blocks are embedded within each other, the timing of when a block starts and finishes is important. Execution shall not continue to the statement following a block until the finish time for the block has been reached—that is, until the block has completely finished executing.

Examples:

Example 1—The following example shows the statements from the example in 9.8.2 written in the reverse order and still producing the same waveform.

```
fork
    #250 -> end_wave;
    #200 r = 'hF7;
    #150 r = 'h00;
    #100 r = 'hE2;
    #50 r = 'h35;
join
```

Example 2—When an assignment is to be made after two separate events have occurred, known as the *joining of events*, a fork-join block can be useful.

```
begin
  fork
    @Aevent;
    @Bevent;
  join
    areg = breg;
end
```

The two events can occur in any order (or even at the same simulation time) and the fork-join block will complete and the assignment will be made. In contrast to this, if the fork-join block was a begin-end block and the Bevent occurred before the Aevent, then the block would be waiting for the next Bevent.

Example 3—This example shows two sequential blocks, each of which will execute when its controlling event occurs. Because the event controls are within a fork-join block, they execute in parallel and the sequential blocks can therefore also execute in parallel.

```
fork
  @enable_a
  begin
    #ta wa = 0;
    #ta wa = 1;
    #ta wa = 0;
  end
  @enable_b
  begin
    #tb wb = 1;
    #tb wb = 0;
    #tb wb = 1;
  end
join
```

9.9 Structured procedures

All procedures in the Verilog HDL are specified within one of the following four statements:

- *initial* construct
- *always* construct
- Task
- Function

The initial and always constructs are enabled at the beginning of a simulation. The initial construct shall execute only once and its activity shall cease when the statement has finished. In contrast, the always construct shall execute repeatedly. Its activity shall cease only when the simulation is terminated. There shall be no implied order of execution between initial and always constructs. The initial constructs need not be scheduled and executed before the always constructs. There shall be no limit to the number of initial and always constructs that can be defined in a module.

Tasks and functions are procedures that are enabled from one or more places in other procedures. Tasks and functions are described in Section 10.

9.9.1 Initial construct

The syntax for the *initial construct* is given in Syntax 9-15.

<pre>initial_construct ::= (From Annex A - A.6.2) initial statement</pre>
--

Syntax 9-15—Syntax for initial construct

Examples:

The following example illustrates use of the initial construct for initialization of variables at the start of simulation.

```
initial begin  
    areg = 0; // initialize a reg  
    for (index = 0; index < size; index = index + 1)  
        memory[index] = 0; //initialize memory word  
end
```

Another typical usage of the initial construct is specification of waveform descriptions that execute once to provide stimulus to the main part of the circuit being simulated.

```
initial begin  
    inputs = 'b000000; //initialize at time zero  
    #10 inputs = 'b011001; // first pattern  
    #10 inputs = 'b011011; // second pattern  
    #10 inputs = 'b011000; // third pattern  
    #10 inputs = 'b001000; // last pattern  
end
```

9.9.2 Always construct

The *always construct* repeats continuously throughout the duration of the simulation. Syntax 9-16 shows the syntax for the always construct.

<pre>always_construct ::= (From Annex A - A.6.2) always statement</pre>
--

Syntax 9-16—Syntax for always construct

The always construct, because of its looping nature, is only useful when used in conjunction with some form of timing control. If an always construct has no control for simulation time to advance, it will create a simulation deadlock condition.

The following code, for example, creates a zero-delay infinite loop.

```
always areg = ~areg;
```

Providing a timing control to the above code creates a potentially useful description as shown in the following:

```
always #half_period areg = ~areg;
```

Section 10

Tasks and functions

Tasks and functions provide the ability to execute common procedures from several different places in a description. They also provide a means of breaking up large procedures into smaller ones to make it easier to read and debug the source descriptions. This section discusses the differences between tasks and functions, describes how to define and invoke tasks and functions, and presents examples of each.

10.1 Distinctions between tasks and functions

The following rules distinguish tasks from functions:

- A function shall execute in one simulation time unit; a task can contain time-controlling statements.
- A function cannot enable a task; a task can enable other tasks and functions.
- A function shall have at least one **input** type argument and shall not have an **output** or **inout** type argument; a task can have zero or more arguments of any type.
- A function shall return a single value; a task shall not return a value.

The purpose of a *function* is to respond to an input value by returning a single value. A *task* can support multiple goals and can calculate multiple result values. However, only the **output** or **inout** type arguments pass result values back from the invocation of a task. A function is used as an operand in an expression; the value of that operand is the value returned by the function.

Example:

Either a task or a *function* can be defined to switch bytes in a 16-bit word. The task would return the switched word in an output argument, so the source code to enable a task called `switch_bytes` could look like the following example:

```
switch_bytes (old_word, new_word);
```

The task `switch_bytes` would take the bytes in `old_word`, reverse their order, and place the reversed bytes in `new_word`.

A word-switching function would return the switched word as the return value of the function. Thus, the function call for the function `switch_bytes` could look like the following example:

```
new_word = switch_bytes (old_word);
```

10.2 Tasks and task enabling

A task shall be enabled from a statement that defines the argument values to be passed to the task and the variables that receive the results. Control shall be passed back to the enabling process after the task has completed. Thus, if a task has timing controls inside it, then the time of enabling a task can be different from the time at which the control is returned. A task can enable other tasks, which in turn can enable still other tasks—with no limit on the number of tasks enabled. Regardless of how many tasks have been enabled, control shall not return until all enabled tasks have completed.

10.2.1 Task declarations

The syntax for defining tasks is given in Syntax 10-1.

```

task_declaration ::= (From Annex A - A.2.7)
    task [ automatic ] task_identifier ;
    { task_item_declaration }
    statement
endtask
| task [ automatic ] task_identifier ( task_port_list ) ;
    { block_item_declaration }
    statement
endtask

task_item_declaration ::=
    block_item_declaration
    | { attribute_instance } input_declaration
    | { attribute_instance } output_declaration
    | { attribute_instance } inout_declaration

task_port_list ::=
    task_port_item { , task_port_item }

task_port_item ::=
    { attribute_instance } input_declaration
    | { attribute_instance } output_declaration
    | { attribute_instance } inout_declaration

block_item_declaration ::= (From Annex A - A.2.8)
    { attribute_instance } block_reg_declaration
    | { attribute_instance } event_declaration
    | { attribute_instance } integer_declaration
    | { attribute_instance } local_parameter_declaration
    | { attribute_instance } parameter_declaration
    | { attribute_instance } real_declaration
    | { attribute_instance } realtime_declaration
    | { attribute_instance } time_declaration

block_reg_declaration ::=
    reg [ signed ] [ range ]
    list_of_block_variable_identifiers ;

list_of_block_variable_identifiers ::=
    block_variable_type { , block_variable_type }

block_variable_type ::=
    variable_identifier
    | variable_identifier dimension { dimension }

```

Syntax 10-1—Syntax for task declaration

There are two alternate task declaration syntaxes.

The first syntax shall begin with the keyword **task**, followed by the optional keyword **automatic**, followed by a name for the task and a semicolon, and ending with the keyword **endtask**. The keyword **automatic** declares an automatic task that is reentrant with all the task declarations allocated dynamically for each concurrent task entry. Task item declarations can specify the following:

- Input arguments
- Output arguments

- Inout arguments
- All data types that can be declared in a procedural block

The second syntax shall begin with the keyword **task**, followed by a name for the task and a parenthesis enclosed *task_port_list*. The *task_port_list* shall consist of zero or more comma separated *task_port_items*. There shall be a semicolon after the close parenthesis. The task body shall follow and then the keyword **endtask**.

In both syntaxes, the declarations have the same syntax as the corresponding declarations in a module definition (see 12.3.3 and 3.2.2). Tasks without the optional keyword **automatic** are static tasks, with all declared items being statically allocated. These items shall be shared across all uses of the task executing concurrently. Task with the optional keyword **automatic** are automatic tasks. All items declared inside automatic tasks are allocated dynamically for each invocation. Automatic task items can not be accessed by hierarchical references. Automatic tasks can be invoked through use of their hierarchical name.

10.2.2 Task enabling and argument passing

The task enabling statement shall pass arguments as a comma-separated list of expressions enclosed in parentheses. The formal syntax of the task enabling statement is given in Syntax 10-2.

```
task_enable ::= (From Annex A - A.6.9)
               hierarchical_task_identifier [ ( expression { , expression } ) ] ;
```

Syntax 10-2—Syntax of the task enabling statement

The list of arguments for a task enabling statement shall be optional. If the list of arguments is provided, the list shall be an ordered list of expressions that has to match the order of the list of arguments in the task definition.

If an argument in the task is declared as an **input**, then the corresponding expression can be any expression. The order of evaluation of the expressions in the argument list is undefined. If the argument is declared as an **output** or an **inout**, then the expression shall be restricted to an expression that is valid on the left-hand side of a procedural assignment (see 9.2). The following items satisfy this requirement:

- **reg**, **integer**, **real**, **realtime**, and **time** variables
- Memory references
- Concatenations of **reg**, **integer**, **real**, **realtime** and **time** variables
- Concatenations of memory references
- Bit-selects and part-selects of **reg**, **integer**, and **time** variables

The execution of the task enabling statement shall pass input values from the expressions listed in the enabling statement to the arguments specified within the task. Execution of the return from the task shall pass values from the task **output** and **inout** type arguments to the corresponding variables in the task enabling statement. All arguments to the task shall be passed by *value* rather than by reference (that is, a *pointer* to the value).

Examples:

Example 1—The following example illustrates the basic structure of a task definition with five arguments.

```

task my_task;
input a, b;
inout c;
output d, e;
begin
    . . .      // statements that perform the work of the task
    . . .
    c = foo1; // the assignments that initialize result regs
    d = foo2;
    e = foo3;
end
endtask

```

Or using the second form of a task declaration, the task could be defined as:

```

task my_task; (input a, b, inout c, output d, e);
begin
    . . .      // statements that perform the work of the task
    . . .
    c = foo1; // the assignments that initialize result regs
    d = foo2;
    e = foo3;
end
endtask

```

The following statement enables the task:

```

my_task (v, w, x, y, z);

```

The task enabling arguments (v, w, x, y, and z) correspond to the arguments (a, b, c, d, and e) defined by the task. At task enabling time, the **input** and **inout** type arguments (a, b, and c) receive the values passed in v, w, and x. Thus, execution of the task enabling call effectively causes the following assignments:

```

a = v;
b = w;
c = x;

```

As part of the processing of the task, the task definition for my_task shall place the computed result values into c, d, and e. When the task completes, the following assignments to return the computed values to the calling process are performed:

```

x = c;
y = d;
z = e;

```

Example 2—The following example illustrates the use of tasks by describing a traffic light sequencer:

```

module traffic_lights;
reg clock, red, amber, green;
parameter on = 1, off = 0, red_tics = 350,
           amber_tics = 30, green_tics = 200;

// initialize colors.
initial red = off;
initial amber = off;
initial green = off;

always begin                                // sequence to control the lights.
    red = on;                                // turn red light on
    light(red, red_tics);                    // and wait.
    green = on;                              // turn green light on
    light(green, green_tics);                // and wait.
    amber = on;                              // turn amber light on
    light(amber, amber_tics);                // and wait.
end

// task to wait for 'tics' positive edge clocks
// before turning 'color' light off.
task light;
output color;
input [31:0] tics;
begin
    repeat (tics) @ (posedge clock);
    color = off;                             // turn light off.
end
endtask

always begin                                // waveform for the clock.
    #100 clock = 0;
    #100 clock = 1;
end
endmodule // traffic_lights.

```

10.2.3 Task memory usage and concurrent activation

A task may be enabled more than once concurrently. All variables of an automatic task shall be replicated on each concurrent task invocation to store state specific to that invocation. All variables of a static task shall be static in that there shall be a single variable corresponding to each declared local variable in a module instance, regardless of the number of concurrent activations of the task. However, static tasks in different instances of a module shall have separate storage from each other.

Variables declared in static tasks shall retain their values between invocations. They shall be initialized to the default initialization value as described in 3.2.2. Variables declared in automatic tasks shall be initialized to the default initialization value whenever execution enters their scope.

Because variables declared in automatic tasks are deallocated at the end of the task invocation, they shall not be used in certain constructs that might refer to them after that point.

- They shall not be assigned values using non-blocking assignments or procedural continuous assignments.
- They shall not be referenced by procedural continuous assignments or procedural force statements.

- They shall not be referenced in intra-assignment event controls of non-blocking assignments.
- They shall not be traced with system tasks such as **\$monitor** and **\$dumpvars**.

10.3 Functions and function calling

The purpose of a function is to return a value that is to be used in an expression. The rest of this clause explains how to define and use functions.

10.3.1 Function declarations

The syntax for defining a function is given in Syntax 10-3.

```

function_declaration ::= (From Annex A - A.2.6)
    function [ automatic ] [ signed ] [ range_or_type ]
        function_identifier ;
        function_item_declaration { function_item_declaration }
        function_statement
    endfunction
| function [ automatic ] [ signed ] [ range_or_type ]
    function_identifier ( function_port_list ) ;
    block_item_declaration { block_item_declaration }
    function_statement
    endfunction

function_item_declaration ::=
    block_item_declaration
| input_declaration

function_port_list ::=
    { attribute_instance } input_declaration { , { attribute_instance }
    input_declaration }

range_or_type ::=
    range | integer | real | realtime | time

block_item_declaration ::= (From Annex A - A.2.8)
    { attribute_instance } block_reg_declaration
| { attribute_instance } event_declaration
| { attribute_instance } integer_declaration
| { attribute_instance } local_parameter_declaration
| { attribute_instance } parameter_declaration
| { attribute_instance } real_declaration
| { attribute_instance } realtime_declaration
| { attribute_instance } time_declaration

block_reg_declaration ::=
    reg [ signed ] [ range ]
        list_of_block_variable_identifiers ;

list_of_block_variable_identifiers ::=
    block_variable_type { , block_variable_type }

block_variable_type ::=
    variable_identifier
| variable_identifier dimension { dimension }

```

Syntax 10-3—Syntax for function declaration

A function definition shall begin with the keyword **function**, followed by the optional keyword **automatic**, followed

by the optional **signed** designator, followed by the range or type of the return value from the function, followed by the name of the function, and then either a semicolon, or a function port list enclosed in parenthesis, and then a semicolon, and then shall end with the keyword **endfunction**. The use of a *range_or_type* shall be optional. A function specified without a range or type defaults to a one bit reg for the return value. If used, *range_or_type* shall specify the return value of the function is a real, an integer, a time, a realtime or a value with a range of [n:m] bits. A function shall have at least one input declared.

The keyword **automatic** declares a recursive function with all the function declarations allocated dynamically for each recursive call. Automatic function items can not be accessed by hierarchical references. Automatic functions can be invoked through the use of their hierarchical name.

Function inputs shall be declared one of two ways. The first method shall have the name of the function followed by a semicolon. After the semicolon one or more input declarations optionally mixed with block item declarations shall follow. After the function item declarations there shall be a behavioral statement and then the endfunction **keyword**.

The second method shall have the name of the function, followed by an open parenthesis, and one or more input declarations, separated by commas. After all the input declarations, there shall be a close parenthesis, and a semicolon. After the semicolon, there shall be zero or more block item declarations, followed by a behavioral statement, and then the endfunction **keyword**.

Example:

The following example defines a function called `getbyte`, using a range specification.

```
function [7:0] getbyte;
input [15:0] address;
begin
    // code to extract low-order byte from addressed word
    . . .
    getbyte = result_expression;
end
endfunction
```

Or using the second form of a function declaration, the function could be defined as:

```
function [7:0] getbyte (input [15:0] address);
begin
    // code to extract low-order byte from addressed word
    . . .
    getbyte = result_expression;
end
endfunction
```

10.3.2 Returning a value from a function

The function definition shall implicitly declare a variable, internal to the function, with the same name as the function. This variable either defaults to a 1-bit reg or is the same type as the type specified in the function declaration. The function definition initializes the return value from the function by assigning the function result to the internal variable with the same name as the function.

It is illegal to declare another object with the same name as the function in the scope where the function is declared. Inside a function, there is an implied variable with the name of the function, which may be used in expressions within the function. It is, therefore, also illegal to declare another object with the same name as the function inside the function scope.

The following line from the example in 10.3.1 illustrates this concept:

```
getbyte = result_expression;
```

10.3.3 Calling a function

A function call is an operand within an expression. The function call has the syntax given in Syntax 10-4.

function_call ::= (From Annex A - A.8.2)
hierarchical_function_identifier { attribute_instance } (expression { , expression })

Syntax 10-4—Syntax for function call

The order of evaluation of the arguments to a function call is undefined.

Example:

The following example creates a word by concatenating the results of two calls to the function `getbyte` (defined in section 10.3.1):

```
word = control ? {getbyte(msbyte), getbyte(lsbyte)}:0;
```

10.3.4 Function rules

Functions are more limited than tasks. The following six rules govern their usage:

- a) A function definition shall not contain any time-controlled statements—that is, any statements introduced with `#`, `@`, or **wait**.
- b) Functions shall not enable tasks.
- c) A function definition shall contain at least one input argument.
- d) A function definition shall not have any argument declared as output or inout.
- e) A function definition shall include an assignment of the function result value to the internal variable that has the same name as the function name.
- f) A function shall not have any non-blocking assignments.

Example:

This example defines a function called `factorial` that returns an integer value. The `factorial` function is called iteratively and the results are printed.

```

module tryfact;

    // define the function
    function automatic integer factorial;
    input [31:0] operand;
    integer i;
    if (operand >= 2)
        factorial = factorial (operand - 1) * operand;
    else
        factorial = 1;
    endfunction

    // test the function
    integer result;
    integer n;
    initial begin
        for (n = 0; n <= 7; n = n+1) begin
            result = factorial(n);
            $display("%0d factorial=%0d", n, result);
        end
    end
endmodule // tryfact

```

The simulation results are as follows:

```

0 factorial=1
1 factorial=1
2 factorial=2
3 factorial=6
4 factorial=24
5 factorial=120
6 factorial=720
7 factorial=5040

```

10.3.5 Use of constant functions

Constant function calls are used to support the building of complex calculations of values at elaboration time (see 12.1.3). A *constant function call* shall be a function invocation of a *constant function* local to the calling module where the arguments to the function are *constant expressions*. *Constant functions* are a subset of normal Verilog functions that shall meet the following constraints:

- They shall contain no hierarchical references.
- Any function invoked within a *constant function* shall be a *constant function* local to the current module. System functions shall not be invoked.
- They shall have no side effects.
- The only system task that may be invoked is \$display, and it shall be ignored when invoked at elaboration time.
- All parameter values used within the function shall be defined before the use of the invoking *constant function call* (i.e. any parameter use in the evaluation of a *constant function call* constitutes a use of that parameter at the site of the original *constant function call*).

- All identifiers which are not parameters or functions shall be declared locally to the current function.
- If they use any parameter value that is affected directly or indirectly by a **defparam** statement (see 12.2.1), the result is undefined. This can produce an error or the constant function can return an indeterminate value.
- They shall not be declared inside a generate scope.
- They shall not themselves use constant functions in any context requiring a constant expression.

Constant function calls are evaluated at elaboration time. Their execution has no effect on the initial values of the variables used either at simulation time or among multiple invocations of a function at elaboration time. In each of these cases, the variables are initialized as they would be for normal simulation.

Example:

This example defines a function called `clogb2` that returns an integer which has the value of the ceiling of the log base 2.

```
module ram_model (address, write, chip_select, data);  
  parameter data_width = 8;  
  parameter ram_depth = 256;  
  localparam adder_width = clogb2(ram_depth);  
  input [adder_width - 1:0] address;  
  input write, chip_select;  
  inout [data_width - 1:0] data;  
  
  //define the clogb2 function  
  function integer clogb2;  
    input depth;  
    integer i,result;  
    begin  
      for (i = 0; 2 ** i < depth; i = i + 1)  
        result = i + 1;  
      clogb2 = result;  
    end  
  endfunction  
  
  reg [data_width - 1:0] data_store[0:ram_depth - 1];  
  //the rest to the ram model
```

An instance of this `ram_model` with parameters assigned:

```
ram_model #(32,421) ram_a0(a_addr,a_wr,a_cs,a_data);
```

Section 11

Disabling of named blocks and tasks

The *disable* statement provides the ability to terminate the activity associated with concurrently active procedures, while maintaining the structured nature of Verilog HDL procedural descriptions. The disable statement gives a mechanism for terminating a task before it executes all its statements, breaking from a looping statement, or skipping statements in order to continue with another iteration of a looping statement. It is useful for handling exception conditions such as hardware interrupts and global resets.

The disable statement has the syntax form shown in Syntax 11-1.

function_call ::= (From Annex A - A.8.2)
hierarchical_function_identifier { attribute_instance } (expression { , expression })

Syntax 11-1—Syntax of disable statement

Either form of disable statement shall terminate the activity of a task or a named block. Execution shall resume at the statement following the block or following the task enabling statement. All activities enabled within the named block or task shall be terminated as well. If task enable statements are nested—that is, one task enables another, and that one enables yet another—then disabling a task within the chain shall disable all tasks downward on the chain. If a task is enabled more than once, then disabling such a task shall disable all activations of the task.

The results of the following activities that may be initiated by a task are not specified if the task is disabled:

- Results of output and inout arguments
- Scheduled, but not executed, nonblocking assignments
- Procedural continuous assignments (**assign** and **force** statements)

The disable statement can be used within blocks and tasks to disable the particular block or task containing the disable statement. The disable statement can be used to disable named blocks within a function, but cannot be used to disable functions. In cases where a disable statement within a function disables a block or a task that called the function, the behavior is undefined. Disabling an automatic task or a block inside an automatic task proceeds as for regular tasks for all concurrent executions of the task.

Examples:

Example 1—This example illustrates how a block disables itself.

```
begin : block_name
    rega = regb;
    disable block_name;
    regc = rega; // this assignment will never execute
end
```

Example 2—This example shows the `disable` statement being used within a named block in a manner similar to a forward *goto*. The next statement executed after the `disable` statement is the one following the named block.

```

begin : block_name
    ...
    ...
    if (a == 0)
        disable block_name;
    ...
end      // end of named block
// continue with code following named block
    ...

```

Example 3—This example shows the `disable` statement being used as an early return from a task. However, a task disabling itself using a `disable` statement is not a short-hand for the *return* statement found in programming languages.

```

task proc_a;
begin
    ...
    ...
    if (a == 0)
        disable proc_a; // return if true
    ...
    ...
end
endtask

```

Example 4—This example shows the `disable` statement being used in an equivalent way to the two statements *continue* and *break* in the C programming language. The example illustrates control code that would allow a named block to execute until a loop counter reaches *n* iterations or until the variable *a* is set to the value of *b*. The named block `break` contains the code that executes until *a* == *b*, at which point the `disable break;` statement terminates execution of that block. The named block `continue` contains the code that executes for each iteration of the `for` loop. Each time this code executes the `disable continue;` statement, the `continue` block terminates and execution passes to the next iteration of the `for` loop. For each iteration of the `continue` block, a set of statements executes if (*a* != 0). Another set of statements executes if (*a* != *b*).

```

begin : break
    for (i = 0; i < n; i = i+1) begin : continue
        @clk
            if (a == 0) // "continue" loop
                disable continue;
            statements
            statements
        @clk
            if (a == b) // "break" from loop
                disable break;
            statements
            statements
    end
end

```

Example 5—This example shows the `disable` statement being used to disable concurrently a sequence of timing controls and the task `action`, when the `reset` event occurs. The example shows a `fork/join` block within which is a named sequential block (`event_expr`) and a `disable` statement that waits for occurrence of the event `reset`. The sequential block and the wait for `reset` execute in parallel. The `event_expr` block waits for one occurrence of event `ev1` and three occurrences of event `trig`. When these four events have happened, plus a delay of `d` time units, the task `action` executes. When the event `reset` occurs, regardless of events within the sequential block, the `fork/join` block terminates—including the task `action`.

```
fork
  begin : event_expr
    @ev1;
    repeat (3) @trig;
    #d action (areg, breg);
  end
  @reset disable event_expr;
join
```

Example 6—The next example is a behavioral description of a retriggerable monostable. The named event `retrig` restarts the monostable time period. If `retrig` continues to occur within 250 time units, then `q` will remain at 1.

```
always begin : monostable
  #250 q = 0;
end

always @retrig begin
  disable monostable;
  q = 1;
end
```


Section 12

Hierarchical structures

The Verilog HDL supports a hierarchical hardware description structure by allowing modules to be embedded within other modules. Higher-level modules create instances of lower-level modules and communicate with them through input, output, and bidirectional ports. These module input/output ports can be scalar or vector.

As an example of a module hierarchy, consider a system consisting of printed circuit boards (PCBs). The system would be represented as the top-level module and would create instances of modules that represent the boards. The board modules would, in turn, create instances of modules that represent ICs, and the ICs could, in turn, create instances of modules such as flip-flops, mux's, and alu's.

To describe a hierarchy of modules, the user provides textual definitions of the various modules. Each module definition stands alone; the definitions are not nested. Statements within the module definitions create instances of other modules, thus describing the hierarchy.

12.1 Modules

This clause gives the formal syntax for a module definition and then gives the syntax for module instantiation, along with an example of a module definition and a module instantiation.

A module definition shall be enclosed between the keywords **module** and **endmodule**. The identifier following the keyword **module** shall be the name of the module being defined. The optional list of parameter definitions shall specify an ordered list of the parameters for the module. The optional list of ports or port declarations shall specify an ordered list of the ports for the module. The order used in defining the list of parameters in the `module_parameter_port_list` and in the list of ports can be significant when instantiating the module (see 12.2.2.1 and 12.3.5). The identifiers in this list shall be declared in input, output, and inout statements within the module definition. Ports declared in the list of port declarations shall not be redeclared within the body of the module. The module items define what constitutes a module and they include many different types of declarations and definitions; many of which have already been introduced.

The keyword **macromodule** can be used interchangeably with the keyword **module** to define a module. An implementation can choose to treat module definitions beginning with **macromodule** keyword differently.

```

module_declaration ::= (From Annex A - A.1.3)
    { attribute_instance } module_keyword module_identifier [ module_parameter_port_list ]
    [ list_of_ports ] ; { module_item }
endmodule
| { attribute_instance } module_keyword module_identifier [ module_parameter_port_list ]
  [ list_of_port_declarations ] ; { non_port_module_item }
endmodule

module_keyword ::= module | macromodule

module_parameter_port_list ::= (From Annex A -A.1.4
    # ( parameter_declaration { , parameter_declaration } )

list_of_ports ::= ( port { , port } )

list_of_port_declarations ::= ( port_declaration { , port_declaration } ) | ( )

port ::= [ port_expression ] . port_identifier ( [ port_expression ] )

port_expression ::= port_reference | { port_reference { , port_reference } }

port_reference ::= port_identifier | port_identifier [ constant_expression ]
    | port_identifier [ range_expression ]

port_declaration ::= { attribute_instance } inout_declaration
    | { attribute_instance } input_declaration
    | { attribute_instance } output_declaration

module_item ::= module_or_generate_item (From Annex A - A.1.5)
    | port_declaration
    | { attribute_instance } generated_instantiation
    | { attribute_instance } local_parameter_declaration
    | { attribute_instance } parameter_declaration
    | { attribute_instance } specify_block
    | { attribute_instance } specparam_declaration

module_or_generate_item ::= { attribute_instance } module_or_generate_item_declaration
    | { attribute_instance } parameter_override
    | { attribute_instance } continuous_assign
    | { attribute_instance } gate_instantiation
    | { attribute_instance } udp_instantiation
    | { attribute_instance } module_instantiation
    | { attribute_instance } initial_construct
    | { attribute_instance } always_construct

module_or_generate_item_declaration ::= net_declaration
    | reg_declaration
    | integer_declaration
    | real_declaration
    | time_declaration
    | realtime_declaration
    | event_declaration
    | genvar_declaration
    | task_declaration
    | function_declaration

non_port_module_item ::= { attribute_instance } generated_instantiation
    | { attribute_instance } local_parameter_declaration
    | { attribute_instance } module_or_generate_item
    | { attribute_instance } parameter_declaration
    | { attribute_instance } specify_block
    | { attribute_instance } specparam_declaration

parameter_override ::= defparam list_of_param_assignments ;

```

Syntax 12-1—Syntax for module

See 12.3 for the definitions of ports.

12.1.1 Top-level modules

Top-level modules are modules that are included in the source text but are not instantiated, as described in 12.1.2.

12.1.2 Module instantiation

Instantiation allows one module to incorporate a copy of another module into itself. Module definitions do not nest. That is, one module definition shall not contain the text of another module definition within its **module-endmodule** keyword pair. A module definition nests another module by *instantiating* it. The *module instantiation statement* creates one or more named *instances* of a defined module.

For example, a counter module might instantiate a D flip-flop module to create multiple instances of the flip-flop.

Syntax 12-2 gives the syntax for specifying instantiations of modules.

```

module_instantiation ::= (From Annex A - A.4.1)
    module_identifier [ parameter_value_assignment ]
        module_instance { , module_instance } ;
parameter_value_assignment ::=
    # ( list_of_parameter_assignments )
list_of_parameter_assignments ::=
    ordered_parameter_assignment { , ordered_parameter_assignment }
    | named_parameter_assignment { , named_parameter_assignment }
ordered_parameter_assignment ::=
    expression
named_parameter_assignment ::=
    . parameter_identifier ( [ expression ] )
module_instance ::=
    name_of_instance ( [ list_of_port_connections ] )
name_of_instance ::=
    module_instance_identifier [ range ]
list_of_port_connections ::=
    ordered_port_connection { , ordered_port_connection }
    | named_port_connection { , named_port_connection }
ordered_port_connection ::=
    { attribute_instance } [ expression ]
named_port_connection ::=
    { attribute_instance } . port_identifier ( [ expression ] )

```

Syntax 12-2—Syntax for module instantiation

The instantiations of modules can contain a range specification. This allows an array of instances to be created. The array of instances are described in 7.1. The syntax and semantics of arrays of instances defined for gates and primitives apply for modules as well.

One or more module instances (identical copies of a module) can be specified in a single module instantiation statement.

The list of port connections shall be provided only for modules defined with ports. The parentheses, however, are always required. When a list of port connections is given using the ordered port connection method, the first element

in the list shall connect to the first port declared in the module, the second to the second port, and so on. See 12.3 for a more detailed discussion of ports and port connection rules.

A connection can be a simple reference to a variable or a net identifier, an expression, or a blank. An expression can be used for supplying a value to a module input port. A blank port connection shall represent the situation where the port is not to be connected.

When connecting ports by name, an unconnected port can be indicated either by omitting it in the port list, or by providing no expression in the parentheses [i.e., `.port_name ()`].

Examples:

Example 1—The following example illustrates a circuit (the lower-level module) being driven by a simple waveform description (the higher-level module) where the circuit module is instantiated inside the waveform module.

```
// Lower level module:
// module description of a nand flip-flop circuit
module ffnand (q, qbar, preset, clear);
output q, qbar;           //declares 2 circuit output nets
input preset, clear;      //declares 2 circuit input nets

// declaration of two nand gates and their interconnections
nand g1 (q, qbar, preset),
      g2 (qbar, q, clear);
endmodule

// Higher-level module:
// a waveform description for the nand flip-flop
module ffnand_wave;
wire out1, out2;         //outputs from the circuit
reg in1, in2;            //variables to drive the circuit
parameter d = 10;

// instantiate the circuit ffnand, name it "ff",
// and specify the IO port interconnections
ffnand ff(out1, out2, in1, in2);

// define the waveform to stimulate the circuit
initial begin
    #d in1 = 0; in2 = 1;
    #d in1 = 1;
    #d in2 = 0;
    #d in2 = 1;
end
endmodule
```

Example 2—The following example creates two instances of the flip-flop module `ffnand` defined in example 1. It connects only to the `q` output in one instance and only to the `qbar` output in the other instance.

```

// a waveform description for testing
// the nand flip-flop, without the output ports
module ffnand_wave;
reg in1, in2; //variables to drive the circuit
parameter d = 10;
// make two copies of the circuit ffnand
// ff1 has qbar unconnected, ff2 has q unconnected
ffnand ff1(out1, , in1, in2),
        ff2(.qbar(out2), .clear(in2), .preset(in1), .q());
// ff3(.q(out3),.clear(in1),,,); is illegal

// define the waveform to stimulate the circuit
initial begin
    #d in1 = 0; in2 = 1;
    #d in1 = 1;
    #d in2 = 0;
    #d in2 = 1;
end
endmodule

```

12.1.3 Generated instantiation

After a Verilog design has been parsed, but before simulation begins, the design must have the modules being instantiated linked to the modules being defined, the parameters propagated among the various modules, and hierarchical references resolved. This phase in understanding a Verilog description is termed elaboration.

Generate instantiations are resolved during elaboration because that is when the parameters associated with a module become defined, hence, allowing the definition of the generated statements and declarations. Genvars are only defined during the evaluation of the generate instantiations and do not exist during simulation of a design.

Generate statements facilitate the creation of parameterized models. When used with constant functions (see 10.3.5), parameters can be used to constrain other parameter(s) or localparam(s) in a generated design.

All generate instantiations are coded within a module scope and require the keywords **generate** - **endgenerate**.

Generate statements allow control over the declaration of variables, functions and tasks, as well as control over instantiations. Generated instantiations are one or more: modules, user defined primitives, Verilog gate primitives, continuous assignments, initial blocks and always blocks. Generated declarations and instantiations can be conditionally instantiated into a design. Generated variable declarations and instantiations can be multiply instantiated into a design. Generated instances have unique identifier names and can be referenced hierarchically as described in 12.4.

To support the interconnection between structural elements and/or procedural blocks, generate statements permit the following Verilog data types to be declared within the generate scope: **net**, **reg**, **integer**, **real**, **time**, **realtime**, and **event**. Generated data types have unique identifier names and can be referenced hierarchically as described in 12.4.

Parameter redefinition using by the ordered or named `parameter = value` assignment or `defparam` statements can also be declared within the generate scope. However, a `defparam` statement within the generate scope or within a hierarchy instantiated within the generate scope shall only modify the value of a parameter declared within the generate scope or within a hierarchy instantiated within the generate scope.

Tasks and functions declarations shall also be permitted within the generate scope, however not in a generate loop. Generated tasks and functions shall have unique identifier names and may be referenced hierarchically as described in 12.4.

Module declarations and module items that shall not be permitted in a generate statement include: parameters, local parameters, input declarations, output declarations, inout declarations and specify blocks.

Connections to generated module instances are handled the same way as they are handled with normal module instances as described in 12.1.2.

Generated statements are created using one of the following three methods: generate-loop, generate-conditional, or generate-case.

The syntax for generate instantiations is given in Syntax 12-3.

```

module_item ::= (From Annex A - A.1.5)
    module_or_generate_item
    | port_declaration
    | { attribute_instance } generated_instantiation
    | { attribute_instance } local_parameter_declaration
    | { attribute_instance } parameter_declaration
    | { attribute_instance } specify_block
    | { attribute_instance } specparam_declaration
module_or_generate_item ::=
    { attribute_instance } module_or_generate_item_declaration
    | { attribute_instance } parameter_override
    | { attribute_instance } continuous_assign
    | { attribute_instance } gate_instantiation
    | { attribute_instance } udp_instantiation
    | { attribute_instance } module_instantiation
    | { attribute_instance } initial_construct
    | { attribute_instance } always_construct
module_or_generate_item_declaration ::=
    net_declaration
    | reg_declaration
    | integer_declaration
    | real_declaration
    | time_declaration
    | realtime_declaration
    | event_declaration
    | genvar_declaration
    | task_declaration
    | function_declaration
generated_instantiation ::= (From Annex A -A.4.2)
    generate { generate_item } endgenerate
generate_item_or_null ::=
    generate_item | ;
generate_item ::=
    generate_conditional_statement
    | generate_case_statement
    | generate_loop_statement
    | generate_block
    | module_or_generate_item
generate_conditional_statement ::=
    if ( constant_expression ) generate_item_or_null [ else generate_item_or_null ]
generate_case_statement ::= case ( constant_expression )
    genvar_case_item { genvar_case_item } endcase
genvar_case_item ::= constant_expression { , constant_expression } :
    generate_item_or_null | default [ : ] generate_item_or_null
generate_loop_statement ::=
    for ( genvar_assignment ; constant_expression ; genvar_assignment )
    begin : generate_block_identifier { generate_item } end
genvar_assignment ::=
    genvar_identifier = constant_expression
generate_block ::=
    begin [ : generate_block_identifier ] { generate_item } end

```

Syntax 12-3—Syntax for generate blocks

12.1.3.1 genvar - generate statement index variable

An index variable that shall only be declared for use in generate statements shall be declared as a *genvar* and is referred to as a *genvar* in the rest of this section.

The syntax for generate statement index variable declarations is given in Syntax 12-4.

```
genvar_declaration ::= (From Annex A - A.2.1.3  
    genvar list_of_genvar_identifiers ;  
list_of_genvar_identifiers ::= (From Annex A - A.2.3)  
    genvar_identifier { , genvar_identifier }
```

Syntax 12-4—Syntax for generate statement index variable declaration

A *genvar* shall be declared within the module where the *genvar* is used. A *genvar* can be declared either inside or outside of a generate scope. A *genvar* is a positive integer that is local to and shall only be used within a generate loop that uses it as an index variable.

Genvars are only defined during the evaluation of the generate blocks (see 12.1.3), and do not exist during simulation of a Verilog design.

The value of a *genvar* shall only be defined by a generate loop. Two generate loops using the same *genvar* as an index variable shall not be nested. The value of a *genvar* can be referenced in any context where the value of a parameter could be referenced.

12.1.3.2 generate-loop

A generate-loop permits one or more variable declarations, modules, user defined primitives, gate primitives, continuous assignments, initial blocks and always blocks to be instantiated multiple times using a for-loop. The index loop variable used in a generate for-loop shall be declared as a *genvar*. Both *genvar* assignments in the for-loop shall assign to the same *genvar*, which is the loop index variable.

Examples:

Example 1—A parameterized gray-code to binary-code converter module using a loop to generate continuous assignments

```
module gray2bin1 (bin, gray);  
    parameter SIZE = 8;          // this module is parameterizable  
    output [SIZE-1:0] bin;  
    input [SIZE-1:0] gray;  
  
    genvar i;  
  
    generate for (i=0; i<SIZE; i=i+1) begin:bit  
        assign bin[i] = ^gray[SIZE-1:i];  
    end endgenerate  
endmodule
```


Example 2—The same gray-code to binary-code converter module in example 1 is built using a loop to generate always blocks

```
module gray2bin2 (bin, gray);  
  parameter SIZE = 8;          // this module is parameterizable  
  output [SIZE-1:0] bin;  
  input  [SIZE-1:0] gray;  
  reg    [SIZE-1:0] bin;  
  
  genvar i;  
  
  generate for (i=0; i<SIZE; i=i+1) begin:bit  
    always @(gray[SIZE-1:i]) // fixed part select  
      bin[i] = ^gray[SIZE-1:i];  
  end endgenerate  
endmodule
```

The models in examples 3 and 4 are parameterized modules of ripple adders using a loop to generate Verilog gate primitives. Example 3 uses a two dimensional net declaration outside of the generate loop to make the connections between the gate primitives while example 4 makes the net declaration inside of the generate loop to generate the wires needed to connect the gate primitives for each iteration of the loop.

Example 3—Generated ripple adder with two-dimensional net declaration outside of the generate loop

```

module addergen1 (co, sum, a, b, ci);
  parameter SIZE = 4;
  output [SIZE-1:0] sum;
  output co;
  input [SIZE-1:0] a, b;
  input ci;
  wire [SIZE :0] c;
  wire [SIZE-1:0] t [1:3];
  genvar i;

  assign c[0] = ci;

  // Generated instance names are:
  // xor gates: bit[0].g1 bit[1].g1 bit[2].g1 bit[3].g1
  //             bit[0].g2 bit[1].g2 bit[2].g2 bit[3].g2
  // and gates: bit[0].g3 bit[1].g3 bit[2].g3 bit[3].g3
  //             bit[0].g4 bit[1].g4 bit[2].g4 bit[3].g4
  // or  gates: bit[0].g5 bit[1].g5 bit[2].g5 bit[3].g5
  // Generated instances are connected with
  // multi-dimensional nets t[1][3:0] t[2][3:0] t[3][3:0]
  // (12 multi-dimensional nets total)
  generate
    for(i=0; i<SIZE; i=i+1) begin:bit
      xor g1 ( t[1][i], a[i], b[i]);
      xor g2 ( sum[i], t[1][i], c[i]);
      and g3 ( t[2][i], a[i], b[i]);
      and g4 ( t[3][i], t[1][i], c[i]);
      or g5 ( c[i+1], t[2][i], t[3][i]);
    end
  endgenerate

  assign co = c[SIZE];
endmodule

```

Example 4—Generated ripple adder with net declaration inside of the generate loop

```

module addergen1 (co, sum, a, b, ci);
  parameter SIZE = 4;
  output [SIZE-1:0] sum;
  output          co;
  input  [SIZE-1:0] a, b;
  input          ci;
  wire    [SIZE :0] c;

  genvar          i;

  assign c[0] = ci;

  // Generated instance names are:
  // xor gates: bit[0].g1 bit[1].g1 bit[2].g1 bit[3].g1
  //             bit[0].g2 bit[1].g2 bit[2].g2 bit[3].g2
  // and gates: bit[0].g3 bit[1].g3 bit[2].g3 bit[3].g3
  //             bit[0].g4 bit[1].g4 bit[2].g4 bit[3].g4
  // or  gates: bit[0].g5 bit[1].g5 bit[2].g5 bit[3].g5
  // Generated instances are connected with
  // generated nets: bit[0].t1 bit[1].t1 bit[2].t1 bit[3].t1
  //                  bit[0].t2 bit[1].t2 bit[2].t2 bit[3].t2
  //                  bit[0].t3 bit[1].t3 bit[2].t3 bit[3].t3
  generate
    for(i=0; i<SIZE; i=i+1) begin:bit
      wire  t1, t2, t3; // generated net declaration

      xor g1 (      t1, a[i], b[i]);
      xor g2 ( sum[i],  t1, c[i]);
      and g3 (      t2, a[i], b[i]);
      and g4 (      t3,  t1, c[i]);
      or  g5 ( c[i+1],  t2,  t3);
    end
  endgenerate

  assign co = c[SIZE];
endmodule

```

The generated instance names in a multi-level generate loop are shown in example 5. The generated name for the scope at each generate loop is created by adding the "[genvar's value]" string to the end of the generate block identifier for the loop. The generated names are now generated identifiers (see 2.7.2) which can be used in hierarchical path names (see 12.4).

Example 5—A multi-level generate loop

```

parameter SIZE = 2;
genvar      i, j, k, m;
generate
  for (i=0; i<SIZE+1; i=i+1) begin:B1 // scope B1[i]
    M1 N1(); // instantiates B1[i].N1[i]
    for (j=0; j<SIZE; j=j+1) begin:B2 // scope B1[i].B2[j]
      M2 N2(); // instantiates B1[i].B2[j].N2
      for (k=0; k<SIZE; k=k+1) begin:B3 // scope B1[i].B2[j].B3[k]
        M3 N3(); // instantiates B1[i].B2[j].B3[k].N3
      end
    end
    if (i>0)
      for (m=0; m<SIZE; m=m+1) begin:B4 // scope B1[i].B4[m]
        M4 N4(); // instantiates B1[i].B4[m].N4
      end
    end
endgenerate

// some of the generated instance names are:
// B1[0].N1  B1[1].N1
// B1[0].B2[0].N2  B1[0].B2[1].N2
// B1[0].B2[0].B3[0].N3  B1[0].B2[0].B3[1].N3
//   B1[0].B2[1].B3[0].N3
// B1[1].B4[0].N4  B1[1].B4[1].N4

```

12.1.3.3 generate-conditional

A generate-conditional is an if-else-if generate construct that permits modules, user defined primitives, Verilog gate primitives, continuous assignments, initial blocks and always blocks to be conditionally instantiated into another module based on an expression that is deterministic at the time the design is elaborated.

Example 6 shows the implementation of a parameterized module. If either of the multiplier's `a_width` or `b_width` parameters are less than 8 (bits), a CLA multiplier is instantiated. If both of the multiplier's `a_width` or `b_width` parameters are greater than or equal to 8 (bits), a Wallace tree multiplier is instantiated.

Example 6—An implementation of a parameterized multiplier module

```

module multiplier(a,b,product);
parameter a_width = 8, b_width = 8;
localparam product_width = a_width+b_width; // can not be modified
// directly with the defparam statement
// or the module instance statement #
input    [a_width-1:0]    a;
input    [b_width-1:0]    b;
output   [product_width-1:0] product;

generate
  if((a_width < 8) || (b_width < 8))
    CLA_multiplier #(a_width,b_width) u1(a, b, product);
    // instance a CLA multiplier
  else
    WALLACE_multiplier #(a_width,b_width) u1(a, b, product);
    // instance a Wallace-tree multiplier
endgenerate
// The generated instance name is u1

endmodule

```

12.1.3.4 generate-case

A generate case construct permits modules, user defined primitives, Verilog gate primitives, continuous assignments, initial blocks and always blocks to be conditionally instantiated into another module based on a select one-of-many case construct. The selecting case expression must be deterministic at the time the design is elaborated.

Example 7—Generate with a case to handle widths less than 3

```

generate
  case (WIDTH)
    1: adder_1bit x1(co, sum, a, b, ci);
    // 1-bit adder implementation
    2: adder_2bit x1(co, sum, a, b, ci);
    // 2-bit adder implementation
    default: adder_cla #(WIDTH) x1(co, sum, a, b, ci);
    // others - carry look-ahead adder
  endcase
// The generated instance name is x1

endgenerate

```

Example 8—A module of memory dimm

```

module dimm;
  parameter [31:0] MEM_SIZE = 8, // in mbytes
              MEM_WIDTH = 16;

  input [11:0] adr;
  input [1:0] ba;
  input      rasx, casx, csx, wex;
  input [7:0] dqm;
  input      cke;
  input [7:0] ds;
  inout [63:0] data;
  input [3:0] clk;

  wire      rasb, casb, csb, web;
  wire [7:0] bex;

  genvar i;

  generate
    case ({MEM_SIZE, MEM_WIDTH})
      {32'd8, 32'd16}: // 8Meg 16 bits wide.
        begin
          for (i=0;i<4;i = i + 1)
            begin:word
              sms_16b216t0 p
                (.clk(clk), .csb(csx), .cke(cke), .ba(ba[0]),
                 .addr(adr[10:0]),...rasb(rasx), .casb(casx),
                 .web(wex), .udqm(dqm[2*i+1]), .ldqm(dqm[2*i]),
                 ...dqi(data[15+16*i:16*i]), .dev_id(dev_id3[4:0])
                );
            end
          task read_mem;
            input [31:0] address;
            output [63:0] data;
            begin
              word[3].p.read_mem(address, data[63:48]);
              word[2].p.read_mem(address, data[47:32]);
              word[1].p.read_mem(address, data[31:16]);
              word[0].p.read_mem(address, data[15:0]);
            end
          endtask
        end
  end

```

```

// The generated instance names are word[3].p, word[2].p,
// word[1].p, word[0].p, and the task read_mem
{32'd16, 32'd8}: // 16Meg 8 bits wide.
    begin
        for (i=0;i<4;i = i + 1)
            begin:byte
                sms_16b208t0 p
                (.clk(clk), .csb(csx), .cke(cke), .ba(ba[0]),
                 .addr(adr[10:0]),
                 ...rasb(rasx), .casb(casx), .web(wex), .dqm(dqm[i]),
                 .dqi(data[8+8*i:8*i]),...dev_id(dev_id7[4:0])
                );
            end
        task read_mem;
            input [31:0] address;
            output [63:0] data;
            begin
                byte[7].p.read_mem(address, data[63:56]);
                byte[6].p.read_mem(address, data[55:48]);
                byte[5].p.read_mem(address, data[47:40]);
                byte[4].p.read_mem(address, data[39:32]);
                byte[3].p.read_mem(address, data[31:24]);
                byte[2].p.read_mem(address, data[23:16]);
                byte[1].p.read_mem(address, data[15:8]);
                byte[0].p.read_mem(address, data[7:0]);
            end
        endtask
    .....
endcase
endgenerate
// The generated instance names are byte[7].p, byte[6].p,
// byte[5].p, byte[4].p, byte[3].p, byte[2].p, byte[1].p,
// byte[0].p and the task read_mem

endmodule

```

12.2 Overriding module parameter values

There are two different ways that parameters can be defined. This first in the *module_parameter_port_list* (see 12.1), and the second is as a *module_item* (see 3.11). A module declaration can contain parameter definitions of either or both types, or no parameter definitions.

