3.14 OPERATORS

Verilog has a number of operators akin to the C language. These are of three types:

1. Unary: the unary operator is associated with a single operand. The operator precedes the operand – for example, ~a.
2. Binary: the binary operator is associated with two operands. The operator appears between the two operands – for example, a&b.
3. Ternary: the ternary operator is associated with three operands. The two operators together constitute a ternary operation. The two operators separate the three operands – for example

```
a?b:c          // Here the operators "?" and ":" together define an operation.
```

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.

3.9 STRENGTHS

The logic levels are also associated with strengths. In many digital circuits, multiple assignments are often combined to reduce silicon area or to reduce pin-outs. To facilitate this, one can assign strengths to logic levels. Verilog has eight strength levels – four of these are of the driving type, three are of capacitive type and one of the hi-Z type. Details are given in Table 3.2 (see also Section 5.4).

When a signal line is driven simultaneously from two sources of different strength levels, the stronger of the two prevails. A few illustrative examples are considered here.

- If a signal line `a` is driven by two sources – `b` at 1 level with strength "`strong1`" and `c` at level 0 with strength "`pull0`" – a will take the value 1.

3.2 Details of strengths in Verilog

Strength name	Strength level (signifies inverse of source impedance)	Specification keyword	Abbreviation	Element modeled
Supply drive	7	`Supply1` `Supply0`	`Su1` `Su0`	Power supply connection
Strong drive	6	`Strong1` `Strong0`	`St1` `St0`	Default gate and assign output strength
Pull drive	5	`Pull1` `Pull0`	`Pu1` `Pu0`	Gate and assign output strength
Large capacitor	4	`Large1` `Large0`	`La1` `La0`	Size of trireg net capacitor
Weak drive	3	`Weak1` `Weak0`	`We1` `We0`	Gate and assign output strength
Medium capacitor	2	`Medium1` `Medium0`	`Me1` `Me0`	Size of trireg net capacitor
Small capacitor	1	`Small1` `Small0`	`Sm1` `Sm0`	Size of trireg net capacitor
High impedance	0	`Highz1` `Highz0`	`Hi1` `Hi0`	Tri-stated line

- If a signal line a is driven by two sources – b at 1 level with strength "**pull1**" and c at level 0 with strength "**strong0**," a will take the value 0.
- If a signal line a is driven by two sources – b at 1 level with strength "**strong1**" and c at level 0 with strength "**strong0**," a will take the value x (indeterminate).
- If a signal line a is driven by two sources – b at 1 level with strength "**weak1**" and c at level 0 with strength "**large0**," a will take the value 0. (Note that **large** signifies a capacitive drive on a tri-stated line whereas **weak** signifies a gate / assigned output drive with a high source impedance; despite this, due to the higher strength level, the **large** signal prevails.)

3.10 DATA TYPES

The data handled in Verilog fall into two categories:

(i) Net data type

(ii) Variable data type

The two types differ in the way they are used as well as with regard to their respective hardware structures. Data type of each variable or signal has to be declared prior to its use. The same is valid within the concerned block or module.

3.10.1 Nets

A net signifies a connection from one circuit unit to another. Such a net carries the value of the signal it is connected to and transmits to the circuit blocks connected to it. If the driving end of a net is left floating, the net goes to the high impedance state. A net can be specified in different ways.

`wire`: It represents a simple wire doing an interconnection. Only one output is connected to a wire and is driven by that.

`tri`: It represents a simple signal line as a wire. Unlike the wire, a tri can be driven by more than one signal outputs.

Functionally, **`wire`** and **`tri`** are identical. Distinct nomenclatures are provided for the convenience of assigning roles. Other types of nets are discussed in Chapter 5.

3.10.2 Variable Data Type

A variable is an abstraction for a storage device. It can be declared through the keyword **reg** and stores the value of a logic level: 0, 1, **x**, or **z**. A net or wire connected to a **reg** takes on the value stored in the **reg** and can be used as input to other circuit elements. But the output of a circuit cannot be connected to a **reg**. The value stored in a **reg** is changed through a fresh assignment in the program. **time**, **integer**, **real**, and **realtime** are the other variable types of data; these are dealt with later.

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.