# MODULE 2: VERILOG SYNTAX AND SEMANTICS

***I. LEARNING OUTCOMES:***

1. Explain different hierarchy and modelling structures of HDL's
2. Compare different modelling structures and its implementation accordingly
3. Differentiate and implement levels of hardware modelling structure of HDL
4. Differentiate the functionality and applications of different HDL technology Platform
   Explain different hierarchy and modelling structures of HDL's

## I. Engage Myself

Any Verilog program begins with a keyword – called a "module." A module is the name given to any system considering it as a black box with input and output terminals as shown in Figure 2.1. The terminals of the module are referred to as 'ports'. These are terminals through which signals are input to the module sometimes; at some other times signals are output from the module through these.

The ports attached to a module can be of three types:

**Input ports** which are one gets entry into the module; they signify the input signal terminals of the module.
**Output ports** through which one exits the module; these signify the output signal terminals of the module.
**inout ports**: These represent ports through which one gets entry into the module or exits the module.

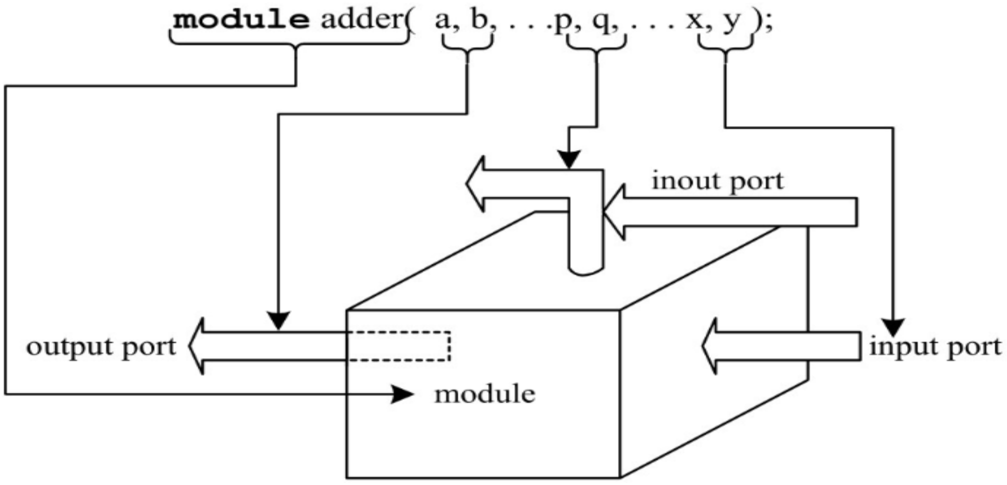


*Figure 2.1 Representation of Module Movement as Black Box*

Whether a module has any of the above ports and how many of each type are present depend solely on the functional nature of the module. Thus one module may not have any port at all, another may have only input ports, while a third may have only output ports, and so on.

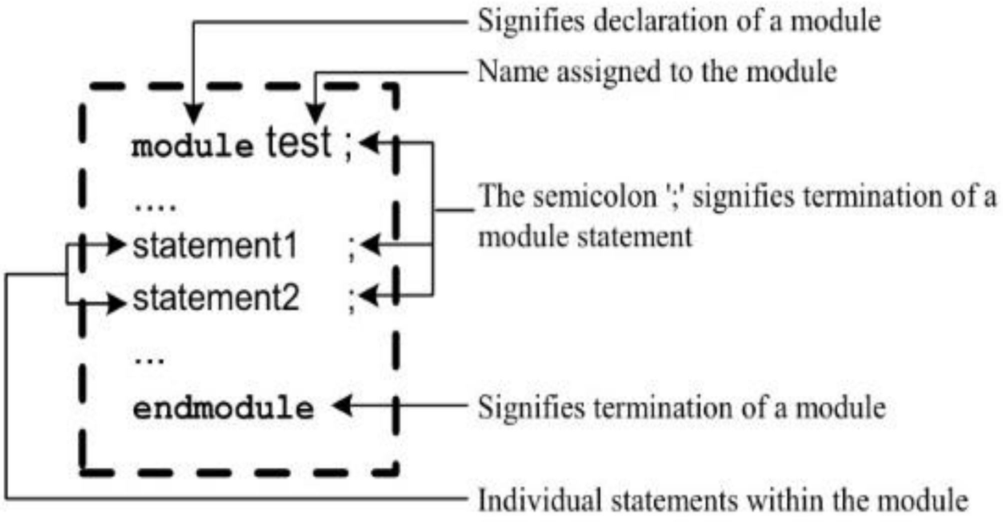

module test ;
....
statement1 ;
statement2 ;
...
endmodule
Signifies declaration of a module
Name assigned to the module
The semicolon ';' signifies termination of a module statement
Signifies termination of a module
Individual statements within the module