A module parameter can have a type specification and a range specification. The effect of parameter overrides on a parameter's type and range shall be in accordance with the following rules:

- A parameter declaration with no type or range specification shall default to the type and range of the final override value assigned to the parameter.
- A parameter with a range specification, but with no type specification, shall be the range of the parameter declaration and shall be unsigned. An override value shall be converted to the type and range of the parameter.
- A parameter with a type specification, but with no range specification, shall be of the type specified. An override value shall be converted to the type of the parameter. A signed parameter shall default to the range of the final override value assigned to the parameter.

- A parameter with a signed type specification and with a range specification shall be a signed, and shall be the range of its declaration. An override value shall be converted to the type and range of the parameter.

Examples:

```
module generic_fifo
  #(parameter MSB=3, LSB=0, DEPTH=4) // These parameters can be overridden
  (
    input [MSB:LSB] in,
    input clk, read, write, reset,
    output [MSB:LSB] out,
    output full, empty
  );

  localparam FIFO_MSB = DEPTH*MSB; // These parameters are local, and
  localparam FIFO_LSB = LSB;       // cannot be overridden. They can be
                                   // affected by altering the public
                                   // parameters above, and the module
                                   // will work correctly.

  reg [FIFO_MSB:FIFO_LSB] fifo;
  reg [LOG2(DEPTH):0] depth;

  always @(posedge clk or reset) begin
    casex ({read,write,reset})
      // implementation of fifo
    endcase
  end
endmodule
```

There are two ways to alter non-local parameter values: the *defparam statement*, which allows assignment to parameters using their hierarchical names, and the *module instance parameter value assignment*, which allows values to be assigned inline during module instantiation. If a defparam assignment conflicts with a module instance parameter, the parameter in the module will take the value specified by the defparam. The module instance parameter value assignment comes in two forms, by ordered list or by name. The next two subclauses describe these two methods.

There are two kinds of parameter declarations. The first kind of parameter declaration has a type and or range qualification, and second does not. When an untyped and unranged parameter's value is overridden, the parameter takes on the size and type of the override.

When a typed and/or ranged parameter is overridden, the new value is converted to the type and size of the destination, and assigned to that parameter.

Example:

```
module foo(a,b);
  real r1,r2;
  parameter [2:0] A = 3'h2;
  parameter B = 3'h2;
  initial begin
    r1 = A;
    r2 = B;
    $display("r1 is %f r2 is %f",r1,r2);
  end
endmodule // foo
```



```
module bar;  
  wire a,b;  
  defparam f1.A = 3.1415;  
  defparam f1.B = 3.1415;  
  foo f1(a,b);  
endmodule // bar
```

Parameter A is a typed and/or ranged parameter, so when its value is redefined, the parameter retains its original type and sign. Therefore, the defparam of f1.A with the value 3.1415 is performed by converting the floating point number 3.1415 into a fixed point number '3' and then the low 3 bits of 3 are assigned to A.

Parameter B is not a typed and/or ranged parameter, so when its value is redefined, the parameter type and range take on the type and range of the new value. Therefore, the defparam of f1.B with the value 3.1415 replaces B's current value of 3'h2 with the floating point number 3.1415.

12.2.1 defparam statement

Using the *defparam statement*, parameter values can be changed in any module instance throughout the design using the hierarchical name of the parameter. However, a defparam statement in a hierarchy under a generate scope or array of instances shall not change a parameter value outside that hierarchy. See 12.4 for hierarchical names.

The expression on the right-hand side of the defparam assignments shall be a constant expression involving only numbers and references to parameters. The referenced parameters (on the right-hand side of the **defparam**) shall be declared in the same module as the defparam statement.

The defparam statement is particularly useful for grouping all of the parameter value override assignments together in one module.

In the case of multiple defparams for a single parameter, the parameter takes the value of the last defparam statement encountered in the source text. When defparams are encountered in multiple source files, e.g., found by library searching, the defparam from which the parameter takes its value is undefined.

Example:

```
module top;
  reg clk;
  reg [0:4] in1;
  reg [0:9] in2;
  wire [0:4] o1;
  wire [0:9] o2;

  vdff m1 (o1, in1, clk);
  vdff m2 (o2, in2, clk);
endmodule

module vdff (out, in, clk);
  parameter size = 1, delay = 1;
  input [0:size-1] in;
  input clk;
  output [0:size-1] out;
  reg [0:size-1] out;

  always @(posedge clk)
    # delay out = in;
endmodule

module annotate;
  defparam
    top.m1.size = 5,
    top.m1.delay = 10,
    top.m2.size = 10,
    top.m2.delay = 20;
endmodule
```

The module `annotate` has the **defparam** statement which overrides `size` and `delay` parameter values for instances `m1` and `m2` in the top-level module `top`. The modules `top` and `annotate` would both be considered top-level modules.

12.2.2 Module instance parameter value assignment

An alternative method for assigning values to parameters within module instances is to use one of the two forms of module instance parameter value assignment. They are assignment by ordered list and assignment by name. The two types of module instance parameter value assignment shall not be mixed; parameter assignments to a particular module instance shall be entirely by order or entirely by name.

Module instance parameter value assignment by ordered list is similar in appearance to the assignment of delay values to gate instances and assignment by name is similar to connecting module ports by name. It supplies values for particular instances of a module to any parameters that have been specified in the definition of that module.

12.2.2.1 Parameter value assignment by ordered list

The order of the assignments in the module instance parameter value assignment by ordered list shall follow the order of declaration of the parameters within the module. It is not necessary to assign values to all of the parameters within a module when using this method. However, it is not possible to skip over a parameter. Therefore, to assign values to a subset of the parameters declared within a module, the declarations of the parameters that make up this subset shall

precede the declarations of the remaining parameters. An alternative is to assign values to all of the parameters, but to use the default value (the same value assigned in the declaration of the parameter within the module definition) for those parameters that do not need new values.

Example:

Consider the following example, where the parameters within module instance `mod_a` are changed during instantiation.

```

module m;
reg clk;
wire [0:4] out_c, in_c;
wire [1:10] out_a, in_a;
wire [1:5] out_b, in_b;

    // create an instance and set parameters
    vdff #(10,15) mod_a(out_a, in_a, clk);
    // create an instance leaving default values
    vdff mod_b(out_b, in_b, clk);
    // create an instance and set one parameter
    vdff #(.delay(12)) mod_c(out_c, in_c, clk);
endmodule

module vdff (out, in, clk);
parameter size = 5, delay = 1;
input [0:size-1] in;
input clk;
output [0:size-1] out;
reg [0:size-1] out;

    always @(posedge clk)
        # delay out = in;
endmodule

```

In this example, the name of the module being instantiated is `vdff`. The construct `#(10,15)` assigns values to parameters used in the `mod_a` instance of `vdff`. The parameter `size` is assigned the value 10 and the parameter `delay` is assigned the value 15 for the instance of module `vdff` called `mod_a`. The construct `#(.delay(12))` assigns the parameter `delay` the value 12 in the instance of module `vdff` called `mod_c`.

12.2.2.2 Parameter value assignment by name

Parameter assignment by name consists of explicitly linking the parameter name and its new value. The name of the parameter shall be the name specified in the instantiated module.

It is not necessary to assign values to all of the parameters within a module when using this method. Only those parameters that are assigned new values need to be specified.

The parameter expression is optional so that the instantiating module can document the existence of a parameter without assigning anything to it. The parentheses are required and in this case the parameter retains its default value. Once a parameter is assigned a value, there shall not be another assignment to this parameter name.

12.2.3 Parameter dependence

A parameter (for example, `memory_size`) can be defined with an expression containing another parameter (for example, `word_size`). Since `memory_size` depends on the value of `word_size`, a modification of `word_size` changes the value of `memory_size`. For example, in the following parameter declaration, an update of `word_size`, whether by `defparam` statement or in an instantiation statement for the module that defined these parameters, automatically updates `memory_size`.

```
parameter
    word_size = 32,
    memory_size = word_size * 4096;
```

12.3 Ports

Ports provide a means of interconnecting a hardware description consisting of modules, primitives, and macromodules. For example, module A can instantiate module B, using port connections appropriate to module A. These port names can differ from the names of the internal nets and variables specified in the definition of module B.

12.3.1 Port definition

The syntax for ports and a list of ports is given in Syntax 12-5.

```
list_of_ports ::= (From Annex A - A.1.4)
    ( port { , port } )
list_of_port_declarations ::=
    ( port_declaration { , port_declaration } )
    | ( )
port ::=
    [ port_expression ]
    . port_identifier ( [ port_expression ] )
port_expression ::=
    port_reference
    | { port_reference { , port_reference } }
port_reference ::=
    port_identifier
    | port_identifier [ constant_expression ]
    | port_identifier [ range_expression ]
port_declaration ::=
    { attribute_instance } inout_declaration
    | { attribute_instance } input_declaration
    | { attribute_instance } output_declaration
```

Syntax 12-5—Syntax for port

12.3.2 List of ports

The port reference for each port in the list of ports at the top of each module declaration can be one of the following:

- A simple identifier or escaped identifier
- A bit-select of a vector declared within the module
- A part-select of a vector declared within the module
- A concatenation of any of the above

The port expression is optional because ports can be defined that do not connect to anything internal to the module. Once a port has been defined, there shall not be another port definition with this same name.

The first type of module port with only a `port_expression` is an implicit port. The second type is the explicit port. This explicitly specifies the `port_identifier` used for connecting module instance ports by name (see 12.3.6) and the `port_expression` which contains identifiers declared inside the module as described in 12.3.3. Use of named port connections shall not be used for implicit ports unless the `port_expression` is a simple `port_identifier`.

12.3.3 Port declarations

Each `port_expression` in the list of ports for the module declaration shall also be declared in the body of the module as one of the following port declarations: **input**, **output**, or **inout** (bidirectional). This is in addition to any other data type declaration for a particular port— for example, a **reg** or **wire**. The syntax for port declarations is given in Syntax 12-6.

```

inout_declaration ::= (From Annex A - A.2.1.2)
    inout [ net_type ] [ signed ] [ range ] list_of_port_identifiers ;
input_declaration ::=
    input [ net_type ] [ signed ] [ range ] list_of_port_identifiers ;
output_declaration ::=
    output [ net_type ] [ signed ] [ range ]
        list_of_port_identifiers ;
    | output [ reg ] [ signed ] [ range ]
        list_of_port_identifiers ;
    | output reg [ signed ] [ range ]
        list_of_variable_port_identifiers ;
    | output [ output_variable_type ]
        list_of_port_identifiers ;
    | output output_variable_type
        list_of_variable_port_identifiers ;
list_of_port_identifiers ::= (From Annex A - A.2.3)
    port_identifier { , port_identifier }

```

Syntax 12-6—Syntax for port declarations

If a port declaration includes a net or variable type, then the port is considered completely declared and it is an error for the port to be declared again as a variable or net data type declaration. Because of this, all other aspects of the port shall be declared in such a port declaration, including the signed and range definitions if needed.

If a port declaration does not include a net or variable type, then the port can be again declared in a net or variable declaration. If the net or variable is declared as a vector, the range specification between the two declarations of a port shall be identical. Once a name is used in a port declaration it shall not be declared again in another port declaration or in a data type declaration.

NOTE—Implementations may limit maximum number of ports in a module definition, but they will at least be 256.

Example:

```

input  aport;    // First declaration - okay.
input  aport;    // Error - multiple declaration, port declaration
output aport;    // Error - multiple declaration, port declaration

```

The signed attribute can be attached either to a port declaration or to the corresponding net or reg declaration, or to

both. If either the port or the net/reg is declared as signed, then the other shall also be considered signed.

Implicit nets shall be considered unsigned. Nets connected to ports without an explicit net declaration shall be considered unsigned, unless the port is declared as signed.

Example:

```

module test(a,b,c,d,e,f,g,h);
input [7:0] a;           // no explicit declaration - net is unsigned
input [7:0] b;
input signed [7:0] c;
input signed [7:0] d;    // no explicit net declaration - net is signed
output [7:0] e;         // no explicit declaration - net is unsigned
output [7:0] f;
output signed [7:0] g;
output signed [7:0] h;  // no explicit net declaration - net is signed

wire signed [7:0] b;    // port b inherits signed attribute from net decl.
wire [7:0] c;           // net c inherits signed attribute from port
reg signed [7:0] f;     // port f inherits signed attribute from reg decl.
reg [7:0] g;           // reg g inherits signed attribute from port

endmodule

module complex_ports ({c,d}, .e(f)); // Nets {c,d} receive the first
// port bits. Name 'f' is declared inside the module.
// Name 'e' is defined outside the module.
// Can't use named port connections of first port.

module split_ports (a[7:4], a[3:0]); // First port is upper 4 bits of
// 'a'.
// Second port is lower 4 bits of 'a'.
// Can't use named port connections because
// of part-select port 'a'.

module same_port (.a(i), .b(i));      // Name 'i' is declared inside the
// module as a inout port. Names 'a' and 'b' are
// defined for port connections.

module renamed_concat (.a({b,c}), f, .g(h[1]));
// Names 'b', 'c', 'f', 'h' are defined inside the module.
// Names 'a', 'f', 'g' are defined for port connections.
// Can use named port connections.

module same_input (a,a);
input a;           // This is legal. The inputs are ored together.

```

12.3.4 Lists of ports declarations

An alternate syntax which minimizes the duplication of data can be used to specify the ports of a module. Modules shall either be declared entirely with the list of ports syntax as described in 12.3.2 or entirely using the *list_of_port_declarations* as described in this section.

Each declared port provides the complete information about the port. The port's direction, width, net, or variable type, and whether the port is signed or unsigned is completely described. The same syntax for input, inout, and output declarations is used in the module header as would be used for the list of port style declaration, except the

list_of_port_declarations is included in the module header rather than separately (after the ; which terminates the module header).

As an example, the module named test given in the previous example could alternatively be declared as:

Example:

```

module test (
    input [7:0] a,
    input signed [7:0] b, c, d, // multiple ports that share all
                                // attributes can be declared together
    output [7:0] e,             // every attribute of the declaration
                                // must be in the one declaration
    output signed reg [7:0] f, g,
    output signed [7:0] h) ;
    // It is illegal to redeclare any ports of the module in the body
    // of the module.
endmodule

```

The *port_reference* type of module port declaration shall not be done using *list_of_port_declarations* style of module declarations. Also ports declared using the *list_of_port_declarations* shall only be simple identifiers. They shall not be bit-selects, part-selects, or concatenations (as in the example *complex_ports*); nor can a port be split (as in the example *split_ports*); nor can they be named ports (as in the example *same_port*).

Designs may freely mix modules declared using each syntax; hence implementations desiring the above special cases of port declaration can be done using the first *list_of_ports* syntax.

12.3.5 Connecting module instance ports by ordered list

One method of making the connection between the port expressions listed in a module instantiation and the ports declared within the instantiated module is the ordered list—that is, the ports expressions listed for the module instance shall be in the same order as the ports listed in the module declaration.

Example:

The following example illustrates a top-level module (*topmod*) that instantiates a second module (*modB*). Module *modB* has ports that are connected by an ordered list. The connections made are as follows:

- Port *wa* in the *modB* definition connects to the bit-select *v[0]* in the *topmod* module.
- Port *wb* connects to *v[3]*.
- Port *c* connects to *w*.
- Port *d* connects to *v[4]*.

In the *modB* definition, ports *wa* and *wb* are declared as *inouts* while ports *c* and *d* are declared as *input*.

```

module topmod;
  wire [4:0] v;
  wire a,b,c,w;

  modB b1 (v[0], v[3], w, v[4]);
endmodule

  module modB (wa, wb, c, d);
    inout wa, wb;
    input c, d;

    tranif1      g1 (wa, wb, cinvert);
    not #(2, 6)  n1 (cinvert, int);
    and #(6, 5) g2 (int, c, d);
endmodule

```

During simulation of the b1 instance of modb, the **and** gate g2 activates first to produce a value on `int`. This value triggers the **not** gate n1 to produce output on `cinvert`, which then activates the **tranif1** gate g1.

12.3.6 Connecting module instance ports by name

The second way to connect module ports consists of explicitly linking the two names for each side of the connection, the port declaration name from the module declaration to the expression — the name used in the module declaration, followed by the name used in the instantiating module. This compound name is then placed in the list of module connections. The port name shall be the name specified in the module declaration. The port name cannot be a bit-select, a part-select, or a concatenation of ports. If the module port declaration was implicit, the `port_expression` shall be a simple `port_identifier` which is used as the port name. If the module port declaration was explicit, the explicit name is used as the name of port.

The port expression can be any valid expression.

The port expression is optional so that the instantiating module can document the existence of the port without connecting it to anything. The parentheses are required.

The two types of module port connections shall not be mixed; connections to the ports of a particular module instance shall be all by order or all by name.

Examples:

Example 1—In the following example, the instantiating module connects its signals `topA` and `topB` to the ports `In1` and `Out` defined by the module `ALPHA`. At least one port provided by `ALPHA` is unused; it is named `In2`. There could be other unused ports not mentioned in the instantiation.

```

ALPHA instance1 (.Out(topB), .In1(topA), .In2());

```


Example 2—This example defines the modules `modB` and `topmod`, and then `topmod` instantiates `modB` using ports connected by name.

```

module topmod;
    wire [4:0] v;
    wire a,b,c,w;

    modB b1 (.wb(v[3]),.wa(v[0]),.d(v[4]),.c(w));
endmodule

module modB(wa, wb, c, d);
    inout wa, wb;
    input c, d;

    tranif1      g1(wa, wb, cinvert);
    not #(6, 2)  n1(cinvert, int);
    and #(5, 6)  g2(int, c, d);
endmodule

```

Since these connections are made by name, the order in which they appear is irrelevant.

Multiple module instance port connections are not allowed, e.g., the following example is illegal:

Example 3—This example shows illegal port connections.

```

module test;
    a ia (.i (a), .i (b),           // illegal connection of input port twice.
         .o (c), .o (d),           // illegal connection of output port twice.
         .e (e), .e (f));          // illegal connection of inout port twice.
endmodule

```

12.3.7 Real numbers in port connections

The `real` data type shall not be directly connected to a port. It shall be connected indirectly, as shown in the following example. The system functions `$realtobits` and `$bitstoreal` shall be used for passing the bit patterns across module ports. (See 17.8 for a description of these system tasks.)

Example:

```
module driver (net_r);
    output net_r;
    real r;
    wire [64:1] net_r = $realtobits(r);
endmodule

module receiver (net_r);
    input net_r;
    wire [64:1] net_r;
    real r;

    initial assign r = $bitstoreal(net_r);
endmodule
```

12.3.8 Connecting dissimilar ports

A port of a module can be viewed as providing a link or connection between two items (nets, regs, expressions, etc.)—one internal to the module instance and one external to the module instance.

Examination of the port connection rules described in 12.3.9 will show that the item receiving the value through the port (the internal item for inputs, the external item for outputs) shall be a structural net expression. The item that provides the value can be any expression.

NOTE—A port that is declared as input (output) but used as an output (input) or inout may be coerced to inout. If not coerced to inout, a warning has to be issued.

12.3.9 Port connection rules

The following rules shall govern the way module ports are declared and the way they are interconnected.

12.3.9.1 Rule 1

An input or inout port shall be of type net.

12.3.9.2 Rule 2

Each port connection shall be a continuous assignment of source to sink, where one connected item shall be a signal source and the other shall be a signal sink. The assignment shall be a continuous assignment from source to sink for input or output ports. The assignment is a nonstrength reducing transistor connection for inout ports. Only nets or structural net expressions shall be the sinks in an assignment.

A *structural net expression* is a port expression whose operands can be the following:

- A scalar net
- A vector net
- A constant bit-select of a vector net
- A part-select of a vector net
- A concatenation of structural net expressions

The following external items shall not be connected to the output or inout ports of modules:

- Variables
- Expressions other than
 - i) A scalar net
 - ii) A vector net
 - iii) A constant bit-select of a vector net
 - iv) A part-select of a vector net
 - v) A concatenation of the expressions listed above

12.3.10 Net types resulting from dissimilar port connections

When different net types are connected through a module port, the nets on both sides of the port can take on the same type. The resulting net type can be determined as shown in Table 12-1. In the table, *external net* means the net specified in the module instantiation, and *internal net* means the net specified in the module definition. The net whose type is used is said to be the *dominating net*. The net whose type is changed is said to be the *dominated net*. It is permissible to merge the dominating and dominated nets into a single net, whose type shall be that of the dominating net. The resulting net is called the *simulated net*, and the dominated net is called a *collapsed net*.

The simulated net shall take the delay specified for the dominating net. If the dominating net is of the type **trireg**, any strength value specified for the trireg net shall apply to the simulated net.

12.3.10.1 Net type resolution rule

When the two nets connected by a port are of different net type, the resulting single net can be assigned one of the following:

- The dominating net type if one of the two nets is dominating, *or*
- The net type external to the module

When a dominating net type does not exist, the external net type shall be used.

12.3.10.2 Net type table

Table 12-1 shows the net type dictated by net type resolution rule.

The simulated net shall take the net type specified in the table and the delay specified for that net. If the simulated net selected is a **trireg**, any strength value specified for the trireg net applies to the simulated net.

Table 12-1—Net types resulting from dissimilar port connections

Internal net	External net							
	wire, tri	wand, triand	wor, trior	trireg	tri0	tri1	supply0	supply1
wire, tri	ext	ext	ext	ext	ext	ext	ext	ext
wand, triand	int	ext	warn	warn	warn	warn	ext	ext

Table 12-1—Net types resulting from dissimilar port connections (*continued*)

Internal net	External net							
	wire, tri	wand, triand	wor, trior	triereg	tri0	tri1	supply0	supply1
wor, trior	int	warn	ext	warn	warn	warn	ext	ext
triereg	int	warn	warn	ext	ext	ext	ext	ext
tri0	int	warn	warn	int	ext	warn	ext	ext
tri1	int	warn	warn	int	warn	ext	ext	ext
supply0	int	int	int	int	int	int	ext	warn
supply1	int	int	int	int	int	int	warn	ext

KEY

ext = The external net type is used

int = The internal net type is used

warn = A warning is issued and the external net type is used

12.3.11 Connecting signed values via ports

The sign attribute shall not cross hierarchy. In order to have the signed type cross hierarchy, the signed keyword must be used in the object's declaration at the different levels of hierarchy. Any expressions on a port shall be treated as any other expression in an assignment. It shall be typed, sized, evaluated and the resulting value assigned to the object on the other side of the port using the same rules as an assignment.

12.4 Hierarchical names

Every identifier in a Verilog HDL description shall have a unique *hierarchical path name*. The hierarchy of modules and the definition of items such as tasks and named blocks within the modules shall define these names. The hierarchy of names can be viewed as a tree structure, where each module instance, generated instance, task, function, or named `begin-end` or `fork-join` block defines a new hierarchical level, or scope, in a particular branch of the tree.

At the top of the name hierarchy are the names of one or more root modules of which no instances have been created. This root or these parallel root modules make up one or more hierarchies in a *design description* or *description*. Inside any module, each module instance (including an arrayed or generated instance), task definition, function definition, and named `begin-end` or `fork-join` block shall define a new branch of the hierarchy. Named blocks within named blocks and within tasks and functions shall create new branches. Only non-recursively referenced automatic tasks and/or functions create visible branches that can be referenced. Recursively called tasks and functions, declared using the automatic keyword and recursively called from within the same task or function, do not create visible branches that can be referenced. See 10.2.1 and 10.3.1 for a discussion of automatic tasks and functions.

Each node in the hierarchical name tree shall be a separate scope with respect to identifiers. A particular identifier can be declared at most once in any scope. See 12.6 for a discussion of scope rules and 3.12 for a discussion of name spaces.

Any named Verilog object or *hierarchical name reference* can be referenced uniquely in its full form by concatenating the names of the modules, module instance names, tasks, functions, or named blocks that contain it. The period character shall be used to separate each of the names in the hierarchy, except for escaped identifiers embedded in the hierarchical name reference, which are followed by separators composed of white space and a period-character. The complete path name to any object shall start at a top-level (root) module. This path name can be used from any level in the hierarchy or from a parallel hierarchy. The first node name in a path name can also be the top of a hierarchy that starts at the level where the path is being used (which allows and enables downward referencing of items) with the exceptions of items of automatic tasks and automatic task item declarations. These declarations can not be accessed by their hierarchical names.

The syntax for hierarchical path names is given in Syntax 12-7.

```

escaped_hierarchical_identifier1 ::= (From Annex A - A.9.3)
    escaped_hierarchical_branch
    [ { .simple_hierarchical_branch | .escaped_hierarchical_branch } ]
escaped_identifier ::=
    \ { Any_ASCII_character_except_white_space } white_space
hierarchical_identifier ::=
    simple_hierarchical_identifier
    | escaped_hierarchical_identifier
simple_hierarchical_identifier2 ::=
    simple_hierarchical_branch [ .escaped_identifier ]
simple_identifier3 ::= [ a-zA-Z_ ] { [ a-zA-Z0-9_$ ] }
simple_hierarchical_branch2 ::= (From Annex A - A.9.4)
    simple_identifier [ [ unsigned_number ] ]
    [ { .simple_identifier [ [ unsigned_number ] ] } ]
escaped_hierarchical_branch1 ::=
    escaped_identifier [ [ unsigned_number ] ]
    [ { .escaped_identifier [ [ unsigned_number ] ] } ]
white_space ::= (From Annex A - A.9.5)
    space | tab | newline | eof4

```

¹The period in escaped_hierarchical_identifier and escaped_hierarchical_branch shall be preceded by white_space, but shall not be followed by white_space.

²The period (.) in simple_hierarchical_identifier and simple_hierarchical_branch shall not be preceded or followed by white_space.

³A simple_identifier and arrayed_reference shall start with an alpha or underscore () character, shall have at least one character, and shall not have any spaces.

⁴End of file.

Syntax 12-7—Syntax for hierarchical path names

Examples:

Example 1—The code in this example defines a hierarchy of module instances and named blocks.

```

module mod (in);
input in;

always @(posedge in) begin : keep
  reg hold;
    hold = in;
end
endmodule

module wave;
reg stim1, stim2;

cct a(stim1, stim2); // instantiate cct

initial begin :wave1
  #100 fork :innerwave
    reg hold;
  join
  #150 begin
    stim1 = 0;
  end
end
endmodule

module cct (stim1, stim2);
input stim1, stim2;

  // instantiate mod
  mod amod(stim1), bmod(stim2);
endmodule

```

Figure 12-1 illustrates the hierarchy implicit in this Verilog code.

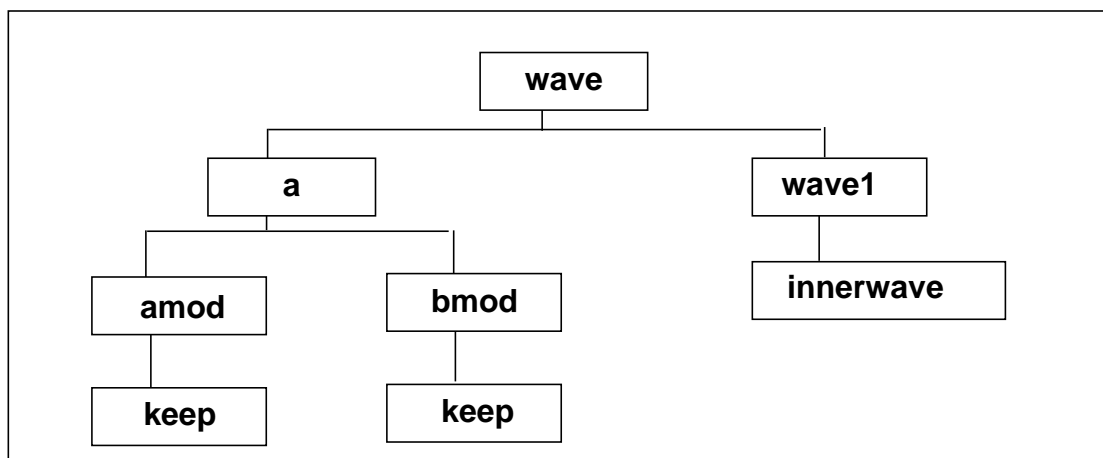


Figure 12-1—Hierarchy in a model

Figure 12-2 is a list of the hierarchical forms of the names of all the objects defined in the code.

wave	wave.a.bmod
wave.stim1	wave.a.bmod.in
wave.stim2	wave.a.bmod.keep
wave.a	wave.a.bmod.keep.hold
wave.a.stim1	wave.wave1
wave.a.stim2	wave.wave1.innerwave
wave.a.amod	wave.wave1.innerwave.hold
wave.a.amod.in	
wave.a.amod.keep	
wave.a.amod.keep.hold	

Figure 12-2—Hierarchical path names in a model

Hierarchical name referencing allows free data access to any object from any level in the hierarchy. If the unique hierarchical path name of an item is known, its value can be sampled or changed from anywhere within the description.

Example 2—The next example shows how a pair of named blocks can refer to items declared within each other.

```
begin
  fork :mod_1
    reg x;
    mod_2.x = 1;
  join
  fork :mod_2
    reg x;
    mod_1.x = 0;
  join
end
```

12.5 Upwards name referencing

The name of a module or module instance is sufficient to identify the module and its location in the hierarchy. A lower-level module can reference items in a module above it in the hierarchy. Variables can be referenced if the name of the higher-level module or its instance name is known. For tasks, functions, and named blocks, Verilog shall look in the enclosing module for the name until it is found or until the root of the hierarchy is reached. It shall only search in higher enclosing modules for the name, not instances. The syntax for an upward reference is given in Syntax 12-8.

upward_name_reference ::= <i>(Not in the Annex A BNF)</i>
module_identifier.item_name
item_name ::=
function_identifier
block_identifier
net_identifier
parameter_identifier
port_identifier
task_identifier
variable_identifier

Syntax 12-8—Syntax for upward name referencing

Upwards name references can also be done with names of the form

```
module_instance_name.item_name
```

A name of this form shall be resolved as follows:

- a) Look in the current module for a module instance named `module_instance_name`. If found, this name reference shall be treated as a downward reference, and the item name shall be resolved in the corresponding module.
- b) Look in the parent module for a module instance named `module_instance_name`. If found, the item name shall be resolved from that instance, which is the sibling of the module containing the reference.
- c) Repeat step b), going up the hierarchy.

There shall be no spaces within the hierarchical name reference, except for escaped identifiers embedded in the hierarchical name reference, which are followed by separators composed of white space and a period-character.

Example:

In this example, there are four modules, `a`, `b`, `c`, and `d`. Each module contains an integer `i`. The highest-level modules in this segment of a model hierarchy are `a` and `d`. There are two copies of module `b` because module `a` and `d` instantiate `b`. There are four copies of `c.i` because each of the two copies of `b` instantiates `c` twice.

```

module a;
integer i;
b a_b1();
endmodule

module b;
integer i;
c b_c1(), b_c2();
initial                                // downward path references two copies of i:
    #10 b_c1.i = 2; // a.a_b1.b_c1.i, d.d_b1.b_c1.i
endmodule

module c;
integer i;
initial begin                            // local name references four copies of i:
    i = 1;                                // a.a_b1.b_c1.i, a.a_b1.b_c2.i,
                                          // d.d_b1.b_c1.i, d.d_b1.b_c2.i
    b.i = 1;                             // upward path references two copies of i:
                                          // a.a_b1.i, d.d_b1.i
end
endmodule

module d;
integer i;
b d_b1();
initial begin                            // full path name references each copy of i
    a.i = 1;                                d.i = 5;
    a.a_b1.i = 2;                            d.d_b1.i = 6;
    a.a_b1.b_c1.i = 3;                        d.d_b1.b_c1.i = 7;
    a.a_b1.b_c2.i = 4;                        d.d_b1.b_c2.i = 8;
end
endmodule

```


12.6 Scope rules

The following four elements define a new scope in Verilog:

- Modules
- Tasks
- Functions
- Named blocks

An identifier shall be used to declare only one item within a scope. This rule means it is illegal to declare two or more variables that have the same name, or to name a task the same as a variable within the same module, or to give a gate instance the same name as the name of the net connected to its output.

If an identifier is referenced directly (without a hierarchical path) within a task, function, or named block, it shall be declared either locally within the task, function, or named block, or within a module, task or named block that is higher in the same branch of the name tree that contains the task, function, or named block. If it is declared locally, then the local item shall be used; if not, the search shall continue upward until an item by that name is found or until a module boundary is encountered. If the item is a variable, it shall stop at a module boundary; if the item is a task, function, or named block it continues to search higher-level modules until found. The search shall cross named block, task, and function boundaries but not module boundaries. This fact means that tasks and functions can use and modify the variables within the containing module by name, without going through their ports.

If an identifier is referenced with a hierarchical name, the path can start with an module name, instance name, task, function, or named block. The names shall be searched first at the current level, then in higher-level modules until found. Since both module names and instance names can be used, precedence is given to instance names if there is a module named the same as an instance name.

Because of the upward searching, path names which are not strictly on a downward path can be used.

Example:

Example 1—In Figure 12-3, each rectangle represents a local scope. The scope available to upward searching extends outward to all containing rectangles—with the boundary of the module A as the outer limit. Thus block G can directly reference identifiers in F, E, and A; it cannot directly reference identifiers in H, B, C, and D.

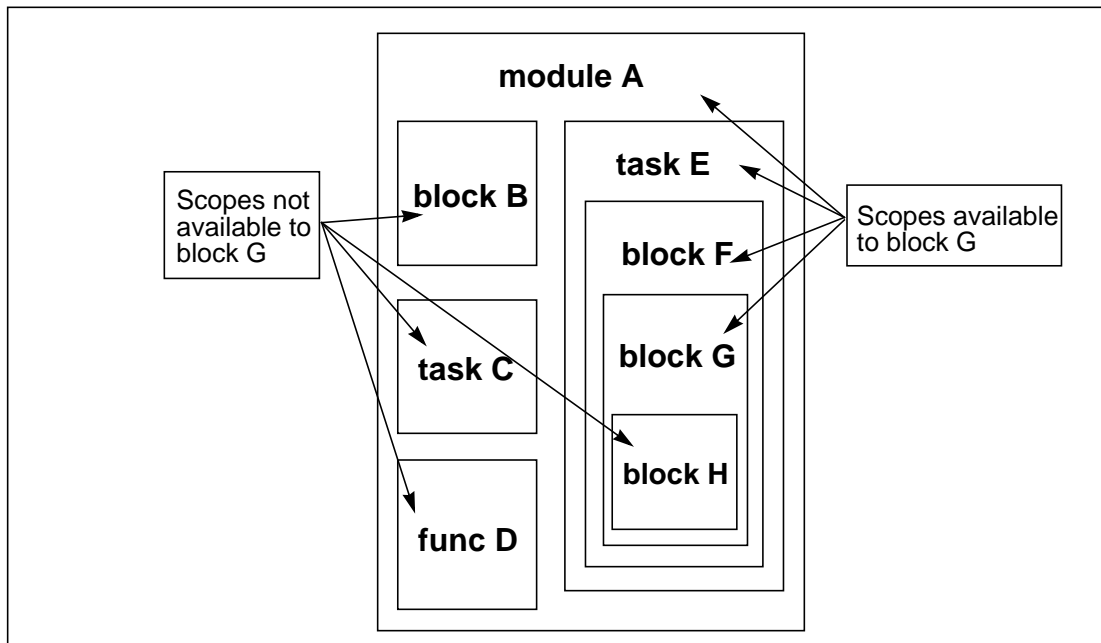


Figure 12-3—Scopes available to upward name referencing

Example 2—The following example shows an incompletely defined downward reference that can be accessed.

```
task t;  
reg r, s;  
begin : b  
    // redundant assignments to reg r  
    t.b.r = 0; // poorly defined but found by upward search  
    t.s = 0;   // fully defined downward reference  
end  
endtask
```

Section 13

Configuring the contents of a design

13.1 Introduction

To facilitate both the sharing of Verilog designs between designers and/or design groups, and the repeatability of the exact contents of a given simulation (or other tool) session, the concept of *configurations* is used in the Verilog language. A configuration is simply an explicit set of rules to specify the exact source description to be used to represent each instance in a design. The operation of selecting a source representation for an instance is referred to as *binding* the instance.

The example below shows a simple configuration problem.

Example:

<pre>file top.v module top(); adder a1(...); adder a2(...); endmodule</pre>	<pre>file adder.v module adder(...); // rtl adder description ... endmodule</pre>	<pre>file adder.vg module adder(...); // gate-level adder description ... endmodule</pre>
---	---	---

Consider using the rtl adder description in adder.v for instance a1 in module top and the gate-level adder description in adder.vg for instance a2. In order to specify this particular set of instance bindings and to avoid having to change the source description to specify a new set, a configuration can be used.

```
config cfg1; // specify rtl adder for top.a1, gate-level adder for top.a2
  design rtlLib.top;
  default liblist rtlLib;
  instance top.a2 liblist gateLib;
endconfig
```

The elements of a *config* are explained in subsequent sections, but this simple example illustrates some important points about *configs*. As evidenced by the **config-endconfig** syntax, the config is a design element, similar to a module, which exists in the Verilog namespace. The config contains a set of rules which are applied when searching for a source description to *bind* to a particular instance of the design.

A Verilog design description starts with a top-level module (or modules), which is not instantiated elsewhere in the design. From this module's source description, the instantiated modules (or children) are found, and then the source descriptions for the module definitions of these subinstances shall be located, and so on until every instance in the design is mapped to a source description.

13.1.1 Library notation

In order to map a Verilog instance to a source description, the concept of a symbolic *library*, which is simply a logical collection of design elements (such as modules, macromodules, primitives, or configs) can be used. These design elements can be referred to as *cells*. The cell name shall be the same as the name of the module/macromodule/primitive/config being processed. Syntax 13-1 specifies a cell from a given library.

`library_cell ::= (Not in the Appendix A BNF)`
`[library_identifier.]cell_identifier[:config]`

Syntax 13-1—Syntax for cell

This notation gives a symbolic method of referring to source descriptions; the method of mapping source descriptions into libraries is shown in greater detail in 13.2.1. The optional `:config` extension shall be used explicitly to refer to a config in the case where a config has the same name as a module/macromodule/primitive.

For the purposes of this example, suppose the files `top.v` and `adder.v`, the rtl descriptions, have been mapped into the library `rtlLib`, and the file `adder.vg`, the gate-level description of the `adder`, has been mapped into the library `gateLib`. The actual mechanism for mapping source descriptions to libraries is detailed in 13.2.

13.1.2 Basic configuration elements

The **design** statement in `config cfg1` of the first example of 13.1 specifies the top-level module in the design and what source description is to be used. In this example, the `rtlLib.top` notation indicates the top-level module description shall be taken from `rtlLib`. Since `top.v` and `adder.v` were mapped to this library, the actual description for the module is known to come from `top.v`.

The **default** statement coupled with the **liblist** clause specifies, by default, all subinstances of `top` (i.e., `top.a1` and `top.a2`) shall be taken from `rtlLib`, which means the descriptions in `top.v` and `adder.v`, which were mapped to this library, shall be used. For a basic design, which can be completely rtl, this can be sufficient to specify completely the binding for the entire design. However, here the `top.a2` instance of `adder` to the gate-level description shall be bound.

The **instance** statement specifies, for the particular instance `top.a2`, the source description shall be taken from `gateLib`. The instance statement overrides the default rule for this particular instance. Since `adder.vg` was mapped to `gateLib`, this statement dictates the gate-level description in `adder.vg` be used for instance `top.a2`.

13.2 Libraries

As mentioned in the previous section, a library is a logical collection of cells which are mapped to particular source description files. The symbolic `lib.cell[:config]` notation supports the separate compilation of source files by providing a file system-independent name to refer to source descriptions when instances in a design are bound. It also allows multiple tools, which can have different invocation use-models, to share the same configuration.

13.2.1 Specifying libraries - the library map file

When parsing a source description file (or files), the parser shall first read the library mapping information from a pre-defined file prior to reading any source files. The name of this file and the mechanism for reading it shall be tool-specific, but all compliant tools shall provide a mechanism to specify one or more library mapping files to be used for a particular invocation of the tool. If multiple mapping files are specified, then they shall be read in the order in which they are specified.

For the purposes of this discussion, assume the existence of a file named `lib.map` in the current working directory, which is automatically read by the parser prior to parsing any source files specified on the command line. The syntax for declaring a library in the library map file is shown in Syntax 13-2.

```

    escaped_hierarchical_identifier1 ::= (From Annex A - A.1.1)
    library_text ::=
        { library_descriptions }
    library_descriptions ::=
        library_declaration
        | include_statement
        | config_declaration
    library_declaration ::=
        library library_identifier file_path_spec [ { , file_path_spec } ]
        [ -incdir file_path_spec [ { , file_path_spec } ] ;
    file_path_spec ::=
        file_path
    include_statement ::=
        include <file_path_spec> ;

```

¹The period in escaped_hierarchical_identifier and escaped_hierarchical_branch shall be preceded by white_space, but shall not be followed by white_space.

Syntax 13-2—Syntax for declaring library in the library map file

NOTES

1—The file_path uses file system-specific notation to specify an absolute or relative path to a particular file or set of files. The following shortcuts/wildcards can be used:

- ? single character wildcard (matches any single character)
- * multiple character wildcard (matches any number of characters in a directory/file name)
- ... hierarchical wildcard (matches any number of hierarchical directories)
- .. specifies the parent directory
- . specifies the directory containing the lib.map

Paths which end in / shall include all files in the specified directory. Identical to /*.

Paths which do not begin with / are relative to the directory in which the current lib.map file is located.

2—The paths ./*.v and *.v are identical and both specify all files with a .v suffix in the current directory.

Any file encountered by the compiler which does not match any library's file_path specification shall by default be compiled into a library named work.

To perform the library mapping discussed in the example in 13.1, use the following library definitions in the lib.map file:

```

library rtlLib *.v;                // matches all files in the current directory with a .v suffix
library gateLib ./*.vg;        // matches all files in the current directory with a .vg suffix

```

13.2.1.1 File path resolution

If a file name potentially matches multiple file path specifications, the path specifications shall be resolved in the following order:

- a) File path specifications which end with an explicit filename
- b) File path specifications which end with a wildcarded filename
- c) File path specifications which end with a directory

If a file name matches path specifications in multiple library definitions (after the above resolution rules have been applied), it shall be an error.

Using these rules with the library definitions in the `lib.map` file, all source files encountered by the parser/compiler can be mapped to a unique library. Once the source descriptions have been mapped to libraries, the cells defined therein are available for binding.

NOTE—Tool implementers may find it convenient to provide a command-line argument to explicitly specify the library into which the file being parsed is to be mapped, which shall override any library definitions in the `lib.map` file. If these libraries do not exist in the `lib.map` file, they can only be accessed via an explicit config.

If multiple cells with the same name map to the same library, then the LAST cell encountered shall be written to the library. This is to support a “separate-compile” use-model (see 13.4.3), where it is assumed encountering a cell after it has previously been compiled is intended to be a recompiling of the cell. In the case where multiple modules with the same name are mapped to the same library in a single invocation of the compiler, then a warning message shall be issued.

13.2.2 Using multiple library mapping files

In addition to specifying library mapping information, a `lib.map` file can also include references to other `lib.map` files. The **include** command is used to insert the entire contents of a library mapping file in another file during parsing. The result is as though the contents of the included mapping file appear in place of the **include** command.

The syntax of a `lib.map` file is limited to library specifications, include statements, and standard Verilog comment syntax. Syntax 13-3 shows the syntax for the **include** command.

<pre>include_statement ::= (From Annex A - A.1.1) include <file_path_spec> ;</pre>

Syntax 13-3—Syntax for include command

If the file path specification, whether in an include or library statement, describes a relative path, it shall be relative to the location of the file which contains the file path. Library providers shall include a local library mapping file in addition to the source contents of the library. Individual users can then simply include the provider’s library mapping file in their own map file to gain access to the contents of the provided library.

13.2.3 Mapping source files to libraries

For each cell definition encountered during parsing/compiling, the name of the source file being parsed is compared to the file path specifications of the library declarations in all of the library map files being used. The cell is mapped into the library whose file path specification matches the source file name.

13.3 Configurations

As mentioned in the introduction of this chapter, a configuration is simply a set of rules to apply when searching for library cells to which to bind instances. The syntax for configurations is shown in 13.3.1.

13.3.1 Basic configuration syntax

The configuration syntax is shown in Syntax 13-4.

```

config_declaration ::= (From Annex A -A.1.2)
    config config_identifier ;
    design_statement
    { config_rule_statement }
    endconfig

design_statement ::=
    design { [library_identifier.]cell_identifier } ;

config_rule_statement ::=
    default_clause liblist_clause
    | inst_clause liblist_clause
    | inst_clause use_clause
    | cell_clause liblist_clause
    | cell_clause use_clause

```

Syntax 13-4—Syntax for configuration

13.3.1.1 Design statement

The **design** statement names the library and cell of the top-level module or modules in the design hierarchy configured by the config. There shall be one and only one design statement, but multiple top-level modules can be listed in the design statement. The cell or cells identified can not be configurations themselves. It is possible the design identifier can have the same name as configs, however.

The **design** statement shall appear before any config rule statements in the config.

If the library identifier is omitted, then the library which contains the config shall be used to search for the cell.

13.3.1.2 The default clause

The syntax for the **default** clause is specified in Syntax 13-5.

```

default_clause ::= (From Annex A - A.1.2)
    default

```

Syntax 13-5—Syntax for default clause

The **default** clause selects all instances which do not match a more specific selection clause. The **use** expansion clause (see 13.3.1.6) can not be used with a **default** selection clause. For other expansion clauses, there can not be more than one **default** clause which specifies the expansion clause.

For simple design configurations, it might be sufficient to specify a **default liblist** (see 13.3.1.5).

