

# 2

*Figure 2-0*  
*Example 2-0*  
*Syntax 2-0*  
*Table 2-0*

## Lexical Conventions

Verilog language source text files are a stream of lexical tokens. A token consists of one or more characters, and each single character is in exactly one token. The layout of tokens in a source file is free format—that is, spaces and newlines are not syntactically significant. However, spaces and newlines are very important for giving a visible structure and format to source descriptions. A good style of format, and consistency in that style, are an essential part of program readability.

The types of lexical tokens in the language are:

- operator
- white space
- comment
- number
- string
- identifier
- keyword

The rest of this chapter defines these tokens.

This manual uses a syntax formalism based on the Backus-Naur Form (BNF) to define the Verilog language syntax. Appendix A contains the complete set of syntax definitions in this format, plus a description of the BNF conventions used in the syntax definitions.

### 2.1 Operators

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

Unary operators appear to the left of their operand. Binary operators appear between their operands. A ternary operator has two operator characters that separate three operands. The Verilog language has one ternary operator the—conditional operator. See Section 4.1.12 for an explanation of the conditional operator.

## 2.2 White Space and Comments

White space can contain the characters for blanks, tabs, newlines, and formfeeds. The Verilog language ignores these characters except when they serve to separate other tokens. However, blanks and tabs are significant in strings.

The Verilog language has two forms to introduce comments. A one-line comment starts with the two characters `//` and ends with a newline. A block comment starts with `/*` and ends with `*/`. Block comments cannot be nested, but a one-line comment can be nested within a block comment.

## 2.3 Numbers

Constant numbers can be specified in decimal, hexadecimal, octal, or binary format. The Verilog language defines two forms to express numbers. The first form is a simple decimal number specified as a sequence of the digits 0 to 9 which can optionally start with a plus or minus. The second takes the following form:

```
<size><base_format><number>
```

The `<size>` element contains decimal digits that specify the size of the constant in terms of its exact number of bits. For example, the `<size>` specification for two hexadecimal digits is `8`, because one hexadecimal digit requires four bits. The `<size>` specification is optional. The `<base_format>` contains a letter specifying the number's base, preceded by the single quote character (`'`). Legal base specifications are one of `d`, `h`, `o`, or `b`, for the bases decimal, hexadecimal, octal, and binary respectively. (Note that these base identifiers can be upper- or lowercase.)

The `<number>` element contains digits that are legal for the specified `<base_format>`. The `<number>` element must physically follow the `<base_format>`, but can be separated from it by spaces. No spaces can separate the single quote and the base specifier character.

Alphabetic letters used to express the `<base_format>` or the hexadecimal digits `a` to `f` can be in upper- or lowercase.

Example 2-1 shows *unsized* 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 2-1: Unsized constant numbers*

Example 2-2 shows *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 2-2: Sized constant numbers*

In the Verilog language a plus or minus preceding the size constant is a sign for the constant number—the size constant does not take a sign. A plus or minus between the <base\_format> and the <number> is illegal syntax. In Example 2-3, the first expression is a syntax error. The second expression legally defines an 8-bit number with a value of minus 6.

---

```
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)
```

---

*Example 2-3: A plus or minus between the <base\_format> and the <number> is illegal*

The number of bits that make up an un-sized number (which is a simple decimal number or a number without the <size> specification) is the host machine word size—for most machines this is 32 bits.

## Lexical Conventions

### Numbers

In the Verilog language, an `x` expresses the unknown value in hexadecimal, octal, and binary constants. A `z` expresses the high impedance value. See Section 3.1 for a discussion of the Verilog value set. An `x` sets four bits to unknown in the hexadecimal base, three bits in the octal base, and one bit in the binary base. Similarly, a `z` sets four, three, and one bit, respectively, to the high impedance value. If the most significant specified digit of a constant number is an `x` or a `z`, then Verilog-XL automatically extends the `x` or `z` to fill the higher order bits of the constant. This makes it easy to specify complete vectors of the unknown and high impedance values. Example 2-4 illustrates this value extension:

---

```
reg [11:0] a;
initial
begin
    a = 'h x;      // yields xxx
    a = 'h 3x;    // yields 03x
    a = 'h 0x;    // yields 00x
end
```

---

*Example 2-4: Automatic extension of x values*

The question mark (?) character is a Verilog HDL alternative for the `z` character. It sets four bits to the high impedance value in hexadecimal numbers, three in octal, and one in binary. Use the question mark to enhance readability in cases where the high impedance value is a don't-care condition. See the discussion of `casez` and `casex` in Section 8.4.1 and the discussion on personality files in Section 22.5.

The underline character is legal anywhere in a number except as the first character. Use this feature to break up long numbers for readability purposes. Example 2-5 illustrates this.

---

```
27_195_000
16'b0011_0101_0001_1111
32 'h 12ab_f001
```

---

*Example 2-5: Use of underline in constant numbers*

Underline characters are also legal in numbers in text files read by the `$readmemb` and `$readmemh` system tasks.