*Figure 2.2 Structure of Simulation Module*

Verilog takes the active statements appearing between the "module" statement and the "endmodule" statement and interprets all of them together as forming the body of the module. Whenever a module is invoked for testing or for incorporation into a bigger design module, the name of the module ("test" here) is used to identify it for the purpose.

## II. Explore it More

### Verilog Syntax

The language is made up of statements, groups of statements, and keywords to identify the different types of statement groups. These are properties of Verilog statements:

- Statements are composed of tokens.
- Statements can be continued across line boundaries, but individual tokens cannot.
- Statements within a given group are usually separated by ";", but statement groups are usually not separated by ";".
- In general, the ";" is a statement separator, not a terminator.

Verilog syntax appears to be something of a cross between C and Pascal.

### Tokens

Verilog is a token-based language. The source stream which a Verilog processor sees is a sequence of tokens. These are the types of Verilog tokens:

- **White Space**
  White space is any sequence of space, tab, newline, or form feed. White space separates tokens and may be arbitrarily long between tokens.

- **Comments**
  Verilog has two kinds of comments:
  ```
  // single line comment
  /* block comment */
  ```
- **Operator**

Verilog uses the same symbols as C does for operators, with a few extras. Operators are single, double, or triple character combinations. There are unary and binary operators, as well as one ternary operator (?:). The +, -, &, |, ^, and ~^ operators can be either unary or binary, depending on context.

- **Constants**
  Verilog uses three types of constants: Integer Constants Real Constants and String

- **Identifiers**
  Identifiers name objects. Objects which can be named are modules, instances, nets, registers, parameters, tasks, functions, blocks, and source macro. Identifiers may be made up of any number of letters, digits, and {_$}:

  simple_identifier ::= [a-zA-Z_]{[a-zA-Z0-9_$]}

  That is, the first character must be alphabetic or "_", and the rest may be alphabetic, numeric, "_", or "$". Identifiers may be as long as 1 million characters and are case-sensitive.

- **Keywords**
  Verilog contains keywords, which are predefined, non-escaped identifiers. An escaped identifier is not treated as a keyword. For example, \begin is not a keyword.

- **System Tasks and Functions**
  System tasks and functions are predefined tasks and functions which provide common operations. Identifiers which start with "$" are considered to be system tasks or system functions. These are used to provide common facilities to the model. This includes output capabilities and information facilities. Like user-defined tasks, system tasks must be invoked as a procedural statement.

**Test Benches**

Any digital circuit that has been designed and wired goes through a testing process before being declared as ready for use. Testing involves studying circuit behavior under simulated conditions for the following:

- Check and ensure that all functions are carried out as desired. It is the test for the static behavior of the circuit. A set of logic input values are applied at selected points and the logic values at another set of points observed.
- Check and ensure that all the functional sequences are carried out as desired. It is one of the tests for the dynamic behavior of the circuit. It may call for the generation of specific input sequences with respect to time, applying them to the circuit and observing selected outputs.
- Check for the timing behavior: One tests for the propagation and other types of delays here.

A variety of tests may have to be carried out. It may involve observation of variations in the signals at selected points, measuring the time delay between specified events, measuring pulse widths, and so on.

Here is an example of Test for an AND gate. The keyword initial is followed by a sequence of statements between the keywords begin and end. Usually the initial banner signifies a setting done on a once or a “once for all” basis. The “# 3” implies a time delay or wait time of 3 time steps in simulation.

Let us summarize:

- Generation of the test signals – under the “initial” banner
- Application of the test signal to the circuit under test – through instantiation
- Observing selected signal values – through the $monitor statement

```
Initial
Begin
a1 = 0;
 a2 = 0;
  #3 a1 = 1;
  #1 a1 = 0;
  #2 a2 = 1;
  #4 a1 = 1;
  #3 a2 = 0;
  #1 a2 = 1;
end
and g1(b, a1, a2);
initial $monitor ( $time, “a1 = %b, a2 = %b, b = %b”’ a1, a2, b);
#100 $finish;
```

*Figure 2.3 Sample test Bench Module*

Thus the sequence implies the following:

- At **0** simulation time the logic variables **a1** and **a2** are assigned the logic level **0** with a delay of **3 ns a1** is reassigned the logic value of **1**.
- With a further delay of **1 ns** – that is, at the 4th ns - **a1** is reverted to the logic level **0**.
- Similarly at the 6th, 10th, 13th and 14th ns values of simulation time, further changes are made to **a1** and **a2.**

Take note that every time value specified here is an increment in simulation time.