13.3.1.3 The instance clause

The **instance** clause is used to specify the specific instance to which the expansion clause shall apply. The syntax for the **instance** clause is specified in Syntax 13-6.

```
inst_clause ::= (From Annex A - A.1.2)
               instance inst_name
inst_name ::=
               toplevel_identifier{.instance_identifier}
```

Syntax 13-6—Syntax for instance clause

The instance name associated with the **instance** clause is a Verilog hierarchical name, starting at the top-level module of the config (i.e., the name of the cell in the **design** statement).

13.3.1.4 The cell clause

The **cell** selection clause names the cell to which it applies. The syntax for the **cell** clause is specified in Syntax 13-7.

```
cell_clause ::= (From Annex A - A.1.2)
               cell [ library_identifier.]cell_identifier
```

Syntax 13-7—Syntax for cell clause

If the optional library name is specified then the selection rule applies to any instance which is bound or is under consideration for being bound to the selected library and cell. It is an error if a library name is included in a **cell** selection clause and the corresponding expansion clause is a library list expansion clause.

13.3.1.5 The liblist clause

The **liblist** clause defines an ordered set of libraries to be searched to find the current instance. The syntax for the **liblist** clause is specified in Syntax 13-8.

```
liblist_clause ::= (From Annex A - A.1.2)
                  liblist [{library_identifier}]
```

Syntax 13-8—Syntax for liblist clause

liblists are inherited hierarchically downward as instances are bound. When searching for a cell to bind to the current unbound instance, and in the absence of an applicable binding expansion clause, the specified library list is searched in the specified order.

The current library list is selected by the selection clauses. If no library list clause is selected, or the selected library list is empty, then the library list contains the single name which is the library in which the cell containing the unbound instance is found (i.e., the parent cell's library).

13.3.1.6 The use clause

The **use** clause specifies a specific binding for the selected cell. The syntax for the **use** clause is specified in Syntax 13-9.


```
use_clause ::= (From Annex A - A.1.2)
             use [library_identifier.]cell_identifier[:config]
```

Syntax 13-9—Syntax for use clause

A **use** clause can only be used in conjunction with an **instance** or **cell** selection clause. It specifies the exact library and cell to which a selected cell or instance is bound. A **use** clause with no library or cell indicates the selected cell is unbound.

The **use** clause has no effect on the current value of the library list. It can be common in practice to specify multiple config rule statements, one of which specifies a binding and the other of which specifies a library list.

If the lib.cell being referred to by the **use** clause is a config which has the same name as a module/macromodule/primitive in the same library, then the optional **:config** suffix can be added to the lib.cell to specify the config explicitly.

If the library name is omitted, the library shall be inherited from the parent cell.

The binding statement can create situations where the unbound instance's module name and the cell name to which it is bound are different. This condition is common in VHDL, but has never before been possible in Verilog.

13.3.2 Hierarchical configurations

For situations where it is desirable to specify a special set of configuration rules for a subsection of a design, it is possible to bind a particular instance directly to a configuration using the binding clause:

```
instance top.a1.foo use lib1.foo:config;
// bind to the config foo in library lib1
```

specifies the instance top.a1.foo is to be replaced with the design hierarchy specified by the configuration lib1.foo:config. The **design** statement in lib1.foo:config shall specify the actual binding for the instance top.a1.foo, and the rules specified in the config shall determine the configuration of all other subinstances under top.a1.foo.

It shall be an error for an instance clause to specify a hierarchical path to an instance which occurs within a hierarchy specified by another config.

```
config bot;
  design lib1.bot;
  default liblist lib1 lib2;
  instance bot.a1 liblist lib3;
endconfig

config top;
  design lib1.top;
  default liblist lib2 lib1;
  instance top.bot use lib1.bot:config;
  instance top.bot.a1 liblist lib4;
  // ERROR - can't set liblist for top.bot.a1 from this config
endconfig
```

13.4 Using libraries and configs

The following section describes potential use-models for referencing configs on the command line. It is included for clarification purposes.

The traditional Verilog simulation use-model takes a file-based approach, where the source descriptions for all cells in the design are specified on the command line for each invocation of the tool. With the advent of compiled-code simulators, the configuration mechanism shall also support a use-model which allows for the source files to be pre-compiled and then for the pre-compiled design objects to be referenced on the command line. This section shall explain how configurations can be used in both of these scenarios.

13.4.1 Precompiling in a single-pass use-model

The single-pass use-model is the traditional use-model with which most users are familiar. In this use-model, all of the source description files shall be provided to the simulator via the command line, and only these source descriptions can be used to bind cell instances in the current design. A precompiling strategy in this scenario actually parses every cell description provided on the command line and map it into the library without regard to whether the cell actually is used in the design. The tool can optionally check to see if the cell already exists in the library, and if it is up-to-date (i.e. the source description has not changed since the last time the cell was compiled) the tool can skip recompiling the cell. After all cells on the command line have been compiled, then the tool can locate the top-level cell (discussed in Section 12), and proceed down the hierarchy, binding each instance as it is encountered in the hierarchy.

NOTE—With this use-model it is not necessary for library objects to persist from one tool invocation to another (although for performance considerations it is recommended they do).

13.4.2 Elaboration-time compiling in a single-pass use-model

An alternate strategy which can be used with a single-pass tool is to parse the source files only to find the top-level module(s), without actually compiling anything into the library during this scanning process. Once the top-level module(s) has been found, then it can be compiled into the library, and the tool can proceed down the hierarchy, only compiling the source descriptions necessary to bind the design successfully. Based on the binding rules in place, only the source files which match the current library specification need to be parsed to find the current cell's source description to compile. As with the precompiled single-pass use-model, it is not necessary for library cells to persist from one invocation to another using this strategy.

13.4.3 Precompiling using a separate compilation tool

When using a separate compilation tool, it is essential library cells persist, and the compiled forms shall therefore exist somewhere in the file system. The exact format and location for holding these compiled forms shall be vendor/tool-specific. Using this separate compiler strategy, the source descriptions shall be parsed and compiled into the library using one or more invocations of the compiler tool. The only restriction is all cells in a design shall be precompiled prior to binding the design (typically via an invocation of a separate tool). Using this strategy, the tool which actually does the binding only needs to be told the top-level module(s) of the design to be bound, and then it shall use the precompiled form of the cell description(s) from the library to determine the subinstances and descend hierarchically down the design binding each cell as it is located.

13.4.4 Command line considerations

In each of the three preceding strategies, the binding rules can either be specified via a config, or the default rules (from the library map file) can be used. In the single-pass use-models, the config can be specified by including its source description file on the command line. In the case where the config includes a design statement, then the specified cell shall be the top-level module, regardless of the presence of any uninstantiated cells in the rest of the source files. When using a separate compilation tool, the tool which actually does the binding only needs to be given the *lib.cell* specification for the top-level cell(s) and/or the config to be used. In this strategy, the config itself shall also be precompiled.

13.5 Configuration examples

Consider the following set of source descriptions:

Example:

```

file top.v          file adder.v          file adder.vg          file lib.map
module top(...);  module adder(...); module adder(...);  library rtlLib top.v;
...                ... // rtl              ... // gate-level library aLib adder.*;
adder a1(...);     foo f1(...);             foo f1(...);        library gateLib
adder a2(...);     foo f2(...);             foo f2(...);        adder.vg;
endmodule        endmodule                endmodule
module foo(...);   module foo(...);           module foo(...);
... // rtl         ... // rtl              ... // gate-level
endmodule        endmodule                endmodule

```

All of the examples in this section shall assume the `top.v`, `adder.v` and `adder.vg` files get compiled with the given `lib.map` file. This yields the following library structure:

```

rtlLib.top // from top.v
rtlLib.foo // from top.v
aLib.adder // from adder.v
aLib.foo // rtl from adder.v
gateLib.adder // from adder.vg
gateLib.foo // from adder.vg

```

13.5.1 Default configuration from library map file

With no configuration, the libraries are searched according to the library declaration order in the library map file. This means all instances of module `adder` shall use `aLib.adder` (since `aLib` is the first library specified which contains a cell named `adder`), and all instances of module `foo` shall use `rtlLib.foo` (since `rtlLib` is the first library which contains `foo`).

13.5.2 Using the default clause

To always use the `foo` definition from file `adder.v`, use the following simple configuration:

```

config cfg1;
    design rtlLib.top
    default liblist aLib rtlLib;
endconfig

```

The **default liblist** statement overrides the library search order in the `lib.map` file, so `aLib` is always searched before `rtlLib`. Since the `gateLib` library is not included in the `liblist`, the gate-level descriptions of `adder` and `foo` shall not be used.

To use the gate-level representations of `adder` and `foo`, add to the config as follows:

```

config cfg2;
    design rtlLib.top
    default liblist gateLib aLib rtlLib;
endconfig

```

This shall cause the gate representation always to be taken before the rtl representation, using the module definitions for `adder` and `foo` from `adder.vg`. The rtl view of `top` shall be taken since there is no gate representation available.

13.5.3 Using the cell clause

To modify the config to use the rtl view of `adder` and the gate-level representation of `foo` from `gateLib`:

```
config cfg3;
  design rtlLib.top
  default liblist aLib rtlLib;
  cell foo use gateLib.foo;
endconfig
```

The cell clause selects all cells named `foo` and explicitly binds them to the gate representation in `gateLib`.

13.5.4 Using the instance clause

To modify the config so the `top.a1` `adder` (and its descendants) use the gate representation and the `top.a2` `adder` (and its descendants) use the rtl representation from `aLib`:

```
config cfg4
  design rtlLib.top
  default liblist gateLib rtlLib;
  instance top.a2 liblist aLib;
endconfig
```

Since the **liblist** is inherited, all of the descendants of `top.a2` inherit its **liblist** from the instance selection clause.

13.5.5 Using a hierarchical config

Now suppose all this work has only been on the `adder` module by itself and a config which uses the `rtlLib.foo` cell for `f1`, and the `gateLib.foo` cell for `f2` has already been developed. Then use:

```
config cfg5;
  design aLib.adder;
  default liblist gateLib aLib;
  instance adder.f1 liblist rtlLib;
endconfig
```

To use this configuration `cfg5` for the `top.a2` instance of `adder` and take the full default `aLib` `adder` for the `top.a1` instance, use the following config:

```
config cfg6;
  design rtlLib.top;
  default liblist aLib rtlLib;
  instance top.a2 use work.cfg5:config
endconfig
```

The binding clause specifies the `work.cfg5:config` configuration is to be used to resolve the bindings of instance `top.a2` and its descendants. It is the design statement in config `cfg5` which defines the exact binding for the `top.a2` instance itself. The rest of `cfg5` defines the rules to bind the descendants of `top.a2`. Notice the instance clause in `cfg5` is relative to its own top-level module, `adder`.

13.6 Displaying library binding information

It shall be possible to display the actual library binding information for module instances during simulation. The format specifier %l or %L shall print out the `library.cell` binding information for the module instance containing the display (or other textual output) command. This is similar to the %m format specifier which prints out the hierarchical path name of the module containing it.

It shall also be able to use the VPI interface to display the binding information. The following new `vpiProperties` shall exist for objects of type `vpiModule`:

- `vpiUseBinding` - the `library.cell` binding information for a module instance
- `vpiLibrary` - the library name into which the module was compiled
- `vpiCell` - the name of the cell bound to the module instance
- `vpiConfig` - the `library.cell` name of the config controlling the binding of the module instance

These properties shall be of `string` type, similar to the `vpiName` and `vpiFullName` properties.

13.7 Reserved words

The keywords **config**, **endconfig**, and **default** shall be treated as reserved words in the language. The following keywords shall be reserved words inside of a `config-endconfig` block only:

design
instance
cell
use
liblist

13.8 Library mapping examples

In the absence of a configuration, it is possible to perform basic control of the library searching order when binding a design.

When a config is used, the config overrides the rules specified here.

13.8.1 Using the command line to control library searching

In the absence of a configuration, it shall be necessary for all compliant tools to provide a mechanism of specifying a library search order on the command line which overrides the default order from the library mapping file. This mechanism shall include specification of library names only, with the definitions of these libraries to be taken from the library mapping file.

NOTE—It is recommended all compliant tools use "-L <library_name>" to specify this search order.

13.8.2 File path specification examples

Example:

Given the following set of files:

```
/proj/lib1/rtl/a.v  
/proj/lib2/gates/a.v  
/proj/lib1/rtl/b.v  
/proj/lib2/gates/b.v
```

From the `/proj` library, the following absolute `file_path_specs` are resolved as shown:

```
/proj/lib*/*/a.v =/proj/lib1/rtl/a.v, /proj/lib2/gates/a.v
../a.v =/proj/lib1/rtl/a.v, /proj/lib2/gates/a.v
/proj/.../b.v =/proj/lib1/rtl/b.v, /proj/lib2/gates/b.v
.../rtl/*.v =/proj/lib1/rtl/a.v, /proj/lib1/rtl/b.v
```

From the `/proj/lib1` directory, the following relative `file_path_specs` are resolved as shown:

```
../lib2/gates/*.v = /proj/lib2/gates/a.v, /proj/lib2/gates/b.v
./rtl/?..v = /proj/lib1/rtl/a.v, /proj/lib1/rtl/b.v
./rtl/ = /proj/lib1/rtl/a.v, /proj/lib1/rtl/b.v
```

13.8.3 Resolving multiple path specifications

Example:

```
library lib1 "/proj/lib1/foo*.v";
library lib2 "/proj/lib1/foo.v";
library lib3 "../lib1/";
library lib4 "/proj/lib1/*ver.v";
```

When evaluated from the directory `/proj/tb` directory, the following source files shall map into the specified library:

<code>../lib1/foobar.v</code>	-	<code>lib1</code> // potentially matches <code>lib1</code> and <code>lib3</code> . Since <code>lib1</code> includes a filename and <code>lib3</code> only specifies a directory; <code>lib1</code> takes precedence
<code>/proj/lib1/foo.v</code>	-	<code>lib2</code> // takes precedence over <code>lib1</code> and <code>lib3</code> path specifications
<code>/proj/lib1/bar.v</code>	-	<code>lib3</code>
<code>/proj/lib1/barver.v</code>	-	<code>lib4</code> // takes precedence over <code>lib3</code> path specification
<code>/proj/lib1/foover.v</code>	-	ERROR // matches <code>lib1</code> and <code>lib4</code>
<code>/test/tb/tb.v</code>	-	<code>work</code> // does not match any library specifications.

Section 14

Specify blocks

Two types of HDL constructs are often used to describe delays for structural models such as ASIC cells. They are

- *Distributed delays*, which specify the time it takes events to propagate through gates and nets inside the module (see 7.14)
- *Module path delays*, which describe the time it takes an event at a source (input port or inout port) to propagate to a destination (output port or inout port)

This section describes how paths are specified in a module and how delays are assigned to these paths.

14.1 Specify block declaration

A block statement called the *specify block* is the vehicle for describing paths between a source and a destination and for assigning delays to these paths. The syntax for specify block is shown in Syntax 14-1.

```
specify_item ::= (From Annex A - A.7.1)
               specparam_declaration
               | pulsestyle_declaration
               | showcanceled_declaration
               | path_declaration
               | system_timing_check
```

Syntax 14-1—Syntax of specify block

The specify block shall be bounded by the keywords **specify** and **endspecify**, and it shall appear inside a module declaration. The specify block can be used to perform the following tasks:

- Describe various paths across the module.
- Assign delays to those paths.
- Perform timing checks to ensure that events occurring at the module inputs satisfy the timing constraints of the device described by the module (see Section 15).

The paths described in the specify block, called *module paths*, pair a signal source with a signal destination. The source may be unidirectional (an input port) or bidirectional (an inout port) and is referred to as the *module path source*. Similarly, the destination may be unidirectional (an output port) or bidirectional (an inout port) and is referred to as the *module path destination*.

Example:

```
specify
  specparam tRise_clk_q = 150, tFall_clk_q = 200;
  specparam tSetup = 70;

  (clk => q) = (tRise_clk_q, tFall_clk_q);

  $setup(d, posedge clk, tSetup);
endspecify
```

The first two lines following the keyword **specify** declare specify parameters, which are discussed in 3.11.3. The line following the declarations of specify parameters describes a module path and assigns delays to that module path. The specify parameters determine the delay assigned to the module path. Specifying module paths is presented in 14.2. Assigning delays to module paths is discussed in 14.3. The line preceding the keyword **endspecify** instantiates one of the system timing checks, which are discussed further in Section 15.

14.2 Module path declarations

There are two steps required to set up module path delays in a specify block:

- a) Describe the module paths
- b) Assign delays to those paths (see 14.3)

The syntax of the module path declaration is described in Syntax 14-2.

<pre>path_declaration ::= (From Annex A - A.7.2) simple_path_declaration ; edge_sensitive_path_declaration ; state_dependent_path_declaration ;</pre>

Syntax 14-2—Syntax of the module path declaration

A module path may be described as a *simple path*, an *edge sensitive path*, or a *state dependent path*. A module path shall be defined inside a specify block as a connection between a source signal and a destination signal. Module paths can connect any combination of vectors and scalars.

Example:

Figure 14-1 illustrates a circuit with module path delays. More than one source (A, B, C, and D) may have a module path to the same destination (Q), and different delays may be specified for each input to output path.

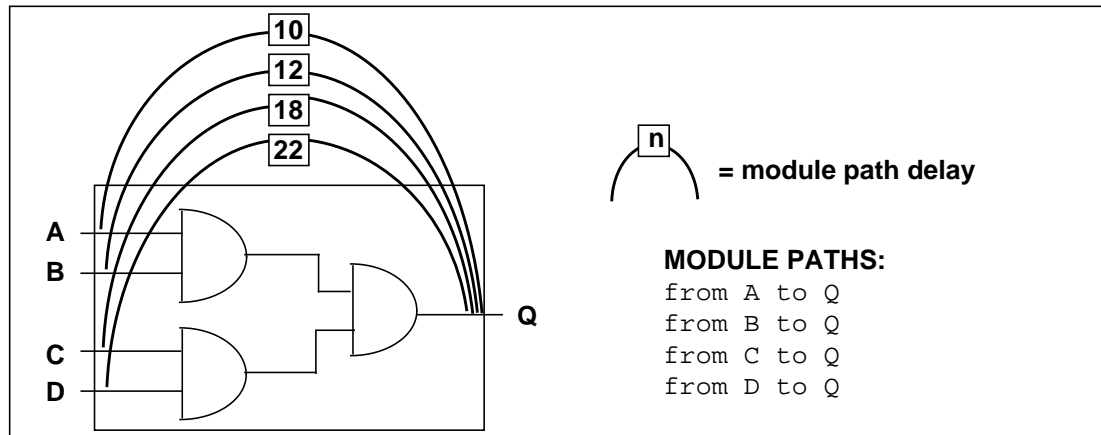


Figure 14-1—Module path delays

14.2.1 Module path restrictions

Module paths have the following restrictions:

- The module path source shall be a net that is connected to a module input port or inout port.
- The module path destination shall be a net or variable that is connected to a module output port or inout port.
- The module path destination shall have only one driver inside the module.

14.2.2 Simple module paths

The syntax for specifying a simple module path is given in Syntax 14-3.

```

simple_path_declaration ::= (From Annex A - A.7.2)
    parallel_path_description = path_delay_value
    | full_path_description = path_delay_value
parallel_path_description ::=
    ( specify_input_terminal_descriptor [ polarity_operator ] =>
      specify_output_terminal_descriptor )
full_path_description ::=
    ( list_of_path_inputs [ polarity_operator ] *> list_of_path_outputs )
list_of_path_inputs ::=
    specify_input_terminal_descriptor { , specify_input_terminal_descriptor }
list_of_path_outputs ::=
    specify_output_terminal_descriptor { , specify_output_terminal_descriptor }
specify_input_terminal_descriptor ::= (From Annex A - A.7.3)
    input_identifier
    | input_identifier [ constant_expression ]
    | input_identifier [ range_expression ]
specify_output_terminal_descriptor ::=
    output_identifier
    | output_identifier [ constant_expression ]
    | output_identifier [ range_expression ]
input_identifier ::=
    input_port_identifier | inout_port_identifier
output_identifier ::=
    output_port_identifier | inout_port_identifier
polarity_operator ::= (From Annex A - A.7.4)
    + | -

```

Syntax 14-3—Syntax for simple module path

Simple path can be declared in one of the two forms:

- source *> destination
- source => destination

The symbols *> and => each represent a different kind of connection between the module path source and the module path destination. The operator *> establishes a *full connection* between source and destination. The operator => establishes a *parallel connection* between source and destination. Refer to 14.2.5 for a description of full connection and parallel connection paths.

Example:

The following three examples illustrate valid simple module path declarations.

```

(A => Q) = 10;
(B => Q) = (12);
(C, D *> Q) = 18;

```

14.2.3 Edge-sensitive paths

When a module path is described using an edge transition at the source, it is called an *edge-sensitive path*. The edge-

sensitive path construct is used to model the timing of input to output delays, which only occur when a specified edge occurs at the source signal.

The syntax of the edge-sensitive path declaration is shown in Syntax 14-4.

```

edge_sensitive_path_declaration ::= (From Annex A - A.7.4)
    parallel_edge_sensitive_path_description = path_delay_value
    | full_edge_sensitive_path_description = path_delay_value
parallel_edge_sensitive_path_description ::=
    ( [ edge_identifier ] specify_input_terminal_descriptor =>
        specify_output_terminal_descriptor [ polarity_operator ] : data_source_expression )
full_edge_sensitive_path_description ::=
    ( [ edge_identifier ] list_of_path_inputs *>
        list_of_path_outputs [ polarity_operator ] : data_source_expression )
data_source_expression ::=
    expression
edge_identifier ::=
    posedge | negedge

```

Syntax 14-4—Syntax of the edge-sensitive path declaration

The edge identifier may be one of the keywords **posedge** or **negedge**, associated with an input terminal descriptor, which may be any input port or inout port. If a vector port is specified as the input terminal descriptor, the edge transition shall be detected on the least significant bit. If the edge transition is not specified, the path shall be considered active on any transition at the input terminal.

An edge-sensitive path may be specified with full connections (*>) or parallel connections (=>). For parallel connections (=>), the destination shall be any scalar output or inout port or one of its bit-selects. For full connections (*>), the destination shall be a list of one or more of the vector or scalar output and inout ports, and bit-selects or part-selects of those ports. Refer to 14.2.5 for a description of parallel paths and full connection paths.

The data source expression is an arbitrary expression, which serves as a description of the flow of data to the path destination. This arbitrary data path description does not affect the actual propagation of data or events through the model; how an event at the data path source propagates to the destination depends on the internal logic of the module. The polarity operator describes whether the data path is inverting or noninverting.

Examples:

Example 1—The following example demonstrates an edge-sensitive path declaration with a positive polarity operator:

```
( posedge clock => ( out +: in ) ) = (10, 8);
```

In this example, at the positive edge of `clock`, a module path extends from `clock` to `out` using a rise delay of 10 and a fall delay of 8. The data path is from `in` to `out`, and `in` is not inverted as it propagates to `out`.

Example 2—The following example demonstrates an edge-sensitive path declaration with a negative polarity operator:

```
( negedge clock[0] => ( out -: in ) ) = (10, 8);
```

In this example, at the negative edge of `clock[0]`, a module path extends from `clock[0]` to `out` using a rise delay of 10 and a fall delay of 8. The data path is from `in` to `out`, and `in` is inverted as it propagates to `out`.

Example 3—The following example demonstrates an edge-sensitive path declaration with no edge identifier:

```
( clock => ( out : in ) ) = (10, 8);
```

In this example, at any change in `clock`, a module path extends from `clock` to `out`.

14.2.4 State-dependent paths

A *state-dependent path* makes it possible to assign a delay to a module path that affects signal propagation delay through the path only if specified conditions are true.

A state-dependent path description includes the following items:

- A conditional expression that, when evaluated true, enables the module path
- A module path description
- A delay expression that applies to the module path

The syntax for the state-dependent path declaration is shown in Syntax 14-5.

```
state_dependent_path_declaration ::= (From Annex A - A.7.4)
    if ( module_path_expression ) simple_path_declaration
    | if ( module_path_expression ) edge_sensitive_path_declaration
    | ifnone simple_path_declaration
```

Syntax 14-5—Syntax of state-dependent paths

14.2.4.1 Conditional expression

The operands in the conditional expression shall be constructed from the following:

- Scalar or vector module input ports or inout ports or their bit-selects or part-selects
- Locally defined variables or nets or their bit-selects or part-selects
- Compile time constants (constant numbers and specify parameters)

Table 14-1 contains a list of valid operators that may be used in conditional expressions:

Table 14-1—List of valid operators in state dependent path delay expression

Operator	Description	Operator	Description
~	bit-wise negation	&	reduction and
&	bit-wise and		reduction or
	bit-wise or	^	reduction xor
^	bit-wise xor	~&	reduction nand
^~ ~^	bit-wise xnor	~	reduction nor
==	logical equality	^~ ~^	reduction xnor
!=	logical inequality	{ }	concatenation
&&	logical and	{ { } }	replication

Table 14-1—List of valid operators in state dependent path delay expression (*continued*)

Operator	Description	Operator	Description
	logical or	?:	conditional
!	logical not		

A conditional expression shall evaluate to true (1) for the state-dependent path to be assigned a delay value. If the conditional expression evaluates to x or z, it shall be treated as true. If the conditional expression evaluates to multiple bits, the least significant bit shall represent the result. The conditional expression can have any number of operands and operators.

14.2.4.2 Simple state-dependent paths

If the path description of a state-dependent path is a simple path, then it is called a *simple state-dependent path*. The simple path description is discussed in 14.2.2.

Examples:

Example 1—The following example uses state-dependent paths to describe the timing of an XOR gate.

```

module XORgate (a, b, out);
input a, b;
output out;

xor x1 (out, a, b);

specify
  specparam noninvrise = 1, noninvfall = 2
  specparam invertrise = 3, invertfall = 4;
  if (a) (b=> out) = (invertrise, invertfall);
  if (b) (a=> out) = (invertrise, invertfall);
  if (~a)(b=> out) = (noninvrise, noninvfall);
  if (~b)(a=> out) = (noninvrise, noninvfall);
endspecify
endmodule

```

In this example, first two state-dependent paths describe a pair of output rise and fall delay times when the XOR gate (x1) inverts a changing input. The last two state-dependent paths describe another pair of output rise and fall delay times when the XOR gate buffers a changing input.

Example 2—The following example models a partial ALU. The state-dependent paths specify different delays for different ALU operations.

```

module ALU (o1, i1, i2, opcode);
input [7:0] i1, i2;
input [2:1] opcode;
output [7:0] o1;

//functional description omitted
specify
    // add operation
    if (opcode == 2'b00) (i1,i2 *> o1) = (25.0, 25.0);
    // pass-through i1 operation
    if (opcode == 2'b01) (i1 => o1) = (5.6, 8.0);
    // pass-through i2 operation
    if (opcode == 2'b10) (i2 => o1) = (5.6, 8.0);
    // delays on opcode changes
    (opcode => o1) = (6.1, 6.5);
endspecify
endmodule

```

In the preceding example, the first three path declarations declare paths extending from operand inputs `i1` and `i2` to the `o1` output. The delays on these paths are assigned to operations on the basis of the operation specified by the inputs on `opcode`. The last path declaration declares a path from the `opcode` input to the `o1` output.

14.2.4.3 Edge-sensitive state-dependent paths

If the path description of a state-dependent path describes an edge-dependent path, then the state-dependent path is called an *edge-sensitive state-dependent path*. The edge-sensitive paths are discussed in 14.2.3.

Different delays can be assigned to the same edge-sensitive path as long as the following criteria are met:

- The edge, condition, or both make each declaration unique.
- The port is referenced in the same way in all path declarations (entire port, bit-select, or part-select).

Examples:

Example 1

```

if ( !reset && !clear )
    ( posedge clock => ( out +: in ) ) = (10, 8) ;

```

In this example, if the positive edge of `clock` occurs when `reset` and `clear` are low, and a module path extends from `clock` to `out` using a rise delay of 10 and a fall delay of 8.

Example 2—The following example shows four edge-sensitive path declarations. Note that each path has a unique edge or condition.

```

specify
  ( posedge clk => ( q[0] : data ) ) = (10, 5);
  ( negedge clk => ( q[0] : data ) ) = (20, 12);

  if (reset)
    ( posedge clk => ( q[0] : data ) ) = (15, 8);
  if (!reset && cntrl)
    ( posedge clk => ( q[0] : data ) ) = (6, 2);
endspecify

```

Example 3—The two state-dependent path declarations shown below are not legal because even though they have different conditions, the destinations are not specified in the same way: the first destination is a part-select, the second is a bit-select.

```

specify
  if (reset)
    ( posedge clk => (q[3:0]:data)) = (10,5);
  if (!reset)
    ( posedge clk => (q[0]:data)) = (15,8);
endspecify

```

14.2.4.4 The ifnone condition

The **ifnone** keyword is used to specify a default state-dependent path delay when all other conditions for the path are false. The **ifnone** condition shall specify the same module path source and destination as the state-dependent module paths. The following rules apply to module paths specified with the **ifnone** condition:

- Only simple module paths may be described with an **ifnone** condition.
- The state-dependent paths that correspond to the **ifnone** path may be either simple module paths or edge-sensitive paths.
- If there are no corresponding state-dependent module paths to the **ifnone** module path, then the **ifnone** module path shall be treated the same as an unconditional simple module path.
- It is illegal to specify both an **ifnone** condition for a module path and an unconditional simple module path for the same module path.

Examples:

Example 1—The following are valid state-dependent path combinations.

```

if (C1) (IN => OUT) = (1,1);
ifnone (IN => OUT) = (2,2);

// add operation
if (opcode == 2'b00) (i1,i2 *> o1) = (25.0, 25.0);
// pass-through i1 operation
if (opcode == 2'b01) (i1 => o1) = (5.6, 8.0);
// pass-through i2 operation
if (opcode == 2'b10) (i2 => o1) = (5.6, 8.0);
// all other operations
ifnone (i2 => o1) = (15.0, 15.0);

(posedge CLK => (Q +: D)) = (1,1);
ifnone (CLK => Q) = (2,2);

```

Example 2—The following module path description combination is illegal because it combines a state-dependent path using an **ifnone** condition and an unconditional path for the same module path.

```

if (a) (b=> out) = (2,2);
if (b) (a=> out) = (2,2);
ifnone (a => out) = (1,1);
(a => out) = (1,1);

```

14.2.5 Full connection and parallel connection paths

The operator **>* shall be used to establish a *full connection* between source and destination. In a full connection, every bit in the source shall connect to every bit in the destination. The module path source need not have the same number of bits as the module path destination.

The full connection can handle most types of module paths, since it does not restrict the size or number of source signals and destination signals. The following situations require the use of full connections:

- To describe a module path between a vector and a scalar
- To describe a module path between vectors of different sizes
- To describe a module path with multiple sources or multiple destinations in a single statement (see 14.2.6)

The operator *=>* shall be used to establish a *parallel connection* between source and destination. In a parallel connection, each bit in the source shall connect to one corresponding bit in the destination. Parallel module paths can be created only between sources and destinations that contain the same number of bits.

Parallel connections are more restrictive than full connections. They only connect one source to one destination, where each signal contains the same number of bits. Therefore, a parallel connection may only be used to describe a module path between two vectors of the same size. Since scalars are one bit wide, either **>* or *=>* may be used to set up bit-to-bit connections between two scalars.

Examples:

Example 1—Figure 14-2 illustrates how a parallel connection differs from a full connection between two 4-bit vectors.

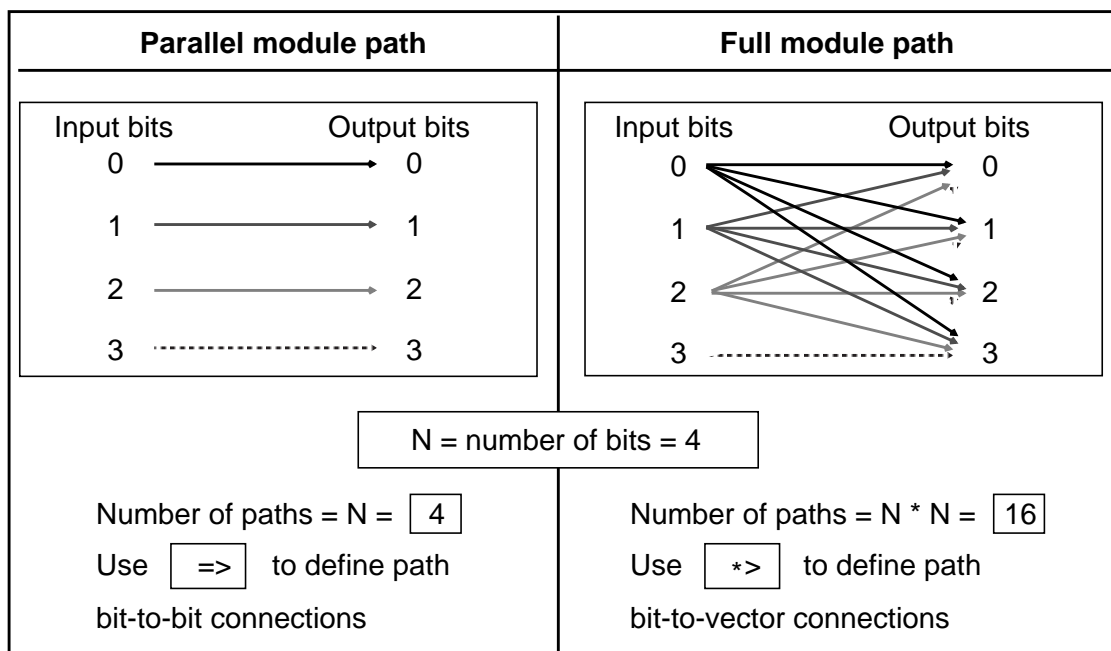


Figure 14-2—The difference between parallel and full connection paths

Example 2—The following example shows module paths for a 2 : 1 multiplexor with two 8-bit inputs and one 8-bit output.

```

module mux8 (in1, in2, s, q) ;
output [7:0] q;
input [7:0] in1, in2;
input s;
// Functional description omitted ...
specify
  (in1 => q) = (3, 4) ;
  (in2 => q) = (2, 3) ;
  (s *> q) = 1;
endspecify
endmodule

```

The module path from *s* to *q* uses a full connection (**>*) because it connects a scalar source—the 1-bit select line—to a vector destination—the 8-bit output bus. The module paths from both input lines *in1* and *in2* to *q* use a parallel connection (*=>*) because they set up parallel connections between two 8-bit buses.

14.2.6 Declaring multiple module paths in a single statement

Multiple module paths may be described in a single statement by using the symbol **>* to connect a comma-separated list of sources to a comma-separated list of destinations. When describing multiple module paths in one statement, the lists of sources and destinations may contain a mix of scalars and vectors of any size.

The connection in a multiple module path declaration is always a full connection.

Example:

```
(a, b, c *> q1, q2) = 10;
```

is equivalent to the following six individual module path assignments:

```
(a *> q1) = 10 ;  
(b *> q1) = 10 ;  
(c *> q1) = 10 ;  
(a *> q2) = 10 ;  
(b *> q2) = 10 ;  
(c *> q2) = 10 ;
```

14.2.7 Module path polarity

The polarity of a module path is an arbitrary specification indicating whether or not the direction of a signal transition is inverted as it propagates from the input to the output. This arbitrary polarity description does not affect the actual propagation of data or events through the model; how a rise or a fall at the source propagates to the destination depends on the internal logic of the module.

Module paths may specify any of three polarities:

- Unknown polarity
- Positive polarity
- Negative polarity

14.2.7.1 Unknown polarity

By default, module paths shall have *unknown polarity*—that is, a transition at the path source may propagate to the destination in an unpredictable way, as follows:

- A rise at the source may cause either a rise transition, a fall transition, or no transition at the destination.
- A fall at the source may cause either a rise transition, a fall transition, or no transition at the destination.

A module path specified either as a full connection or a parallel connection, but without a polarity operator + or −, shall be treated as a module path with unknown polarity.

14.2.7.2 Positive polarity

For module paths with *positive polarity*, any transition at the source may cause the same transition at the destination, as follows:

- A rise at the source may cause either a rise transition or no transition at the destination.
- A fall at the source may cause either a fall transition or no transition at the destination.

A module path with positive polarity shall be specified by prefixing the + polarity operator to => or *>.

14.2.7.3 Negative polarity

For module paths with *negative polarity*, any transition at the source may cause the opposite transition at the destination, as follows:

- A rise at the source may cause either a fall transition or no transition at the destination.
- A fall at the source may cause either a rise transition or no transition at the destination.

A module path with negative polarity shall be specified by prefixing the – polarity operator to => or *>.

Examples:

The following examples show each type of path polarity:

```
// Positive polarity
(In1 +=> q) = In_to_q ;
(s    +*> q) = s_to_q ;

// Negative polarity
(In1 -=> q) = In_to_q ;
(s    -*> q) = s_to_q ;

// Unknown polarity
(In1 => q) = In_to_q ;
(s    *> q) = s_to_q ;
```

14.3 Assigning delays to module paths

The delays that occur at the module outputs where paths terminate shall be specified by assigning delay values to the module path descriptions. The syntax for specifying delay values is shown in Syntax 14-6.

```
path_delay_value ::= (From Annex A - A.7.4)
    list_of_path_delay_expressions
    | ( list_of_path_delay_expressions )
list_of_path_delay_expressions ::=
    t_path_delay_expression
    | trise_path_delay_expression , tfall_path_delay_expression
    | trise_path_delay_expression , tfall_path_delay_expression , tz_path_delay_expression
    | t01_path_delay_expression , t10_path_delay_expression , t0z_path_delay_expression ,
      tz1_path_delay_expression , t1z_path_delay_expression , tz0_path_delay_expression
    | t01_path_delay_expression , t10_path_delay_expression , t0z_path_delay_expression ,
      tz1_path_delay_expression , t1z_path_delay_expression , tz0_path_delay_expression ,
      t0x_path_delay_expression , tx1_path_delay_expression , t1x_path_delay_expression ,
      tx0_path_delay_expression , txz_path_delay_expression , txx_path_delay_expression
t_path_delay_expression ::=
    path_delay_expression
```

Syntax 14-6—Syntax for path delay value

In module path delay assignments, a module path description (see 14.2) is specified on the left-hand side, and one or more delay values are specified on the right-hand side. The delay values may be optionally enclosed in a pair of parentheses. There may be one, two, three, six, or twelve delay values assigned to a module path, as described in 14.3.1. The delay values shall be constant expressions containing literals or specparams, and there may be a delay expression of the form min:typ:max.

Example:

```
specify
  // Specify Parameters
  specparam tRise_clk_q = 45:150:270, tFall_clk_q=60:200:350;
  specparam tRise_Control = 35:40:45, tFall_control=40:50:65;

  // Module Path Assignments
  (clk => q) = (tRise_clk_q, tFall_clk_q);
  (clr, pre *> q) = (tRise_control, tFall_control);
endspecify
```

In the example above, the specify parameters declared following the **specparam** keyword specify values for the module path delays. The module path assignments assign those module path delays to the module paths.

14.3.1 Specifying transition delays on module paths

Each path delay expression may be a single value—representing the typical delay—or a colon-separated list of three values—representing a *minimum*, *typical*, and *maximum* delay, in that order. If the path delay expression results in a negative value, it shall be treated as zero. Table 14-2 describes how different path delay values shall be associated with various transitions. The path delay expression names refer to the names used in Syntax 14-6.

Table 14-2—Associating path delay expressions with transitions

Transitions	Number of path delay expressions specified				
	1	2	3	6	12
0 -> 1	t	trise	trise	t01	t01
1 -> 0	t	tfall	tfall	t10	t10
0 -> z	t	trise	tz	t0z	t0z
z -> 1	t	trise	trise	tz1	tz1
1 -> z	t	tfall	tz	t1z	t1z
z -> 0	t	tfall	tfall	tz0	tz0
0 -> x	*	*	*	*	t0x
x -> 1	*	*	*	*	tx1
1 -> x	*	*	*	*	t1x
x -> 0	*	*	*	*	tx0
x -> z	*	*	*	*	txz
z -> x	*	*	*	*	tzx

* See 14.3.2.

Example:

```
// one expression specifies all transitions
(C => Q) = 20;
(C => Q) = 10:14:20;

// two expressions specify rise and fall delays
specparam tPLH1 = 12, tPHL1 = 25;
specparam tPLH2 = 12:16:22, tPHL2 = 16:22:25;
(C => Q) = ( tPLH1, tPHL1 ) ;
(C => Q) = ( tPLH2, tPHL2 ) ;

// three expressions specify rise, fall, and z transition delays
specparam tPLH1 = 12, tPHL1 = 22, tPz1 = 34;
specparam tPLH2 = 12:14:30, tPHL2 = 16:22:40, tPz2 = 22:30:34;
(C => Q) = (tPLH1, tPHL1, tPz1);
(C => Q) = (tPLH2, tPHL2, tPz2);

// six expressions specify transitions to/from 0, 1, and z
specparam t01 = 12, t10 = 16, t0z = 13,
           tz1 = 10, t1z = 14, tz0 = 34 ;
(C => Q) = ( t01, t10, t0z, tz1, t1z, tz0 ) ;
specparam T01 = 12:14:24, T10 = 16:18:20, T0z = 13:16:30 ;
specparam Tz1 = 10:12:16, T1z = 14:23:36, Tz0 = 15:19:34 ;
(C => Q) = ( T01, T10, T0z, Tz1, T1z, Tz0 ) ;

// twelve expressions specify all transition delays explicitly
specparam t01=10, t10=12, t0z=14, tz1=15, t1z=29, tz0=36,
           t0x=14, tx1=15, t1x=15, tx0=14, txz=20, tzx=30 ;
(c => Q) = (t01, t10, t0z, tz1, t1z, tz0,
           t0x, tx1, t1x, tx0, txz, tzx) ;
```

14.3.2 Specifying x transition delays

If the x transition delays are not explicitly specified, the calculation of delay values for x transitions is based on the following two pessimistic rules:

- Transitions from a known state to x shall occur as quickly as possible—that is, the shortest possible delay shall be used for any transition to x.
- Transitions from x to a known state shall take as long as possible—that is, the longest possible delay shall be used for any transition from x.

Table 14-3 presents the general algorithm for calculating delay values for x transitions, along with specific examples. The following two groups of x transitions are represented in the table:

- a) Transition from a known state s to x: s ' x
- b) Transition from x to a known state s: x ' s

Table 14-3—Calculating delays for x transitions

X transition	Delay value
General algorithm	
s -> x	minimum (s -> other known signals)
x -> s	maximum (other known signals -> s)
Specific transitions	
0 -> x	minimum (0 -> z delay, 0 -> 1 delay)
1 -> x	minimum (1 -> z delay, 1 -> 0 delay)
z -> x	minimum (z -> 1 delay, z -> 0 delay)
x -> 0	maximum (z -> 0 delay, 1 -> 0 delay)
x -> 1	maximum (z -> 1 delay, 0 -> 1 delay)
x -> z	maximum (1 -> z delay, 0 -> z delay)
Usage: (C => Q) = (5, 12, 17, 10, 6, 22) ;	
0 -> x	minimum (17, 5) = 5
1 -> x	minimum (6, 12) = 6
z -> x	minimum (10, 22) = 10
x -> 0	maximum (22, 12) = 22
x -> 1	maximum (10, 5) = 10
x -> z	maximum (6, 17) = 17

14.3.3 Delay selection

The simulator shall determine the proper delay to use when a specify path output must be scheduled to transition. There may be specify paths to the output from more than one input, and the simulator must decide which specify path to use.

The simulator shall do this by first determining which specify paths to the output are active. Active specify paths are those whose input has transitioned most recently in time, and which have either no condition or whose conditions are true. In the presence of simultaneous input transitions, it is possible for many specify paths to an output to be simultaneously active.

Once the active specify paths are identified, a delay must be selected from among them. This is done by comparing the correct delay for the specific transition being scheduled from each specify path, and choosing the smallest.

Examples:

Example 1:

```
(A => Y) = (6, 9);
(B => Y) = (5, 11);
```

For a Y transition from 0 to 1, if A transitioned more recently than B a delay of 6 will be chosen. But if B transitioned more recently than A, a delay of 5 will be chosen. And if the last time they transitioned A and B did so simultaneously, then the smallest of the two rise delays would be chosen, which is the rise delay from B of 5. The fall delay from A of 9 would be chosen if Y was instead to transition from 1 to 0.

Example 2:

```

if (MODE < 5) (A => Y) = (5, 9);
if (MODE < 4) (A => Y) = (4, 8);
if (MODE < 3) (A => Y) = (6, 5);
if (MODE < 2) (A => Y) = (3, 2);
if (MODE < 1) (A => Y) = (7, 7);

```

Anywhere from zero to five of these specify paths might be active depending upon the value of MODE. For instance, when MODE is 2 the first three specify paths are active. A rise transition would select a delay of 4, because that is the smallest rise delay among the first three. A fall transition would select a delay of 5, because that is the smallest fall delay among the first three.

14.4 Mixing module path delays and distributed delays

If a module contains module path delays and distributed delays (delays on primitive instances within the module), the larger of the two delays for each path shall be used.

Examples:

Example 1—Figure 14-3 illustrates a simple circuit modeled with a combination of distributed delays and path delays (only the D input to Q output path is illustrated). Here, the delay on the module path from input D to output Q = 22, while the sum of the distributed delays = 0 + 1 = 1. Therefore, a transition on Q caused by a transition on D will occur 22 time units after the transition on D.

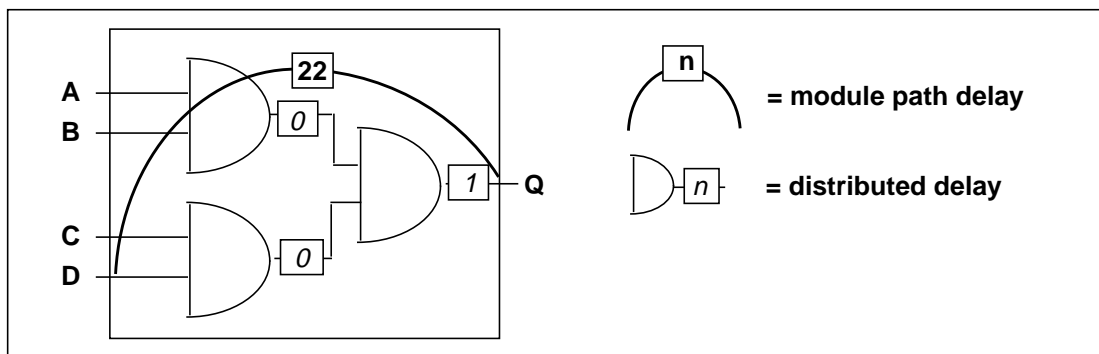


Figure 14-3—Module path delays longer than distributed delays

Example 2—In Figure 14-4, the delay on the module path from D to Q = 22, but the distributed delays along that module path now add up to 10 + 20 = 30. Therefore, an event on Q caused by an event on D will occur 30 time units after the event on D.