**Please note:** A sized negative number is not sign-extended when assigned to a register data type.

## 2.4 Strings

A string is a sequence of characters enclosed by double quotes and must all be contained on a single line. Verilog treats strings used as operands in expressions and assignments as a sequence of eight-bit ASCII values, with one eight-bit ASCII value representing one character.

Examples of strings:

```
"this is a string"
```

```
"print out a message\n"
```

```
"bell!\007"
```

### 2.4.1 String Variable Declaration

To declare a variable to store a string, declare a register large enough to hold the maximum number of characters the variable will hold. Note that no extra bits are required to hold a termination character; Verilog does not store a string termination character.

For example, to store the string "Hello world!" requires a register  $8 \times 12$ , or 96 bits wide, as shown in Example 2-6.

---

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

---

*Example 2-6: Storage needed for strings*

## 2.4.2 String Manipulation

Verilog permits strings to be manipulated using the standard Verilog HDL operators. Keep in mind that the value being manipulated by an operator is a sequence of 8-bit ASCII values, with no special termination character.

The code in Example 2-7 declares a string variable large enough to hold 14 characters and assigns a value to it. The code then manipulates this string value using the concatenation operator.

Note that when a variable is larger than required to hold a value being assigned, Verilog pads the contents on the left with zeros after the assignment. This is consistent with the padding that occurs during assignment of non-string values.

---

```
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
```

---

*Example 2-7: String manipulation*

The following strings display as a result of executing Verilog-XL on Example 2-7:

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

### 2.4.3 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.

EscapeString	Character Produced by Escape String
\n	new line character
\t	tab character
\\	\ character
\"	" character
\ddd	a character specified in 1-3 octal digits (0 <= d <= 7)
%%	% character

Table 2-1: Specifying special characters in strings

## 2.5 Identifiers, Keywords, and System Names

An identifier is used to give an object, such as a register or a module, a name so that it can be referenced from other places in a description. An identifier is any sequence of letters, digits, dollar signs (\$), and the underscore (\_) symbol.

The first character must NOT be a digit or \$; it can be a letter or an underscore.

Upper- and lowercase letters are considered to be different (unless the upper case option is used when compiling). Identifiers can be up to 1024 characters long.

## Lexical Conventions

### Identifiers, Keywords, and System Names

Examples of identifiers follow:

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

### 2.5.1 Escaped Identifiers

Escaped identifiers start with the backslash character (\) and 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). An escaped identifier ends with white space (blank, tab, newline). Neither the leading back-slash character nor the terminating white space is considered to be part of the identifier.

The primary application of escaped identifiers is for translators from other hardware description languages and CAE systems, where special characters may be allowed in identifiers. Escaped identifiers should not be used under normal circumstances.

Examples of escaped identifiers follow:

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

**Please note:** Remember to terminate escaped identifiers with white space, otherwise characters that should follow the identifier are considered as part of it.

### 2.5.2 Keywords

Keywords are predefined non-escaped 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 and therefore must be typed in lowercase in source files (unless the upper case option is used when compiling).

## 2.6 Text Substitutions

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. Text macros can also be defined and used in the interactive mode where they can be helpful for predefining those interactive commands that you use often.

The syntax for text macro definitions is as follows:

```
<text_macro_definition>  
 ::= `define <text_macro_name> <MACRO_TEXT>  
<text_macro_name>  
 ::= <IDENTIFIER>
```

*Syntax 2-1: Syntax for <text\_macro\_definition>*

<MACRO\_TEXT> is any arbitrary text specified on the same line as the <text\_macro\_name>. If a one-line comment (that is, a comment specified with the characters //) is included in the text, then the comment does not become part of the text substituted. The text for <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 as follows:

```
<text_macro_usage>  
 ::= `<text_macro_name>
```

*Syntax 2-2: Syntax for <text\_macro\_usage>*

## Lexical Conventions

### Text Substitutions

Once a text macro name has been defined (that is, assigned <MACRO\_TEXT>), it can be used anywhere in a source description or in an interactive command; that is, there are no scope restrictions. However, to use a text macro the compiler directive symbol ` (open quote, also known as “accent grave”) must precede the text macro name. Example 2-8 shows two definitions of macro text and a use of each of the defined macros.

---

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

`define typ_nand nand #5 //define a nand w/typical delay
`typ_nand g121 (q21, n10, n11);
```

---

*Example 2-8: Using macro text*

The text specified for <MACRO\_TEXT> must not be split across the following lexical tokens:

- comments
- numbers
- strings
- identifiers
- keywords
- double or triple character operators

For example, the following is illegal syntax in the Verilog language because it is split across a string:

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

Text macro names can re-use names being used as ordinary identifiers. For example, `signal_name` and ``signal_name` are different. 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.