- The values of **a1** and **a2** are not changed beyond the 14th ns. The statement initial # 100 $finish; implies that the simulation is to be continued up to the 100th ns of simulation time and then stopped.
- The above constitutes the generation of the test sequence for testing. Such test signals are applied to the designed circuit through instantiation; the statement **and g1 (b, a1, a2)**; implies as much.
- The statement **initial $monitor ( $time, “a1 = %b, a2 = %b, b = %b”’ a1, a2, b)**; monitors **a1**, **a2**, and **a3** for changes; whenever any of them changes, all of them are sampled and the sampled values displayed.

## III. Explain Further

***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

*Table 1.4 Sample identifiers an description*

| **Sample Identifier** | **Description** |
|---|---|
| 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.

```
shiftreg_a
busa_index
error_condition
merge12
_bus23
n$657
```

Escaped Identifiers:

```
\wire*
\busa+index
\-clock
\***error-condition***
\net1/\net2
```

Hierarchical identifier

```
abc.def
top.foo.bar.xyz
system.board.chip.wire123
```

***Numbers***

Numbers can be integer type or real type. **Integer types** are represented in whole numbers, signed and unsigned while **real numbers** can be specified in a decimal or scientific notation.

***Real Numbers***

The decimal notation has the form **-a.b** where a, b, the negative sign, and the decimal point have the 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.

***Integers***

Integer 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:

These are examples of integer constants:

```
123
1'b1
8'h81
'o7773
12'bx
16'd5
3'b1xz
32'h8f_32_ab_f7
```

Sized vs. Unsized

A sized constant has the size specifier present, while an unsized constant does not. Unsized numbers have a default size of 32 bits.

Signed vs. Unsigned

If both the size and radix are omitted, the constant is a signed number, represented in 2's complement. Otherwise, the constant is an unsigned, positive number. This is only visible if the constant is preceded by a unary minus sign.

Radix Specifiers

The radix specifier indicates that the digits following are decimal ('d), hexadecimal ('h), octal ('o), or binary ('b). Each digit represents the appropriate number of bits for the radix. You can use capital letters too to represent these radix specifier.