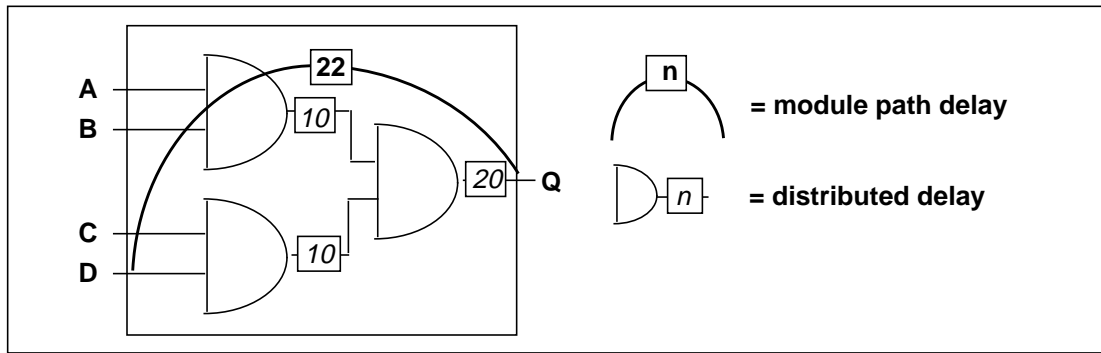


Figure 14-4—Module path delays shorter than distributed delays

14.5 Driving wired logic

Module path output nets shall not have more than one driver within the module. Therefore, wired logic is not allowed at module path outputs.

Figure 14-6 illustrates a violation of this wired-output rule and a method of avoiding the rule violation.

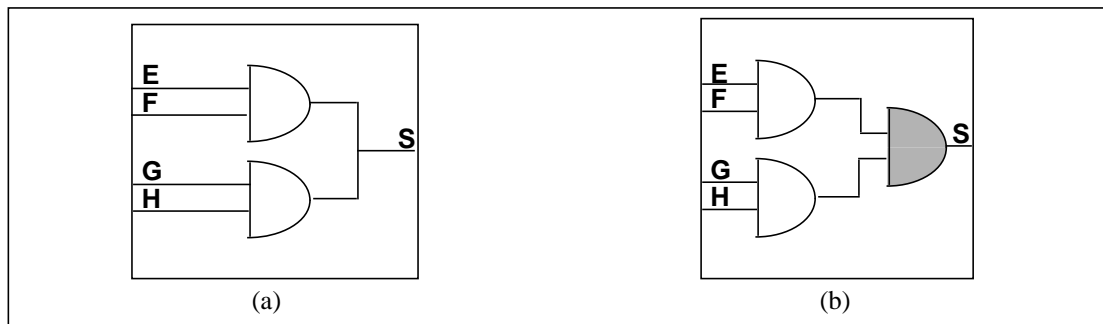
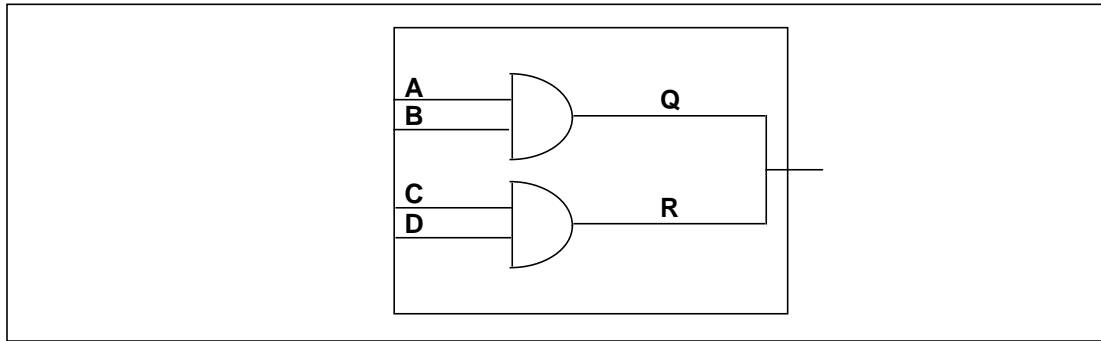


Figure 14-5—Legal and illegal module paths

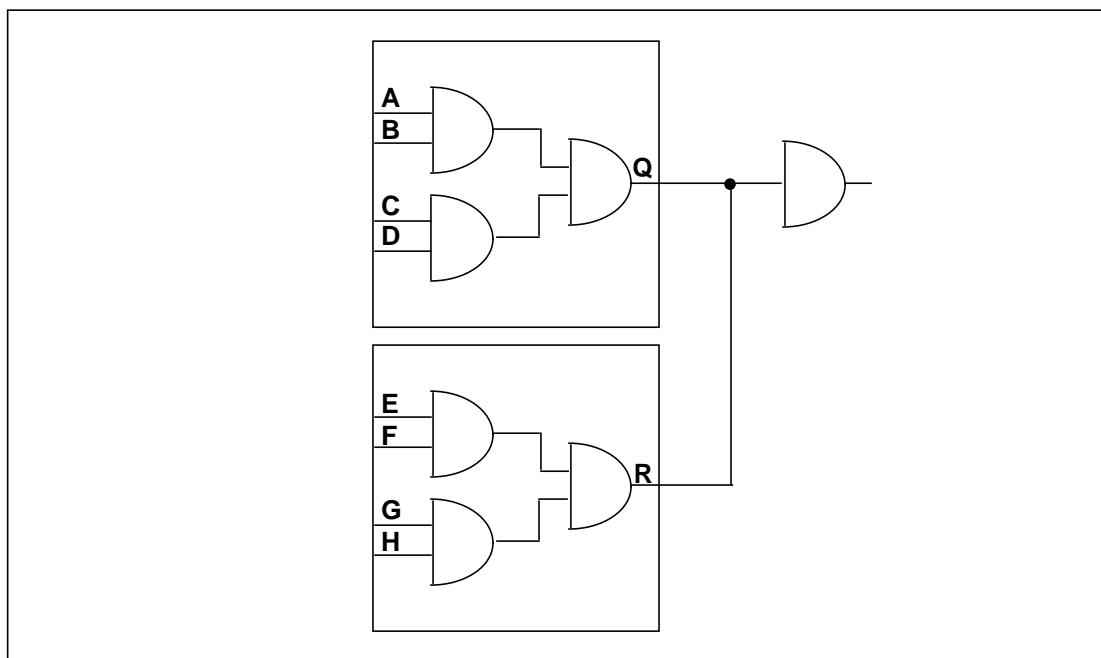
In Figure 14-5 (a), any module path to S is illegal because the path destination has two drivers.

Assuming signal S in Figure 14-5 (a) is a *wired-and*, this limitation can be circumvented by replacing wired logic with gated logic to create a single driver to the output. Figure 14-5 (b) shows how adding a third **and** gate—the shaded gate—solves the problem for the module in Figure 14-5 (a).

The example in Figure 14-6 is also illegal. In this example, when the outputs Q and R are wired together, it creates a condition where both paths have multiple drivers from within the same module.

**Figure 14-6—Illegal module paths**

Although multiple output drivers to a path destination are prohibited *inside* the same module, they are allowed *outside* the module. The example in Figure 14-7 is legal since Q and R each have only one driver within the module in which the module paths are specified.

**Figure 14-7—Legal module paths**

14.6 Detailed control of pulse filtering behavior

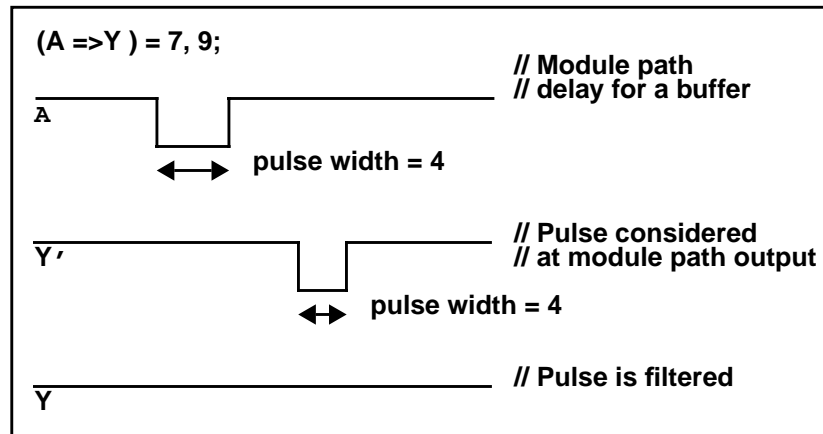
Two consecutive scheduled transitions closer together in time than the module path delay is deemed a pulse. By default, pulses on a module path output are rejected. Consecutive transitions cannot be closer together than the module path delay, and this is known as the inertial delay model of pulse propagation.

Pulse width ranges control how to handle a pulse presented at a module path output. They are:

- A pulse width range for which a pulse shall be rejected
- A pulse width range for which a pulse shall be allowed to propagate to the path destination
- A pulse width range for which a pulse shall generate a logic x on the path destination

Two pulse limit values define the pulse width ranges associated with each module path transition delay. The pulse limit values are called the error limit (e-limit) and the rejection limit (r-limit). The e-limit must always be at least as large as the r-limit. Pulses greater than or equal to the e-limit pass unfiltered. Pulses less than the e-limit but greater than or equal to the r-limit are filtered to X. Pulses less than the r-limit are rejected and no pulse emerges. By default, both the e-limit and the r-limit are set equal to the delay. These default values yield full inertial pulse behavior, rejecting all pulses smaller than the delay.

Example:



The rise delay from input A to output Y is 7, and the fall delay is 9. By default, the e-limit and the r-limit for the rise delay are both 7. The e-limit and the r-limit for the fall delay are both 9. The pulse limits associated with the delay forming the trailing edge of the pulse determine if and how the pulse should be filtered. Waveform Y' shows the waveform resulting from no pulse filtering. The width of the pulse is 2, which is less than the reject limit for the rise delay of 7, and so the pulse is filtered as shown in waveform Y.

There are three ways to modify the pulse limits from their default values. First, the Verilog language provides the **PATHPULSE\$** specparam to modify the pulse limits from their default values. Second, invocation options can specify percentages applying to all module path delays to form the corresponding e-limits and r-limits. Third, SDF annotation can individually annotate the e-limit and r-limit of each module path transition delay.

14.6.1 Specify block control of pulse limit values

Pulse limit values may be set from within the specify block with the **PATHPULSE\$** specparam. The syntax for using **PATHPULSE\$** to specify the reject limit and error limit values is given in Syntax 14-7.

```
pulse_control_specparam ::= (From Annex A - A.2.4)
    PATHPULSE$ = ( reject_limit_value [ , error_limit_value ] ) ;
    | PATHPULSE$specify_input_terminal_descriptor$specify_output_terminal_descriptor
      = ( reject_limit_value [ , error_limit_value ] ) ;
error_limit_value ::=
    limit_value
reject_limit_value ::=
    limit_value
limit_value ::=
    constant_mintypmax_expression
```

*Syntax 14-7—Syntax for **PATHPULSE\$** pulse control*

If only the reject limit value is specified, it shall apply to both the reject limit and the error limit.

The reject limit and error limit may be specified for a specific module path. When no module path is specified, the reject limit and error limit shall apply to all module paths defined in a module. If both path-specific **PATHPULSE\$** specparams and a non-path-specific **PATHPULSE\$** specparam appear in the same module, then the path-specific specparams shall take precedence for the specified paths.

The module path input terminals and output terminals shall conform to the rules for module path inputs and outputs, with the following restriction: the terminals may not be a bit-select or part-select of a vector.

When a module path declaration declares multiple paths, the **PATHPULSE\$** specparam shall only be specified for the first path input terminal and the first path output terminal. The reject limit and error limit specified shall apply to all other paths in the multiple path declaration. A **PATHPULSE\$** specparam which specifies anything other than the first path input and path output terminals shall be ignored.

Example:

In the following example, the path (clk=>q) acquires a reject limit of 2 and an error limit of 9, as defined by the first **PATHPULSE\$** declaration. The paths (clr*>q) and (pre*>q) receive a reject limit of 0 and an error limit of 4, as specified by the second **PATHPULSE\$** declaration. The path (data=>q) is not explicitly defined in any of the **PATHPULSE\$** declarations, and so it acquires reject and error limit of 3, as defined by the last **PATHPULSE\$** declaration.

```

specify
  (clk => q) = 12;
  (data => q) = 10;
  (clr, pre *> q) = 4;

  specparam
    PATHPULSE$clk$q = (2,9),
    PATHPULSE$clr$q = (0,4),
    PATHPULSE$ = 3;
endspecify

```

14.6.2 Global control of pulse limit values

Two invocation options can specify percentages applying globally to all module path transition delays. The error limit invocation option specifies the percentage of each module path transition delay used for its error limit value. The reject limit invocation option specifies the percentage of each module path transition delay used for its reject limit value. The percentage values shall be an integer between 0 and 100.

The default values for both the reject and error limit invocation options are 100%. When neither option is present then 100% of each module transition delay is used as the reject and error limits.

It is an error if the error limit percentage is smaller than the reject limit percentage. In such cases the error limit percentage is set equal to the reject limit percentage.

When both **PATHPULSE\$** and global pulse limit invocation options are present, the **PATHPULSE\$** values shall take precedence.

14.6.3 SDF annotation of pulse limit values

SDF annotation can be used to specify the pulse limit values of module path transition delays. Section 16 describes this in greater detail.

When both **PATHPULSE\$**, global pulse limit invocation options, and SDF annotation of pulse limit values are present, SDF annotation values shall take precedence.

14.6.4 Detailed pulse control capabilities

The default style of pulse filtering behavior has two drawbacks. First, pulse filtering to the X state may be insufficiently pessimistic with an X state duration too short to be useful. Second, unequal delays can result in pulse rejection whenever the trailing edge precedes the leading edge, leaving no indication that a pulse was rejected. This section introduces more detailed pulse control capabilities.

14.6.4.1 On-event versus on-detect pulse filtering

When an output pulse must be filtered to X, greater pessimism can be expressed if the module path output transitions immediately to X (on-detect) instead of at the already scheduled transition time of the leading edge of the pulse (on-event).

The on-event method of pulse filtering to X is the default. When an output pulse must be filtered to X, the leading edge of the pulse becomes a transition to X and the trailing edge a transition from X. The times of transition of the edges do not change.

Just like on-event, the on-detect method of pulse filtering changes the leading edge of the pulse into a transition to X and the trailing edge to a transition from X, but the time of the leading edge is changed to occur immediately upon detection of the pulse.

Figure 14-8 illustrates this behavior using a simple buffer with asymmetric rise/fall times and both the r-limits and e-limits equal to 0. An output waveform is shown for both on-detect and on-event approaches.

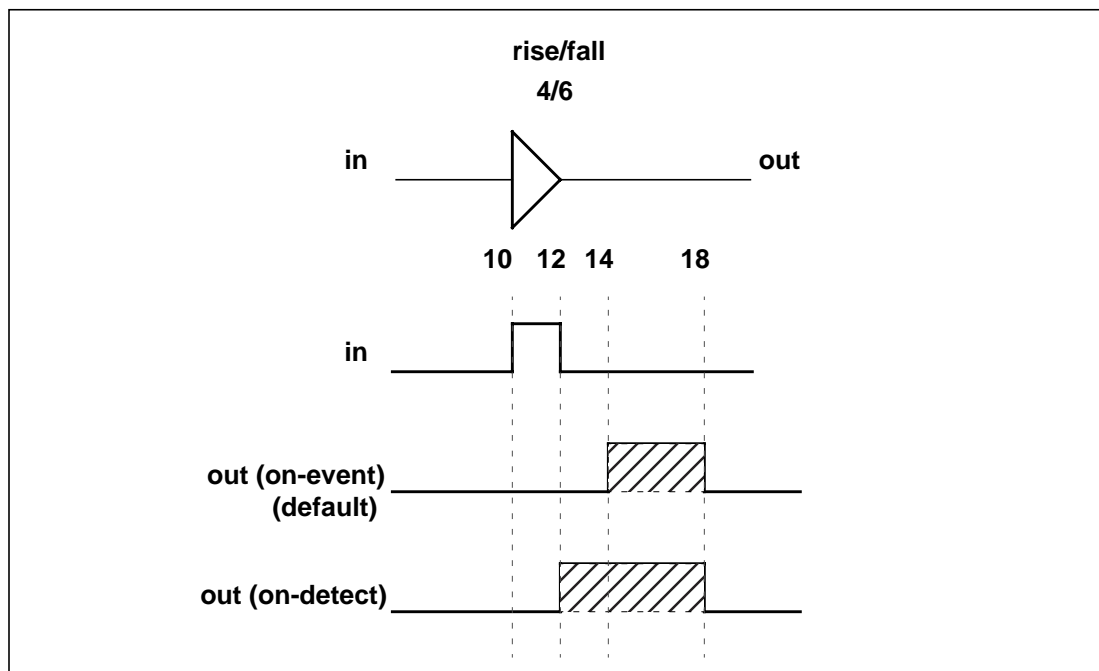


Figure 14-8—On-detect -vs.- on-event

On-detect versus on-event behavior can be selected in two different ways. First, one may be selected globally for all module path outputs through use of the on-detect or on-event invocation option. Second, one may be selected locally through use of specify block pulse style declarations.

The syntax for pulse style declarations is shown in Syntax 14-8.

```
pulsestyle_declaration ::= (From Annex A- A.7.1)
    pulsestyle_oneevent list_of_path_outputs ;
    | pulsestyle_ondetect list_of_path_outputs ;
```

Syntax 14-8—Syntax for pulse style declarations

It is an error if a module path output appears in a pulse style declaration after it has already appeared in a module path declaration.

The pulse style invocation options take precedence over pulse style specify block declarations.

14.6.4.2 Negative pulse detection

When the delays to a module path output are unequal, it is possible for the trailing edge of a pulse to be scheduled for a time earlier than the schedule time of the leading edge, yielding a pulse with a negative width. Under normal operation, if the schedule for a trailing pulse edge is earlier than the schedule for a leading pulse edge, then the leading edge is cancelled. No transition takes place when the initial and final states of the pulse are the same, leaving no indication a schedule was ever present.

Negative pulses can be indicated with the X state by use of the *showcancelled* style of behavior. When the trailing edge of a pulse would be scheduled before the leading edge, this style causes the leading edge to be scheduled to X, and the trailing edge to be scheduled from X. With on-event pulse style, the schedule to X replaces the leading edge schedule. With on-detect pulse style, the schedule to X is made immediately upon detection of the negative pulse.

showcancelled behavior can be enabled in two different ways. First, it may be enabled globally for all module path outputs through use of the **showcancelled** and **noshowcancelled** invocation options. Second, it may be enabled locally through use of specify block **showcancelled** declarations.

The syntax for *showcancelled* declarations is shown in Syntax 14-9.

```
showcancelled_declaration ::= (From Annex A- A.7.1)
    showcancelled list_of_path_outputs ;
    | noshowcancelled list_of_path_outputs ;
```

Syntax 14-9—Syntax for showcancelled declarations

It is an error if a module path output appears in a *showcancelled* declaration after it has already appeared in a module path declaration. The *showcancelled* invocation options take precedence over the *showcancelled* specify block declarations.

The *showcancelled* behavior is illustrated in Figure 14-9, which shows a narrow pulse presented at the input to a buffer with unequal rise/fall delays. This causes the trailing edge of the pulse to be scheduled earlier than leading edge. The leading edge of the input pulse schedules an output event 6 units later at the point marked by A. The pulse trailing edge occurs one time unit later, which schedules an output event 4 units later marked by point B. This second schedule on the output is for a time prior to the already existing schedule for the leading output pulse edge.

The output waveform is shown for three different operating modes. The first waveform shows the default behavior with *showcancelled* behavior not enabled and with the default on-event style. The waveform shows *showcancelled*

behavior in conjunction with on-event. The waveform shows showcancelled behavior in conjunction with on-detect.

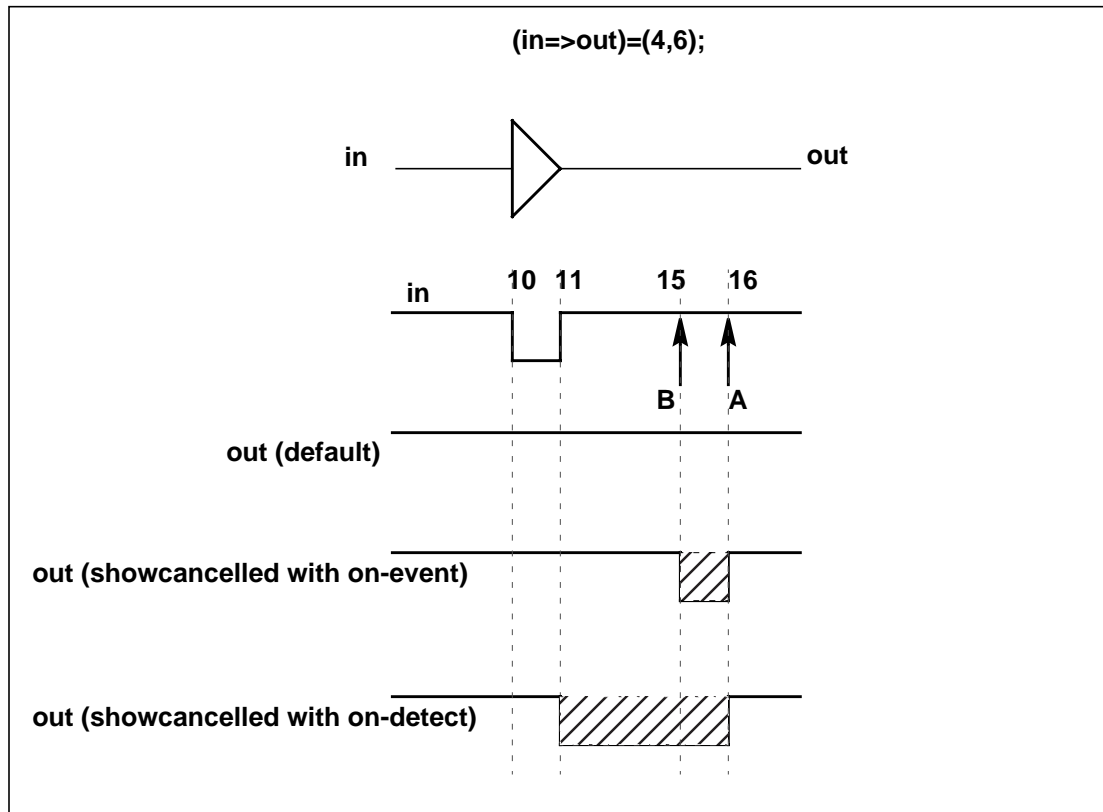
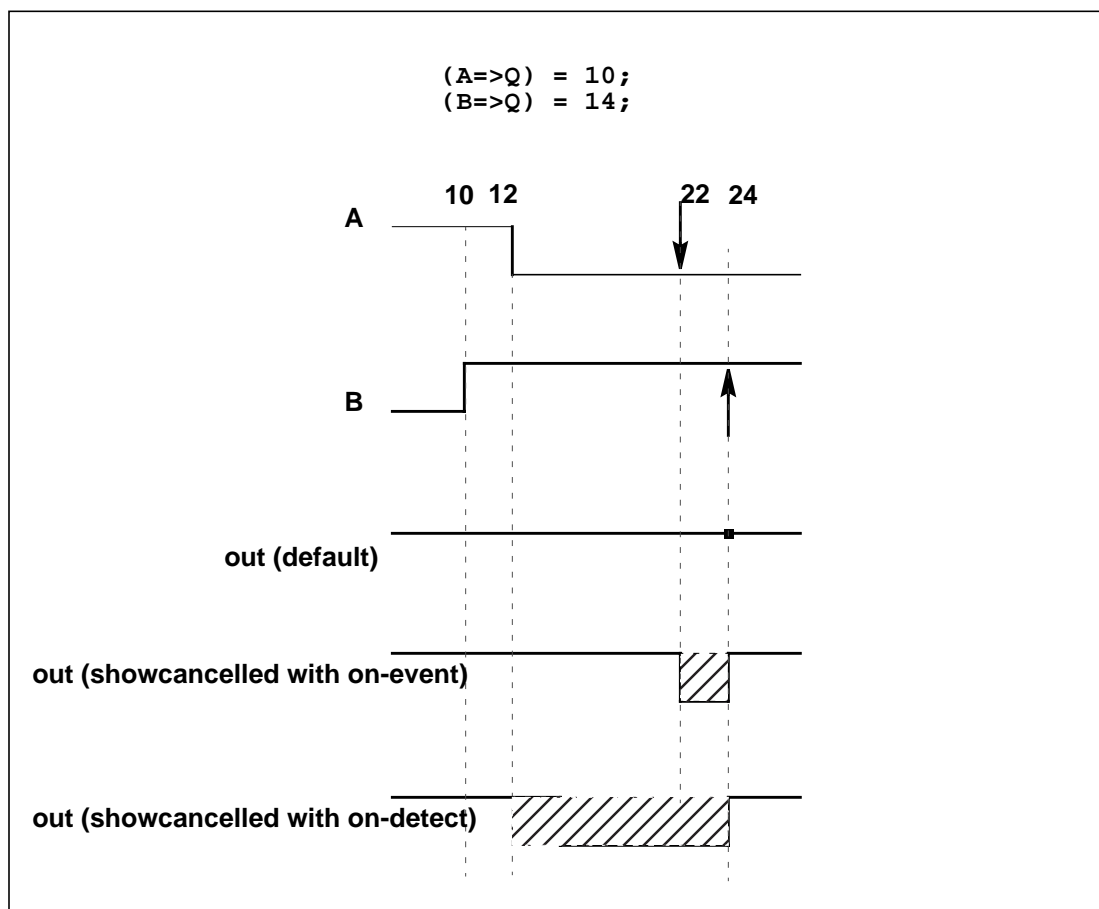


Figure 14-9—Current event cancellation problem and correction

This same situation can also arise with nearly simultaneous input transitions, which is defined as two inputs transitioning closer together in time than the difference in their respective delays to the output. Figure 14-10 shows waveforms for a 2-input NAND gate where initially A is high and B is low. B transitions 0→1 at time 10, causing a 1→0 output schedule at time 24. A transitions 1→0 at time 12, causing a 0→1 schedule at time 22. Arrows mark the output transitions caused by the transitions on inputs A and B.

The output waveform is shown for three different operating modes. The first waveform shows the default behavior with showcancelled behavior not enabled and with the default on-event style. The second shows showcancelled behavior in conjunction with on-event. The third shows showcancelled behavior in conjunction with on-detect.



**Figure 14-10—NAND gate with nearly simultaneous input switching
where one event is scheduled prior to another that has not matured**

One drawback of the on-event style with showcancelled behavior is that as the output pulse edges draw closer together, the duration of the resulting X state becomes smaller. Figure 14-11 illustrates how the on-detect style solves this problem.

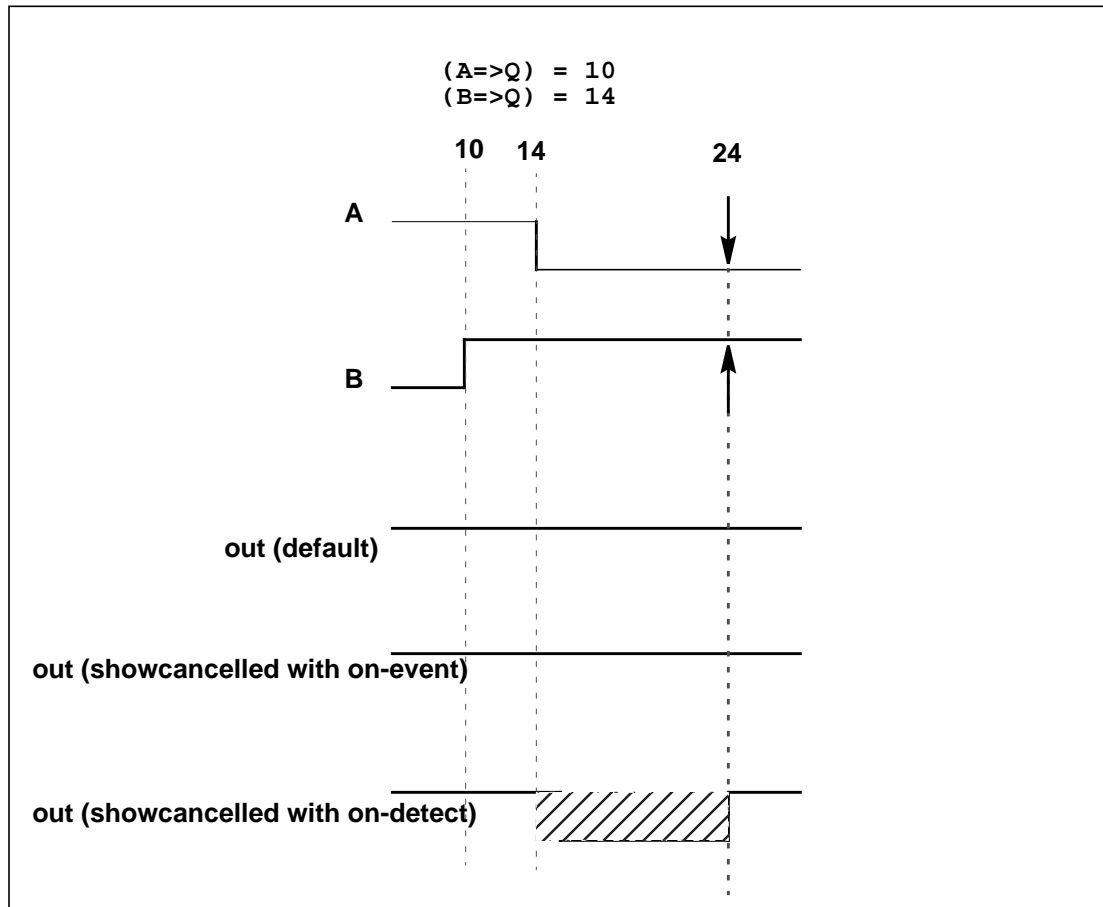


Figure 14-11—Input NAND gate with nearly simultaneous input switching with output event scheduled at same time.

Examples:

Example 1:

```

specify
  (a=>out)=(2,3);
  (b =>out)=(3,4);
endspecify

```

Since no pulse style or showcancelled declarations appear within the specify block, the compiler applies the default modes of on-event and noshowcancelled.

Example 2:

```

specify
  (a=>out)=(2,3);
  showcancelled out;
  (b =>out)=(3,4);
endspecify

```

This showcancelled declaration is in error because it follows use of out in a module path declaration. It would be contradictory for out to have noshowcancelled behavior from input a, but showcancelled behavior from input b.

Example 2—Both these specify blocks produce the same result. Outputs `out` and `out_b` are both declared `showcancelled` and `on_detect`.

```
specify
  showcanceled out;
  pulsestyle_ondetect out;
  (a =>out)=(2,3);
  (a=>out)=(4,5);
  showcanceled out_b;
  pulsestyle_ondetect out_b;
  (b=>out_b)=(5,6);
  (b=>out_b)=(3,4);
endspecify

specify
  showcanceled out,out_b;
  pulsestyle_ondetect out,out_b;
  (a =>out)=(2,3);
  (b=>out)=(3,4);
  (a =>out_b)=(3,4);
  (b=>out_b)=(5,6);
endspecify
```


Section 15

Timing checks

This section describes how timing checks are used in specify blocks to determine if signals obey the timing constraints.

15.1 Overview

Timing checks can be placed in specify blocks to verify the timing performance of a design by making sure critical events occur within given time limits. The syntax for system timing checks is given in Syntax 15-1.

```

system_timing_check ::= (From Annex A - A.7.5.1)
    $setup_timing_check
    | $hold_timing_check
    | $setuphold_timing_check
    | $recovery_timing_check
    | $removal_timing_check
    | $recrem_timing_check
    | $skew_timing_check
    | $timeskew_timing_check
    | $fullskew_timing_check
    | $period_timing_check
    | $width_timing_check
    | $nochange_timing_check

$setup_timing_check ::=
    $setup ( data_event , reference_event , timing_check_limit [ , [ notify_reg ] ] );

$hold_timing_check ::=
    $hold ( reference_event , data_event , timing_check_limit [ , [ notify_reg ] ] );

$setuphold_timing_check ::=
    $setuphold ( reference_event , data_event , timing_check_limit , timing_check_limit
        [ , [ notify_reg ] [ , [ stamptime_condition ] [ , [ checktime_condition ]
        [ , [ delayed_reference ] [ , [ delayed_data ] ] ] ] ] );

$recovery_timing_check ::=
    $recovery ( reference_event , data_event , timing_check_limit [ , [ notify_reg ] ] );

$removal_timing_check ::=
    $removal ( reference_event , data_event , timing_check_limit [ , [ notify_reg ] ] );

$recrem_timing_check ::=
    $recrem ( reference_event , data_event , timing_check_limit , timing_check_limit
        [ , [ notify_reg ] [ , [ stamptime_condition ] [ , [ checktime_condition ]
        [ , [ delayed_reference ] [ , [ delayed_data ] ] ] ] ] );

$skew_timing_check ::=
    $skew ( reference_event , data_event , timing_check_limit [ , [ notify_reg ] ] );

$timeskew_timing_check ::=
    $timeskew ( reference_event , data_event , timing_check_limit
        [ , [ notify_reg ] [ , [ event_based_flag ] [ , [ remain_active_flag ] ] ] ] );

$fullskew_timing_check ::=
    $fullskew ( reference_event , data_event , timing_check_limit , timing_check_limit
        [ , [ notify_reg ] [ , [ event_based_flag ] [ , [ remain_active_flag ] ] ] ] );

$period_timing_check ::=
    $period ( controlled_reference_event , timing_check_limit [ , [ notify_reg ] ] );

$width_timing_check ::=
    $width ( controlled_reference_event , timing_check_limit ,
        threshold [ , [ notify_reg ] ] );

$nochange_timing_check ::=
    $nochange ( reference_event , data_event , start_edge_offset ,
        end_edge_offset [ , [ notify_reg ] ] );

```

Syntax 15-1—Syntax for system timing checks

The syntax for check time conditions and timing check events are given in Syntax 15-2.

```

checktime_condition ::= (From Annex A - A.7.5.2)
    mintypmax_expression
controlled_reference_event ::=
    controlled_timing_check_event
data_event ::=
    timing_check_event
delayed_data ::=
    terminal_identifier
    | terminal_identifier [ constant_mintypmax_expression ]
delayed_reference ::=
    terminal_identifier
    | terminal_identifier [ constant_mintypmax_expression ]
end_edge_offset ::= mintypmax_expression
event_based_flag ::= constant_expression
notify_reg ::= variable_identifier
reference_event ::= timing_check_event
remain_active_flag ::= constant_mintypmax_expression
stamp_time_condition ::= mintypmax_expression
start_edge_offset ::= mintypmax_expression
threshold ::= constant_expression
timing_check_limit ::= expression
timing_check_event ::= (From Annex A - A.7.5.3)
    [timing_check_event_control] specify_terminal_descriptor [ &&& timing_check_condition ]
controlled_timing_check_event ::=
    timing_check_event_control specify_terminal_descriptor [ &&& timing_check_condition ]
timing_check_event_control ::= posedge | negedge | edge_control_specifier
specify_terminal_descriptor ::=
    specify_input_terminal_descriptor
    | specify_output_terminal_descriptor
edge_control_specifier ::= edge [ edge_descriptor [ , edge_descriptor ] ]
edge_descriptor1 ::= 01 | 10 | z_or_x zero_or_one | zero_or_one z_or_x
zero_or_one ::= 0 | 1
z_or_x ::= x | X | z | Z
timing_check_condition ::=
    scalar_timing_check_condition
    | ( scalar_timing_check_condition )
scalar_timing_check_condition ::=
    expression
    | ~ expression
    | expression == scalar_constant
    | expression === scalar_constant
    | expression != scalar_constant
    | expression !== scalar_constant
scalar_constant ::= 1'b0 | 1'b1 | 1'B0 | 1'B1 | 'b0 | 'b1 | 'B0 | 'B1 | 1 | 0

```

¹Embedded spaces are illegal.

Syntax 15-2—Syntax for check time conditions and timing check events

For ease of presentation, the timing checks are divided into two groups. The first group of timing checks are described

in terms of stability time windows:

\$setup	\$hold	\$setuphold
\$recovery	\$removal	\$recrem

The timing checks in the second group check clock and control signals, and are described in terms of the difference in time between two events (the **\$nochange** check involves three events):

\$skew	\$timeskew	\$fullskew
\$width	\$period	\$nochange

Although they begin with a \$, timing checks are not system tasks. The leading \$ is present because of historical reasons, and timing checks shall not be confused with system tasks. In particular, no system task can appear in a specify block, and no timing check can appear in procedural code.

Some timing checks can accept negative limit values. This topic is discussed in detail in 15.8.

All timing checks have both a reference event and a data event, and boolean conditions can be associated with each. Some of the checks have two signal arguments, one of which is the reference event and the other the data event. Other checks have only one signal argument, and the reference and data events are derived from it. Reference events and data events shall only be detected by timing checks when their associated conditions are true. See 15.6 for more information about conditions in timing checks.

Timing check evaluation is based upon the times of two events, called the timestamp event and the timecheck event. A transition on the timestamp event signal causes the simulator to record (stamp) the time of transition for future use in evaluating the timing check. A transition on the timecheck event signal causes the simulator to actually evaluate the timing check to determine whether a violation has taken place.

For some checks the reference event is always the timestamp event, while the data event is always the timecheck event, while for other checks the reverse is true. And for yet other checks the decision as to which is the timestamp and which the timecheck event is based upon factors to be discussed later in greater detail.

Every timing check can include an optional notifier which toggles whenever the timing check detects a violation. The model can use the notifier to make behavior a function of timing check violations. Notifiers are discussed in greater detail in 15.5.

Like expressions for module path delays, timing check limit values are constant expressions which can include spec-params.

15.2 Timing checks using a stability window

These timing checks are discussed in this section:

\$setup	\$hold	\$setuphold
\$recovery	\$removal	\$recrem

These checks accept two signals, the reference event and the data event, and define a time window with respect to one signal while checking the time of transition of the other signal with respect to the window. In general they all perform the following steps:

- Define a time window with respect to the reference signal using the specified limit or limits;
- Check the time of transition of the data signal with respect to the time window;
- Report a timing violation if the data signal transitions within the time window.

15.2.1 \$setup

The **\$setup** timing check syntax is shown in Syntax 15-3.

```

$setup_timing_check ::= (From Annex A - A.7.5.1)
$setup ( data_event , reference_event , timing_check_limit [ , [ notify_reg ] ] ) ;
data_event ::= (From Annex A - A.7.5.2)
    timing_check_event
notify_reg ::=
    variable_identifier
reference_event ::=
    timing_check_event
timing_check_limit ::=
    expression

```

Syntax 15-3—Syntax for \$setup

Table 15-1 defines the **\$setup** timing check.

Table 15-1—\$setup arguments

Argument	Description
data_event	Timestamp event
reference_event	Timecheck event
limit	Non-negative constant expression
notifier (optional)	Reg

The data event is usually a data signal, while the reference event is usually a clock signal.

The endpoints of the time window are determined as follows:

```

(beginning of time window) = (timecheck time) - limit
(end of time window) = (timecheck time)

```

The **\$setup** timing check reports a timing violation in the following case:

```

(beginning of time window) < (timestamp time) < (end of time window)

```

The endpoints of the time window are not part of the violation region. When the limit is zero, the **\$setup** check shall never issue a violation.

15.2.2 \$hold

The **\$hold** timing check syntax is shown in Syntax 15-4.

```

$hold_timing_check ::= (From Annex A - A.7.5.1)
$hold ( reference_event , data_event , timing_check_limit [ , [ notify_reg ] ] ) ;
data_event ::= (From Annex A - A.7.5.2)
    timing_check_event
notify_reg ::=
    variable_identifier
reference_event ::=
    timing_check_event
timing_check_limit ::=
    expression

```

Syntax 15-4—Syntax for \$hold

Table 15-2 defines the **\$hold** timing check.

Table 15-2—\$hold arguments

Argument	Description
reference_event	Timestamp event
data_event	Timecheck event
limit	Non-negative constant expression
notifier (optional)	Reg

The data event is usually a data signal, while the reference event is usually a clock signal.

The endpoints of the time window are determined as follows:

```

(beginning of time window) = (timestamp time)
(end of time window) = (timestamp time) + limit

```

The **\$hold** timing check reports a timing violation in the following case:

```

(beginning of time window) <= (timecheck time) < (end of time window)

```

Only the end of the time window is not part of the violation region. When the limit is zero, the **\$hold** check shall never issue a violation.

15.2.3 \$setuphold

The **\$setuphold** timing check syntax is shown in Syntax 15-5.


```

$setuphold_timing_check ::= (From Annex A - A.7.5.1)
    $setuphold ( reference_event , data_event , timing_check_limit , timing_check_limit
        [ , [ notify_reg ] [ , [ stamptime_condition ] [ , [ checktime_condition ]
            [ , [ delayed_reference ] [ , [ delayed_data ] ] ] ] ] ) ;
checktime_condition ::= (From Annex A - A.7.5.2)
    mintypmax_expression
data_event ::=
    timing_check_event
delayed_data ::=
    terminal_identifier
    | terminal_identifier [ constant_mintypmax_expression ]
delayed_reference ::=
    terminal_identifier
    | terminal_identifier [ constant_mintypmax_expression ]
notify_reg ::=
    variable_identifier
reference_event ::=
    timing_check_event
stamptime_condition ::=
    mintypmax_expression
timing_check_limit ::=
    expression

```

Syntax 15-5—Syntax for \$setuphold

Table 15-3 defines the **\$setuphold** timing check.

Table 15-3—\$setuphold arguments

Argument	Description
reference_event	Timecheck or timestamp event when setup limit is positive Timestamp event when setup limit is negative
data_event	Timecheck or timestamp event when hold limit is positive Timestamp event when hold limit is negative
setup_limit	Constant expression
hold_limit	Constant expression
notifier (optional)	Reg
timestamp_cond (optional)	Timestamp condition for negative timing checks
timecheck_cond (optional)	Timecheck condition for negative timing checks
delayed_reference (optional)	Delayed reference signal for negative timing checks
delayed_data (optional)	Delayed data signal for negative timing checks

The **\$setuphold** timing check can accept negative limit values. This is discussed in greater detail in 15.8.

The data event is usually a data signal, while the reference event is usually a clock signal.

When both the setup limit and the hold limit are positive, either the reference event or the data event can be the timecheck event. It shall depend upon which occurs first in the simulation.

When either the setup limit or the hold limit is negative the restriction becomes:

$$\text{setup_limit} + \text{hold_limit} > (\text{simulation unit of precision})$$

The **\$setuphold** timing check combines the functionality of the **\$setup** and **\$hold** timing checks into a single timing check. Therefore, the following invocation:

```
$setuphold( posedge clk, data, tSU, tHLD );
```

is equivalent in functionality to the following, if tSU and tHLD are not negative:

```
$setup( data, posedge clk, tSU );  
$hold( posedge clk, data, tHLD );
```

When both setup and hold limits are positive and the data event occurs first, the endpoints of the time window are determined as follows:

```
(beginning of time window) = (timecheck time) - limit  
(end of time window) = (timecheck time)
```

And the **\$setuphold** timing check reports a timing violation in the following case:

$$(\text{beginning of time window}) < (\text{timecheck time}) \leq (\text{end of time window})$$

Only the beginning of the time window is not part of the violation region. The **\$setuphold** check shall report a timing violation when the reference and data events occur simultaneously.

When both setup and hold limits are positive and the data event occurs second, the endpoints of the time window are determined as follows:

```
beginning of time window) = (timestamp time)  
(end of time window) = (timestamp time) + limit
```

And the **\$setuphold** timing check reports a timing violation in the following case:

$$(\text{beginning of time window}) \leq (\text{timecheck time}) < (\text{end of time window})$$

Only the end of the time window is not part of the violation region. The **\$setuphold** check shall report a timing violation when the reference and data events occur simultaneously.

When both limits are zero, the **\$setuphold** check shall never issue a violation.

15.2.4 \$removal

The **\$removal** timing check syntax is shown in Syntax 15-6.

```

$removal_timing_check ::= (From Annex A - A.7.5.1)
$removal ( reference_event , data_event , timing_check_limit [ , [ notify_reg ] ] ) ;
data_event ::= (From Annex A - A.7.5.2)
    timing_check_event
notify_reg ::=
    variable_identifier
reference_event ::=
    timing_check_event
timing_check_limit ::=
    expression

```

Syntax 15-6—Syntax for \$removal

Table 15-4 defines the **\$removal** timing check.

Table 15-4—\$removal arguments

Argument	Description
reference_event	Timestamp event
data_event	Timecheck event
limit	Non-negative constant expression
notifier (optional)	Reg

The reference event is usually a control signal like clear, reset or set, while the data event is usually a clock signal.

The endpoints of the time window are determined as follows:

```

(beginning of time window) = (timecheck time) - limit
(end of time window) = (timecheck time)

```

The **\$removal** timing check reports a timing violation in the following case:

```

(beginning of time window) < (timestamp time) < (end of time window)

```

The endpoints of the time window are not part of the violation region. When the limit is zero, the **\$removal** check shall never issue a violation.

15.2.5 \$recovery

The **\$recovery** timing check syntax is shown in Syntax 15-7.

```

$recovery_timing_check ::= (From Annex A - A.7.5.1)
    $recovery ( reference_event , data_event , timing_check_limit [ , [ notify_reg ] ] ) ;
data_event ::= (From Annex A - A.7.5.2)
    timing_check_event
notify_reg ::=
    variable_identifier
reference_event ::=
    timing_check_event
timing_check_limit ::=
    expression

```

Syntax 15-7—Syntax for \$recovery

Table 15-5 defines the **\$recovery** timing check.

Table 15-5—\$recovery arguments

Argument	Description
reference_event	Timestamp event
data_event	Timecheck event
limit	Non-negative constant expression
notifier (optional)	Reg

The reference event is usually a control signal like clear, reset or set, while the data event is usually a clock signal.

The endpoints of the time window are determined as follows:

```

(beginning of time window) = (timestamp time)
(end of time window) = (timestamp time) + limit

```

The **\$recovery** timing check reports a timing violation in the following case:

```

(beginning of time window) <= (timecheck time) < (end of time window)

```

Only the end of the time window is not part of the violation region. When the limit is zero, the **\$recovery** check shall never issue a violation.

15.2.6 \$recrem

The **\$recrem** timing check syntax is shown in Syntax 15-8.

```

$recrem_timing_check ::= (From Annex A - A.7.5.1)
    $recrem ( reference_event , data_event , timing_check_limit , timing_check_limit
        [ , [ notify_reg ] [ , [ stamptime_condition ] [ , [ checktime_condition ]
            [ , [ delayed_reference ] [ , [ delayed_data ] ] ] ] ] ) ;
checktime_condition ::= (From Annex A - A.7.5.2)
    mintypmax_expression
data_event ::=
    timing_check_event
delayed_data ::=
    terminal_identifier
    | terminal_identifier [ constant_mintypmax_expression ]
delayed_reference ::=
    terminal_identifier
    | terminal_identifier [ constant_mintypmax_expression ]
notify_reg ::=
    variable_identifier
reference_event ::=
    timing_check_event
stamptime_condition ::=
    mintypmax_expression
timing_check_limit ::=
    expression

```

Syntax 15-8—Syntax for \$recrem

Table 15-6 defines the **\$recrem** timing check.

