3.2 KEYWORDS

The keywords define the language constructs. A keyword signifies an activity to be carried out, initiated, or terminated. As such, a programmer cannot use a keyword for any purpose other than that it is intended for. All keywords in

Verilog are in small letters and require to be used as such (since Verilog is a case-sensitive language). All keywords appear in the text in New Courier Bold-type letters.

Examples

`module` ← signifies the beginning of a module definition.
`endmodule` ← signifies the end of a module definition.
`begin` ← signifies the beginning of a block of statements.
`end` ← signifies the end of a block of statements.
`if` ← signifies a conditional activity to be checked
`while` ← signifies a conditional activity to be carried out.

A list of keywords in Verilog with the significance of each is given in Appendix A.

3.3 IDENTIFIERS

Any program requires blocks of statements, signals, *etc.*, to be identified with an attached nametag. Such nametags are identifiers. It is good practice for us to use identifiers, closely related to the significance of variable, signal, block, *etc.*, concerned. This eases understanding and debugging of any program.

e.g., clock, enable, gate_1, . . .

There are some restrictions in assigning identifier names. All characters of the alphabet or an underscore can be used as the first character. Subsequent characters can be of alphanumeric type, or the underscore (_), or the dollar ($) sign – for example

name, _name. Name, name1, name_$, . . . ← all these are allowed as identifiers
name aa ← not allowed as an identifier because of the blank ("name" and "aa" are interpreted as two different identifiers)
$name ← not allowed as an identifier because of the presence of "$" as the first character.
1_name ← not allowed as an identifier, since the numeral "1" is the first character
@name ← not allowed as an identifier because of the presence of the character "@".
A+b ← not allowed as an identifier because of the presence of the character "+".

An alternative format makes it is possible to use any of the printable ASCII characters in an identifier. Such identifiers are called "escaped identifiers"; they have to start with the backslash (\) character. The character set between the first backslash character and the first white space encountered is treated as an identifier. The backslash itself is not treated as a character of the identifier concerned.

Examples

```
\b=c
\control-signal
\&logic
\abc // Here the combination "abc" forms the identifier.
```

It is preferable to use the former type of identifiers and avoid the escaped identifiers; they may be reserved for use in files which are available as inputs to the design from other CAD tools.

3.6 NUMBERS

Frequently numbers need to be specified in a design description. Logic status of signal lines, buses, delay values, and numbers to be loaded in registers are examples. The numbers can be of integer type or real type.

3.6.1 Integer Numbers

Integers can be represented in two ways. In the first case it is a decimal number – signed or unsigned; an unsigned number is automatically taken as a positive number. Some examples of valid number representations of this category are given below:

2
25
253
–253

The following are invalid since nondecimal representations are not permissible.

2a
B8
–2a
–B8

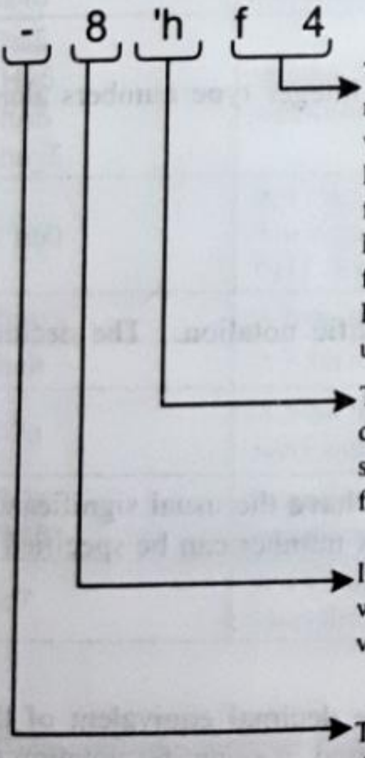
Normally the number is taken as 32 bits wide. Thus all the following numbers are assigned 32 bits of width:

2
25

253
-2
-25
-253

If a design description has a number specified in the form given here, the circuit synthesizer program will assign 32 bits of width to it and to all the related circuits. Hence all such number specifications – despite their simplicity – may be avoided in design descriptions. Number representation in this form may preferably be restricted to test benches.

The alternate form of number representation is more specific – though elaborate. The number can be specified in binary, octal, decimal, or hexadecimal form. The representation has three tokens with an optional sign preceding it. Figure 3.1 shows typical number representations with the significance of each field explained separately.