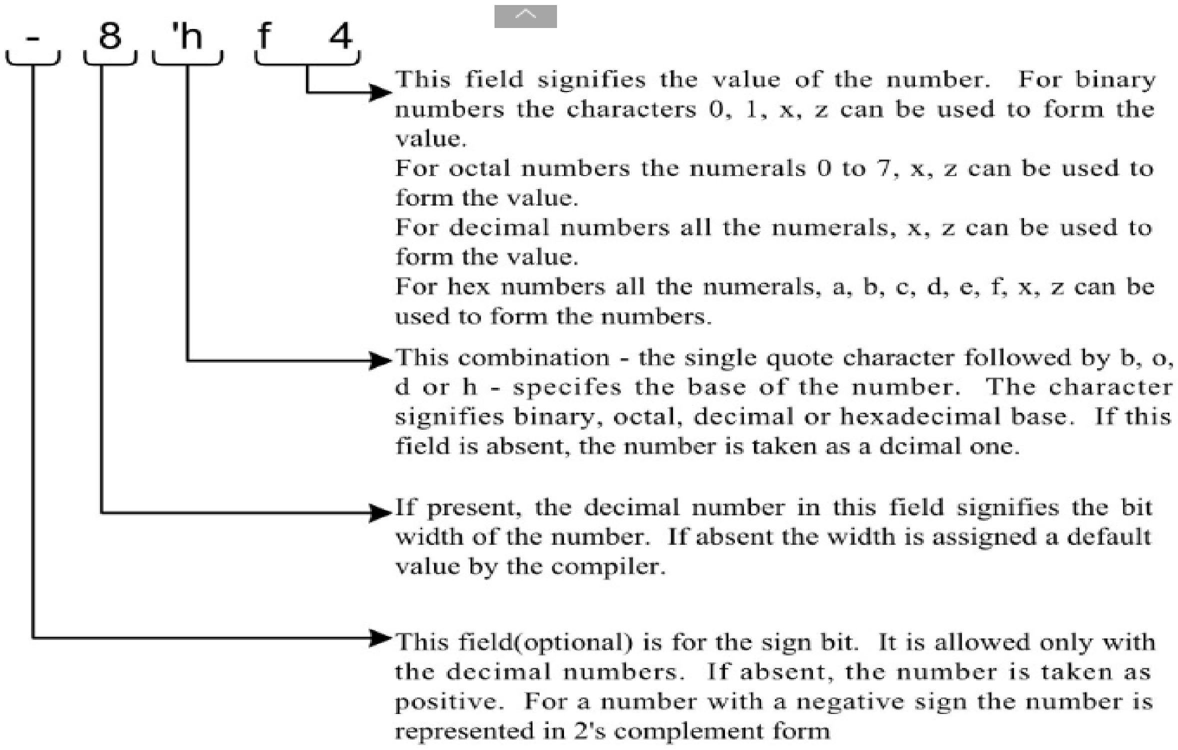
Padding and Truncation

The number of bits represented by the digits in the value part of the constant may be more or less than the given size. If there are more, then the high order bits are truncated. For example,

7'h8f is equivalent to 7'h0f

If the size is greater than the number of bits in the value part (which is a much more common case), then the number is padded on the left (high order part) with 0. However, if the left-most digit in the value part is x or z, then the number is padded with x or z.

```
12'h3    is equivalent to 12'b000000000011
12'h3x   is equivalent to 12'b00000011xxxx
12'bx    is equivalent to 12'bxxxxxxxxxxxx
12'oz37 is equivalent to 12'bzzzzzz011111
```



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 2.4 Representation of a number in Verilog*

A string constant is a sequence of characters enclosed in “” (double quotes). String constants may be used wherever a vector is allowed, and, in general, is equivalent to a vector whose width is 8 times the number of characters in the string. The value of the vector is the same as if the ASCII values of each character were concatenated.
For example,

"Hello World" is represented as 88'h48656c6c6f_20_576f726c64

Strings may be assigned to vectors, or used with operators in expressions. They are most commonly used as arguments to system tasks (like $display). Note that, though you can use a string constant where ever a vector is allowed, the reverse is not true. That is, you cannot put a format string into a variable and use that in a $display task invocation. (You can do it, it just won't do what you would expect.)

## IV. Elaborate Application

Verilog defines several scopes in which identifiers are defined. They are:

### Namespaces

*Table 1.5 Namespaces in identifiers*

| | |
|---|---|
| 1. Module | Net names, top-level register names, task names, function names, module and primitive instance names, and port names exist in the module's scope. That is, two different modules can each have a net named "net1 or a module instance named "foo_inst". However, there can be only one "net1" or "foo_inst" in a single module. |
| 2. Task, Function, Block | Tasks, functions, and named blocks allow registers, parameters, and named blocks to be defined within them. These names exist in hierarchical name spaces. That is, an identifier in an outer scope (module, task, function, or block) may be redefined in an inner scope (task, function, or block). |
| 3. Global | There is a single scope which contains all module types (i.e. the name used in the module definition). Thus, there can be only one module of type "DF99". |
| 4. Macros | Source macros have a single global scope that crosses module boundaries. |

A string is a sequence of characters enclosed within double quotes. A string must be contained on a single line; that is, it cannot be carried over to two lines with a carriage return. Special characters are specified by preceding them with the "\" character. Verilog treats a string as a sequence of ASCII characters – for example,

"This is a string"

"This string is one \t with a gap in between"

"This is called a \"string\"".