Table 15-6—\$recrem arguments

Argument	Description
reference_event	Timecheck or timestamp event when removal limit is positive Timestamp event when removal limit is negative
data_event	Timecheck or timestamp event when recovery limit is positive Timestamp event when recovery limit is negative
recovery_limit	Constant expression
removal_limit	Constant expression
notifier (optional)	Reg
timestamp_cond (optional)	Timestamp condition for negative timing checks
timecheck_cond (optional)	Timecheck condition for negative timing checks
delayed_reference (optional)	Delayed reference signal for negative timing checks
delayed_data (optional)	Delayed data signal for negative timing checks

The **\$recrem** timing check can accept negative limit values. This is discussed in greater detail in 15.8.

When both the removal limit and the recovery limit are positive, either the reference event or the data event can be the timecheck event. It shall depend upon which occurs first in the simulation.

When either the removal limit or the recovery limit is negative the restriction becomes:

$$\text{removal_limit} + \text{recovery_limit} > (\text{simulation unit of precision})$$

The **\$recrem** timing check combines the functionality of the **\$removal** and **\$recovery** timing checks into a single timing check. Therefore, the following invocation:

```
$recrem( posedge clear, posedge clk, tREC, tREM );
```

is equivalent in functionality to the following, if tREC and tREM are not negative:

```
$removal( posedge clear, posedge clk, tREM );
$recovery( posedge clear, posedge clk, tREC );
```

When both removal and recovery limits are positive and the data event occurs first, the endpoints of the time window are determined as follows:

$$\begin{aligned} (\text{beginning of time window}) &= (\text{timecheck time}) - \text{limit} \\ (\text{end of time window}) &= (\text{timecheck time}) \end{aligned}$$

And the **\$recrem** timing check reports a timing violation in the following case:

$$(\text{beginning of time window}) < (\text{timecheck time}) \leq (\text{end of time window})$$

Only the beginning of the time window is not part of the violation region. The **\$recrem** check shall report a timing violation when the reference and data events occur simultaneously.

When both removal and recovery limits are positive and the data event occurs second, the endpoints of the time window are determined as follows:

$$\begin{aligned} (\text{beginning of time window}) &= (\text{timestamp time}) \\ (\text{end of time window}) &= (\text{timestamp time}) + \text{limit} \end{aligned}$$

And the **\$recrem** timing check reports a timing violation in the following case:

$$(\text{beginning of time window}) \leq (\text{timecheck time}) < (\text{end of time window})$$

Only the end of the time window is not part of the violation region. The **\$recrem** check shall report a timing violation when the reference and data events occur simultaneously.

When both limits are zero, the **\$setuphold** check shall never issue a violation.

15.3 Timing checks for clock and control signals

The following timing checks are discussed in this section:

\$skew **\$timeskew** **\$fullskew** **\$period** **\$width** **\$nochange**

These checks accept one or two signals and verify transitions on them are never separated by more than the limit. For those checks specifying only one signal, the reference event and data event are derived from that one signal. In general these checks all perform the following steps:

- a) Determine the elapsed time between two events;
- b) Compare the elapsed time to the specified limit;

- c) Report a timing violation if the elapsed time violates the limit.

The skew checks have two different violation detection mechanisms, event-based and timer-based. Event-based skew checking is performed only when a signal transitions, while timer-based skew checking takes place as soon as the simulation time equal to the skew limit has elapsed.

The **\$nochange** check involves three events rather than two.

15.3.1 \$skew

The **\$skew** timing check syntax is shown in Syntax 15-9.

```
$skew_timing_check ::= (From Annex A - A.7.5.1)
    $skew ( reference_event , data_event , timing_check_limit [ , [ notify_reg ] ] ) ;
data_event ::= (From Annex A - A.7.5.2)
    timing_check_event
notify_reg ::=
    variable_identifier
reference_event ::=
    timing_check_event
timing_check_limit ::=
    expression
```

Syntax 15-9—Syntax for \$skew

Table 15-7 defines the **\$skew** timing check.

Table 15-7—\$skew arguments

Argument	Description
reference_event	Timestamp event
data_event	Timecheck event
limit	Non-negative constant expression
notifier (optional)	Reg

The **\$skew** timing check reports a violation in the following case:

$$(\text{timecheck time}) - (\text{timestamp time}) > \text{limit}$$

Simultaneous transitions on the reference and data signals can never cause **\$skew** to report a timing violation, even when the skew limit value is zero.

The **\$skew** timing check is event-based; it is evaluated only after a data event. If there is never a data event (i.e., the data event is infinitely late), the **\$skew** timing check shall never be evaluated, and no timing violation shall ever be reported. In contrast, the **\$timeskew** and **\$fullskew** checks are timer-based by default, and they shall be used if violation reports are absolutely required and the data event can be very late or even absent altogether. These checks are discussed in 15.3.2 and 15.3.3.

\$skew shall wait indefinitely for the data event once it has detected a reference event and it shall not report a timing

violation until the data event takes place. A second consecutive reference event shall cancel the old wait for the data event and begin a new one.

After a reference event, the **\$skew** timing check shall never stop checking data events for a timing violation. **\$skew** shall report timing violations for all data events occurring beyond the limit after a reference event.

15.3.2 \$timeskew

The syntax for **\$timeskew** is shown in Syntax 15-10.

```
$timeskew_timing_check ::= (From Annex A - A.7.5.1)
    $timeskew ( reference_event , data_event , timing_check_limit
        [ , [ notify_reg ] [ , [ event_based_flag ] [ , [ remain_active_flag ] ] ] ) ;
data_event ::= (From Annex A - A.7.5.2)
    timing_check_event
event_based_flag ::=
    constant_expression
notify_reg ::=
    variable_identifier
reference_event ::=
    timing_check_event
remain_active_flag ::=
    constant_mintypmax_expression
timing_check_limit ::=
    expression
```

Syntax 15-10—Syntax for **\$timeskew**

Table 15-8 defines the **\$timeskew** timing check arguments.

Table 15-8—**\$timeskew** arguments

Argument	Description
reference_event	Timestamp event
data_event	Timecheck event
limit	Non-negative constant expression
notifier (optional)	Reg
event_based_flag (optional)	Constant expression
remain_active_flag (optional)	Constant expression

The **\$timeskew** timing check reports a violation only in the following cases:

$$(\text{timecheck time}) - (\text{timestamp time}) > \text{limit}$$

Simultaneous transitions on the reference and data signals can never cause **\$timeskew** to report a timing violation, even when the skew limit value is zero.

The default behavior for **\$timeskew** is timer-based. Violations are reported immediately upon an elapse of time after

the reference event equal to the limit, and the check shall become dormant and report no more violations (even in response to data events) until after the next reference event. This check shall also become dormant if it detects a reference event when its condition is false.

The **\$timeskew** check's default timer-based behavior can be altered to event-based using the event based flag. It behaves like the **\$skew** check when both the event based flag and the remain active flag are set. The **\$timeskew** check behaves like the **\$skew** check when only the event based flag is set, except it becomes dormant after reporting the first violation.

Example:

```
$timeskew (posedge CP &&& MODE, negedge CPN, 50);
```

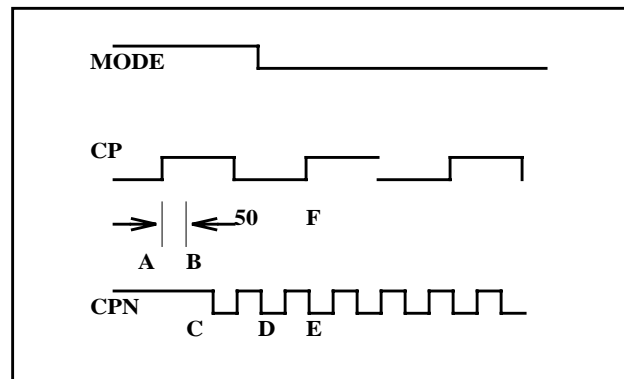


Figure 15-1—Sample \$timeskew

Case 1: Event based flag and remain active flag not set.

After the first reference event on CP at A, a violation is reported at B as soon as 50 time units have passed. No further violations are reported.

Case 2: Event based flag set, remain active flag not set.

The first three negative transitions on CPN at points C, D and E shall cause timing violations. The second reference event at F occurs while MODE is false, turning the **\$timeskew** check dormant, and no further violations are reported.

Case 3: Event based flag and remain active flag both set.

Every negative edge on CPN is reported as a violation, which is identical to **\$skew** behavior.

15.3.3 \$fullskew

The syntax for **\$fullskew** is shown in Syntax 15-11.

```

$fullskew_timing_check ::= (From Annex A - A.7.5.1)
    $fullskew ( reference_event , data_event , timing_check_limit , timing_check_limit
        [ , [ notify_reg ] [ , [ event_based_flag ] [ , [ remain_active_flag ] ] ] ) ;
data_event ::= (From Annex A - A.7.5.2)
    timing_check_event
event_based_flag ::=
    constant_expression
notify_reg ::=
    variable_identifier
reference_event ::=
    timing_check_event
remain_active_flag ::=
    constant_mintypmax_expression
timing_check_limit ::=
    expression

```

Syntax 15-11—Syntax for \$fullskew

Table 15-9 defines the **\$fullskew** timing check arguments.

Table 15-9—\$fullskew arguments

Argument	Description
reference_event	Timestamp or timecheck event
data_event	Timestamp or timecheck event
limit 1	Non-negative constant expression
limit 2	Non-negative constant expression
notifier (optional)	Reg
event_based_flag (optional)	Constant expression
remain_active_flag (optional)	Constant expression

\$fullskew is identical to **\$timeskew** except the reference and data events can transition in either order. The first limit is the maximum time by which the data event can follow the reference event. The second limit is the maximum time by which the reference event can follow the data event.

The reference event is the timestamp event and the data event is the timecheck event when the reference event precedes the data event. The data event is the timestamp event and the reference event is the timecheck event when the data event precedes the reference event.

The **\$fullskew** timing check reports a violation only in the following case, where limit is set to limit1 when the reference event transitions first, and to limit2 when the data event transitions first:

$$(\text{timecheck time}) - (\text{timestamp time}) > \text{limit}$$

Simultaneous transitions on the reference and data signals shall never cause **\$fullskew** to report a timing violation, even when the skew limit value is zero.

The default behavior for **\$fullskew** is timer-based. Violations shall be reported immediately upon an elapse of time after the timestamp event equal to the limit. It then becomes dormant and reports no more violations, even in response to timecheck events, until after the next timestamp event. This check shall also become dormant if it detects a timestamp event when the associated condition is false.

The **\$fullskew** check's default timer-based behavior can be altered to event-based using the event based flag. It behaves like the **\$skew** check when both the event based flag and the remain active flag are set. The **\$timeskew** check behaves like the **\$skew** check when only the event based flag is set, except it becomes dormant after it reports the first violation.

Example:

```
$fullskew (posedge CP &&& MODE, negedge CPN, 50, 70);
```

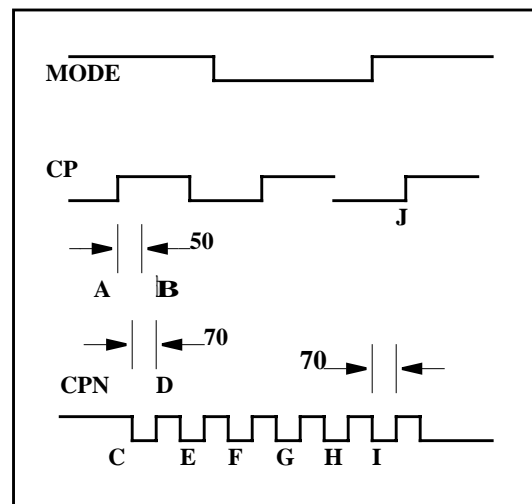


Figure 15-2—Sample \$fullskew

Case 1: Event based flag and remain active flag not set.

The transition at A of CP while MODE is true begins a wait for a negative transition on CPN, and a violation is reported at B as soon as a period of time equal to 50 time units has passed. This resets the check and readies it for the next active transition.

A negative transition on CPN occurs next at C, beginning a wait for a positive transition on CP while MODE is true. At D a time equal to 70 time units has passed without a positive edge on CP while MODE is true, so a violation is reported and the check is again reset to await the next active transition.

A transition on CPN at E also results in a timing violation, as does the transition at F, because even though CP transitions, MODE is no longer true. Transitions at G and H also result in timing violations, but not the transition at I, because it is followed by a positive transition on CP while MODE is true.

Case 2: Event based flag set, remain active flag not set.

The transition at A of CP while MODE is true begins a wait for a negative transition on CPN, and a violation is reported at C on CPN because it occurs beyond the 50 time unit limit. This transition at B also begins a wait of 70 time units for a positive transition on CP while MODE is true. But for transitions on CPN at B through H there is no positive transition on CP while MODE is true, and so no timing violations are reported. The transition at I on CPN begins a wait of 70 time units, and this is satisfied by the positive transition on CP at J while MODE is true.

Case 3: Event based flag and remain active flag both set.

The transition at A of CP while MODE is true begins a wait for a negative transition on CPN, and a violation is reported at C on CPN, and it shall also begin a wait for a positive transition on CP while MODE is true. No such transition on CP ever takes place after CPN transitions C through H, but no violations are reported because CP never experiences a positive transition while MODE is true. Transition I also reports no violation because a positive transition at I on CP while MODE is true occurs within the 70 time unit skew limit.

15.3.4 \$width

The **\$width** timing check syntax is shown in Syntax 15-12.

```
$width_timing_check ::= (From Annex A - A.7.5.1)
$width ( controlled_reference_event , timing_check_limit ,
        threshold [ , [ notify_reg ] ] ) ;
controlled_reference_event ::= (From Annex A - A.7.5.2)
        controlled_timing_check_event
notify_reg ::=
        variable_identifier
threshold ::=
        constant_expression
timing_check_limit ::=
        expression
```

Syntax 15-12—Syntax for \$width

If the comma before the threshold is present, the comma before the notifier shall also be present, even though both arguments are optional.

Table 15-10 defines the **\$width** timing check.

Table 15-10—\$width arguments

Argument	Description
reference_event	Timestamp edge triggered event
(data_event - implicit)	Timecheck edge triggered event
limit	Non-negative constant expression
threshold (optional)	Non-negative constant expression
notifier (optional)	Reg

The **\$width** timing check monitors the width of signal pulses by measuring the time from the timestamp event to the timecheck event. Since a data event is not passed to **\$width**, it is derived from the reference event, as follows:

data event = reference event signal with opposite edge

Because of the way the data event is derived for **\$width**, an edge triggered event has to be passed as the reference event. A compilation error shall occur if the reference event is not an edge specification.

While the **\$width** timing check can be defined in terms of a time window, it is simpler to express it as the difference between the timecheck and timestamp times. The **\$width** timing check reports a violation in the following case:

```
threshold < (timecheck time) - (timestamp time) < limit
```

The pulse width has to be greater than or equal to limit in order to avoid a timing violation, but no violation is reported for glitches smaller than the threshold.

The threshold argument shall be included if the notifier argument is required. It is permissible to not specify both the threshold and notifier arguments, making the default value for the threshold zero (0). If the notifier is present, a non-null value for the threshold shall also be present. Here is a legal **\$width** check when the notifier is required and the threshold is not:

```
$width (posedge clk, 6, 0, ntfr_reg);
```

The data event and the reference event shall never occur at the same simulation time because these events are triggered by opposite transitions.

Example:

The following example demonstrates some examples of legal and illegal calls:

```
// Legal Calls
$width ( negedge clr, lim );
$width ( negedge clr, lim, thresh, notif );
$width ( negedge clr, lim, 0, notif );

// Illegal Calls
$width ( negedge clr, lim, , notif );
$width ( negedge clr, lim, notif );
```

15.3.5 \$period

The **\$period** timing check syntax is shown in Syntax 15-13.

```
$period_timing_check ::= (From Annex A - A.7.5.1)
$period ( controlled_reference_event , timing_check_limit [ , [ notify_reg ] ] );
controlled_reference_event ::= (From Annex A - A.7.5.2)
    controlled_timing_check_event
notify_reg ::=
    variable_identifier
timing_check_limit ::=
    expression
```

Syntax 15-13—Syntax for \$period

Table 15-11 defines the **\$period** timing check.

Table 15-11—\$period arguments

Argument	Description
reference_event	Timestamp edge triggered event
(data_event - implicit)	Timestamp edge triggered event

Table 15-11—\$period arguments (*continued*)

limit	Non-negative constant expression
notifier (optional)	Reg

Since the data event is not passed as an argument to **\$period**, it is derived from the reference event, as follows:

data event = reference event signal with the same edge

Because of the way the data event is derived for **\$period**, an edge triggered event shall be passed as the reference event. A compilation error shall occur if the reference event is not an edge specification.

While the **\$period** timing check can be defined in terms of a time window, it is simpler to express it as the difference between the timecheck and timestamp times. The **\$period** timing check reports a violation in the following case:

(timecheck time) - (timestamp time) < limit

15.3.6 \$nochange

The **\$nochange** syntax is shown in Syntax 15-14.

```
$nochange_timing_check ::= (From Annex A - A.7.5.1)
$nochange ( reference_event , data_event , start_edge_offset ,
            end_edge_offset [ , [ notify_reg ] ] ) ;
data_event ::= (From Annex A - A.7.5.2)
            timing_check_event
end_edge_offset ::=
            mintypmax_expression
notify_reg ::=
            variable_identifier
reference_event ::=
            timing_check_event
start_edge_offset ::=
            mintypmax_expression
```

Syntax 15-14—Syntax for \$nochange

Table 15-12 defines the **\$nochange** timing check arguments.

Table 15-12—\$nochange arguments

Argument	Description
reference_event	Edge triggered timestamp and/or timecheck event
data_event	Timestamp or timecheck event
start_edge_offset	Constant expression
end_edge_offset	Constant expression
notifier (optional)	Reg

The **\$nochange** timing check reports a timing violation if the data event occurs during the specified level of the control signal (the reference event). The reference event can be specified with the **posedge** or the **negedge** keyword, but the edge control specifiers (see 15.4) can not be used.

The start edge and end edge offsets can expand or shrink the timing violation region, which is defined by the duration of the reference event signal after the edge. For example, if the reference event is a posedge, then the duration is the period during which the reference signal is high. A positive offset for start edge extends the region by starting the timing violation region earlier; a negative offset for start edge shrinks the region by starting the region later. Similarly, a positive offset for the end edge extends the timing violation region by ending it later, while a negative offset for the end edge shrinks the region by ending it earlier. If both the offsets are zero, the size of the region shall not change.

Unlike other timing checks, **\$nochange** involves three, rather than two, transitions. The leading edge of the reference event defines the beginning of the time window, while the trailing edge of the reference event defines the end of the time window. A violation results if the data event occurs anytime within the time window.

The endpoints of the time window are determined as follows:

```
(beginning of time window) =
(leading reference edge time) - start_edge_offset
(end of time window) = (trailing reference edge time) + end_edge_offset
```

The **\$nochange** timing check reports a timing violation in the following case:

```
beginning of time window) < (data event time) < (end of time window)
```

The endpoints of the time window are not included. The values of `start_edge_offset` and `end_edge_offset` play a significant role in determining which signal, the reference event or the data event, is the timestamp or timecheck event.

Example:

```
$nochange( posedge clk, data, 0, 0) ;
```

In this example, **\$nochange** system task shall report a violation if the `data` signal changes while `clk` is high. It shall not be a violation if `posedge clk` and a transition on `data` occur simultaneously.

15.4 Edge-control specifiers

The edge-control specifiers can be used to control events in timing checks based on specific edge transitions between 0, 1, and x. Syntax 15-15 shows the syntax for edge-control specifiers.

```
edge_control_specifier ::= (From Annex A - A.7.5.3)
    edge [ edge_descriptor [ , edge_descriptor ] ]
edge_descriptor1 ::=
    01
    | 10
    | z_or_x zero_or_one
    | zero_or_one z_or_x
zero_or_one ::= 0 | 1
z_or_x ::= x | X | z | Z
```

¹Embedded spaces are illegal.

Syntax 15-15—Syntax for edge control specifier

Edge-control specifiers contain the keyword **edge** followed by a square bracketed list of from one to six pairs of edge transitions between 0, 1 and x, as follows:

01	Transition from 0 to 1
0x	Transition from 0 to x
10	Transition from 1 to 0
1x	Transition from 1 to x
x0	Transition from x to 0
x1	Transition from x to 1

Edge transitions involving z are treated the same way as edge transitions involving x.

The **posedge** and **negedge** keywords can be used as a shorthand for certain edge-control specifiers. For example, the construct:

```
posedge clr
```

is equivalent to the following:

```
edge[01, 0x, x1] clr
```

Similarly, the construct

```
negedge clr
```

is the same as the following:

```
edge[10, x0, 1x] clr
```

However, edge-control specifiers offer the flexibility to declare edge transitions other than **posedge** and **negedge**.

15.5 Notifiers: user-defined responses to timing violations

Timing check notifiers detect timing check violations behaviorally, and, therefore, take an action as soon as a violation occurs. Such notifiers can be used to print an informative error message describing the violation or to propagate an x value at the output of the device which reported the violation.

The notifier is a reg—declared in the module where timing check tasks are invoked—which is passed as the last argument to a system timing check. Whenever a timing violation occurs, the system task updates the value of the notifier.

The notifier is an optional argument to all system timing checks and can be omitted from the system task call without adversely affecting its operation.

Table 15-13 shows how the notifier values are toggled when timing violations occur.

Table 15-13—User-defined responses to timing violations

BEFORE violation	AFTER violation
x	0
0	1
1	0
z	z

Examples:

Example 1

```
$setup( data, posedge clk, 10, notify_reg ) ;
$width( posedge clk, 16, notify_reg ) ;
```

Example 2—Consider a more complex example of how to use notifiers in a behavioral model. The following example uses a notifier to set the D flip-flop output to x when a timing violation occurs in an edge-sensitive UDP.

```
primitive posdff_udp(q, clock, data, preset, clear, notifier);
output q; reg q;
input clock, data, preset, clear, notifier;
table
//clock data  p c notifier state  q
//-----
r      0      1 1      ?      :  ?  : 0 ;
r      1      1 1      ?      :  ?  : 1 ;

p      1      ? 1      ?      :  1  : 1 ;
p      0      1 ?      ?      :  0  : 0 ;

n      ?      ? ?      ?      :  ?  : - ;
?      *      ? ?      ?      :  ?  : - ;

?      ?      0 1      ?      :  ?  : 1 ;
?      ?      * 1      ?      :  1  : 1 ;

?      ?      1 0      ?      :  ?  : 0 ;
?      ?      1 *      ?      :  0  : 0 ;
?      ?      ? ?      *      :  ?  : x ; // At any notifier event
                                           // output x
endtable
endprimitive
```

```

module dff(q, qbar, clock, data, preset, clear);
output q, qbar;
input clock, data, preset, clear;
reg notifier;

and (enable, preset, clear);
not (qbar, ffout);
buf (q, ffout);
posdff_udp (ffout, clock, data, preset, clear, notifier);

specify
    // Define timing check specparam values
    specparam tSU = 10, tHD = 1, tPW = 25, tWPC = 10, tREC = 5;
    // Define module path delay rise and fall min:typ:max values
    specparam tPLHc = 4:6:9 , tPHLc = 5:8:11;
    specparam tPLHpc = 3:5:6 , tPHLpc = 4:7:9;

    // Specify module path delays
    (clock *> q,qbar) = (tPLHc, tPHLc);
    (preset,clear *> q,qbar) = (tPLHpc, tPHLpc);

    // Setup time : data to clock, only when preset and clear are 1
    $setup(data, posedge clock &&& enable, tSU, notifier);

    // Hold time: clock to data, only when preset and clear are 1
    $hold(posedge clock, data &&& enable, tHD, notifier);

    // Clock period check
    $period(posedge clock, tPW, notifier);
    // Pulse width : preset, clear
    $width(negedge preset, tWPC, 0, notifier);
    $width(negedge clear, tWPC, 0, notifier);

    // Recovery time: clear or preset to clock
    $recovery(posedge preset, posedge clock, tREC, notifier);
    $recovery(posedge clear, posedge clock, tREC, notifier);
endspecify
endmodule

```

NOTE—This model applies to edge-sensitive UDPs only; for level-sensitive models, an additional UDP for x propagation has to be generated.

15.5.1 Requirements for accurate simulation

In order to accurately model negative value timing checks:

- a) A timing violation shall be triggered if the signal changes in the violation window, exclusive of the endpoints. Violation windows smaller than two units of simulation precision can not yield timing violations.
- b) The value of the latched data shall be the one which is stable during the violation window, again, exclusive of the endpoints.

To facilitate these modeling requirements, delayed copies of the data and reference signals are generated in the timing checks, and these are used internally for timing check evaluation at run-time. The setup and hold times used internally

are adjusted so as to shift the violation window and make it overlap the reference signal.

Delayed data and reference signals can be declared within the timing check so they can be used in the model's functional implementation to insure accurate simulation. If no delayed signals are declared in the timing check, and if a negative setup or hold value is present, then implicit delayed signals are created. Since implicit delayed signals can not be used in defining model behavior, such a model can possibly behave incorrectly.

Examples:

Example 1:

```
$setuphold(posedge CLK, DATA, -10, 20);
```

Implicit delayed signals shall be created for CLK and DATA, but it shall not be possible to access them. The **\$setuphold** check shall be properly evaluated, but functional behavior shall not always be accurate. The old DATA value shall be incorrectly clocked in if DATA transitions between `posedge CLK` and 10 time units later.

Example 2:

```
$setuphold(posedge CLK, DATA1, -10, 20);  
$setuphold(posedge CLK, DATA2, -15, 18);
```

Implicit delayed signals shall be created for CLK, DATA1 and DATA2, one for each. Even though CLK is referenced in two different timing checks, only one implicit delayed signal is created, and it is used for both timing checks.

Example 3:

If a given signal has a delayed signal in some timing checks but not in others, the delayed signal shall be used in both cases:

```
$setuphold(posedge CLK, DATA1, -10, 20,,, del_CLK, del_DATA1);  
$setuphold(posedge CLK, DATA2, -15, 18);
```

Explicit delayed signals of `del_CLK` and `del_DATA1` are created for CLK and DATA1, while an implicit delayed signal is created for DATA2. In other words, CLK has only one delayed signal created for it, `del_CLK`, rather than one explicit delayed signal for the first check, and another implicit delayed signal for the second check.

The delayed versions of the signals, whether implicit or explicit, shall be used in the **\$setup**, **\$hold**, **\$setuphold**, **\$recovery**, **\$removal**, **\$recrem**, **\$width**, **\$period** and **\$nochange** timing checks, and these checks shall have their limits adjusted accordingly. This ensures the notifier shall be toggled at the proper moment. If the adjusted limit becomes less than or equal to 0, the limit shall be set to 0 and the simulator shall issue a warning.

The delayed versions of the signals shall not be used for the **\$skew**, **\$fullskew** and **\$timeskew** timing checks because it can possibly result in the reversal of the order of signal transitions. This causes the notifiers for these timing checks to toggle at the wrong time relative to the rest of the model, perhaps resulting in transitions to X due to a timing check violation being canceled. This issue shall be addressed in the model, possibly by using separate notifiers for these checks.

It is possible for a set of negative timing check values to be mutually inconsistent and produce no solution for the delay values of delayed signals. In these situations the simulator shall issue a warning message. The inconsistency shall be resolved by changing the smallest negative limit value to 0 and recalculating the delays for the delayed signals, and this shall be repeated until a solution is reached. This procedure shall always produce a solution because in the worst case all negative limit values become 0, and no delayed signals are needed.

The delayed timing check signals are only actually delayed when negative limit values are present. If a timing check signal becomes delayed by more than the propagation delay from that signal to an output, that output shall take longer than its propagation delay to change. It shall instead transition at the same time which the delayed timing check signal

changes. Thus, the output shall behave as if its specify path delay were equal to the delay applied to the timing check signal. This situation can only arise when unique setup/hold or removal/recovery times are given for each edge of the data signal.

Example:

```
(CLK = Q) = 6;
$setuphold (posedge CLK, posedge D, -3, 8, , , , dCLK, dD);
$setuphold (posedge CLK, negedge D, -7, 13, , , , dCLK, dD);
```

The setup time of -7 (the larger in absolute value of -3 and -7) creates a delay of 7 for dCLK, and so output Q shall not change until 7 time units after a positive edge on CLK, rather than the 6 time units given in the specify path.

15.5.2 Conditions in negative timing checks

Conditions can be associated with both the reference and data signals by using the &&& operator, but when either the setup or hold time is negative the conditions need to be paired with reference and data signals in a more flexible way. This example illustrates why.

This pair of **\$setup** and **\$hold** checks work together to provide the same check as a single **\$setuphold**:

```
$setup (data, clk&&&cond1, tsetup, ntfr);
$hold (clk, data&&&cond1, thold, ntfr);
```

clk is the timecheck event for the **\$setup** check, while data is the timecheck event for the **\$hold** check. This can not be represented in a single **\$setuphold** check, and so additional arguments are provided to make this possible. These arguments are timestamp_cond and timecheck_cond, and they immediately follow the notifier (see 15.2.3). This **\$setuphold** check is equivalent to the separate **\$setup** and **\$hold** checks shown above:

```
$setuphold( clk, data, tsetup, thold, ntfr, , cond1);
```

The timestamp_cond argument is null, while the timecheck_cond argument is cond1.

The timestamp_cond and timecheck_cond arguments are associated with either the reference or data signals based on which delayed version of these signals occurs first. timestamp_cond is associated with the delayed signal which transitions first, while timecheck_cond is associated with the delayed signal which transitions second.

Delayed signals are only created for the reference and data signals, and not for any condition signals associated with them. Therefore, timestamp_cond and timecheck_cond are not implicitly delayed by the simulator. Delayed condition signals for the timestamp_cond and timecheck_cond fields can be created by making them a function of the delayed signals.

Example:

```
assign TE_cond_D = (dTE !== 1'b1);
assign TE_cond_TI = (dTE !== 1'b0);
assign DXTI_cond = (dTI !== dD);

specify
  $setuphold(posedge CP, D, -10, 20, notifier, ,TE_cond_D, dCP, dD);
  $setuphold(posedge CP, TI, 20, -10, notifier, ,TE_cond_TI, dCP, dTI);
  $setuphold(posedge CP, TE, -4, 8, notifier, ,DXTI_cond, dCP, dTE);
endspecify
```

The assign statements create condition signals which are functions of the delayed signals. Creating delayed signal conditions synchronizes the conditions with the delayed versions of the reference and data signals used to perform the checks.

The first **\$setuphold** has a negative setup time, and so the timecheck condition `TE_cond_D` is associated with data signal `D`. The second **\$setuphold** has a negative hold time, and so the timecheck condition `TE_cond_TI` is associated with reference signals `CP`. The third **\$setuphold** has a negative setup time, and so the timecheck condition `DXTI_cond` is associated with data signal `TE`.

The violation windows for the example are shown in Figure 15-3.

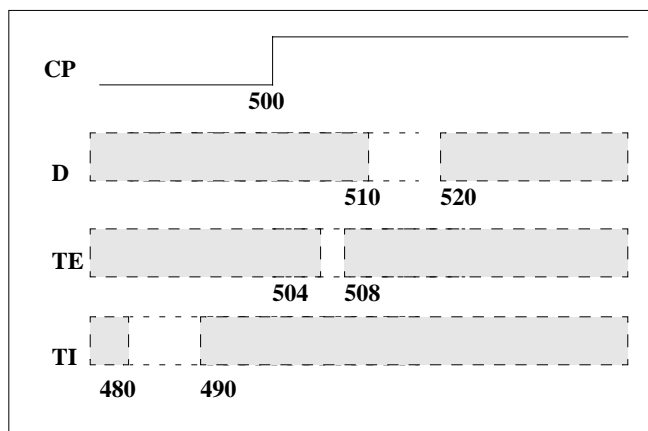


Figure 15-3—Timing check violation windows

These are the delay values calculated for the delayed signals:

dCP	10.01
dD	0.00
dTI	20.02
dTE	2.02

Use of delayed signals in creating the signals for the `timestamp_cond` and `timecheck_cond` arguments is not required, but it is usually closer to actual device behavior.

15.5.3 Notifiers in negative timing checks

Because the reference and data signals are delayed internally, the detection of the timing violation is also delayed. Notifier regs in negative timing checks shall be toggled when the timing check detects a timing violation, which occurs when the delayed signals as measured by the adjusted timing check values are in violation, not when the undelayed signals at the model inputs as measured by the original timing check values are in violation.

15.5.4 Option behavior

As already mentioned, the ability of Verilog simulators to handle negative values in **\$setuphold** and **\$recrem** timing checks shall be enabled with an invocation option. It is possible models written to accept negative timing check values with delayed reference and/or delayed data signals can be run without this invocation option enabled. In this circumstance the delayed reference and data signals become copies of the original reference and data signals. The same occurs if an invocation option turning off all timing checks is used.

15.6 Enabling timing checks with conditioned events

A construct called a conditioned event ties the occurrence of timing checks to the value of a conditioning signal. Syntax 15-16 shows the syntax for controlled timing check event.

```

timing_check_event ::= (From Annex A - A.7.5.3)
    [timing_check_event_control] specify_terminal_descriptor [ &&& timing_check_condition ]
controlled_timing_check_event ::=
    timing_check_event_control specify_terminal_descriptor [ &&& timing_check_condition ]
timing_check_event_control ::=
    posedge
    | negedge
    | edge_control_specifier
specify_terminal_descriptor ::=
    specify_input_terminal_descriptor
    | specify_output_terminal_descriptor
timing_check_condition ::=
    scalar_timing_check_condition
    | ( scalar_timing_check_condition )
scalar_timing_check_condition ::=
    expression
    | ~ expression
    | expression == scalar_constant
    | expression === scalar_constant
    | expression != scalar_constant
    | expression !== scalar_constant
scalar_constant ::=
    'b0 | 'b1 | 'B0 | 'B1 | 'b0 | 'b1 | 'B0 | 'B1 | 1 | 0

```

Syntax 15-16—Syntax for controlled timing check event

The comparisons used in the condition can be deterministic, as in ==, !=, ~, or no operation, or nondeterministic, as in == or !=. When comparisons are deterministic, an x value on the conditioning signal shall not enable the timing check. For nondeterministic comparisons, an x on the conditioning signal shall enable the timing check.

The conditioning signal shall be a scalar net; if a vector net or an expression resulting in a multi-bit value is used, then the least significant bit of the vector net or the expression value is used.

If more than one conditioning signal is required for conditioning timing checks, appropriate logic shall be combined in a separate signal outside the specify block, which can be used as the conditioning signal.

Examples:

Example 1—To illustrate the difference between conditioned and unconditioned timing check events, consider the following example with unconditioned timing check:

```
$setup( data, posedge clk, 10 );
```

Here, a setup timing check shall occur every time there is a positive edge on the signal `clk`.

To trigger the setup check on the positive edge on the signal `clk` only when the signal `clr` is high, rewrite the command as

```
$setup( data, posedge clk &&& clr, 10 );
```

Example 2—This example shows two ways to trigger the same timing check as in example 1 (on the positive `clk` edge) only when `clr` is low. The second method uses the === operator, which makes the comparison deterministic.

```

$setup( data, posedge clk &&& (~clr), 10 ) ;
$setup( data, posedge clk &&& (clr===0), 10 ) ;

```

Example 3—To perform the previous sample setup check on the positive clk edge only when clr and set are high, add the following statement outside the specify block:

```

and new_gate( clr_and_set, clr, set );

```

Then add the condition to the timing check using the signal clr_and_set as follows:

```

$setup( data, posedge clk &&& clr_and_set, 10 );

```

15.7 Vector signals in timing checks

Either or both signals in a timing check can be a vector. This shall be interpreted as a single timing check where the transition of one or more bits of a vector is considered a single transition of that vector.

Example:

```

module DFF (Q, CLK, DAT);
input CLK;
input [7:0] DAT;
output [7:0] Q;
always @(posedge clk)
  Q = DAT;
specify
  $setup (DAT, posedge CLK, 10);
endspecify
endmodule

```

If DAT transitions from 'b00101110 to 'b01010011 at time 100, and CLK transitions from 0 to 1 at time 105, then the **\$setup** timing check shall still only report a single timing violation.

Simulators can provide an option causing vectors in timing checks to result in the creation of multiple single-bit timing checks. For timing checks with only a single signal, such as **\$period** or **\$width**, a vector of width N results in N unique timing checks. For timing checks with two signals, such as **\$setup**, **\$hold**, **\$setuphold**, **\$skew**, **\$timeskew**, **\$fullskew**, **\$recovery**, **\$removal**, **\$recrem** and **\$nochange**, where M and N are the widths of the signals, the result is M*N unique timing checks. If there is a notifier, all the timing checks trigger that notifier.

With such an option enabled, the above example yields six timing violation because six bits of DAT transitioned.

15.8 Negative timing checks

Both the **\$setuphold** and **\$recrem** timing checks can accept negative values when the negative timing check option is enabled. The behavior of these two timing checks is identical with respect to negative values. The descriptions in this section are for the **\$setuphold** timing check, but apply equally to the **\$recrem** timing check.

The setup and hold timing check values define a timing violation window with respect to the reference signal edge during which the data shall remain constant. Any change of the data during the specified window causes a timing violation. The timing violation is reported and, through the notifier reg, other actions can take place in the model, such as forcing the output of a flip-flop to X when it detects a timing violation.

A positive value for both setup and hold times implies this violation window straddles the reference signal shown in Figure 15-4.

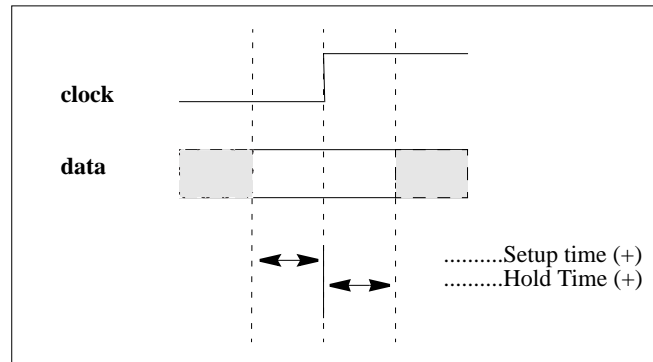


Figure 15-4—Data constraint interval, positive setup/hold

A negative hold or setup time means the violation window is shifted to either before or after the reference edge. This can happen in a real device because of disparate internal device delays between the internal clock and data signal paths. These internal device delays are illustrated in Figure 15-5 showing how significant differences in these delays can cause negative setup or hold values.

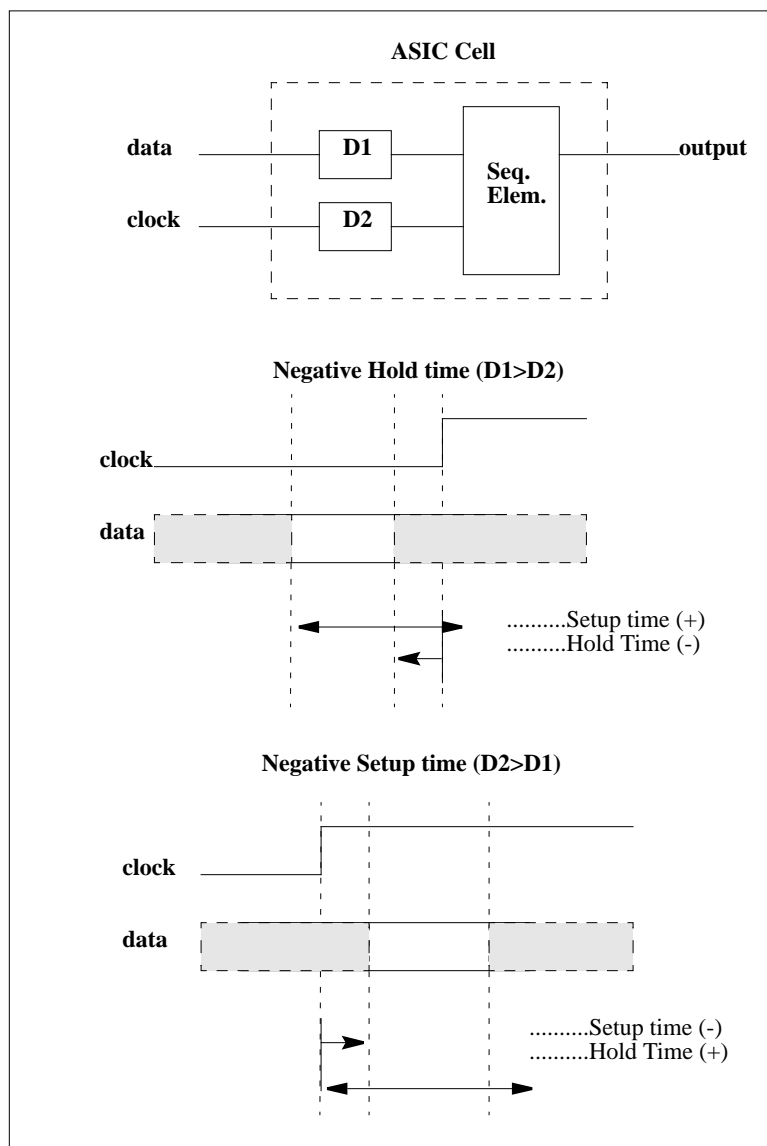


Figure 15-5—Data constraint interval, negative setup/hold

Section 16

Backannotation using the Standard Delay Format (SDF)

SDF files contain timing values for specify path delays, specparam values, timing check constraints, and interconnect delays. SDF files can also contain other information in addition to simulation timing, but these need not concern Verilog simulation. The timing values in SDF files usually come from ASIC delay calculation tools that take advantage of connectivity, technology, and layout geometry information.

Verilog backannotation is the process by which timing values from the SDF file update specify path delays, specparam values, timing constraint values, and interconnect delays.

All this information is covered further in *IEEE Std 1497-1999, Standard for Standard Delay Format (SDF) for the Electronic Design Process* [B2].

16.1 The SDF annotator

The term SDF Annotator refers to any tool capable of backannotating SDF data to a Verilog simulator. It shall report a warning for any data it is unable to annotate.

An SDF file can contain many constructs which are not related to specify path delays, specparam values, timing check constraint values, or interconnect delays. An example is any construct in the `TIMINGENV` section of the SDF file. All constructs unrelated to Verilog timing shall be ignored without any warnings issued.

Any Verilog timing value for which the SDF file does not provide a value shall not be modified during the backannotation process, and its pre-backannotation value shall be unchanged.

16.2 Mapping of SDF constructs to Verilog

SDF timing values appear within a `CELL` declaration, which can contain one or more of `DELAY`, `TIMINGCHECK` and `LABEL` sections. The `DELAY` section contains propagation delay values for specify paths and interconnect delays. The `TIMINGCHECK` section contains timing check constraint values. The `LABEL` section contains new values for specparams. Backannotation into Verilog is done by matching SDF constructs to the corresponding Verilog declarations, then replacing the existing Verilog timing values with those from the SDF file.

16.2.1 Mapping of SDF delay constructs to Verilog declarations

When annotating `DELAY` constructs that are not interconnect delays (covered in 16.2.3), the SDF annotator looks for specify paths where the names and conditions match. When annotating `TIMINGCHECK` constructs, the SDF annotator looks for timing checks of the same type where the names and conditions match. Table 16-1 shows which Verilog structures can be annotated by each SDF construct in the `DELAY` section.

Table 16-1—Mapping of SDF delay constructs to Verilog declarations

SDF Construct	Verilog annotated structure
(PATHPULSE...	Conditional and non-conditional specify path pulse limits
(PATHPULSEPERCENT...	Conditional and non-conditional specify path pulse limits
(IOPATH...	Conditional and non-conditional specify path delays/pulse limits
(IOPATH (RETAIN...	Conditional and non-conditional specify path delays/pulse limits, RETAIN ignored without warning
(COND (IOPATH...	Conditional specify path delays/pulse limits
(COND (IOPATH (RETAIN...	Conditional specify path delays/pulse limits, RETAIN ignored without warning
(CONDELSE (IOPATH...	ifnone
(CONDELSE (IOPATH (RETAIN...	ifnone, RETAIN ignored without warning
(DEVICE...	All specify paths to module outputs. If no specify paths, all primitives driving module outputs.
(DEVICE port_instance...	If port_instance is a module instance, all specify paths to module outputs. If no specify paths, all primitives driving module outputs. If port_instance is a module instance output, all specify paths to that module output. If no specify path, all primitives driving that module output.

In this example the source SDF signal `sel` matches the source Verilog signal, and the destination SDF signal `zout` also matches the destination Verilog signal, and so the rise/fall times of `1.3` and `1.7` are annotated to the specify path.

SDF file:

```
(IOPATH sel zout (1.3) (1.7))
```

Verilog specify path:

```
(sel => zout) = 0;
```

A conditional IOPATH delay between two ports shall annotate only to Verilog specify paths between those same two ports with the same condition. In this example the rise/fall times of `1.3` and `1.7` are annotated only to the second specify path.

SDF file:

```
(COND mode (IOPATH sel zout (1.3) (1.7)))
```

Verilog specify paths:

```
if (!mode) (sel => zout) = 0;
if (mode) (sel => zout) = 0;
```

A non-conditional IOPATH delay between two ports shall annotate to all Verilog specify paths between those same two ports. In this example the rise/fall times of `1.3` and `1.7` are annotated to both specify paths.

SDF file:

```
(IOPATH sel zout (1.3) (1.7))
```

Verilog specify paths:

```
if (!mode) (sel => zout) = 0;
if (mode) (sel => zout) = 0;
```

16.2.2 Mapping of SDF timing check constructs to Verilog

Table 16-2 shows which Verilog timing checks are annotated to by each type of SDF timing check. *v1* is the first value of a timing check, *v2* is the second value, while *x* indicates no value is annotated.

Table 16-2—Mapping of SDF timing check constructs to Verilog

SDF Timing Check	Annotated Verilog Timing checks
(SETUP <i>v1</i> ...	<code>\$setup(v1), \$setuphold(v1,x)</code>
(HOLD <i>v1</i> ...	<code>\$hold(v1), \$setuphold(x,v1)</code>
(SETUPHOLD <i>v1 v2</i> ...	<code>\$setup(v1), \$hold(v2), \$setuphold(v1,v2)</code>
(RECOVERY <i>v1</i> ...	<code>\$recovery(v1), \$recrem(v1,x)</code>
(REMOVAL <i>v1</i> ...	<code>\$removal(v1), \$recrem(x,v1)</code>
(RECREM <i>v1 v2</i> ...	<code>\$recovery(v1), \$removal(v2), \$recrem(v1,v2)</code>
(SKEW <i>v1</i> ...	<code>\$skew(v1)</code>
(TIMESKEW <i>v1</i> ... ¹	<code>\$timeskew(v1)</code>
(FULLSKEW <i>v1 v2</i> ... ¹	<code>\$fullskew(v1,v2)</code>
(WIDTH <i>v1</i> ...	<code>\$width(v1,x)</code>
(PERIOD <i>v1</i> ...	<code>\$period(v1)</code>
(NOCHANGE <i>v1 v2</i> ...	<code>\$nochange(v1,v2)</code> ²

¹Not part of current SDF standard

²Not usually implemented in Verilog simulators

The reference and data signals of timing checks can have logical condition expressions and edges associated with them. An SDF timing check with no conditions or edges on any of its signals shall match all corresponding Verilog timing checks regardless of whether conditions are present or not. In this example the SDF timing check shall annotate to all the Verilog timing checks:

SDF file:

```
(SETUPHOLD data clk (3) (4))
```

Verilog timing checks:

```
$setuphold (posedge clk&&& mode, data, 1, 1, ntfr);
$setuphold (negedge clk&&&!mode, data, 1, 1, ntfr);
```

When conditions and/or edges are associated with the signals in an SDF timing check, then they shall match those in any corresponding Verilog timing check before annotation shall happen. In this example the SDF timing check shall annotate to the first Verilog timing check, but not the second:

SDF file:

```
(SETUPHOLD data (posedge clk) (3) (4))
```

Verilog timing checks:

```
$setuphold (posedge clk&&& mode, data, 1, 1, ntfr); // Annotated
$setuphold (negedge clk&&&!mode, data, 1, 1, ntfr); // Not annotated
```

Here, the SDF timing check shall not annotate to any of the Verilog timing checks:

SDF file:

```
(SETUPHOLD data (COND !mode (posedge clk)) (3) (4))
```

Verilog timing checks:

```
$setuphold (posedge clk&&& mode, data, 1, 1, ntfr); // Not annotated
$setuphold (negedge clk&&&!mode, data, 1, 1, ntfr); // Not annotated
```

16.2.3 SDF annotation of specparams

The SDF LABEL construct annotates to specparams. Any expression containing one or more specparams is reevaluated when annotated to from an SDF file.