This field signifies the value of the number. For binary numbers the characters 0, 1, x, z can be used to form the value.
For octal numbers the numerals 0 to 7, x, z can be used to form the value.
For decimal numbers all the numerals, x, z can be used to form the value.
For hex numbers all the numerals, a, b, c, d, e, f, x, z can be used to form the numbers.

This combination - the single quote character followed by b, o, d or h - specifes the base of the number. The character signifies binary, octal, decimal or hexadecimal base. If this field is absent, the number is taken as a dcimal one.

If present, the decimal number in this field signifies the bit width of the number. If absent the width is assigned a default value by the compiler.

This field(optional) is for the sign bit. It is allowed only with the decimal numbers. If absent, the number is taken as positive. For a number with a negative sign the number is represented in 2's complement form

Figure 3.1 Representation of a number in Verilog: One can use capital letters instead of small letters in the last two fields.

Observations:

- The characters used to specify the base number, the sign or the magnitude can be in either case (Thus A, B, C, D, E, or F can be used in place of a, b, c, d, e, or f, respectively, to specify the concerned hex digit. **X** or **Z** can be used in place of **x** or **z** value, respectively).
- The single quote character in the base field has to be immediately followed by the character representing the base. Intervening white spaces are not allowed. However, such white spaces can precede the magnitude field.
- Negative numbers are represented in 2's complement form.
- The question mark character – "?" – can be used in place of **z**. The underscore character can be used anywhere after the first character. It adds to the readability. It is normally ignored.
- If the number size is smaller than the size specified, the size is made up by padding 0's to the left. However, if the leftmost bit is a **x** or **z**, the same is padded to the left.
- Left truncation and right extension can often be confusing. It is preferable to specify the numbers fully.

Table 3.1 shows the format of specifications of the integer type numbers along with illustrative examples.

3.6.2 Real Numbers

Real numbers can be specified in decimal or scientific notation. The decimal notation has the form

-a.b

where a, b, the negative sign, and the decimal point have the usual significance. The fields a and b must be present in the number. A number can be specified in scientific notation as

4.3e2

where 4.3 is the mantissa and 2 the exponent. The decimal equivalent of this number is 430. Other examples of numbers represented in scientific notation are –4.3e2, –4.3e–2, and 4.3e–2. The representations are common.

3.15 SYSTEM TASKS

During the simulation of any design, a number of activities are to be carried out to monitor and control simulation. A number of such tasks are provided / available in Verilog. Some other tasks serve other functions. However, a few of these are used commonly; these are described here. The "$" symbol identifies a system task. A task has the format

```
$<keyword>
```

3.15.1 $display

When the system encounters this task, the specified items are displayed in the formats specified and the system advances to a new line. The structure, format, and rules for these are the same as for the "printf" / "scanf" function in C. Refer to a standard text in "C" language for the text formatting codes in common usage [Gottfried].

Examples

`$display` ("The value of a is : a = , %d", a);
Execution of this line results in printing the value of a as a decimal number (specified by "%d"). The string present within the inverted commas specifies this. Thus if a has the value 3.5, we get the display

The value of a is : a = 3.5.

After printing the above line, the system advances to the next line.

`$display;` /* This is a display task without any arguments. It advances output to a new line. */

3.15.2 $monitor

The **`$monitor`** task monitors the variables specified whenever any one of those specified changes. During the running of the program the monitor task is invoked and the concerned quantities displayed whenever any one of these changes. Following this, the system advances to the next line. A monitor statement need appear only once in a simulation program. All the quantities specified in it are continuously monitored. In contrast, the **`$display`** command displays the quantities concerned only once – that is, when the specific line is encountered during execution. The format for the **`$monitor`** task is identical to that of the **`$display`** task.

Examples

`$monitor` ("The value of a is : a = , %d", a);

With the task, whenever the value of a changes during execution of a program, its new value is printed according to the format specified. Thus if the value of a changes to 2.4 at any time during execution of the program, we get the following display on the monitor.

```
The value of a is: a = 2.4.
```

3.15.3 Tasks for Control of Simulation

Two system tasks are available for control of simulation:

`$finish` task, when encountered, exits simulation. Control is reverted to the Operating System. Normally the simulation time and location are also printed out by default as part of the exit operation.

`$stop` task, suspends simulation; if necessary the simulation can be resumed by user intervention. Thus with the stop task, the simulator is in an interactive mode. In contrast with $finish, simulation has to be started afresh.