*Table 1.6 Different ways of number representation in Verilog*

| **Representation** | **Remarks** |
|---|---|
| 33‘d33 | Both of these represent decimal numbers of unspecified size – normally interpreted by Verilog as 32 bitwide, i.e., 0000 0000 0000 0000 0000 0000 0010 0001 |
| 9’D439<br>9’d439<br>9’D4_39 | All these represent 3 digit decimal numbers. D & d both specify decimal numbers. “_” (underscore) is ignored |
| 9’b1_1011__1x01<br>9’b11011x01<br>9’B11011x01 | All these represent binary numbers of value 11011x01. B & b specify binary numbers. “_” is ignored. x signifies the concerned bit to be of unknown value. |
| 9’o123<br>9’O123<br>9’o1x3<br>9’o12z | All these represent 9-bit octal numbers. The binary equivalents are001 010 011, 001 010 011, 001 xxx 011, 001 010 zzz respectively. z signifies the concerned bits to be in the high impedance state. |
| ‘o213 | An octal number of unspecified size having octal value 213. |
| 8’HA5<br>8’ha5<br>8’hA5<br>8’ha_5 | All these are 8 bit-wide-hex numbers of hex value a5h. The equivalent binary value is 1010 0101. |
| 11’hb0 | A 11 bit number with a hex assignment. Its value is 000 1011 0000. The number of bits specified is more than that indicated in the value field. Enough zeros are padded to the left as shown. |
| 9’hza | A hex number of 9 bits. Its value is taken as zzzzz 1010. |
| 5’hza | A 5-bit hex number. Its value is taken as z 1010. |
| 5’h?a | A 5-bit hex number. Its value is taken as z 1010. ‘?’ is another representation for ‘z’. |
| -3’b101<br>-5’h1a | Negative numbers. Negative numbers are represented in 2’s complement form. |
| -4’d7 | A 4 bit negative number. Its value in 2’s complement form is 7. Thus the number is actually – (16 – 7) = –9. |

**Keywords in Verilog**

These are the keywords in Verilog:

```
Always
Assign
Buf
bufif1
Casex
Cmos
Default
Disable
Else
Endcase
endfunction
endspecify
Endtask
For
Fork
highz0
If
inout
integer
large
medium
nand
nmos
not
notif1
output
pmos
primitive
pull1
pullup
real
reg
repeat
rpmos
rtranif0
scalared
specify
strength
strong1
supply1
task
tran
tranif1
tri0
triand
trireg
wait
weak0
while
wor
xor
and
begin
bufif0
case
casez
deassign
defparam
edge
end
endmodule
endprimitive
endtable
event
force
function
highz1
initial
input
join
macromodule
module
negedge
nor
notif0
or
parameter
posedge
pull0
pulldown
rcmos
realtime
release
rnmos
rtran
rtranif1
small
specparam
strong0
supply0
table
time
tranif0
tri
tri1
trior
vectored
wand
weak1
wire
xnor
```

## Come to think of it…

Verilog is similar in syntax to the C programming language. Hardware designers with previous C programming experience will find Verilog easy to learn.

Lexical conventions for operators, comments, whitespace, numbers, strings, and identifiers were discussed. Various data types are available in Verilog. There are four logic values, each with different strength levels. Available data types include nets, registers, vectors, numbers, simulation time, arrays, memories, parameters, and strings. Data types represent actual hardware elements very closely.

Verilog provides useful system tasks to do functions like displaying, monitoring, suspending, and finishing a simulation. Compiler directive `define is used to define text macros, and `include is used to include other Verilog files.

### References:

*Brown, S. D., & Vranesic, Z. G. (2014). Fundamentals of digital logic with Verilog design. New York: McGraw-Hill Higher Education.*
*Harris, S. L., & Harris, D. M. (2018). Digital design and computer architecture. Amsterdam: Elsevier / Morgan Kaufmann.*
*Paltnikar, Samir (2003). Verilog HDL: A Guide to Digital Design and Synthesis, Second Edition / Prentice Hall PTR*
*Padmanabhan, T.R. & B. Bala Tripura Sundari (2004). Design through Verilog HDL / IEEE Press*
*Quick Quartus: Verilog. (n.d.). Retrieved August 18, 2020, from*
*http://www.swarthmore.edu/NatSci/echeeve1/Ref/embedRes/QQS_V/QuickQuartusVerilog.html*
*Verilog Simulation. (n.d.). Retrieved August 18, 2020, from https://www.chipverify.com/verilog/verilog-simulation*

Name:_______________________________________________________Score:___________
Course/Sec:________________Schedule:_______________Date Submitted:________________

## V. Evaluate my Learning

**I.** Are the following **legal strings**? If not, write the correct strings

| Declaration | Verilog string |
|---|---|
| *"This is a string displaying the % sign"* | |
| *"out = in1 + in2"* | |
| *"Please ring a bell \007"* | |
| *"This is a backslash \ character\n"* | |

**II.** Are these **legal** or **illegal** identifiers? If it is illegal, write the correct identifier

*a. system1*

*b. 1reg*

*c. $latch*

*d. exec$*