This example shows SDF LABEL constructs annotating to specparams in a Verilog module. The specparams are used in procedural delays to control when the clock transitions. The SDF LABEL construct annotates the values of dhigh and dlow, thereby setting the period and duty cycle of the clock.

SDF file:

```
(LABEL
  (ABSOLUTE
    (dhigh 60)
    (dlow 40)))
```

Verilog file:

```
module clock(clk);
output clk;
reg clk;
specparam dhigh=0, dlow=0;
initial clk = 0;
always
begin
#dhigh clk = 1; // Clock remains low for time dlow
                // before transitioning to 1
#dlow  clk = 0; // Clock remains high for time dhigh
                // before transitioning to 0
end;
endmodule
```

This example shows a specparam in an expression of a specify path. The SDF LABEL construct can be used to change the value of the specparam and cause reevaluation of the expression:

```
specparam cap = 0;
...
specify
  (A => Z) = 1.4 * cap + 0.7;
endspecify
```

16.2.4 SDF annotation of interconnect delays

SDF interconnect delay annotation differs from annotation of other constructs described above in that there exists no corresponding Verilog declaration to which to annotate. In Verilog simulation, interconnect delays are an abstraction that represents the signal propagation delay from an output or inout module port to an input or inout module port. The INTERCONNECT construct includes a source, a load, and delay values, while the PORT and NETDELAY constructs include only a load and delay values. Interconnect delays can only be annotated between module ports, never between primitive pins. Table 16-3 shows how the SDF interconnect constructs in the DELAY section are annotated:

Table 16-3—SDF annotation of interconnect delays

SDF Construct	Verilog annotated structure
(PORT...	Interconnect delay
(NETDELAY ¹	Interconnect delay
(INTERCONNECT...	Interconnect delay

¹Only OVI SDF version 1.0, 2.0, and 2.1, and IEEE SDF version 4.0

Interconnect delays can be annotated to both single source and multi-source nets.

When annotating a PORT construct, the SDF annotator shall search for the port and if it exists shall annotate an interconnect delay to that port which shall represent the delay from all sources on the net to that port.

When annotating a NETDELAY construct, the SDF annotator shall check to see if it is annotating to a port or a net. If it is a port then the SDF annotator shall annotate an interconnect delay to that port. If it is a net then it shall annotate an interconnect delay to all load ports connected to that net. If the port or net has more than one source then the delay shall represent the delay from all sources. NETDELAY delays can only be annotated to input or inout module ports, or to nets.

In the case of multi-source nets, unique delays can be annotated between each source/load pair using the INTERCONNECT construct. When annotating this construct, the SDF annotator shall find the source port and the load port, and if both exist it shall annotate an interconnect delay between the two. If the source port is not found, or if the source port and the load port are not actually on the same net, then a warning message is issued, but the delay to the load port is annotated anyway. If this happens for a load port that is part of a multi-source net, then the delay is treated as if it were the delay from all source ports, which is the same as the annotation behavior for a PORT delay. Source ports shall be output or input ports, while load ports shall be input or inout ports.

Interconnect delays share many of the characteristics of specify path delays. The same rules for specify path delays for filling in missing delays and pulse limits also apply for interconnect delays. Interconnect delays have twelve transition delays, and unique reject and error pulse limits are associated with each of the twelve. An unlimited number of future schedules are permitted.

In a Verilog module, a reference to an annotated port, wherever it occurs, whether in \$monitor and \$display statements or in expressions, shall provide the delayed signal value. A reference to the source shall yield the unde-

layed signal value, while a reference to the load shall yield the delayed signal value. In general, references to the signal value hierarchically before the load shall yield the undelayed signal value, while references to the signal at or hierarchically after the load shall yield the delayed signal value. An annotation to a hierarchical port shall affect all connected ports at higher or lower hierarchical levels, depending on the direction of annotation. An annotation from a source port shall be interpreted as being from all sources hierarchically higher or lower than that source port.

Up-hierarchy annotations shall be properly handled. This situation arises when the load is hierarchically above the source. The delay to all ports hierarchically above the load or which connect to the net at points hierarchically above the load is the same as the delay to that load.

Down-hierarchy annotation shall also be properly handled. This situation arises when the source is hierarchically above the load. The delay to the load is interpreted as being from all ports at or above the source or which connect to the net at points hierarchically above the source.

Hierarchically overlapping annotations are permitted. This occurs when annotations to or from the same port take place at different hierarchical levels, and therefore do not correspond to the same hierarchical subset of ports. In this example, the first INTERCONNECT statement annotates to all ports of the net that are at or hierarchically within i53/selmode, while the second annotates to a smaller subset of ports, only those at or hierarchically within i53/u21/in:

```
(INTERCONNECT i14/u5/out i53/selmode (1.43) (2.17))
(INTERCONNECT i14/u5/out i53/u21/in (1.58) (1.92))
```

Overlapping annotations can occur in many different ways, particularly on multi-source/multi-load nets, and SDF annotation shall properly resolve all the interactions.

16.3 Multiple annotations

SDF annotation is an ordered process. The constructs from the SDF file are annotated in their order of occurrence. This means that annotation of an SDF construct can be changed by annotation of a subsequent construct that either modifies (INCREMENT) or overwrites (ABSOLUTE) it. These do not have to be the same construct. This example first annotates pulse limits to an IOPATH, then annotates the entire IOPATH, thereby overwriting the pulse limits that were just annotated:

```
(DELAY
  (ABSOLUTE
    (PATHPULSE A Z (2.1) (3.4))
    (IOPATH A Z (3.5) (6.1))
```

Overwriting the pulse limits can be avoided by using empty parentheses to hold the current values of the pulse limits:

```
(DELAY
  (ABSOLUTE
    (PATHPULSE A Z (2.1) (3.4))
    (IOPATH A Z ((3.5) () ()) ((6.1) () ()) )
```

The above annotation can be simplified into a single statement like this:

```
(DELAY
  (ABSOLUTE
    (IOPATH A Z ((3.5) (2.1) (3.4)) ((6.1) (2.1) (3.4)) )
```

A PORT annotation followed by an INTERCONNECT annotation to the same load shall cause only the delay from the INTERCONNECT source to be affected. For this net with three sources and a single load, the delay from all sources

except i13/out remains 6:

```
(DELAY
  (ABSOLUTE
    (PORT i15/in (6))
    (INTERCONNECT i13/out i15/in (5))
```

An INTERCONNECT annotation followed by a PORT annotation shall cause the INTERCONNECT annotation to be overwritten. Here, the delays from all sources to the load shall become 6.

```
(DELAY
  (ABSOLUTE
    (INTERCONNECT i13/out i15/in (5))
    (PORT i15/in (6))
```

16.4 Multiple SDF files

More than one SDF file can be annotated. Each call to the **\$sdf_annotate** task annotates the design with timing information from an SDF file. Annotated values either modify (INCREMENT) or overwrite (ABSOLUTE) values from earlier SDF files. Different regions of a design can be annotated from different SDF files by specifying the region's hierarchy scope as the second argument to **\$sdf_annotate**.

16.5 Pulse limit annotation

For SDF annotation of delays (not timing constraints), the default values annotated for pulse limits shall be calculated using the percentage settings for the reject and error limits. By default these limits are 100%, but they can be modified through invocation options. For example, assuming invocation options have set the reject limit to 40% and the error limit to 80%, this SDF construct shall annotate a delay of 5, a reject limit of 2, and an error limit of 4:

```
(DELAY
  (ABSOLUTE
    (IOPATH A Z (5))
```

Given that the specify path delay was originally 0, this annotation results in a delay of 5 and pulse limits of 0:

```
(DELAY
  (ABSOLUTE
    (IOPATH A Z ((5) () ())) )
```

Annotations in INCREMENT mode can result in pulse limits less than 0, in which case they shall be adjusted to 0. For example, if the specify path pulse limits were both 3, this annotation results in a 0 value for both pulse limits:

```
(DELAY
  (INCREMENT
    (IOPATH A Z (( ) (-4) (-5)) )
```

There are two SDF constructs that annotate only to pulse limits, PATHPULSE and PATHPULSEPERCENT. They do not affect the delay. When PATHPULSE sets the pulse limits to values greater than the delay Verilog shall exhibit the same behavior as if the pulse limits had been set equal to the delay.

16.6 SDF to Verilog delay value mapping

Verilog specify paths and interconnects can have unique delays for up to twelve state transitions (see 14.3.1). All other constructs, such as gate primitives and continuous assignments, can have only three state transition delays (see

7.14).

For Verilog specify path and interconnect delays, the number of transition delay values provided by SDF might be less than twelve.

Table 16-4 shows how fewer than twelve SDF delays are extended to be twelve delays. The Verilog transition types are shown down the left hand side, while the number of SDF delays provided is shown across the top. The SDF values are given the names v1 through v12.

Table 16-4—SDF to Verilog delay value mapping

Verilog transition	Number of SDF delay values provided				
	1 value	2 values	3 values	6 values	12 values
0 -> 1	v1	v1	v1	v1	v1
1 -> 0	v1	v2	v2	v2	v2
0 -> z	v1	v1	v3	v3	v3
z -> 1	v1	v1	v1	v4	v4
1 -> z	v1	v2	v3	v5	v5
z -> 0	v1	v2	v2	v6	v6
0 -> x	v1	v1	min(v1,v3)	min(v1,v3)	v7
x -> 1	v1	v1	v1	max(v1,v4)	v8
1 -> x	v1	v2	min(v2,v3)	min(v2,v5)	v9
x -> 0	v1	v2	v2	max(v2,v6)	v10
x -> z	v1	max(v1,v2)	v3	max(v3,v5)	v11
z -> x	v1	min(v1,v2)	min(v1,v2)	min(v4,v6)	v12

For other delays that can have at most three values, the expansion of less than three SDF delays into three Verilog delays is covered in Table 7-9. More than three SDF delays are reduced to three Verilog delays by simply ignoring the extra delays. The delay to the X-state is created from the minimum of the other three delays.

Section 17

System tasks and functions

This section describes system tasks and functions that are considered part of the Verilog HDL. These system tasks and functions are divided into ten categories as follows:

Display tasks	[17.1]	PLA modeling tasks	[17.5]
\$display	\$strobe	\$async\$and\$array	\$async\$and\$plane
\$displayb	\$strobeb	\$async\$nand\$array	\$async\$nand\$plane
\$displayh	\$strobeh	\$async\$or\$array	\$async\$or\$plane
\$displayo	\$strobo	\$async\$nor\$array	\$async\$nor\$plane
\$monitor	\$write	\$sync\$and\$array	\$sync\$and\$plane
\$monitorb	\$writeb	\$sync\$nand\$array	\$sync\$nand\$plane
\$monitorh	\$writeh	\$sync\$or\$array	\$sync\$or\$plane
\$monitro	\$writeo	\$sync\$nor\$array	\$sync\$nor\$plane
\$monitoroff	\$monitoron		
File I/O tasks	[17.2]	Stochastic analysis tasks	[17.6]
\$fclose	\$fopen	\$q_initialize	\$q_add
\$fdisplay	\$fstrobe	\$q_remove	\$q_full
\$fdisplayb	\$fstrobeb	\$q_exam	
\$fdisplayh	\$fstrobeh		
\$fdisplayo	\$fstrobo		
\$fgetc	\$fputc	Simulation time functions	[17.7]
\$fflush	\$ferror	\$realtime	\$stime
\$fgets	\$frewind	\$time	
\$fmonitor	\$fwrite		
\$fmonitorb	\$fwriteb	Conversion functions	[17.8]
\$fmonitorh	\$fwriteh	\$bitstoreal	\$realto bits
\$fmonitro	\$fwroteo	\$itor	\$rtoi
\$readmemb	\$readmemh	\$signed	\$unsigned
\$swrite	\$swriteb		
\$swriteo	\$swriteh		
\$sformat	\$sdf_annotate		
\$fscanf	\$sscanf	Probabilistic distribution functions	[17.9]
\$fread	\$ftell	\$dist_chi_square	\$dist_erlang
\$fseek		\$dist_exponential	\$dist_normal
		\$dist_poisson	\$dist_t
		\$dist_uniform	\$random
Timescale tasks	[17.3]		
\$prnttimescale	\$timeformat		
Simulation control tasks	[17.4]	Command line input	[17.10]
\$finish	\$stop	\$test\$plusargs	\$value\$plusargs

These utility tasks and functions provide some broadly useful capabilities. The following clauses describe the behavior of these tasks and functions. Additional tasks for value change dump (VCD) are described in Section 18.

17.1 Display system tasks

The display group of system tasks are divided into three categories: the display and write tasks, strobed monitoring tasks, and continuous monitoring tasks.

17.1.1 The display and write tasks

```
display_tasks ::= (Not in the Annex A BNF)
display_task_name ( list_of_arguments ) ;
display_task_name ::=
    $display | $displayb | $displayo | $displayh
    | $write | $writeb | $writeo | $writeh
```

Syntax 17-1—Syntax for \$display and \$write system tasks

These are the main system task routines for displaying information. The two sets of tasks are identical except that **\$display** automatically adds a newline character to the end of its output, whereas the **\$write** task does not.

The **\$display** and **\$write** tasks display their arguments in the same order as they appear in the argument list. Each argument can be a quoted string, an expression that returns a value, or a null argument.

The contents of string arguments are output literally except when certain escape sequences are inserted to display special characters or to specify the display format for a subsequent expression.

Escape sequences are inserted into a string in three ways:

- The special character `\` indicates that the character to follow is a literal or nonprintable character (see Table 17-1).
- The special character `%` indicates that the next character should be interpreted as a format specification that establishes the display format for a subsequent expression argument (see Table 17-2). For each `%` character that appears in a string, a corresponding expression argument shall be supplied after the string.
- The special character string `%%` indicates the display of the percent sign character `%` (see Table 17-1).

Any null argument produces a single space character in the display. (A null argument is characterized by two adjacent commas in the argument list.)

The **\$display** task, when invoked without arguments, simply prints a newline character. A **\$write** task supplied without parameters prints nothing at all.

17.1.1.1 Escape sequences for special characters

The escape sequences given in Table 17-1, when included in a string argument, cause special characters to be displayed.

Table 17-1—Escape sequences for printing special characters

Argument	Description
\n	The newline character
\t	The tab character
\\	The \ character
\"	The " character
\ddd	A character specified by 1 to 3 octal digits
%%	The % character

Example:

```

module disp;
initial begin
    $display( "\\t\\n\"123" );
end
endmodule

\      \
"S

```

17.1.1.2 Format specifications

Table 17-2 shows the escape sequences used for format specifications. Each escape sequence, when included in a string argument, specifies the display format for a subsequent expression. For each % character (except %m) that appears in a string, a corresponding expression shall follow the string in the argument list. The value of the expression replaces the format specification when the string is displayed.

Any expression argument that has no corresponding format specification is displayed using the default decimal format in **\$display** and **\$write**, binary format in **\$displayb** and **\$writeb**, octal format in **\$displayo** and **\$writeo**, and hexadecimal format in **\$displayh** and **\$writeh**.

Table 17-2—Escape sequences for format specifications

Argument	Description
%h or %H	Display in hexadecimal format
%d or %D	Display in decimal format
%o or %O	Display in octal format
%b or %B	Display in binary format
%c or %C	Display in ASCII character format
%l or %L	Display library binding information
%v or %V	Display net signal strength

Table 17-2—Escape sequences for format specifications (*continued*)

%m or %M	Display hierarchical name
%s or %S	Display as a string
%t or %T	Display in current time format
%u or %U	Unformatted 2 value data
%z or %Z	Unformatted 4 value data

The formatting specification %l (or %L) is defined for displaying the library information of the specific module. This information shall be displayed as "*library.cell*" corresponding to the library name the current module instance was extracted from and the cell name of the current module instance. See Section 13 for information on libraries and configuring designs.

The formatting specification %u (or %U) is defined for writing data without formatting (binary values). The application shall transfer the 2 value binary representation of the specified data to the output stream. This escape sequence may be used with any of the existing display system tasks, although \$fwrite should be the preferred one to use. Any unknown or high-impedance bits in the source shall be treated as zero. This formatting specifier is intended to be used to support transferring data to and from external programs that have no concept of x and z. Applications that require preservation of x and z are encouraged to use the %z I/O format specification.

The data shall be written to the file in the native endian format of the underlying system (i.e., in the same endian order as if the PLI was used, and the C language write(2) system call was used). The data will be written in units of 32 bits with the word containing the LSB written first.

NOTE—For POSIX applications: It may be necessary to open files for unformatted I/O with the wb, wb+, or w+b specifiers, to avoid the systems implementation of I/O altering patterns in the unformatted stream that match special characters.

The formatting specification %z (or %Z) is defined for writing data without formatting (binary values). The application shall transfer the 4 value binary representation of the specified data to the output stream. This escape sequence may be used with any of the existing display system tasks, although \$fwrite should be the preferred one to use.

This formatting specifier is intended to be used to support transferring data to and from external programs that recognize and support the concept of x and z. Applications that do not require the preservation of x and z are encouraged to use the %u I/O format specification.

The data shall be written to the file in the native endian format of the underlying system (i.e., in the same endian order as if the PLI was used, and the data were in a s_vpi_vecval structure (See 27.14, Figure 27-8), and the C language write(2) system call was used to write the structure to disk). The data will be written in units of the native size of an integer on the machine, which is typically 32 bits.

NOTE—For POSIX applications: It may be necessary to open files for unformatted I/O with the wb, wb+ or w+b specifiers, to avoid the systems implementation of I/O altering patterns in the unformatted stream that match special characters.

The format specifications in Table 17-3 are used with real numbers and have the full formatting capabilities available in the C language. For example, the format specification %10.3g specifies a minimum field width of 10 with 3 fractional digits.

Table 17-3—Format specifications for real numbers

Argument	Description
%e or %E	Display ‘real’ in an exponential format
%f or %F	Display ‘real’ in a decimal format
%g or %G	Display ‘real’ in exponential or decimal format, whichever format results in the shorter printed output

The net signal strength, hierarchical name, and string format specifications are described in 17.1.1.5 through 17.1.1.7.

The `%t` format specification works with the **\$timeformat** system task to specify a uniform time unit, time precision, and format for reporting timing information from various modules that use different time units and precisions. The **\$timeformat** task is described in 17.3.2.

Example:

```

module disp;
reg [31:0] rval;
pulldown (pd);
initial begin
    rval = 101;
    $display("rval = %h hex %d decimal",rval,rval);
    $display("rval = %o octal\nrval = %b bin",rval,rval);
    $display("rval has %c ascii character value",rval);
    $display("pd strength value is %v",pd);
    $display("current scope is %m");
    $display("%s is ascii value for 101",101);
    $display("simulation time is %t", $time);
end
endmodule

rval = 000000065 hex          101 decimal
rval = 00000000145 octal
rval = 0000000000000000000000001100101 bin
rval has e ascii character value
pd strength value is StX
current scope is disp
e is ascii value for 101
simulation time is              0

```

17.1.1.3 Size of displayed data

For expression arguments, the values written to the output file (or terminal) are sized automatically.

For example, the result of a 12-bit expression would be allocated three characters when displayed in hexadecimal format and four characters when displayed in decimal format, since the largest possible value for the expression is FFF (hexadecimal) and 4095 (decimal).

When displaying decimal values, leading zeros are suppressed and replaced by spaces. In other radices, leading zeros are always displayed.

The automatic sizing of displayed data may be overridden by inserting a zero between the % character and the letter that indicates the radix, as shown in the following example.

```
$display("d=%0h a=%0h", data, addr);
```

Example:

```
module printval;
reg [11:0] r1;
initial begin
    r1 = 10;
    $display( "Printing with maximum size - :%d: :%h:", r1,r1 );
    $display( "Printing with minimum size - :%0d: :%0h:", r1,r1 );
end
endmodule

Printing with maximum size - : 10: :00a:
Printing with minimum size - :10: :a:
```

In this example, the result of a 12-bit expression is displayed. The first call to **\$display** uses the standard format specifier syntax and produces results requiring four and three columns for the decimal and hexadecimal radices, respectively. The second **\$display** call uses the %0 form of the format specifier syntax and produces results requiring two columns and one column, respectively.

17.1.1.4 Unknown and high impedance values

When the result of an expression contains an unknown or high impedance value, the following rules apply to displaying that value.

In decimal (%d) format

- If all bits are at the unknown value, a single lowercase x character is displayed.
- If all bits are at the high impedance value, a single lowercase z character is displayed.
- If some, but not all, bits are at the unknown value, the uppercase X character is displayed.
- If some, but not all, bits are at the high impedance value, the uppercase Z character is displayed.
- Decimal numerals always appear right-justified in a fixed-width field.

In hexadecimal (%h) and octal (%o) formats

- Each group of 4 bits is represented as a single hexadecimal digit; each group of 3 bits is represented as a single octal digit.
- If all bits in a group are at the unknown value, a lowercase x is displayed for that digit.
- If all bits in a group are at a high impedance state, a lowercase z is printed for that digit.
- If some, but not all, bits in a group are unknown, an uppercase X is displayed for that digit.
- If some, but not all, bits in a group are at a high impedance state, then an uppercase Z is displayed for that digit.

In binary (%b) format, each bit is printed separately using the characters 0, 1, x, and z.

Example:

STATEMENT	RESULT
<code>\$display("%d", 1'b0);</code>	x
<code>\$display("%h", 14'b01010);</code>	xxXa
<code>\$display("%h %o", 12'b001xxx101x01, 12'b001xxx101x01);</code>	XXX 1x5X

17.1.1.5 Strength format

The %v format specification is used to display the strength of scalar nets. For each %v specification that appears in a string, a corresponding scalar reference shall follow the string in the argument list.

The strength of a scalar net is reported in a three-character format. The first two characters indicate the strength. The third character indicates the current logic value of the scalar and may be any one of the values given in Table 17-4.

Table 17-4—Logic value component of strength format

Argument	Description
0	For a logic 0 value
1	For a logic 1 value
X	For an unknown value
Z	For a high impedance value
L	For a logic 0 or high impedance value
H	For a logic 1 or high impedance value

The first two characters—the strength characters—are either a two-letter mnemonic or a pair of decimal digits. Usually, a mnemonic is used to indicate strength information; however, in less typical cases, a pair of decimal digits may be used to indicate a range of strength levels. Table 17-5 shows the mnemonics used to represent the various strength levels.

Table 17-5—Mnemonics for strength levels

Mnemonic	Strength name	Strength level
Su	Supply drive	7
St	Strong drive	6
Pu	Pull drive	5
La	Large capacitor	4
We	Weak drive	3
Me	Medium capacitor	2

Table 17-5—Mnemonics for strength levels (*continued*)

Mnemonic	Strength name	Strength level
Sm	Small capacitor	1
Hi	High impedance	0

Note that there are four driving strengths and three charge storage strengths. The driving strengths are associated with gate outputs and continuous assignment outputs. The charge storage strengths are associated with the **triereg** type net. (See Section 7 for strength modeling.)

For the logic values 0 and 1, a mnemonic is used when there is no range of strengths in the signal. Otherwise, the logic value is preceded by two decimal digits, which indicate the maximum and minimum strength levels.

For the unknown value, a mnemonic is used when both the 0 and 1 strength components are at the same strength level. Otherwise, the unknown value X is preceded by two decimal digits, which indicate the 0 and 1 strength levels respectively.

The high impedance strength cannot have a known logic value; the only logic value allowed for this level is Z.

For the values L and H, a mnemonic is always used to indicate the strength level.

Examples:

always

```
#15 $display($time, , "group=%b signals=%v %v %v", {s1,s2,s3}, s1, s2, s3);
```

The example below shows the output that might result from such a call, while Table 17-6 explains the various strength formats that appear in the output.

```
0 group=111 signals=St1 Pu1 St1
15 group=011 signals=Pu0 Pu1 St1
30 group=0xz signals=520 PuH HiZ
45 group=0xx signals=Pu0 65X StX
60 group=000 signals=Me0 St0 St0
```

Table 17-6—Explanation of strength formats

Argument	Description
St1	Means a strong driving 1 value
Pu0	Means a pull driving 0 value
HiZ	Means the high-impedance state
Me0	Means a 0 charge storage of medium capacitor strength
StX	Means a strong driving unknown value

Table 17-6—Explanation of strength formats (*continued*)

Argument	Description
PuH	Means a pull driving strength of 1 or high-impedance value
65X	Means an unknown value with a strong driving 0 component and a pull driving 1 component
520	Means an 0 value with a range of possible strength from pull driving to medium capacitor

17.1.1.6 Hierarchical name format

The %m format specifier does not accept an argument. Instead, it causes the display task to print the hierarchical name of the module, task, function, or named block that invokes the system task containing the format specifier. This is useful when there are many instances of the module that calls the system task. One obvious application is timing check messages in a flip-flop or latch module; the %m format specifier will pinpoint the module instance responsible for generating the timing check message.

17.1.1.7 String format

The %s format specifier is used to print ASCII codes as characters. For each %s specification that appears in a string, a corresponding parameter shall follow the string in the argument list. The associated argument is interpreted as a sequence of 8-bit hexadecimal ASCII codes, with each 8 bits representing a single character. If the argument is a variable, its value should be right-justified so that the rightmost bit of the value is the least-significant bit of the last character in the string. No termination character or value is required at the end of a string, and leading zeros are never printed.

17.1.2 Strobed monitoring

```

strobe_tasks ::= (Not in the Annex A BNF)
    strobe_task_name ( list_of_arguments );
strobe_task_name ::=
    $strobe | $strobeb | $strobeo | $strobeh

```

Syntax 17-2—Syntax for \$strobe system tasks

The system task **\$strobe** provides the ability to display simulation data at a selected time. That time is the end of the current simulation time, when all the simulation events that have occurred for that simulation time, just before simulation time is advanced. The arguments for this task are specified in exactly the same manner as for the **\$display** system task—including the use of escape sequences for special characters and format specifications (see 17.1.1).

Example:

```

forever @(negedge clock)
    $strobe ("At time %d, data is %h", $time, data);

```

In this example, **\$strobe** will write the time and data information to the standard output and the log file at each negative edge of the clock. The action will occur just before simulation time is advanced and after all other events at that time have occurred, so that the data written is sure to be the correct data for that simulation time.

17.1.3 Continuous monitoring

```
monitor_tasks ::= (Not in the Annex A BNF)
    monitor_task_name [ ( list_of_arguments ) ] ;
    | $monitoron ;
    | $monitoroff ;
monitor_task_name ::=
    $monitor | $monitorb | $monitoro | $monitorh
```

Syntax 17-3—Syntax for \$monitor system tasks

The **\$monitor** task provides the ability to monitor and display the values of any variables or expressions specified as arguments to the task. The arguments for this task are specified in exactly the same manner as for the **\$display** system task—including the use of escape sequences for special characters and format specifications (see 17.1.1).

When a **\$monitor** task is invoked with one or more arguments, the simulator sets up a mechanism whereby each time a variable or an expression in the argument list changes value—with the exception of the **\$time**, **\$stime** or **\$realtime** system functions—the entire argument list is displayed at the end of the time step as if reported by the **\$display** task. If two or more arguments change value at the same time, only one display is produced that shows the new values.

Only one **\$monitor** display list can be active at any one time; however, a new **\$monitor** task with a new display list may be issued any number of times during simulation.

The **\$monitoron** and **\$monitoroff** tasks control a monitor flag that enables and disables the monitoring. Use **\$monitoroff** to turn off the flag and disable monitoring. The **\$monitoron** system task can be used to turn on the flag so that monitoring is enabled and the most recent call to **\$monitor** can resume its display. A call to **\$monitoron** shall produce a display immediately after it is invoked, regardless of whether a value change has taken place; this is used to establish the initial values at the beginning of a monitoring session. By default, the monitor flag is turned on at the beginning of simulation.

17.2 File input-output system tasks and functions

The system tasks and functions for file-based operations are divided into three categories:

- Functions and tasks that open and close files
- Tasks that output values into files
- Tasks that output values into variables
- Tasks and functions that read values from files and load into variables or memories

17.2.1 Opening and closing files

```
file_open_function ::= (Not in the Annex A BNF)
    integer multi_channel_descriptor = $fopen ( " file_name " );
    | integer fd = $fopen ( " file_name ", type );
file_close_task ::=
    $fclose ( multi_channel_descriptor );
    | $fclose ( fd );
```

Syntax 17-4—Syntax for \$fopen and \$fclose system tasks

The function **\$fopen** opens the file specified as the **filename** argument and returns either a 32 bit multi channel descriptor, or a 32 bit file descriptor, determined by the absence or presence of the **type** argument.

filename is a character string, or a reg containing a character string that names the file to be opened.

type is a character string, or a reg containing a character string of one of the following forms in the table below, which indicates how the file should be opened. If **type** is omitted, the file is opened for writing, and a multi channel descriptor **mcd** is returned. If **type** is supplied, the file is opened as specified by the value of **type**, and a file descriptor **fd** is returned.

The multi channel descriptor **mcd** is a 32 bit reg in which a single bit is set indicating which file is opened. The least significant bit (bit 0) of a **mcd** always refers to the standard output. Output is directed to two or more files opened with multi channel descriptors by bitwise oring together their **mcds** and writing to the resultant value.

The most significant bit (bit 32) of a multi channel descriptor is reserved, and will always be cleared, limiting an implementation to at most 31 files opened for output via multi channel descriptors.

The file descriptor **fd** is a 32 bit value. The most significant bit (bit 32) of a **fd** is reserved, and shall always be set; this allows implementations of the file input and output functions to determine how the file was opened. The remaining bits hold a small number indicating what file is opened. Three file descriptors are pre opened; they are **STDIN**, **STDOUT** and **STDERR**, which have the values 32'h8000_0000, 32'h8000_0001 and 32'h8000_0002, respectively. **STDIN** is pre opened for reading, and **STDOUT** and **STDERR** are pre opened for append.

Unlike multi channel descriptors, file descriptors can not be combined via bitwise or in order to direct output to multiple files. Instead, files are opened via file descriptor for input, output, input and output, as well as for append operations, based on the value of **type**, according to the following table:

Table 17-7—Types for file descriptors

Argument	Description
"r" or "rb"	open for reading
"w" or "wb"	truncate to zero length or create for writing
"a" or "ab"	append; open for writing at end of file, or create for writing
"r+", "r+b", or "rb+"	open for update (reading and writing)
"w+", "w+b", or "wb+"	truncate or create for update
"a+", "a+b", or "ab+"	append; open or create for update at end-of-file

If a file can not be opened (either the file doesn't exist, and the **type** specified is "r", "rb", "r+", "r+b", or "rb+", or the permissions do not allow the file to be opened at that path, a zero is returned for either the **mcd** or the **fd**. Applications may call **\$ferror** to determine the cause of the most recent error (see 17.2.7).

The "b" in the above types exists to distinguish binary files from text files. Many systems (such as Unix) make no distinction between binary and text files, and on these systems the "b" is ignored. However, some systems (such as machines running NT or Windows) will perform data mappings on certain binary values written to and read from files that are opened for text access.

The **\$fclose** system tasks closes the file specified by **fd** or closes the file(s) specified by the multi channel descriptor **mcd**. No further output to or input from any file descriptor(s) closed by **\$fclose** is allowed. Active **\$fmonitor** and/or **\$fstrobe** operations on a file descriptor or multi channel descriptor are implicitly cancelled by an **\$fclose** operation. The **\$fopen** function shall reuse channels that have been closed.

NOTE—The number of simultaneous input and output channels that may be open at any one time is dependent on the operating system. Some operating systems may not support opening files for update.

17.2.2 File output system tasks

```

file_output_tasks ::= (Not in the Annex A BNF)
    file_output_task_name ( multi_channel_descriptor , list_of_arguments ) ;
    | file_output_task_name ( fd , list_of_arguments ) ;
file_output_task_name ::=
    $fdisplay | $fdisplayb | $fdisplayh | $fdisplayo
    | $fwrite | $fwriteb | $fwriteh | $fwriteo
    | $fstrobe | $fstrobeb | $fstrobeh | $fstrobeo
    | $fmonitor | $fmonitorb | $fmonitorh | $fmonitoro

```

Syntax 17-5—Syntax for file output system tasks

Each of the four formatted display tasks—**\$display**, **\$write**, **\$monitor**, and **\$strobe**—has a counterpart that writes to specific files as opposed to the standard output. These counterpart tasks—**\$fdisplay**, **\$fwrite**, **\$fmonitor**, and **\$fstrobe**—accept the same type of arguments as the tasks upon which they are based, with one exception: The first parameter shall be either a multi channel descriptor or a file descriptor, which indicates where to direct the file output. Multi channel descriptors are described in detail in 17.2.1. A multichannel descriptor is either a variable or the result of an expression that takes the form of a 32-bit unsigned integer value.

The **\$fstrobe** and **\$fmonitor** system tasks work just like their counterparts, **\$strobe** and **\$monitor**, except that they write to files using the multi channel descriptor for control. Unlike **\$monitor**, any number of **\$fmonitor** tasks can be set up to be simultaneously active. However, there is no counterpart to **\$monitoron** and **\$monitroff** tasks. The task **\$fclose** is used to cancel an active **\$fstrobe** or **\$fmonitor** task.

Example:

This example shows how to set up multi channel descriptors. In this example, three different channels are opened using the **\$fopen** function. The three multi channel descriptors that are returned by the function are then combined in a bit-wise or operation and assigned to the integer variable `messages`. The `messages` variable can then be used as the first parameter in a file output task to direct output to all three channels at once. To create a descriptor that directs output to the standard output as well, the `messages` variable is a bit-wise logical or with the constant 1, which effectively enables channel 0.

```

integer
    messages,          broadcast,
    cpu_chann,         alu_chann, mem_chann;
initial begin
    cpu_chann = $fopen("cpu.dat");
    if (cpu_chann == 0) $finish;
    alu_chann = $fopen("alu.dat");
    if (alu_chann == 0) $finish;
    mem_chann = $fopen("mem.dat");
    if (mem_chann == 0) $finish;
    messages = cpu_chann | alu_chann | mem_chann;
    // broadcast includes standard output
    broadcast = 1 | messages;
end
endmodule

```

The following file output tasks show how the channels opened in the preceding example might be used:

```

$fdisplay( broadcast, "system reset at time %d", $time );

$fdisplay( messages, "Error occurred on address bus",
           " at time %d, address = %h", $time, address );

forever @(posedge clock)
    $fdisplay( alu_chann, "acc= %h f=%h a=%h b=%h", acc, f, a, b );

```

17.2.3 Formatting data to a string

```

string_output_tasks ::= (Not in the Annex A BNF)
    string_output_tasks_name ( output_reg, list_of_arguments );
string_output_task_name ::=
    $swrite | $swriteb | $swriteh | $swriteo
variable_format_string_output_task ::=
    $sformat ( output_reg, format, list_of_arguments );

```

Syntax 17-6—Syntax for formatting data tasks

The syntax for the string output system tasks is

```

$swrite( output_reg, list_of_arguments );
$sformat( output_reg, format_string, list_of_arguments );
length = $sformat( output_reg, format_string, list_of_arguments );

```

The **\$swrite** family of tasks are based on the **\$fwrite** family of tasks, and accept the same type of arguments as the tasks upon which they are based, with one exception: The first parameter to **\$swrite** shall be a reg variable to which the resulting string shall be written, instead of a variable specifying the file to which to write the resulting string.

The variable *output_reg* is assigned using the Verilog's string assignment to variable rules, as specified in 4.2.3.

The system task `$sformat` is similar to the system task `$swrite`, with a one major difference.

Unlike the display and write family of output system tasks, `$sformat` always interprets its second argument, and only its second argument as a format string. This format argument can be a static string, such as "data is %d", or can be a reg variable whose content is interpreted as the format string. No other arguments are interpreted as format strings. `$sformat` supports all the format specifiers supported by `$display`, as documented in 17.1.1.2.

The remaining arguments to `$sformat` are processed using any format specifiers in the *format_string*, until all such format specifiers are used up. If not enough arguments are supplied for the format specifiers, or too many are supplied, then the application shall issue a warning, and continue execution. The application, if possible, may statically determine a mismatch in format specifiers and number of arguments, and issue a compile time error message.

NOTE—If the *format_string* is a reg, it may not be possible to determine its value at compile time.

The variable *output_reg* is assigned using the Verilog's string assignment to variable rules, as specified in 4.2.3.

17.2.4 Reading data from a file

Files opened using file descriptors may be read from, and only those files opened with **type** of either the "r" or "r+" values. See 17.2.1 for more information about opening files.

17.2.4.1 Reading a character at a time

```
c = $fgetc ( fd );
```

Read a byte from the file specified by *fd*. If an error occurs reading from the file, then **c** is set to EOF (-1). Care should be taken to define the width of the reg which gets the return value of `$fgetc` to be wider than 8 bits so that the return of EOF (-1) may be determined from a return of a byte with the value 0xff. Applications may call `$ferror` to determine the cause of the most recent error (see 17.2.7).

```
code = $ungetc ( c, fd );
```

Insert the character specified by **c** into the buffer specified by file descriptor *fd*. The character **c** will be returned by the next `$fgetc` call on that file descriptor. The file itself is unchanged. Note that the features of the underlying implementation of fileio on the host system will limit the number of characters that may be pushed back onto a stream. Note also that operations like `$fseek` might erase any pushed back characters. If an error occurs pushing a character onto a file descriptor, then **code** is set to EOF. Otherwise **code** is set to zero. Applications may call `$ferror` to determine the cause of the most recent error (see 17.2.7).

17.2.4.2 Reading a line at a time

```
integer code = $fgets ( str, fd );
```

Read characters from the file specified by *fd* into the reg *str* until either *str* is filled, or a newline character is read and transferred to *str*, or an end-of-file condition is encountered. If *str* is not an integral number of bytes in length, the most significant partial byte is not used in order to determine the size.

If an error occurs reading from the file, then **code** is set to zero. Otherwise the number of characters read is returned in **code**. Applications may call `$ferror` to determine the cause of the most recent error (see below).

17.2.4.3 Reading formatted data

```
integer code = $fscanf ( fd, format, args );  
integer code = $sscanf ( str, format, args );
```


\$fscanf reads from the file descriptor `fd`.

\$sscanf reads from the reg `str`.

Both functions read characters, interpret them according to a format, and stores the results in its arguments. Both expect as arguments a control string, `format`, and a set of arguments specifying where to place the results. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are ignored.

If an argument is too small to hold the converted input, then in general, the least significant bits are transferred. Arguments of any length that is supported by Verilog may be used. However if the destination is a **real** or **realtime** then the value `+Inf` (or `-Inf`) is transferred. The format may be a string constant or a reg containing a string constant. The string contains conversion specifications, which direct the conversion of input into the arguments. The control string may contain

- a) White-space characters (blanks, tabs, new-lines, or form-feeds) that, except in one case described below, cause input to be read up to the next non-white-space character.
- b) An ordinary character (not `%`) that must match the next character of the input stream.
- c) Conversion specifications consisting of the character `%` an optional assignment suppression character `*`, a decimal digit string that specifies an optional numerical maximum field width, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the variable specified in the corresponding argument unless assignment suppression was indicated by the character `*`; in this case no argument shall be supplied.

The suppression of assignment provides a way of describing an input field that is to be skipped. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the maximum field width, if one is specified, is exhausted. For all descriptors except the character `c`, white space leading an input field is ignored.

- `%` A single `%` is expected in the input at this point; no assignment is done.
- `b` Matches a binary number, consisting of a sequence from the set `0,1,X,x,Z,z,?` and `_`.
- `o` Matches a octal number, consisting of a sequence of characters from the set `0,1,2,3,4,5,6,7,X,x,Z,z,?` and `_`.
- `d` Matches an optionally signed decimal number, consisting of the optional sign from the set `+` or `-`, followed by a sequence of characters from the set `0,1,2,3,4,5,6,7,8,9` and `_`, or a single value from the set `x,X,z,Z,?`.
- `h` or `x` Matches a hexadecimal number, consisting of a sequence of characters from the set `0,1,2,3,4,5,6,7,8,9,a,A,b,B,c,C,d,D,e,E,f,F,x,X,z,Z,?` and `_`.
- `f` `e` or `g` Matches a floating point number. The format of a floating point number is an optional sign (either `+` or `-`), followed by a string of digits from the set `0,1,2,3,4,5,6,7,8,9` optionally containing a decimal point character (`.`), then an optional exponent part including `e` or `E` followed by an optional sign, followed by a string of digits from the set `0,1,2,3,4,5,6,7,8,9`.
- `v` Matches a net signal strength, consisting of three character sequence as specified in 17.1.1.5. This conversion is not extremely useful, as strength values are really only usefully assigned to nets and **\$fscanf** can only assign values to regs (if assigned to regs, the values are converted to the 4 value equivalent).
- `t` Matches a floating point number. The format of a floating point number is an optional sign (either `+` or `-`), followed by a string of digits from the set `0,1,2,3,4,5,6,7,8,9` optionally containing a decimal point character (`.`), then an optional exponent part including `e` or `E` followed by an optional sign, followed by a string of digits from the set `0,1,2,3,4,5,6,7,8,9`. The value matched is then scaled and rounded according to the current

time scale as set by **\$timeformat**. For example, if the timescale is ``timescale 1ns/100ps` and the time format is **\$timeformat**(-3,2," ms",10);, then a value read with **\$sscanf**("10.345", "%t", t) would return 10350000.0.

- c Matches a single character, whose 8 bit ASCII value is returned.
- s Matches a string, which is a sequence of non white space characters.
- u Matches unformatted (binary) data. The application shall transfer sufficient data from the input to fill the target reg. Typically the data is obtained from a matching `$fwrite (" %u",data)`, or from an external application written in another programming language such as C, Perl or FORTRAN.

The application shall transfer the 2 value binary data from the input stream to the destination reg, expanding the data to the four value format. This escape sequence may be used with any of the existing input system tasks, although `$fscanf` should be the preferred one to use. As the input data can not represent x or z, it is not possible to obtain an x or z in the result reg. This formatting specifier is intended to be used to support transferring data to and from external programs that have no concept of x and z.

Applications that require preservation of x and z are encouraged to use the `%z i/o` format specification.

The data shall be read from the file in the native endian format of the underlying system (i.e., in the same endian order as if the PLI was used, and the C language `read(2)` system call was used).

For POSIX applications: It may be necessary to open files for unformatted I/O with the "rb", "rb+" or "r+b" specifiers, to avoid the systems implementation of I/O altering patterns in the unformatted stream that match special characters.

The formatting specification `%z` (or `%Z`) is defined for reading data without formatting (binary values). The application shall transfer the 4 value binary representation of the specified data from the input stream to the destination reg. This escape sequence may be used with any of the existing input system tasks, although `$fscanf` should be the preferred one to use.

This formatting specifier is intended to be used to support transferring data to and from external programs that recognize and support the concept of x and z. Applications that do not require the preservation of x and z are encouraged to use the `%u i/o` format specification.

The data shall be read from the file in the native endian format of the underlying system (i.e., in the same endian order as if the PLI was used, and the data were in a `s_vpi_vecval` structure (See 27.14, Figure 27-8), and the C language `read(2)` system call was used to read the data from disk).

For POSIX applications: It may be necessary to open files for unformatted I/O with the "rb", "rb+" or "r+b" specifiers, to avoid the systems implementation of I/O altering patterns in the unformatted stream that match special characters.

- m Returns the current hierarchical path as a string. Does not read data from the input file or str argument. If an invalid conversion character follows the %, the results of the operation are implementation dependent.

If the format string, or the `str` argument to **\$sscanf** contains unknown bits (x or z) then the system task shall return EOF.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (if any) is terminated with an input failure.

If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream. Trailing white space (including new-line characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable.

The number of successfully matched and assigned input items is returned in `code`; this number can be 0 in the event of an early matching failure between an input character and the control string. If the input ends before the first matching failure or conversion, EOF is returned. Applications may call **\$ferror** to determine the cause of the most recent error (see below).

17.2.4.4 Reading binary data

```
integer code = $fread( myreg, fd);
integer code = $fread( mem, fd);
integer code = $fread( mem, fd, start);
integer code = $fread( mem, fd, start, count);
integer code = $fread( mem, fd, , count);
```

Read a binary data from the file specified by `fd` into the reg `myreg` or the memory `mem`.

`Start` is an optional argument. If present, `start` will be used as the starting location in the memory. If not present the least significant location in the memory shall be used.

`Count` is an optional argument. If present, `count` will be the maximum number of locations in `mem` that will be loaded. If not supplied the memory will be filled with what data is available.

`Start` and `count` are be ignored if **\$fread** is loading a reg.

\$fread shall store data into a memory starting with the lowest numbered location, continuing up to the higher location. For the memory declared **up[10:20]**, the first location loaded will be **up[10]**, next will be **up[11]**, up to **up[20]**. For the memory declared **down[20:10]**, the first location loaded will be **down[10]**, then **down[11]**, down to **down[20]**.

`start` is the word offset from the lowest element in the memory. For `start = 2` and the memory **up[10:20]**, the first data would be loaded at **up[12]**. For the memory **down[20:10]**, the first location loaded would be **down[12]**, then **down[13]**.

The data in the file shall be read byte by byte to fulfill the request. An 8-bit wide memory is loaded using one byte per memory word, while a 9-bit wide memory is loaded using 2 bytes per memory word. The data is read from the file in a big endian manner; the first byte read is used to fill the most significant location in the memory element. If the memory width is not evenly divisible by 8 (8, 16, 24, 32), not all data in the file will be loaded into memory because of truncation.

The data loaded from the file is taken as "two value" data. A bit set in the data is interpreted as a 1, and bit not set is interpreted as a 0. It is not possible to read a value of `x` or `z` using **\$fread**.

If an error occurs reading from the file, then `code` is set to zero. Otherwise the number of characters read is returned in `code`. Applications may call **\$ferror** to determine the cause of the most recent error (see 17.2.7).

Note that there is not a "binary" mode and a "ASCII" mode; one may freely intermingle binary and formatted read commands from the same file.

17.2.5 File positioning

```
integer pos = $ftell ( fd );
```

Returns in `pos` the offset from the beginning of the file of the current byte of the file `fd` which will be read or written by a subsequent operation on that file descriptor.

This value may be used by subsequent **\$fseek** calls to reposition the file to this point. Note that any repositioning will cancel any **\$ungetc** operations. If an error occurs EOF is returned. Applications may call **\$ferror** to determine the cause of the most recent error (see 17.2.7).

```
code = $fseek ( fd, offset, operation );
code = $rewind ( fd );
```

Sets the position of the next input or output operation on the file specified by `fd`. The new position is at the signed distance offset bytes from the beginning, from the current position, or from the end of the file, according to an operation value of 0, 1 and 2 as follows:

- 0 set position equal to offset bytes
- 1 set position to current location plus offset
- 2 set position to EOF plus offset

\$rewind is equivalent to **\$fseek** (`fd`, 0, 0);

Repositioning the current file position with **\$fseek** or **\$rewind** shall cancel any **\$ungetc** operations.

\$fseek() allows the file position indicator to be set beyond the end of the existing data in the file. If data is later written at this point, subsequent reads of data in the gap will return zero until data is actually written into the gap. **\$fseek**, by itself, does not extend the size of the file.

When a file is opened for append (that is, when `type` is "a", or "a+"), it is impossible to overwrite information already in the file. **\$fseek** may be used to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is disregarded. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output.

If an error occurs repositioning the file, then `code` is set to -1. Otherwise `code` is set to 0. Applications may call **\$ferror** to determine the cause of the most recent error (see 17.2.7).

17.2.6 Flushing output

```
$fflush ( mcd );
$fflush ( fd );
$fflush ( );
```

Writes any buffered output to the file(s) specified by `mcd`, the file specified by `fd` or if **\$fflush** is invoked with no arguments, writes any buffered output to all open files.

17.2.7 I/O error status

Should any error be detected by one of the fileio routines, an error code is returned. Often this is sufficient for normal operation; (i.e., if the opening of a optional configuration file fails, the application typically would simply continue using default values.) However sometimes it is useful to obtain more information about the error for correct application operation. In this case the **\$ferror** function may be used:

```
integer errno = $ferror ( fd, str );
```

A string description of type of error encountered by the most recent file I/O operation is written into `str` which should be at least 640 bits wide. The integral value of the error code is returned in `errno`. If the most recent operation did not result in an error, then the value returned will be zero, and the reg `str` shall be cleared.

17.2.8 Loading memory data from a file

```
load_memory_tasks ::= (Not in the Annex A BNF)
    $readmemb ( " file_name ", memory_name [ , start_addr [ , finish_addr ] ] ) ;
    | $readmemh ( " file_name ", memory_name [ , start_addr [ , finish_addr ] ] ) ;
```

Syntax 17-7—Syntax for memory load system tasks

Two system tasks—**\$readmemb** and **\$readmemh**—read and load data from a specified text file into a specified memory. Either task may be executed at any time during simulation. The text file to be read shall contain only the following:

- White space (spaces, new lines, tabs, and form-feeds)
- Comments (both types of comment are allowed)
- Binary or hexadecimal numbers

The numbers shall have neither the length nor the base format specified. For **\$readmemb**, each number shall be binary. For **\$readmemh**, the numbers shall be hexadecimal. The unknown value (x or X), the high impedance value (z or Z), and the underscore (_) can be used in specifying a number as in a Verilog HDL source description. White space and/or comments shall be used to separate the numbers.

In the following discussion, the term “address” refers to an index into the array that models the memory.

As the file is read, each number encountered is assigned to a successive word element of the memory. Addressing is controlled both by specifying start and/or finish addresses in the system task invocation and by specifying addresses in the data file.

When addresses appear in the data file, the format is an “at” character (@) followed by a hexadecimal number as follows:

@hh . . . h

Both uppercase and lowercase digits are allowed in the number. No white space is allowed between the @ and the number. As many address specifications as needed within the data file may be used. When the system task encounters an address specification, it loads subsequent data starting at that memory address.

If no addressing information is specified within the system task, and no address specifications appear within the data file, then the default start address is the left-hand address given in the declaration of the memory. Consecutive words are loaded until either the memory is full or the data file is completely read. If the start address is specified in the task without the finish address, then loading starts at the specified start address and continues towards the right-hand address given in the declaration of the memory.

If both start and finish addresses are specified as parameters to the task, then loading begins at the start address and continues toward the finish address, regardless of how the addresses are specified in the memory declaration.

When addressing information is specified both in the system task and in the data file, the addresses in the data file shall be within the address range specified by the system task parameters; otherwise, an error message is issued and the load operation is terminated.

A warning message is issued if the number of data words in the file differs from the number of words in the range implied by the start through finish addresses.

Example:

```
reg [7:0] mem[1:256];
```

Given this declaration, each of the following statements will load data into mem in a different manner:

```
initial $readmemh("mem.data", mem);
initial $readmemh("mem.data", mem, 16);
initial $readmemh("mem.data", mem, 128, 1);
```

The first statement will load up the memory at simulation time 0 starting at the memory address 1. The second statement will begin loading at address 16 and continue on towards address 256. For the third and final statement, loading will begin at address 128 and continue down towards address 1.

In the third case, when loading is complete, a final check is performed to ensure that exactly 128 numbers are contained in the file. If the check fails, a warning message is issued.

17.2.9 Loading timing data from an SDF file

The syntax for the **\$sdf_annotate** system task is shown in Syntax 17-8.

```
sdf_annotate_task ::= (Not in the Annex A BNF)
    $sdf_annotate("sdf_file" [, [ module_instance ] [, [ "config_file" ]
        [, [ "log_file" ] [, [ "mtm_spec" ]
        [, [ "scale_factors" ] [, [ "scale_type" ] ] ] ] ] ] );
```

Syntax 17-8—Syntax for \$sdf_annotate system task

The **\$sdf_annotate** system task reads timing data from an SDF file into a specified region of the design.

sdf_file is a character string, or a reg containing a character string naming the file to be opened.

module_instance is an optional argument specifying the scope to which to annotate the information in the SDF file. The SDF annotator uses the hierarchy level of the specified instance for running the annotation. Array indices are permitted. If the *module_instance* not specified, the SDF Annotator uses the module containing the call to the **\$sdf_annotate** system task as the *module_instance* for annotation.

config_file is an optional character string argument providing the name of a configuration file. Information in this file can be used to provide detailed control over many aspects of annotation.

log_file is an optional character string argument providing the name of the log file used during SDF annotation. Each individual annotation of timing data from the SDF file results in an entry in the log file.

mtm_spec is an optional character string argument specifying which member of the min/typ/max triples will be annotated. The legal values for this string are described in Table 17-8. This overrides any MTM_SPEC keywords in the configuration file.

Table 17-8—mtm spec argument

Keyword	Description
MAXIMUM	Annotate the maximum value
MINIMUM	Annotate the minimum value
TOOL_CONTROL (default)	Annotate the value as selected by the simulator
TYPICAL	Annotate the typical value

scale_factors is an optional character string argument specifying the scale factors to be used while annotating timing values. For example, "1.6:1.4:1.2" causes minimum values to be multiplied by 1.6, typical values by 1.4, and maximum values by 1.2. The default values are 1.0:1.0:1.0. The *scale_factors* argument overrides any SCALE_FACTORS keywords in the configuration file.

scale_type is an optional character string argument specifying how the scale factors should be applied to the min/typ/max triples. The legal values for this string are shown in Table 17-9. This overrides any SCALE_TYPE keywords in the configuration file.

Table 17-9—scale type argument

Keyword	Description
FROM_MAXIMUM	Apply scale factors to maximum value
FROM_MINIMUM	Apply scale factors to minimum value
FROM_MTM (default)	Apply scale factors to min/typ/max values
FROM_TYPICAL	Apply scale factors to typical value

17.3 Timescale system tasks

The following system tasks display and set timescale information:

- a) **\$printtimescale**
- b) **\$timeformat**

17.3.1 \$printtimescale

The **\$printtimescale** system task displays the time unit and precision for a particular module. The syntax for the system task is shown in Syntax 17-9.

<pre>printtimescale_task ::= (Not in the Annex A BNF) \$printtimescale [(hierarchical_identifier)] ;</pre>

Syntax 17-9—Syntax for \$printtimescale

This system task can be specified with or without an argument.

- When no argument is specified, **\$printtimescale** displays the time unit and precision of the module that is the current scope.
- When an argument is specified, **\$printtimescale** displays the time unit and precision of the module passed to it.

The timescale information appears in the following format:

```
Time scale of (module_name) is unit / precision
```

Example:

```
`timescale 1 ms / 1 us
module a_dat;
initial
    $printtimescale(b_dat.c1);
endmodule

`timescale 10 fs / 1 fs
module b_dat;
    c_dat c1 ();
endmodule

`timescale 1 ns / 1 ns
module c_dat;
    .
    .
    .
endmodule
```

In this example, module `a_dat` invokes the **\$printtimescale** system task to display timescale information about another module `c_dat`, which is instantiated in module `b_dat`.

The information about `c_dat` is displayed in the following format:

```
Time scale of (b_dat.c1) is 1ns / 1ns
```

17.3.2 \$timeformat

The syntax for **\$timeformat** system task is shown in Syntax 17-10.

timeformat_task ::= (Not in the Annex A BNF)

\$timeformat [(units_number , precision_number , suffix_string , minimum_field_width)] ;

Syntax 17-10—Syntax for \$timeformat

The **\$timeformat** system task performs the following two functions:

- It specifies how the %t format specification reports time information for the **\$write**, **\$display**, **\$strobe**, **\$monitor**, **\$fwrite**, **\$fdisplay**, **\$fstrobe**, and **\$fmonitor** group of system tasks.
- It specifies the time unit for delays entered interactively.

The units number argument shall be an integer in the range from 0 to -15. This argument represents the time unit as shown in Table 17-10.

Table 17-10—\$timeformat units_number arguments

Unit number	Time unit	Unit number	Time unit
0	1 s	-8	10 ns
-1	100 ms	-9	1 ns
-2	10 ms	-10	100 ps
-3	1 ms	-11	10 ps
-4	100 us	-12	1 ps
-5	10 us	-13	100 fs
-6	1 us	-14	10 fs
-7	100 ns	-15	1 fs

The **\$timeformat** system task performs the following two operations:

- It sets the time unit for all later-entered delays entered interactively.
- It sets the time unit, precision number, suffix string, and minimum field width for all %t formats specified in all modules that follow in the source description until another **\$timeformat** system task is invoked.

The default **\$timeformat** system task arguments are given in Table 17-11.

Table 17-11—\$timeformat default value for arguments

Argument	Default
units_number	The smallest time precision argument of all the `timescale compiler directives in the source description
precision_number	0
suffix_string	A null character string
minimum_field_width	20

Example:

The following example shows the use of %t with the **\$timeformat** system task to specify a uniform time unit, time precision, and format for timing information.

```
`timescale 1 ms / 1 ns
module cntrl;
initial
    $timeformat(-9, 5, " ns", 10);
endmodule

`timescale 1 fs / 1 fs
module a1_dat;
reg in1;
integer file;
buf #10000000 (o1,in1);
initial begin
    file = $fopen("a1.dat");
    #00000000 $fmonitor(file,"%m: %t in1=%d o1=%h", $realtime,in1,o1);
    #10000000 in1 = 0;
    #10000000 in1 = 1;
end
endmodule

`timescale 1 ps / 1 ps
module a2_dat;
reg in2;
integer file2;
buf #10000 (o2,in2);
initial begin
    file2=$fopen("a2.dat");
    #00000 $fmonitor(file2,"%m: %t in2=%d o2=%h",$realtime,in2,o2);
    #10000 in2 = 0;
    #10000 in2 = 1;
end
endmodule
```

The contents of file a1.dat are as follows:

```
a1_dat: 0.00000 ns in1= x o1=x
a1_dat: 10.00000 ns in1= 0 o1=x
a1_dat: 20.00000 ns in1= 1 o1=0
a1_dat: 30.00000 ns in1= 1 o1=1
```

The contents of file a2.dat are as follows:

```
a2_dat: 0.00000 ns in2=x o2=x
a2_dat: 10.00000 ns in2=0 o2=x
a2_dat: 20.00000 ns in2=1 o2=0
a2_dat: 30.00000 ns in2=1 o2=1
```

In this example, the times of events written to the files by the **\$fmonitor** system task in modules a1_dat and a2_dat are reported as multiples of 1 ns—even though the time units for these modules are 1 fs and 1 ps

respectively—because the first argument of the **\$timeformat** system task is -9 and the %t format specification is included in the arguments to **\$fmonitor**. This time information is reported after the module names with five fractional digits, followed by an ns character string in a space wide enough for 10 ASCII characters.

17.4 Simulation control system tasks

There are two simulation control system tasks:

- a) **\$finish**
- b) **\$stop**

17.4.1 \$finish

Syntax 17-11 shows the syntax for **\$finish** system task.

```
finish_task ::= (Not in the Annex A BNF)
$finish [ ( n ) ] ;
```

Syntax 17-11—Syntax for \$finish

The **\$finish** system task simply makes the simulator exit and pass control back to the host operating system. If an expression is supplied to this task, then its value determines the diagnostic messages that are printed before the prompt is issued. If no argument is supplied, then a value of 1 is taken as the default.

Table 17-12—Diagnostics for \$finish

Parameter value	Diagnostic message
0	Prints nothing
1	Prints simulation time and location
2	Prints simulation time, location, and statistics about the memory and CPU time used in simulation

17.4.2 \$stop

The syntax for **\$stop** system task is shown in Syntax 17-12.

```
stop_task ::= (Not in the Annex A BNF)
$stop [ ( n ) ] ;
```

Syntax 17-12—Syntax for \$stop

The **\$stop** system task causes simulation to be suspended. This task takes an optional expression argument (0, 1, or 2) that determines what type of diagnostic message is printed. The amount of diagnostic messages output increases with the value of the optional argument passed to **\$stop**.

17.5 PLA modeling system tasks

The modeling of PLA devices is provided in the Verilog HDL by a group of system tasks. This clause describes the syntax and use of these system tasks and the formats of the logic array personality file. The syntax for PLA modeling system task is shown in Syntax 17-13.

```

pla_system_task ::= (Not in the Annex A BNF)
    $array_type$logic$format ( memory_type , input_terms , output_terms ) ;
array_type ::=
    sync | async
logic ::=
    and | or | nand | nor
format ::=
    array | plane
input_terms ::=
    expression
output_terms ::=
    variable_lvalue

```

Syntax 17-13—Syntax for PLA modeling system task

NOTE—The input terms can be nets or variables whereas the output terms shall only be variables.

The PLA syntax allows for the system tasks as shown in Table 17-13.

Table 17-13—PLA modeling system tasks

\$async\$and\$array	\$sync\$and\$array	\$async\$and\$plane	\$sync\$and\$plane
\$async\$nand\$array	\$sync\$nand\$array	\$async\$nand\$plane	\$sync\$nand\$plane
\$async\$or\$array	\$sync\$or\$array	\$async\$or\$plane	\$sync\$or\$plane
\$async\$nor\$array	\$sync\$nor\$array	\$async\$nor\$plane	\$sync\$nor\$plane

17.5.1 Array types

The modeling of both synchronous and asynchronous arrays is provided by the PLA system tasks. The synchronous forms control the time at which the logic array will be evaluated and the outputs will be updated. For the asynchronous forms, the evaluations are automatically performed whenever an input term changes value or any word in the personality memory is changed.

For both the synchronous and asynchronous forms, the output terms are updated without any delay.

Examples:

An example of an asynchronous system call is as follows:

```

wire      a1, a2, a3, a4, a5, a6, a7;
reg       b1, b2, b3;
wire [1:7] awire;
reg  [1:3] breg;

```

```

$async$and$array(mem, {a1,a2,a3,a4,a5,a6,a7}, {b1,b2,b3});
or
$async$and$array(mem,awire, breg);

```

An example of a synchronous system call is as follows:

```

$sync$or$plane(mem, {a1,a2,a3,a4,a5,a6,a7}, {b1,b2,b3});

```

17.5.2 Array logic types

The logic arrays are modeled with and, or, nand, and nor logic planes. This applies to all array types and formats.

Examples:

An example of a nor plane system call is as follows:

```

$async$nor$plane(mem, {a1,a2,a3,a4,a5,a6,a7}, {b1,b2,b3});

```

An example of a nand plane system call is as follows:

```

$sync$nand$plane(mem, {a1,a2,a3,a4,a5,a6,a7}, {b1,b2,b3});

```

17.5.3 Logic array personality declaration and loading

The logic array personality is declared as an array of regs that is as wide as the number of input terms and as deep as the number of output terms.

The personality of the logic array is normally loaded into the memory from a text data file using the system tasks **\$readmemb** or **\$readmemh**. Alternatively, the personality data may be written directly into the memory using the procedural assignment statements. PLA personalities may be changed dynamically at any time during simulation simply by changing the contents of the memory. The new personality will be reflected on the outputs of the logic array at the next evaluation.

Example:

The following example shows a logic array with *n* input terms and *m* output terms.

```

reg [1:n] mem[1:m];

```

NOTE—Put PLA input terms, output terms, and memory in ascending order, as shown in examples in this clause.

17.5.4 Logic array personality formats

Two separate personality formats are supported by the Verilog HDL and are differentiated by using either an array system call or a plane system call. The array system call allows for a 1 or 0 in the memory that has been declared. A 1 means take the input value and a 0 means do not take the input value.

The plane system call complies with the University of California at Berkeley format for Espresso. Each bit of the data stored in the array has the following meaning:

- 0 Take the complemented input value
- 1 Take the true input value
- x* Take the “worst case” of the input value

z Don't-care; the input value is of no significance

$?$ Same as z

Examples:

Example 1—The following example illustrates an array with logic equations:

```
b1 = a1 & a2
b2 = a3 & a4 & a5
b3 = a5 & a6 & a7
```

The PLA personality is as follows:

```
1100000 in mem[1]
0011100 in mem[2]
0000111 in mem[3]
```

The module for the PLA is as follows:

```
module async_array(a1,a2,a3,a4,a5,a6,a7,b1,b2,b3);
input a1, a2, a3, a4, a5, a6, a7 ;
output b1, b2, b3;
reg [1:7] mem[1:3]; // memory declaration for array personality
reg b1, b2, b3;
initial begin
    // setup the personality from the file array.dat
    $readmemb("array.dat", mem);
    // setup an asynchronous logic array with the input
    // and output terms expressed as concatenations
    $async$and$array(mem, {a1,a2,a3,a4,a5,a6,a7}, {b1,b2,b3});
end
endmodule
```

Where the file array.dat contains the binary data for the PLA personality:

```
1100000
0011100
0000111
```

A synchronous version of this example has the following description:

```

module sync_array(a1,a2,a3,a4,a5,a6,a7,b1,b2,b3,clk);
input a1, a2, a3, a4, a5, a6, a7, clk;
output b1, b2, b3;
reg [1:7] mem[1:3]; // memory declaration
reg b1, b2, b3;
initial begin
    // setup the personality
    $readmemb("array.dat", mem);
    // setup a synchronous logic array to be evaluated
    // when a positive edge on the clock occurs
    forever @(posedge clk)
        $async$and$array(mem, {a1,a2,a3,a4,a5,a6,a7}, {b1,b2,b3});
end
endmodule

```

Example 2—An example of the usage of the plane format tasks follows. The logical function of this PLA is shown first, followed by the PLA personality in the new format, the Verilog HDL description using the **\$async\$and\$plane** system task, and finally the result of running the simulation.

The logical function of the PLA is as follows:

```

b[1] = a[1] & ~a[2];
b[2] = a[3];
b[3] = ~a[1] & ~a[3];
b[4] = 1;

```

The PLA personality is as follows:

```

3'b10?
3'b??1
3'b0?0
3'b???

```

```

module pla;
`define rows 4
`define cols 3
reg [1:`cols] a, mem[1:`rows];
reg [1:`rows] b;
initial begin
    // PLA system call
    $async$and$plane(mem,a[1:3],b[1:4]);
    mem[1] = 3'b10?;
    mem[2] = 3'b??1;
    mem[3] = 3'b0?0;
    mem[4] = 3'b???;
    // stimulus and display
    #10 a = 3'b111;
    #10 $displayb(a, " -> ", b);
    #10 a = 3'b000;
    #10 $displayb(a, " -> ", b);
    #10 a = 3'bxxx;
    #10 $displayb(a, " -> ", b);
    #10 a = 3'b101;
    #10 $displayb(a, " -> ", b);
end
endmodule

```

The output is as follows:

```

111 -> 0101
000 -> 0011
xxx -> xxx1
101 -> 1101

```

17.6 Stochastic analysis tasks

This clause describes a set of system tasks and functions that manage queues and generate random numbers with specific distributions. These tasks facilitate implementation of stochastic queueing models.

The set of tasks and functions that create and manage queues follow:

```

$q_initialize (q_id, q_type, max_length, status);
$q_add (q_id, job_id, inform_id, status);
$q_remove (q_id, job_id, inform_id, status);
$q_full (q_id, status);
$q_exam (q_id, q_stat_code, q_stat_value, status);

```

17.6.1 \$q_initialize

The **\$q_initialize** system task creates new queues. The *q_id* parameter is an integer input that shall uniquely identify the new queue. The *q_type* parameter is an integer input. The value of the *q_type* parameter specifies the type

of the queue as shown in Table 17-14.

Table 17-14—Types of queues of \$q_type values

q_type value	Type of queue
1	first-in, first-out
2	last-in, first-out

The maximum length parameter is an integer input that specifies the maximum number of entries that will be allowed on the queue. The success or failure of the creation of the queue is returned as an integer value in status. The error conditions and corresponding values of status are described in Table 17-14.

17.6.2 \$q_add

The **\$q_add** system task places an entry on a queue. The `q_id` parameter is an integer input that indicates to which queue to add the entry. The `job_id` parameter is an integer input that identifies the job.

The `inform_id` parameter is an integer input that is associated with the queue entry. Its meaning is user-defined. For example, `inform_id` parameter can represent execution time for an entry in a CPU model. The status parameter reports on the success of the operation or error conditions as described in Table 17-14.

17.6.3 \$q_remove

The **\$q_remove** system task receives an entry from a queue. The `q_id` parameter is an integer input that indicates from which queue to remove. The `job_id` parameter is an integer output that identifies the entry being removed. The `inform_id` parameter is an integer output that the queue manager stored during **\$q_add**. Its meaning is user-defined. The status parameter reports on the success of the operation or error conditions as described in Table 17-14.

17.6.4 \$q_full

The **\$q_full** system function checks whether there is room for another entry on a queue. It returns 0 when the queue is not full and 1 when the queue is full.

17.6.5 \$q_exam

The **\$q_exam** system task provides statistical information about activity at the queue `q_id`. It returns a value in `q_stat_value` depending on the information requested in `q_stat_code`. The values of `q_stat_code` and the corresponding information returned in `q_stat_value` are described in Table 17-15.

Table 17-15—Parameter values for \$q_exam system task

Value requested in q_stat_code	Information received back from q_stat_value
1	Current queue length
2	Mean interarrival time
3	Maximum queue length
4	Shortest wait time ever
5	Longest wait time for jobs still in the queue
6	Average wait time in the queue

17.6.6 Status codes

All of the queue management tasks and functions return an output status parameter. The status parameter values and corresponding information are described in Table 17-16.

Table 17-16—Status parameter values

Status parameter values	What it means
0	OK
1	Queue full, cannot add
2	Undefined q_id
3	Queue empty, cannot remove
4	Unsupported queue type, cannot create queue
5	Specified length ≤ 0 , cannot create queue
6	Duplicate q_id, cannot create queue
7	Not enough memory, cannot create queue

17.7 Simulation time system functions

The following system functions provide access to current simulation time:

\$time **\$stime** **\$realtime**

17.7.1 \$time

The syntax for **\$time** system function is shown in Syntax 17-14.

time_function ::= *(Not in the Annex A BNF)*
\$time

Syntax 17-14—Syntax for \$time

The **\$time** system function returns an integer that is a 64-bit time, scaled to the timescale unit of the module that invoked it.

Example:

```
`timescale 10 ns / 1 ns
module test;
reg set;
parameter p = 1.55;
initial begin
    $monitor($time, , "set=", set);
    #p set = 0;
    #p set = 1;
end
endmodule

// The output from this example is as follows:
// 0 set=x
// 2 set=0
// 3 set=1
```

In this example, the reg `set` is assigned the value 0 at simulation time 16 ns, and the value 1 at simulation time 32 ns. Note that these times do not match the times reported by `$time`. The time values returned by the `$time` system function are determined by the following steps:

- a) The simulation times 16ns and 32 ns are scaled to 1.6 and 3.2 because the time unit for the module is 10 ns, so time values reported by this module are multiples of 10 ns.
- b) The value 1.6 is rounded to 2, and 3.2 is rounded to 3 because the `$time` system function returns an integer. The time precision does not cause rounding of these values.

17.7.2 \$time

The syntax for `$time` system function is shown in Syntax 17-15.

stime_function ::= (Not in the Annex A BNF)
\$time

Syntax 17-15—Syntax for \$time

The `$time` system function returns an unsigned integer that is a 32-bit time, scaled to the timescale unit of the module that invoked it. If the actual simulation time does not fit in 32 bits, the low order 32 bits of the current simulation time are returned.

17.7.3 \$realtime

The syntax for `$realtime` system function is shown in Syntax 17-16.

realtime_function ::= **\$realtime** (Not in the Annex A BNF)

Syntax 17-16—Syntax for \$realtime

The **\$realtime** system function returns a real number time that, like **\$time**, is scaled to the time unit of the module that invoked it.

Example:

```
`timescale 10 ns / 1 ns
module test;
reg set;
parameter p = 1.55;
initial begin
    $monitor($realtime, , "set=", set);
    #p set = 0;
    #p set = 1;
end
endmodule

// The output from this example is as follows:
// 0 set=x
// 1.6 set=0
// 3.2 set=1
```

In this example, the event times in the reg **set** are multiples of 10 ns because 10 ns is the time unit of the module. They are real numbers because **\$realtime** returns a real number.

17.8 Conversion functions

The following functions handle **real** values:

integer	\$rtoi (real_val) ;
real	\$itor (int_val) ;
[63:0]	\$realtobits (real_val) ;
real	\$bitstoreal (bit_val) ;

\$rtoi converts real values to integers by truncating the real value (for example, 123.45 becomes 123)

\$itor converts integers to real values (for example, 123 becomes 123.0)

\$realtobits passes bit patterns across module ports; converts from a real number to the 64-bit representation (vector) of that real number

\$bitstoreal is the reverse of **\$realtobits**; converts from the bit pattern to a real number.

The real numbers accepted or generated by these functions shall conform to the *IEEE Std 754-1985* [B1] representation of the real number. The conversion shall round the result to the nearest valid representation.

Example:

The following example shows how the **\$realtobits** and **\$bitstoreal** functions are used in port connections:

```

module driver (net_r);
output net_r;
real r;
wire [64:1] net_r = $realtobits(r);
endmodule

module receiver (net_r);
input net_r;
wire [64:1] net_r;
real r;
initial assign r = $bitstoreal(net_r);
endmodule

```

See 4.5 for a description of **\$signed** and **\$unsigned**.

17.9 Probabilistic distribution functions

There are a set of random number generators that return integer values distributed according to standard probabilistic functions.

17.9.1 \$random function

The syntax for the system function **\$random** is shown in Syntax 17-17.

random_function ::= (Not in the Annex A BNF)
\$random [(seed)] ;

Syntax 17-17—Syntax for \$random

The system function **\$random** provides a mechanism for generating random numbers. The function returns a new 32-bit random number each time it is called. The random number is a signed integer; it can be positive or negative. For further information on probabilistic random number generators, see 17.9.2.

The seed parameter controls the numbers that **\$random** returns such that different seeds generate different random streams. The seed parameter shall be either a reg, an integer, or a time variable. The seed value should be assigned to this variable prior to calling **\$random**.

Examples:

Example 1—Where b is greater than 0, the expression (**\$random** % b) gives a number in the following range: [(-b+1) : (b-1)]. The following code fragment shows an example of random number generation between -59 and 59:

```

reg [23:0] rand;
rand = $random % 60;

```

Example 2—The following example shows how adding the concatenation operator to the preceding example gives rand a positive value from 0 to 59.

```
reg [23:0] rand;
rand = {$random} % 60;
```

17.9.2 \$dist_ functions

```
dist_functions ::= (Not in the Annex A BNF)
    $dist_uniform ( seed , start , end ) ;
    | $dist_normal ( seed , mean , standard_deviation ) ;
    | $dist_exponential ( seed , mean ) ;
    | $dist_poisson ( seed , mean ) ;
    | $dist_chi_square ( seed , degree_of_freedom ) ;
    | $dist_t ( seed , degree_of_freedom ) ;
    | $dist_erlang ( seed , k_stage , mean ) ;
```

Syntax 17-18—Syntax for the probabilistic distribution functions

All parameters to the system functions are integer values. For the exponential, poisson, chi-square, t, and erlang functions, the parameters mean, degree of freedom, and k_stage shall be greater than 0.

Each of these functions returns a pseudo-random number whose characteristics are described by the function name. That is, **\$dist_uniform** returns random numbers uniformly distributed in the interval specified by its parameters.

For each system function, the seed parameter is an in-out parameter; that is, a value is passed to the function and a different value is returned. The system functions will always return the same value given the same seed. This facilitates debugging by making the operation of the system repeatable. The argument for the seed parameter should be an integer variable that is initialized by the user and only updated by the system function. This will ensure that the desired distribution is achieved.

In the **\$dist_uniform** function, the start and end parameters are integer inputs that bound the values returned. The start value should be smaller than the end value.

The mean parameter, used by **\$dist_normal**, **\$dist_exponential**, **\$dist_poisson**, and **\$dist_erlang**, is an integer input that causes the average value returned by the function to approach the value specified.

The standard deviation parameter used with the **\$dist_normal** function is an integer input that helps determine the shape of the density function. Larger numbers for standard deviation will spread the returned values over a wider range.

The degree of freedom parameter used with the **\$dist_chi_square** and **\$dist_t** functions is an integer input that helps determine the shape of the density function. Larger numbers will spread the returned values over a wider range.

17.9.3 Algorithm for probabilistic distribution functions

Table 17-17 shows the Verilog probabilistic distribution functions listed with their corresponding C functions.

Table 17-17—Verilog to C function cross-listing

Verilog function	C function
\$dist_uniform	rtl_dist_uniform
\$dist_normal	rtl_dist_normal
\$dist_exponential	rtl_dist_exponential

Table 17-17—Verilog to C function cross-listing (*continued*)

Verilog function	C function
\$dist_poisson	rtl_dist_poisson
\$dist_chi_square	rtl_dist_chi_square
\$dist_t	rtl_dist_t
\$dist_erlang	rtl_dist_erlang
\$random	rtl_dist_uniform(seed, LONG_MIN, LONG_MAX)

The algorithm for these functions is defined by the following C code.

```

/*
 * Algorithm for probabilistic distribution functions.
 *
 * IEEE Std 1364-2000 Verilog Hardware Description Language (HDL).
 */

#include <limits.h>

static double uniform( long *seed, long start, long end );
static double normal( long *seed, long mean, long deviation);
static double exponential( long *seed, long mean);
static long poisson( long *seed, long mean);
static double chi_square( long *seed, long deg_of_free);
static double t( long *seed, long deg_of_free);
static double erlangian( long *seed, long k, long mean);

long
rtl_dist_chi_square( seed, df )
    long *seed;
    long df;
{
    double r;
    long i;

    if(df>0)
    {
        r=chi_square(seed,df);
        if(r>=0)
        {
            i=(long)(r+0.5);
        }
        else
        {
            r = -r;
            i=(long)(r+0.5);
            i = -i;
        }
    }
    else

```

```

    {
        print_error("WARNING: Chi_square distribution must have positive
                    degree of freedom\n");
        i=0;
    }

    return (i);
}

long
rtl_dist_erlang( seed, k, mean )
    long *seed;
    long k, mean;
{
    double r;
    long i;

    if(k>0)
    {
        r=erlangian(seed,k,mean);
        if(r>=0)
        {
            i=(long)(r+0.5);
        }
        else
        {
            r = -r;
            i=(long)(r+0.5);
            i = -i;
        }
    }
    else
    {
        print_error("WARNING: k-stage erlangian distribution must have
                    positive k\n");
        i=0;
    }

    return (i);
}

long
rtl_dist_exponential( seed, mean )
    long *seed;
    long mean;
{
    double r;
    long i;

    if(mean>0)
    {
        r=exponential(seed,mean);
        if(r>=0)

```



```

        {
            i=(long)(r+0.5);
        }
        else

        {
            r = -r;
            i=(long)(r+0.5);
            i = -i;
        }
    }
    else
    {
        print_error("WARNING: Exponential distribution must have a
                    positive mean\n");
        i=0;
    }

    return (i);
}

long
rtl_dist_normal( seed, mean, sd )
    long *seed;
    long mean, sd;
{
    double r;
    long i;

    r=normal(seed,mean,sd);
    if(r>=0)
    {
        i=(long)(r+0.5);
    }
    else
    {
        r = -r;
        i=(long)(r+0.5);
        i = -i;
    }

    return (i);
}

long
rtl_dist_poisson( seed, mean )
    long *seed;
    long mean;
{
    long i;

    if(mean>0)
    {
        i=poisson(seed,mean);
    }
    else

```

```

        {
            print_error("WARNING: Poisson distribution must have a positive
                        mean\n");
            i=0;
        }
        return (i);
    }

long
rtl_dist_t( seed, df )
    long *seed;
    long df;
{
    double r;
    long i;

    if(df>0)
    {
        r=t(seed,df);
        if(r>=0)
        {
            i=(long)(r+0.5);
        }
        else
        {
            r = -r;
            i=(long)(r+0.5);
            i = -i;
        }
    }
    else
    {
        print_error("WARNING: t distribution must have positive degree
                    of freedom\n");
        i=0;
    }
    return (i);
}

long
rtl_dist_uniform(seed, start, end)
    long *seed;
    long start, end;
{
    double r;
    long i;

    if (start >= end) return(start);

    if (end != LONG_MAX)
    {
        end++;
        r = uniform( seed, start, end );
        if (r >= 0)

```

```

        {
            i = (long) r;
        }
        else
        {
            i = (long) (r-1);
        }
        if (i<start) i = start;
        if (i>=end) i = end-1;
    }
    else if (start!=LONG_MIN)
    {
        start--;
        r = uniform( seed, start, end) + 1.0;
        if (r>=0)
        {
            i = (long) r;
        }
        else
        {
            i = (long) (r-1);
        }
        if (i<=start) i = start+1;
        if (i>end) i = end;
    }
    else
    {
        r =(uniform(seed,start,end)+2147483648.0)/4294967295.0=;
        r = r*4294967296.0-2147483648.0;
        if (r>=0)
        {
            i = (long) r;
        }
        else
        {
            i = (long) (r-1);
        }
    }

    return (i);
}

static double
uniform( seed, start, end )
    long *seed, start, end;
{
    union u_s
    {
        float s;
        unsigned stemp;
    } u;

    double d = 0.00000011920928955078125;
    double a,b,c;

```

```

        if ((*seed) == 0)
            *seed = 259341593;

        if (start >= end)
        {
            a = 0.0;
            b = 2147483647.0;
        }
        else
        {
            a = (double) start;
            b = (double) end;
        }
        *seed = 69069 * (*seed) + 1;
        u.stemp = *seed;

        /*
         * This relies on IEEE floating point format
         */
        u.stemp = (u.stemp >> 9) | 0x3f800000;

        c = (double) u.s;

        c = c+(c*d);
        c = ((b - a) * (c - 1.0)) + a;

        return (c);
    }

static double
normal(seed,mean,deviation)
long *seed,mean,deviation;
{
    double v1,v2,s;
    double log(), sqrt();

    s = 1.0;
    while((s >= 1.0) || (s == 0.0))
    {
        v1 = uniform(seed,-1,1);
        v2 = uniform(seed,-1,1);
        s = v1 * v1 + v2 * v2;
    }
    s = v1 * sqrt(-2.0 * log(s) / s);
    v1 = (double) deviation;
    v2 = (double) mean;
    return(s * v1 + v2);
}

static double
exponential(seed,mean)
long *seed,mean;
{
    double log(),n;
    n = uniform(seed,0,1);

```

```

        if(n != 0)
        {
            n = -log(n) * mean;
        }
        return(n);
    }

static long
poisson(seed,mean)
long *seed,mean;
{
    long n;
    double p,q;
    double exp();

    n = 0;
    q = -(double)mean;
    p = exp(q);
    q = uniform(seed,0,1);
    while(p < q)
    {
        n++;
        q = uniform(seed,0,1) * q;
    }
    return(n);
}

static double
chi_square(seed,deg_of_free)
long *seed,deg_of_free;
{
    double x;
    long k;
    if(deg_of_free % 2)
    {
        x = normal(seed,0,1);
        x = x * x;
    }
    else
    {
        x = 0.0;
    }
    double log(),n;

    n = uniform(seed,0,1);
    if(n != 0)
    {
        n = -log(n) * mean;
    }
    return(n);
}

static long
poisson(seed,mean)
long *seed,mean;

```

```

{
    long n;
    double p,q;
    double exp();

    n = 0;
    q = -(double)mean;

    p = exp(q);
    q = uniform(seed,0,1);
    while(p < q)
    {
        n++;
        q = uniform(seed,0,1) * q;
    }
    return(n);
}

static double
chi_square(seed,deg_of_free)
long *seed,deg_of_free;
{
    double x;
    long k;
    if(deg_of_free % 2)
    {
        x = normal(seed,0,1);
        x = x * x;
    }
    else
    {
        x = 0.0;
    }
}

static double
t(seed,deg_of_free)
long *seed,deg_of_free;
{
    double sqrt(),x;
    double chi2 = chi_square(seed,deg_of_free);
    double div = chi2 / (double)deg_of_free;
    double root = sqrt(div);
    x = normal(seed,0,1) / root;
    return(x);
}

static double
erlangian(seed,k,mean)
long *seed,k,mean;
{
    double x,log(),a,b;
    long i;

    x=1.0;
    for(i=1;i<=k;i++)

```

```

    {
        x = x * uniform(seed,0,1);
    }
    a=(double)mean;
    b=(double)k;
    x= -a*log(x)/b;
    return(x);
}

```

17.10 Command line input

An alternative to reading a file to obtain information for use in the simulation is specifying information with the command to invoke the simulator. This information is in the form of a optional argument provided to the simulation. These arguments are visually distinguished from other simulator arguments by the starting with the plus (+) character.

These arguments, referred to below as *plusargs*, are accessible through the following system functions.

17.10.1 \$test\$plusargs (string)

This system function searches the list of plusargs for the provided string. The plusargs present on the command line are searched in the order provided. If the prefix of one of the supplied plusargs matches all characters in the provided string, a non-zero integer is returned. If no plusarg from the command line matches the string provided, the integer value zero (0) is returned.

Examples:

Run simulator with command: +HELLO

The Verilog code is:

```

initial begin
    if ($test$plusargs("HELLO"))    $display("Hello argument found.");
    if ($test$plusargs("HE"))        $display("The HE subset string is
detected.");
    if ($test$plusargs("H"))          $display("Argument starting with H found.");
    if ($test$plusargs("HELLO_HERE"))$display("Long argument.");
    if ($test$plusargs("HI"))         $display("Simple greeting.");
    if ($test$plusargs("LO"))         $display("Does not match.");
end

```

This would produce the following output:

```

Hello argument found.
The HE subset string is detected.
Argument starting with H found.

```

17.10.2 \$value\$plusargs (user_string, variable)

This system function searches the list of plusargs (like the **\$test\$plusargs** system function) for a user specified plus-arg string. The string is specified in the first argument to the system function as either a string or a register which is interpreted as a string. If the string is found, the remainder of the string is converted to the type specified in the *user_string* and the resulting value stored in the variable provided. If a string is found, the function returns a non-zero integer. If no string is found matching, the function returns the integer value zero and the variable provided is not modified. No warnings shall be generated when the function returns zero (0).

The *user_string* shall be of the form: "*plusarg_string format_string*". The format strings are the same as the **\$display** system tasks. These are the only valid ones (upper and lower case as well as a leading 0 forms are valid):

%d	decimal conversion
%o	octal conversion
%h	hexadecimal conversion
%b	binary conversion
%e	real exponential conversion
%f	real decimal conversion
%g	real decimal or exponential conversion
%s	string (no conversion)

The first string, from the list of *plusargs* provided to the simulator, which matches the *plusarg_string* portion of the *user_string* specified shall be the *plusarg* string available for conversion. The remainder string of the matching *plusarg* (the remainder is the part of the *plusarg* string after the portion which matches the users *plusarg_string*) shall be converted from a string into the format indicated by the format string and stored in the variable provided. If there is no remaining string, the value stored into the variable shall either be a zero (0) or an empty string value.

If the size of the variable is larger than the value after conversion, the value stored is zero (0) padded to the width of the variable. If the variable can not contain the value after conversion, the value shall be truncated. If the value is negative, the value shall be considered larger than the variable provided. If characters exist in the string available for conversion, which are illegal for the specified conversion, the variable shall be written with the value 'bx.

Examples:

```
+FINISH=10000 +TESTNAME=this_test +FREQ+5.6666 +FREQUENCY +TEST12

// Get clock to terminate simulation if specified.
real frequency;
reg 8*32:1 testname;
integer stop_clock;
if ($value$plusargs("FINISH=%d", stop_clock))
    begin
        repeat (stop_clock) @(posedge clk);
        $finish;
    end

// Get testname from plusarg.
if ($value$plusargs("TESTNAME=%s", testname))
    begin
        $display("Running test %0s.", testname);
        startTest();
    end

// Get frequency from command line; set default if not specified.
if (!$value$plusargs("FREQ+%0F", frequency))
    frequency = 8.33333; // 166MHz;

forever
    begin
        #frequency clk = 0;
        #frequency clk = 1;
    end

reg [64*8:1] pstring;
pstring = "+TEST%d";
```



```
if ($value$plusargs(pstring, test[31:0]))
  begin
    $display("Running test number %0d.", test);
    startTest();
  end
```

This code would have the following effects:

- The variable `test` would get the value `'d12`.
- The variable `stop_clock` obtains the value `10000`.
- The variable `testname` obtains the value `this_test`.
- The variable `frequency` obtains the value `5.6666`; note the final *plusarg* `+FREQUENCY` does not affect the value of the variable `frequency`.

The output is:

```
Running test this_test.
Running test number 12.
```


Section 18

Value change dump (VCD) files

A *value change dump (VCD) file* contains information about value changes on selected variables in the design stored by value change dump system tasks. Two types of VCD files exist:

- a) Four state: to represent variable changes in 0, 1, x, and z with no strength information.
- b) Extended: to represent variable changes in all states and strength information.

This section describes how to generate both types of VCD files and their format.

18.1 Creating the four state value change dump file

The steps involved in creating the four state VCD file are listed below and illustrated in Figure 18-1.

- a) Insert the VCD system tasks in the Verilog source file to define the dump file name and to specify the variables to be dumped.
- b) Run the simulation.

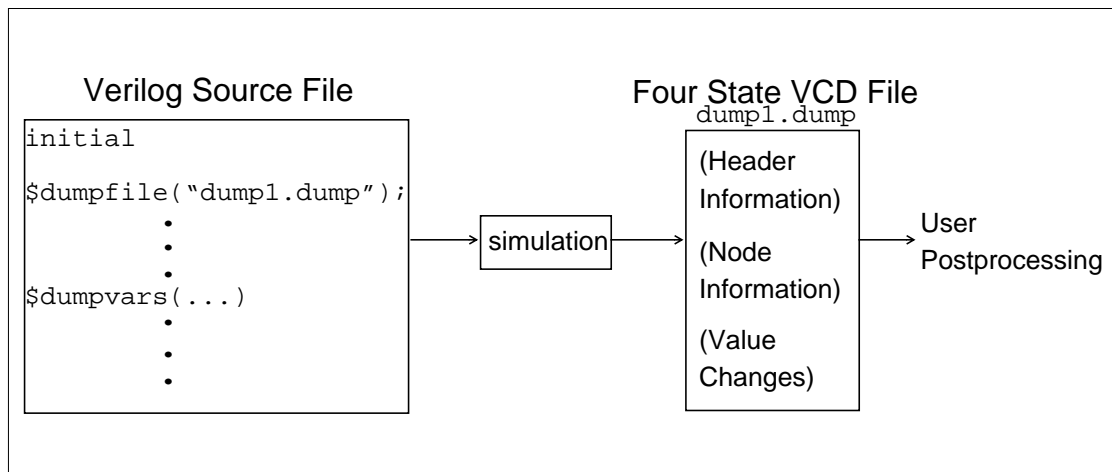


Figure 18-1—Creating the four state VCD file

A VCD file is an ASCII file which contains header information, variable definitions, and the value changes for all variables specified in the task calls.

Several system tasks can be inserted in the source description to create and control the VCD file.

18.1.1 Specifying the name of the dump file (\$dumpfile)

The **\$dumpfile** task shall be used to specify the name of the VCD file. The syntax for the task is given in Syntax 18-1.

```
dumpfile_task ::= (Not in the Annex A BNF)  
$dumpfile ( filename ) ;
```

Syntax 18-1—Syntax for \$dumpfile task

The *filename* syntax is given in Syntax 18-2.

```
filename ::= (Not in the Annex A BNF)  
          literal_string  
          | variable  
          | expression
```

Syntax 18-2—Syntax for filename

The *filename* is optional and defaults to the literal string `dump.vcd` if not specified.

Example:

```
initial    $dumpfile ( "module1.dump" ) ;
```

18.1.2 Specifying the variables to be dumped (\$dumpvars)

The **\$dumpvars** task shall be used to list which variables to dump into the file specified by **\$dumpfile**. The **\$dumpvars** task can be invoked as often as desired throughout the model (for example, within various blocks), but the execution of all the **\$dumpvars** tasks shall be at the same simulation time.

The **\$dumpvars** task can be used with or without arguments. The syntax for the **\$dumpvars** task is given in Syntax 18-3.

```
dumpvars_task ::= (Not in the Annex A BNF)  
$dumpvars ;  
| $dumpvars ( levels [ , list_of_modules_or_variables ] ) ;  
list_of_modules_or_variables ::= (Not in the Annex A BNF)  
    module_or_variable { , module_or_variable }  
module_or_variable ::=  
    module_identifier  
    | variable_identifier
```

Syntax 18-3—Syntax for \$dumpvars task

When invoked with no arguments, **\$dumpvars** dumps all the variables in the model to the VCD file.

When the **\$dumpvars** task is specified with arguments, the first argument indicates how many *levels* of the hierarchy below each specified module instance to dump to the VCD file. Subsequent arguments specify which scopes of the model to dump to the VCD file. These arguments can specify entire modules or individual variables within a module.

Setting the first argument to 0 causes a dump of all variables in the specified module and in all module instances below the specified module. The argument 0 applies only to subsequent arguments which specify module instances, and not to individual variables.

Examples:

Example 1

```
$dumpvars (1, top);
```

Because the first argument is a 1, this invocation dumps all variables within the module `top`; it does not dump variables in any of the modules instantiated by module `top`.

Example 2

```
$dumpvars (0, top);
```

In this example, the **\$dumpvars** task shall dump all variables in the module `top` and in all module instances below module `top` in the hierarchy.

Example 3—This example shows how the **\$dumpvars** task can specify both modules and individual variables:

```
$dumpvars (0, top.mod1, top.mod2.net1);
```

This call shall dump all variables in module `mod1` and in all module instances below `mod1`, along with variable `net1` in module `mod2`. The argument 0 applies only to the module instance `top.mod1` and not to the individual variable `top.mod2.net1`.

18.1.3 Stopping and resuming the dump (\$dumpoff/\$dumpon)

Executing the **\$dumpvars** task causes the value change dumping to start at the end of the current simulation time unit. To suspend the dump, the **\$dumpoff** task can be invoked. To resume the dump, the **\$dumpon** task can be invoked. The syntax of these two tasks is given in Syntax 18-4.

<pre>dumpoff_task ::= (Not in the Annex A BNF) \$dumpoff ; dumpon_task ::= (Not in the Annex A BNF) \$dumpon ;</pre>
--

Syntax 18-4—Syntax for \$dumpoff and \$dumpon tasks

When the **\$dumpoff** task is executed, a checkpoint is made in which every selected variable is dumped as an `x` value. When the **\$dumpon** task is later executed, each variable is dumped with its value at that time. In the interval between **\$dumpoff** and **\$dumpon**, no value changes are dumped.

The **\$dumpoff** and **\$dumpon** tasks provide the mechanism to control the simulation period during which the dump shall take place.

Example:

```

initial  begin
    #10    $dumpvars( . . . );

    #200   $dumpoff;

    #800   $dumpon;

    #900   $dumpoff;
end

```

This example starts the value change dump after 10 time units, stops it 200 time units later (at time 210), restarts it again 800 time units later (at time 1010), and stops it 900 time units later (at time 1910).

18.1.4 Generating a checkpoint (\$dumpall)

The **\$dumpall** task creates a checkpoint in the VCD file which shows the current value of all selected variables. The syntax is given in Syntax 18-5.

dumpall_task ::= (Not in the Annex A BNF)
\$dumpall ;

Syntax 18-5—Syntax for \$dumpall task

When dumping is enabled, the value change dumper records the values of the variables which change during each time increment. Values of variables which do not change during a time increment are not dumped.

18.1.5 Limiting the size of the dump file (\$dumplimit)

The **\$dumplimit** task can be used to set the size of the VCD file. The syntax for this task is given in Syntax 18-6.

dumplimit_task ::= (Not in the Annex A BNF)
\$dumplimit (filesize) ;

Syntax 18-6—Syntax for \$dumplimit task

The *filesize* argument which specifies the maximum size of the VCD file in bytes. When the size of VCD file reaches this number of bytes, the dumping stops and a comment is inserted in the VCD file indicating the dump limit was reached.

18.1.6 Reading the dump file during simulation (\$dumpflush)

The **\$dumpflush** task can be used to empty the VCD file buffer of the operating system to ensure all the data in that buffer is stored in the VCD file. After executing a **\$dumpflush** task, dumping is resumed as before so no value changes are lost. The syntax for the task is given in Syntax 18-7.

`dumpflush_task ::= (Not in the Annex A BNF)
$dumpflush ;`

Syntax 18-7—Syntax for \$dumpflush task

A common application is to call **\$dumpflush** to update the dump file so an application program can read the VCD file during a simulation.

Examples:

Example 1—This example shows how the **\$dumpflush** task can be used in a Verilog HDL source file:

```
initial begin  
    $dumpvars ;  
        .  
        .  
        .  
  
    $dumpflush ;  
  
    $(applications program) ;  
  
end
```

Example 2—The following is a simple source description example to produce a VCD file.

In this example, the name of the dump file is `verilog.dump`. It dumps value changes for all variables in the model. Dumping begins when an event `do_dump` occurs. The dumping continues for 500 clock cycles, then stops and waits for the event `do_dump` to be triggered again. At every 10000 time steps, the current values of all VCD variables are dumped.

```
module dump;
  event do_dump;

  initial $dumpfile("verilog.dump");
  initial @do_dump
    $dumpvars;          //dump variables in the design

  always @do_dump      //to begin the dump at event do_dump
  begin
    $dumpon;           //no effect the first time through
    repeat (500) @(posedge clock); //dump for 500 cycles
    $dumpoff;          //stop the dump
  end

  initial @(do_dump)
    forever #10000 $dumpall; //checkpoint all variables
endmodule
```

18.2 Format of the four state VCD file

The dump file is structured in a free format. White space is used to separate commands and to make the file easily readable by a text editor.

18.2.1 Syntax of the four state VCD file

The syntax of the four state VCD file is given in Syntax 18-8.


```

value_change_dump_definitions ::= (Not in the Annex A BNF)
    { declaration_command } { simulation_command }
declaration_command ::=
    declaration_keyword
    [ command_text ]
    $end
simulation_command ::=
    simulation_keyword { value_change } $end
    | $comment [ comment_text ] $end
    | simulation_time
    | value_change
declaration_keyword ::=
    $comment | $date | $enddefinitions | $scope | $timescale | $upscope
    | $var | $version
simulation_keyword ::=
    $dumpall | $dumpoff | $dumpon | $dumpvars
simulation_time ::=
    # decimal_number
value_change ::=
    scalar_value_change
    | vector_value_change
scalar_value_change ::=
    value identifier_code
value ::=
    0 | 1 | x | X | z | Z
vector_value_change ::=
    b binary_number identifier_code
    | B binary_number identifier_code
    | r real_number identifier_code
    | R real_number identifier_code
identifier_code ::=
    { ASCII character }

```

Syntax 18-8—Syntax of the output four state VCD file

The VCD file starts with header information giving the date, the version number of the simulator used for the simulation, and the timescale used. Next, the file contains definitions of the scope and type of variables being dumped, followed by the actual value changes at each simulation time increment. Only the variables which change value during a time increment are listed.

The simulation time recorded in VCD file is the absolute value of the simulation time for the changes in variable values which follow.

Value changes for real variables are specified by real numbers. Value changes for all other variables are specified in binary format by 0, 1, x, or z values. Strength information and memories are not dumped.

A real number is dumped using a `% .16g printf()` format. This preserves the precision of that number by outputting all 53 bits in the mantissa of a 64-bit *IEEE Std 754-1985* [B1] double-precision number. Application programs can read a real number using a `%g` format to `scanf()`.

The value change dumper generates character identifier codes to represent variables. The identifier code is a code composed of the printable characters which are in the ASCII character set from ! to ~ (decimal 33 to 126).

NOTES

1—The VCD format does not support a mechanism to dump *part* of a vector. For example, bits 8 to 15 (8:15) of a 16-bit vector cannot be dumped in VCD file; instead, the entire vector (0:15) has to be dumped. In addition, expressions, such as $a + b$, cannot be dumped in the VCD file.

2—Data in the VCD file is case sensitive.

18.2.2 Formats of variable values

Variables can be either scalars or vectors. Each type is dumped in its own format. Dumps of value changes to scalar variables shall not have any white space between the value and the identifier code.

Dumps of value changes to vectors shall not have any white space between the base letter and the value digits, but they shall have one white space between the value digits and the identifier code.

The output format for each value is right-justified. Vector values appear in the shortest form possible: redundant bit values which result from left-extending values to fill a particular vector size are eliminated.

The rules for left-extending vector values are given in Table 18-1.

Table 18-1—Rules for left-extending vector values

When the value is	VCD left-extends with
1	0
0	0
Z	Z
X	X

Table 18-2 shows how the VCD can shorten values.

Table 18-2—How the VCD can shorten values

The binary value	Extends to fill a 4-bit reg as	Appears in the VCD file as
10	0010	b10
X10	XX10	bX10
ZX0	ZZX0	bZX0
0X10	0X10	b0X10

Events are dumped in the same format as scalars; for example, $1 * \%$. For events, however, the value (1 in this example) is irrelevant. Only the identifier code ($* \%$ in this example) is significant. It appears in the VCD file as a marker to indicate the event was triggered during the time step.

Examples:

1*@ No space between the value 1 and the identifier code *@

b1100x01z (k No space between the b and 1100x01z,
but a space between b1100x01z and (k

18.2.3 Description of keyword commands

The general information in the VCD file is presented as a series of sections surrounded by keywords. Keyword commands provide a means of inserting information in the VCD file. Keyword commands can be inserted either by the dumper or manually.

This sub clause deals with the keyword commands given in Table 18-3.

Table 18-3—Keyword commands

Declaration keywords		Simulation keywords
\$comment	\$timescale	\$dumpall
\$date	\$upscope	\$dumpoff
\$enddefinitions	\$var	\$dumpon
\$scope	\$version	\$dumpvars

18.2.3.1 \$comment

The **\$comment** section provides a means of inserting a comment in the VCD file. The syntax for the section is given in Syntax 18-9.

vcd_declaration_comment ::= (Not in the Annex A BNF)
\$comment comment_text **\$end**

Syntax 18-9—Syntax for \$comment section

Examples:

\$comment This is a single-line comment **\$end**
\$comment This is a
multiple-line comment
\$end

18.2.3.2 \$date

The **\$date** section indicates the date on which the VCD file was generated. The syntax for the section is given in Syntax 18-10.

```
vcd_declaration_date ::= (Not in the Annex A BNF)
    $date date_text $end
```

Syntax 18-10—Syntax for \$date section

Example:

```
$date
    June 25, 1989 09:24:35
$end
```

18.2.3.3 \$enddefinitions

The **\$enddefinitions** section marks the end of the header information and definitions. The syntax for the section is given in Syntax 18-11.

```
vcd_declaration_enddefinitions ::= (Not in the Annex A BNF)
    $enddefinitions $end
```

Syntax 18-11—Syntax for \$enddefinitions section

18.2.3.4 \$scope

The **\$scope** section defines the scope of the variables being dumped. The syntax for the section is given in Syntax 18-12.

```
vcd_declaration_scope ::= (Not in the Annex A BNF)
    $scope scope_type scope_identifier $end
scope_type ::=
    begin
    | fork
    | function
    | module
    | task
```

Syntax 18-12—Syntax for \$scope section

The scope type indicates one of the following scopes:

<i>module</i>	Top-level module and module instances
<i>task</i>	Tasks
<i>function</i>	Functions
<i>begin</i>	Named sequential blocks
<i>fork</i>	Named parallel blocks

Example:

```
$scope
    module top
$end
```

18.2.3.5 \$timescale

The **\$timescale** keyword specifies what timescale was used for the simulation. The syntax for the keyword is given in Syntax 18-13.

```
vcd_declaration_timescale ::= (Not in the Annex A BNF)
    $timescale time_number time_unit $end
time_number ::=
    1 | 10 | 100
time_unit ::=
    s | ms | us | ns | ps | fs
```

Syntax 18-13—Syntax for \$timescale

Example:

```
$timescale 10 ns $end
```

18.2.3.6 \$upscope

The **\$upscope** section indicates a change of scope to the next higher level in the design hierarchy. The syntax for the section is given in Syntax 18-14.

```
vcd_declaration_upscope ::= (Not in the Annex A BNF)
    $upscope $end
```

Syntax 18-14—Syntax for \$upscope section

18.2.3.7 \$version

The **\$version** section indicates which version of the VCD writer was used to produce the VCD file and the **\$dumpfile** system task used to create the file. If a variable or an expression was used to specify the *filename* within **\$dumpfile**, the unevaluated variable or expression literal shall appear in the **\$version** string. The syntax for the **\$version** section is given in Syntax 18-15.

```
vcd_declaration_version ::= (Not in the Annex A BNF)
    $version version_text system_task $end
```

Syntax 18-15—Syntax for \$version section

Example:

```
$version
    VERILOG-SIMULATOR 1.0a
    $dumpfile( "dump1.dump" )
$end
```

18.2.3.8 \$var

The **\$var** section prints the names and identifier codes of the variables being dumped. The syntax for the section is given in Syntax 18-16.

```
vcd_declaration_vars ::= (Not in the Annex A BNF)
    $var var_type size identifier_code reference $end
var_type ::=
    event | integer | parameter | real | reg | supply0 | supply1 | time
    | tri | triand | trior | trireg | tri0 | tri1 | wand | wire | wor
size ::=
    decimal_number
reference ::=
    identifier [ bit_select_index ]
    | identifier [ msb_index : lsb_index ]
index ::=
    decimal_number
```

Syntax 18-16—Syntax for \$var section

Size specifies how many bits are in the variable.

The identifier code specifies the name of the variable using printable ASCII characters, as previously described.

- The msb index indicates the most significant index; the lsb index indicates the least significant index.
- More than one reference name can be mapped to the same identifier code. For example, net10 and net15 can be interconnected in the circuit and therefore have the same identifier code.
- The individual bits of vector nets can be dumped individually.
- The identifier is the name of the variable being dumped in the model.

Example:

```
$var
    integer 32 (2 index
$end
```

18.2.3.9 \$dumpall

The **\$dumpall** keyword specifies current values of all variables dumped. The syntax for the keyword is given in Syntax 18-17.

```
vcd_simulation_dumpall ::= (Not in the Annex A BNF)
$dumpall { value_changes } $end
```

Syntax 18-17—Syntax for \$dumpall keyword

Example:

```
$dumpall 1* @ x*# 0*$ bx (k $end
```

18.2.3.10 \$dumpoff

The **\$dumpoff** keyword indicates all variables dumped with X values. The syntax for the keyword is given in Syntax 18-18.

```
vcd_simulation_dumpoff ::= (Not in the Annex A BNF)
$dumpoff { value_changes } $end
```

Syntax 18-18—Syntax for \$dumpoff keyword

Example:

```
$dumpoff x* @ x*# x*$ bx (k $end
```

18.2.3.11 \$dumpon

The **\$dumpon** keyword indicates resumption of dumping and lists current values of all variables dumped. The syntax for the keyword is given in Syntax 18-19.

```
vcd_simulation_dumpon ::= (Not in the Annex A BNF)
$dumpon { value_changes } $end
```

Syntax 18-19—Syntax for \$dumpon keyword

Example:

```
$dumpon x* @ 0*# x*$ b1 (k $end
```

18.2.3.12 \$dumpvars

The section beginning with **\$dumpvars** keyword lists initial values of all variables dumped. The syntax for the keyword is given in Syntax 18-20.

<p>vcd_simulation_dumpvars ::= <i>(Not in the Annex A BNF)</i> \$dumpvars { value_changes } \$end</p>

Syntax 18-20—Syntax for \$dumpvars keyword

Example:

\$dumpvars x*@ z*\$ b0 (k **\$end**

18.2.4 Four state VCD file format example

The following example illustrates the format of the four state VCD file.

```

$date June 26, 1989 10:05:41
$end
$version VERILOG-SIMULATOR 1.0a
$end
$timescale 1 ns
$end
$scope module top $end
$scope module m1 $end
$var trireg 1 *@ net1 $end
$var trireg 1 *# net2 $end
$var trireg 1 *$ net3 $end
$upscope $end
$scope task t1 $end
$var reg 32 (k accumulator[31:0] $end
$var integer 32 {2 index $end
$upscope $end
$upscope $end
$enddefinitions $end
$comment
    Note: $dumpvars was executed at time '#500'.
          All initial values are dumped at this time.
$end

```

#500	(Continued from left column)
\$dumpvars	
x*@	bz (k
x*#	b1111000101z01x {2
x*\$	\$end
bx (k	#540
bx {2	1*\$
\$end	#1000
#505	\$dumpoff
0*@	x*@
1*#	x*#
1*\$	x*\$
b10zx1110x11100 (k	bx (k
b1111000101z01x {2	bx {2
#510	\$end
0*\$	#2000
#520	\$dumpon
1*\$	z*@
#530	1*#
0*\$	0*\$
bz (k	b0 (k
#535	bx {2
\$dumpall 0*@ 1*# 0*\$	\$end
	#2010
	1*\$
(Continued in right column)	

18.3 Creating the extended value change dump file

The steps involved in creating the extended VCD file are listed below and illustrated in Figure 18-2.

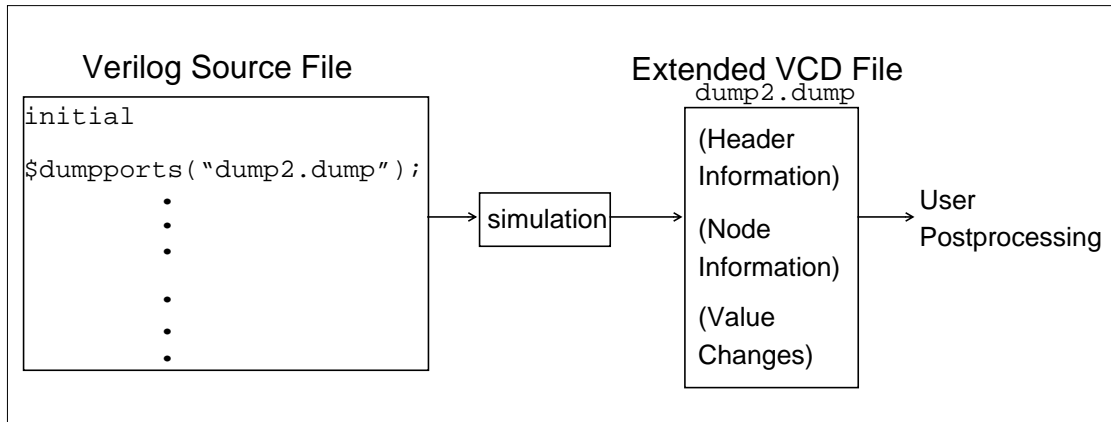


Figure 18-2—Creating the extended VCD file

- a) Insert the extended VCD system tasks in the Verilog source file to define the dump file name and to specify the variables to be dumped.
- b) Run the simulation.

The four state VCD file rules and syntax apply to the extended VCD file unless otherwise stated in this section.

18.3.1 Specifying the dumpfile name and the ports to be dumped (\$dumpports)

The **\$dumpports** task shall be used to specify the name of the VCD file and the ports to be dumped. The syntax for the task is given in Syntax 18-21.

```

dumpports_task ::= (Not in the Annex A BNF)
$dumpports ( scope_list , file_pathname ) ;
scope_list ::=
    module_identifier { , module_identifier }
file_pathname ::=
    literal_string
    | variable
    | expression
  
```

Syntax 18-21—Syntax for \$dumpports task

Where the arguments are optional and are defined as:

scope_list one or more module identifiers. Only modules are allowed (not variables). If more than one *module_identifier* is specified, they shall be separated by a comma. Pathnames to modules are allowed, using the period hierarchy separator. Literal strings are not allowed for the *module_identifier*.

If no *scope_list* value is provided, the scope shall be the module from which **\$dumpports** is called.

`file_pathname` can be a double quoted pathname (literal string), a reg type variable, or an expression which denotes the file which shall contain the port VCD information. If no `file_pathname` is provided, the file shall be written to the current working directory with the name `dumpports.vcd`. If that file already exists, it shall be silently overwritten. All file writing checks shall be made by the simulator (write rights, correct pathname, etc.) and appropriate errors or warnings issued.

The following rules apply to the use of the `$dumpports` system task:

- All the ports in the model from the point of the **\$dumpports** call are considered primary I/O pins and shall be included in the VCD file. However, any ports which exist in instantiations below `scope_list` are not dumped.
- If no arguments are specified for the task, **\$dumpports**; and `$dumpports()` are allowed. In both of these cases, the default values for the arguments shall be used.
- If the first argument is null, a comma shall be used before specifying the second argument in the argument list.
- Each scope specified in the `scope_list` shall be unique. If multiple calls to **\$dumpports** are specified, the `scope_list` values in these calls shall also be unique.
- The **\$dumpports** task can be used in source code which also contains the **\$dumpvars** task.
- When **\$dumpports** executes, the associated value change dumping shall start at the end of the current simulation time unit.
- The **\$dumpports** task can be invoked multiple times throughout the model, but the execution of all **\$dumpports** tasks shall be at the same simulation time. Specifying the same `file_pathname` multiple times is not allowed.

18.3.2 Stopping and resuming the dump (\$dumpportsoff/\$dumpportson)

The **\$dumpportsoff** and **\$dumpportson** system tasks provide a means to control the simulation period for dumping port values. The syntax for these system tasks is given in Syntax 18-22.

```

dumpportsoff_task ::= (Not in the Annex A BNF)
    $dumpportsoff ( file_pathname ) ;
dumpportson_task ::=
    $dumpportson ( file_pathname ) ;
file_pathname ::=
    literal_string
    | variable
    | expression

```

Syntax 18-22—Syntax for `$dumpportsoff` and `$dumpportson` system tasks

The `file_pathname` argument can be a double quoted pathname (literal string), a reg type variable, or an expression which denotes the `file_pathname` specified in the **\$dumpports** system task.

\$dumpportsoff. When this task is executed, a checkpoint is made in the `file_pathname` where each specified port is dumped with an X value. Port values are no longer dumped from that simulation time forward. If `file_pathname` is not specified, all dumping to files opened by **\$dumpports** calls shall be suspended.

\$dumpportson. When this task is executed, all ports specified by the associated **\$dumpports** call shall have their values dumped. This system task is typically used to resume dumping after the execution of **\$dumpportsoff**. If `file_pathname` is not specified, dumping shall resume for all files specified by **\$dumpports** calls, if dumping to those files was stopped.

If **\$dumpportson** is executed while ports are already being dumped to `file_pathname`, the system task is ignored. If

\$dumpportsoff is executed while port dumping is already suspended for *file_pathname*, the system task is ignored.

18.3.3 Generating a checkpoint (\$dumpportsall)

The **\$dumpportsall** system task creates a checkpoint in the VCD file which shows the value of all selected ports at that time in the simulation, regardless of whether the port values have changed since the last timestep. The syntax for this system task is given in Syntax 18-23.

```
dumpportsall_task ::= (Not in the Annex A BNF)
$dumpportsall ( file_pathname ) ;
file_pathname ::=
    literal_string
    | variable
    | expression
```

Syntax 18-23—Syntax for \$dumpportsall system task

The *file_pathname* argument can be a double quoted pathname (literal string), a reg type variable, or an expression which denotes the *file_pathname* specified in the **\$dumpports** system task.

If the *file_pathname* is not specified, checkpointing occurs for all files opened by calls to **\$dumpports**.

18.3.4 Limiting the size of the dump file (\$dumpportslimit)

The **\$dumpportslimit** system task allows control of the VCD file size. The syntax for this system task is given in Syntax 18-24.

```
dumpportslimit_task ::= (Not in the Annex A BNF)
$dumpportslimit ( filesize , file_pathname ) ;
file_size ::=
    integer
file_pathname ::=
    literal_string
    | variable
    | expression
```

Syntax 18-24—Syntax for \$dumpportslimit system task

The *filesize* argument is required and it specifies the maximum size in bytes for the associated *file_pathname*. When this *filesize* is reached, the dumping stops and a comment is inserted into *file_pathname* indicating the size limit was attained.

The *file_pathname* argument can be a double quoted pathname (literal string), a reg type variable, or an expression which denotes the *file_pathname* specified in the **\$dumpports** system task.

If the *file_pathname* is not specified, the *filesize* limit applies to all files opened for dumping due to calls to **\$dumpports**.

18.3.5 Reading the dump file during simulation (\$dumpportsflush)

To facilitate performance, simulators often buffer VCD output and write to the file at intervals, instead of line by line. The **\$dumpportsflush** system task writes all port values to the associated file, clearing a simulator's VCD buffer.

The syntax for this system task is given in Syntax 18-25.

```

dumpportsflush_task ::= (Not in the Annex A BNF)
$dumpportsflush ( file_pathname ) ;
file_pathname ::=
    literal_string
    | variable
    | expression

```

Syntax 18-25—Syntax for \$dumpportsflush system task

The *file_pathname* argument can be a double quoted pathname (literal string), a reg type variable, or an expression which denotes the *file_pathname* specified in the **\$dumpports** system task.

If the *file_pathname* is not specified, the VCD buffers shall be flushed for all files opened by calls to **\$dumpports**.

18.3.6 Description of keyword commands

The general information in the extended VCD file is presented as a series of sections surrounded by keywords. Keyword commands provide a means of inserting information in the extended VCD file. Keyword commands can be inserted either by the dumper or manually. Extended VCD provides one additional keyword command to that of the four state VCD.

18.3.6.1 \$vcdclose

The **\$vcdclose** keyword indicates the final simulation time at the time the extended VCD file is closed. This allows accurate recording of the end simulation time, regardless of the state of signal changes, in order to assist parsers which require this information. The syntax for the keyword is given in Syntax 18-26.

```

vcdclose_task ::= (Not in the Annex A BNF)
$vcdclose final_simulation_time $end

```

Syntax 18-26—Syntax for \$vcdclose keyword

Example:

```
$vcdclose #13000 $end
```

18.3.7 General rules for extended VCD system tasks

For each extended VCD system task, the following rules apply:

- If a *file_pathname* is specified which does not match a *file_pathname* specified in a **\$dumpports** call, the control task shall be ignored.
- If no arguments are specified for the tasks which have only optional arguments, the system task name can be used with no arguments or the name followed by () can be specified. For example: **\$dumpportsflush**; or **\$dumpportsflush()**. In both of these cases, the default actions for the arguments shall be executed.

18.4 Format of the extended VCD file

The format of the extended VCD file is similar to that of the four state VCD file, as it is also structured in a free format. White space is used to separate commands and to make the file easily readable by a text editor.

18.4.1 Syntax of the extended VCD file

The syntax of the extended VCD file is given in Syntax 18-27. A four state VCD construct name which matches an extended VCD construct shall be considered equivalent, except if preceded by an *.

```

value_change_dump_definitions ::= {declaration_command} {simulation_command}
declaration_command ::= declaration_keyword [command_text] $end
simulation_command ::= (Not in the Annex A BNF)
    simulation_keyword { value_change } $end
    | $comment [comment_text] $end
    | simulation_time
    | value_change
* declaration_keyword ::=
    $comment | $date | $enddefinitions | $scope | $timescale | $supscope | $var
    | $vcdclose | $version
command_text ::=
    comment_text | close_text | date_section | scope_section | timescale_section
    | var_section | version_section
* simulation_keyword ::= $dumpports | $dumpportsoff | $dumpportson |
    $dumpportsall
simulation_time ::= #decimal_number
value_change ::= value identifier_code
value ::= pport_value 0_strength_component 1_strength_component
port_value ::= input_value | output_value | unknown_direction_value
input_value ::= D | U | N | Z | d | u
output_value ::= L | H | X | T | l | h
unknown_direction_value ::= 0 | 1 | ? | F | A | a | B | b | C | c | f
strength_component ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
* identifier_code ::= <{integer}
comment_text ::= {ASCII_character}
close_text ::= final_simulation_time
date_section ::= date_text
date_text ::= day month date time year
scope_section ::= scope_type scope_identifier
* scope_type ::= module
timescale_section ::= number time_unit
number ::= 1 | 10 | 100
time_unit ::= fs | ps | ns | us | ms | s
var_section ::= var_type size identifier_code reference
* var_type ::= port
* size ::= 1 | vector_index
vector_index ::= [ msb_index : lsb_index ]
index ::= decimal_number
* reference ::= port_identifier
identifier ::= {printable_ASCII_character}
version_section ::= version_text
* version_text ::= version_identifier {dumpports_command}
dumpports_command ::=
    $dumpports (scope_identifier , string_literal
    | variable
    | expression )

```

Syntax 18-27—Syntax of the output extended VCD file

The extended VCD file starts with header information giving the date, the version number of the simulator used for the simulation, and the timescale used. Next, the file contains definitions of the scope of the ports being dumped, followed by the actual value changes at each simulation time increment. Only the ports which change value during a time increment are listed.

The simulation time recorded in the extended VCD file is the absolute value of the simulation time for the changes in port values which follow.

Value changes for all ports are specified in binary format by 0, 1, x, or z values and include strength information.

A real number is dumped using a `% .16g printf ()` format. This preserves the precision of that number by outputting all 53 bits in the mantissa of a 64-bit *IEEE Std 754-1985* [B1] double-precision number. Application programs can read a real number using a `%g` format to `scanf ()`.

NOTES

1—The extended VCD format does not support a mechanism to dump *part* of a vector. For example, bits 8 to 15 (8:15) of a 16-bit vector cannot be dumped in VCD file; instead, the entire vector (0:15) has to be dumped. In addition, expressions, such as `a + b`, cannot be dumped in the VCD file.

2—Data in the extended VCD file is case sensitive.

18.4.2 Extended VCD node information

The node information section (also referred to as the variable definitions section) is affected by the **\$dumpports** task as Syntax 18-28 shows.

```
$var var_type size < identifier_code reference $end
var_type ::= (Not in the Annex A BNF)
port
size ::=
    1
    | vector_index
vector_index ::=
    [msb_index : lsb_index]
index ::=
    decimal_number
identifier_code ::=
    integer
reference ::=
    port_identifier
```

Syntax 18-28—Syntax of extended VCD node information

Where the constructs are defined as:

var_type the keyword **port**. No other keyword is allowed.

size a decimal number indicating the number of bits in the port. If the port is a single bit, the value shall be 1. If the port is a bus, the actual index is printed. The *msb* indicates the most significant index; *lsb* the least significant index.

identifier_code an integer preceded by < which starts at zero and ascends in one unit increments for each port, in the order found in the module declaration.

reference identifier indicating the port name.

Example:

```

module test_device(count_out, carry, data, reset)
output count_out, carry ;
input [0:3] data;
input reset;
. . .
initial
    begin
        $dumpports(testbench.DUT, "testoutput.vcd");
    . . .
    end

```

This example produces the following node information in the VCD file:

```

$scope module testbench.DUT $end
$var port      1 <0          count_out  $end
$var port      1 <1          carry      $end
$var port     [0:3] <2          data      $end
$var port      1 <3          reset      $end
$upscope $end

```

At least one space shall separate each syntactical element. However, the formatting of the information is the choice of the simulator vendor. All four state VCD syntax rules for the *vector_index* apply.

If the *vector_index* appears in the port declaration, this shall be the index dumped. If the *vector_index* is not in the port declaration, the *vector_index* in the net or reg declaration matching the port name shall be dumped. If no *vector_index* is found, the port is considered scalar (1 bit wide).

Concatenated ports shall appear in the extended VCD file as separate entries.

Example:

```

module addbit ({A, b}, ci, sum, co);
    input      A, b, ci;
    output     sum, co;
. . .

```

The VCD file output looks like:

```

$scope module addbit $end
$var port 1 <0 A $end
$var port 1 <1 b $end
$var port 1 <2 ci $end
$enddefinitions $end
. . .

```

18.4.3 Value changes

The value change section of the VCD file is also affected by **\$dumpports**, as Syntax 18-29 shows.

p port_value 0_strength_component 1_strength_component identifier_code

Syntax 18-29—Syntax of value change section

Where the constructs are defined as:

p	key character which indicates a port. There is no space between the p and the <i>port_value</i> .
<i>port_value</i>	state character (described below).
<i>0_strength_component</i>	one of the 8 Verilog strengths which indicates the <i>strength0</i> specification for the port.
<i>1_strength_component</i>	one of the 8 Verilog strengths which indicates the <i>strength1</i> specification for the port.

The Verilog strength values are (append keyword with 0 or 1 as appropriate for the strength component):

0	highz
1	small
2	medium
3	weak
4	large
5	pull
6	strong
7	supply

identifier_code the integer preceded by the < character as defined in the \$var construct for the port.

18.4.3.1 State characters

The following state information is listed in terms of input values from a test fixture, the output values of the device under test (DUT), and the states representing unknown direction:

INPUT (TESTFIXTURE)

D	low
U	high
N	unknown
Z	tri-state
d	low (two or more drivers active)
u	high (two or more drivers active)

OUTPUT (DUT)

L	low
H	high
X	unknown (don't care)
T	tri-state
l	low (two or more drivers active)
h	high (two or more drivers active)

UNKNOWN DIRECTION

0	low (both input and output are active with 0 value)
1	high (both input and output are active with 1 value)
?	unknown

F	tri-state (input and output unconnected)
A	unknown (input 0 and output 1)
a	unknown (input 0 and output X)
B	unknown (input 1 and output 0)
b	unknown (input 1 and output X)
C	unknown (input X and output 0)
c	unknown (input X and output 1)
f	unknown (input and output tri-stated)

18.4.3.2 Drivers

Where drivers are considered only in terms of primitives, continuous assignments, and procedural continuous assignments. Value 0/1 means both input and output are active with value 0/1. 0 and 1 are conflict states. The following rules apply to conflicts:

- If both input and output are driving the same value with the same range of strength, then this is a conflict. The resolved value is 0/1 and the strength is the stronger of the two.
- If the input is driving a strong strength (range) and the output is driving a weak strength (range), the resolved value is d/u and the strength is the strength of the input.
- If the input is driving a weak strength (range) and the output is driving a strong strength (range), then the resolved value is l/h and the strength is the strength of the output.

Where range is:

- Strength supply 7 to 5 (large) - strong strength
- Strength 4 to 1 - weak strength

18.4.4 Extended VCD file format example

The following example illustrates the format of the extended VCD file.

A module declaration:

```

module adder(data0, data1, data2, data3, carry, as, rdn, reset, test,
              write);
    inout data0, data1, data2, data3;
    output carry;
    input as, rdn, reset, test, write;
    . . .

```

And the resulting VCD fragment:

```

$scope module testbench.adder_instance $end
$var port      1 <0          data0 $end
$var port      1 <1          data1 $end
$var port      1 <2          data2 $end
$var port      1 <3          data3 $end
$var port      1 <4          carry $end
$var port      1 <5          as $end
$var port      1 <6          rdn $end
$var port      1 <7          reset $end
$var port      1 <8          test $end
$var port      1 <9          write $end
$upscope $end

```

\$enddefinitions \$end

#0

\$dumpports

pX 6 6 <0

pX 6 6 <1

pX 6 6 <2

pX 6 6 <3

pX 6 6 <4

pN 6 6 <5

pN 6 6 <6

pU 0 6 <7

pD 6 0 <8

pN 6 6 <9

\$end

#180

pH 0 6 <4

#200000

pD 6 0 <5

pU 0 6 <6

pD 6 0 <9

#200500

pf 0 0 <0

pf 0 0 <1

pf 0 0 <2

pf 0 0 <3

Section 19

Compiler directives

All Verilog compiler directives are preceded by the (`) character. This character is called accent grave. It is different from the character ('), which is the single quote character. The scope of compiler directives extends from the point where it is processed, across all files processed, to the point where another compiler directive supersedes it or the processing completes.

This section describes the following compiler directives:

<code>`celldefine</code>	[19.1]
<code>`default_nettype</code>	[19.2]
<code>`define</code>	[19.3]
<code>`else</code>	[19.4]
<code>`elsif</code>	[19.4]
<code>`endcelldefine</code>	[19.1]
<code>`endif</code>	[19.4]
<code>`ifdef</code>	[19.4]
<code>`ifndef</code>	[19.4]
<code>`include</code>	[19.5]
<code>`line</code>	[19.7]
<code>`nounconnected_drive</code>	[19.9]
<code>`resetall</code>	[19.6]
<code>`timescale</code>	[19.8]
<code>`unconnected_drive</code>	[19.9]
<code>`undef</code>	[19.3]

19.1 ``celldefine` and ``endcelldefine`

The directives ``celldefine` and ``endcelldefine` tag modules as cell modules. Cells are used by certain PLI routines for applications, such as delay calculations. It is advisable to pair each ``celldefine` with an ``endcelldefine`. 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 the module definition.

19.2 ``default_nettype`

The directive ``default_nettype` controls the net type created for implicit net declarations (see 3.5). It can be used only outside of module definitions. It affects all modules that follow the directive, even across source file boundaries. 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 19-1 contains the syntax of the directive.

```
default_nettype_compiler_directive ::= (Not in the Annex A BNF)
`default_nettype net_type
net_type ::= wire | tri | tri0 | wand | triand | wor | trior | trireg | none
```

Syntax 19-1—Syntax for default nettype compiler directive

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 must be explicitly declared. If a net is not explicitly declared, an error is generated.

19.3 ``define` and ``undef`

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.

19.3.1 ``define`

The directive ``define` creates a macro for text substitution. This directive can be used both inside and outside module definitions. 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 ``macro_name`. All compiler directives shall be considered predefined macro names; 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 19-2.

```
text_macro_definition ::= (Not in the Annex A BNF)
                        `define text_macro_name macro_text
text_macro_name ::=
    text_macro_identifier [ ( list_of_formal_arguments ) ]
list_of_formal_arguments ::=
    formal_argument_identifier { , formal_argument_identifier }
text_macro_identifier ::= (From Annex A - A.9.3)
    simple_identifier
```

Syntax 19-2—Syntax for text macro definition

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 shall be preceded by a backslash (`\`). The first newline not preceded by a backslash shall end the macro text. The newline preceded by a backslash shall be replaced in the expanded macro with a newline (but without the preceding backslash character).

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 a one-line comment (that is, a comment specified with the characters `//`) is included in the text, then the comment shall not become part of the substituted text. 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 19-3.

```

text_macro_usage ::= (Not in the Annex A BNF)
`text_macro_identifier [ ( list_of_actual_arguments ) ]
list_of_actual_arguments ::=
    actual_argument { , actual_argument }
actual_argument ::=
    expression

```

Syntax 19-3—Syntax for text macro usage

For a macro without arguments, the text shall be substituted “as is” for every occurrence of ``text_macro_name`. 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.

Once a text macro name has been defined, it can be used anywhere in a source description; that is, there are no scope restrictions. Text macros can be defined and used interactively. The text macro name shall be a simple identifier.

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

- Comments
- Numbers
- Strings
- Identifiers
- Keywords
- Operators

Examples:

```

`define wordsize 8
reg [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 following is illegal syntax because it is split across a string:

```

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

```

NOTES

1—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.

2—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 Verilog HDL source description can be the same as compiler directive keywords (although this is not recommended). The following problems should be considered:

- a) Text macro names may not be the same as compiler directive keywords.
- b) Text macro names can re-use 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.

19.3.2 `undef

The directive **`undef** shall undefine a previously defined text macro. An attempt to undefine a text macro that was not previously defined using a **`define** compiler directive can result in a warning. The syntax for **`undef** compiler directive is given in Syntax 19-4.

```
undefine_compiler_directive ::= (Not in the Annex A BNF)
`undef text_macro_identifier
```

Syntax 19-4—Syntax for undef compiler directive

An undefined text macro has no value.

19.4 `ifdef, `else, `elsif, `endif, `ifndef

These conditional compilation compiler directives are used to include optionally lines of a Verilog HDL source description during compilation. The **`ifdef** compiler directive checks for the definition of a `text_macro_name`. If the `text_macro_name` is defined, then the lines following the **`ifdef** directive are included. If the `text_macro_name` is not defined and an **`else** directive exists, then this source is compiled. The **`ifndef** compiler directive checks for the definition of a `text_macro_name`. If the `text_macro_name` is not defined, then the lines following the **`ifndef** directive are included. If the `text_macro_name` is defined and an **`else** directive exists, then this source is compiled.

If the **`elsif** directive exists (instead of the **`else**) the compiler checks for the definition of the `text_macro_name`. If the name exists the lines following the **`elsif** directive are included. The **`elsif** directive is equivalent to the compiler directive sequence **`else `ifdef ... `endif**. This directive does not need a corresponding **`endif** directive. This directive must be preceded by an **`ifdef** or **`ifndef** directive.

These directives may appear anywhere in the source description.

Situations where the **`ifdef**, **`else**, **`elsif**, **`endif**, and **`ifndef** compiler directives may be useful include:

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

The **`ifdef**, **`else**, **`elsif**, **`endif**, and **`ifndef** compiler directives have the syntax shown in Syntax 19-5.


```

conditional_compilation_directive ::= (Not in the Annex A BNF)
    ifdef_directive
    | ifndef_directive
ifdef_directive ::=
    `ifdef text_macro_identifier
    ifdef_group_of_lines
    { `elsif text_macro_identifier elsif_group_of_lines }
    [ `else else_group_of_lines ]
    `endif
ifndef_directive ::=
    `ifndef text_macro_identifier
    ifndef_group_of_lines
    { `elsif text_macro_identifier elsif_group_of_lines }
    [ `else else_group_of_lines ]
    `endif

```

Syntax 19-5—Syntax for conditional compilation directives

The `text_macro_identifier` is a Verilog HDL *simple_identifier*. The `ifdef_group_of_lines`, `ifndef_group_of_lines`, `elsif_group_of_lines` and the `else_group_of_lines` are parts of a Verilog HDL source description. The ``else` and ``elsif` compiler directives and all of the groups of lines are optional.

The ``ifdef`, ``else`, ``elsif`, and ``endif` compiler directives work together in the following manner:

- When an ``ifdef` is encountered, the `ifdef` text macro identifier is tested to see if it is defined as a text macro name using ``define` within the Verilog HDL source description.
- If the `ifdef` text macro identifier is defined, the `ifdef` group of lines is compiled as part of the description and if there are ``else` or ``elsif` compiler directives, these compiler directives and corresponding groups of lines are ignored.
- If the `ifdef` text macro identifier has not been defined, the `ifdef` group of lines is ignored.
- If there is an ``elsif` compiler directive, the `elsif` text macro identifier is tested to see if it is defined as a text macro name using ``define` within the Verilog HDL source description.
- If the `elsif` text macro identifier is defined, the `elsif` group of lines is compiled as part of the description and if there are other ``elsif` or ``else` compiler directives, the other ``elsif` or ``else` directives and corresponding groups of lines are ignored.
- If the first `elsif` text macro identifier has not been defined, the first `elsif` group of lines is ignored.
- If there are multiple ``elsif` compiler directives, they are evaluated like the first ``elsif` compiler directive in the order they are written in the Verilog HDL source description.
- If there is an ``else` compiler directive, the `else` group of lines is compiled 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`, ``ifndef`, and ``elsif`.

The ``ifndef`, ``else`, ``elsif`, and ``endif` compiler directives work together in the following manner:

- When an ``ifndef` is encountered, the `ifndef` text macro identifier is tested to see if it is defined as a text macro name using ``define` within the Verilog HDL source description.
- If the `ifndef` text macro identifier is not defined, the `ifndef` group of lines is compiled as part of the description and if there are ``else` or ``elsif` compiler directives, these compiler directives and corresponding groups of lines are ignored.
- If the `ifndef` text macro identifier is defined, the `ifndef` group of lines is ignored.
- If there is an ``elsif` compiler directive, the `elsif` text macro identifier is tested to see if it is defined as a text macro name using ``define` within the Verilog HDL source description.

- If the `elsif` text macro identifier is defined, the `elsif` group of lines is compiled as part of the description and if there are other ``elsif` or ``else` compiler directives, the other ``elsif` or ``else` directives and corresponding groups of lines are ignored.
- If the first `elsif` text macro identifier has not been defined, the first `elsif` group of lines is ignored.
- If there are multiple ``elsif` compiler directives, they are evaluated like the first ``elsif` compiler directive in the order they are written in the Verilog HDL source description.
- If there is an ``else` compiler directive, the `else` group of lines is compiled 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`, ``ifndef`, and ``elseif`.

Nesting of ``ifdef`, ``ifndef`, ``else`, ``elsif`, and ``endif` compiler directives shall be permitted.

NOTE—Any group of lines that the compiler ignores still has to follow the Verilog HDL lexical conventions for white space, comments, numbers, strings, identifiers, keywords, and operators.

Examples:

Example 1—The example below 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("nest_two is defined");
    `else
        initial $display("nest_two 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

```

19.5 ``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 **``include`** compiler directive can be used to include global or commonly used definitions and tasks without encapsulating repeated code within module boundaries.

Advantages of using the **``include`** compiler directive include the following:

- Providing an integral part of configuration management
- Improving the organization of Verilog HDL source descriptions
- Facilitating the maintenance of Verilog HDL source descriptions

The syntax for the **``include`** compiler directive is given in Syntax 19-6.

<p>include_compiler_directive ::= (Not in the Annex A BNF) <code>`include</code> "filename"</p>
--

Syntax 19-6—Syntax for include compiler directive

The compiler directive **``include`** can be specified anywhere within the Verilog HDL description. 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.

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

A file included in the source using the **``include`** compiler directive may contain other **``include`** compiler directives. The number of nesting levels for included files shall be finite.

Examples:

Examples of legal comments for the **``include`** compiler directive are as follows:

```
`include "parts/count.v"  
`include "fileB"  
`include "fileB" // including fileB
```

NOTE—Implementations may limit the maximum number of levels to which include files can be nested, but the limit shall be at least 15.

19.6 ``resetall`

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

The recommended usage is to place **``resetall`** at the beginning of each source text file, followed immediately by the directives desired in the file.

19.7 ``line`

The compiler is expected to maintain the current line and the filename of the file being compiled. The line number

(**line**) compiler directive is used to reset the current line number and filename of the current file to the line number and filename presented. This can be used to reflect the location in an original file; if the actual source file has been modified by addition or reduction of lines. After specifying the new line number or file name, the compiler can correctly refer to the original source file location. For example error messages, source code debugging, etc. can direct the user to the actual original line.

The syntax for the **line** compiler directive is given in Syntax 19-7.

line_compiler_directive ::= (Not in the Annex A BNF)
line number "filename" level

Syntax 19-7—Syntax for line compiler directive

The directive can be specified anywhere within the Verilog HDL source description. The number parameter is the new line number of the next line. The filename parameter is the new name of the file. The filename can be a full or relative path name. The level parameter indicates whether an include file has been entered (value is 1), an include file is exited (value is 2), or neither has been done (value is 0).

The results of this directive are not affected by the compiler directive **resetall**. 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 filename shall be updated to the new current file being processed. When beginning to read include files, the current line and filename shall be stored for restoration at the termination of the include file. The updated line number and filename information shall be available for PLI access. The mechanism of library searching is not affected by the effects of the **line** compiler directive.

19.8 **timescale**

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

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

- The **timescale** compiler directive to specify the unit of measurement for time and precision of time in the modules in the design
- The **\$prinntimescale** system task to display the time unit and precision of a module
- 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 unit of measurement for time and delay values and the degree of accuracy for delays in all modules that follow this directive until another **timescale** compiler directive is read.

The syntax for the **timescale** directive is given in Syntax 19-8.

timescale_compiler_directive ::= (Not in the Annex A BNF)
timescale time_unit / time_precision

Syntax 19-8—Syntax for timescale compiler directive

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 values used are accurate to within the unit of timespecified here, even if there is a smaller `time_precision` argument elsewhere in the design. The smallest `time_precision` argument of all the **``timescale`** compiler directives in the design determines the precision of the time unit of the 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**.

The units of measurement specified by these character strings are given in Table 19-1.

Table 19-1—Arguments of `time_precision`

Character string	Unit of measurement
s	seconds
ms	milliseconds
us	microseconds
ns	nanoseconds
ps	picoseconds
fs	femtoseconds

Examples:

The following example shows how this directive is used:

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

Here, all time values in the modules 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 modules 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;
  reg 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 and, therefore, the value stored in parameter `d` is scaled to a delay of 16 ns. This means that the value 0 is assigned to `reg 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, so the delay of parameter `d` is scaled from 1.6 to 16.
- c) The assignment of 0 to `reg 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.

19.9 **`unconnected_drive** and **`nounconnected_drive**

All unconnected input ports of a module 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. These directives shall be specified in pairs, and outside of the module declarations.

