UNIT-1

Syllabus: INTRODUCTION TO VERILOG: Verilog as HDL, Levels of design Description, Concurrency ,Simulation and Synthesis, Functional Verification, System Tasks Programming Language Interface (PLI), Module Simulation and Synthesis Tools, Test Benches.

LANGUAGE CONSTRUCTS AND CONVENTIONS: Introduction, Keywords, Identifiers, White Space Characters, Comments, Numbers, Strings, Logic Values, Strengths, Data Types, Scalars and Vectors, Parameters, Operators.

Objectives: *To make the student learn and understand*

- Acquire a basic knowledge of the Verilog HDL
- Language constructs and conventions in Verilog
- Basic Concepts of Verilog HDL like Data Types, System Tasks and Compiler Directives.

Outcomes: *The student will be able to*

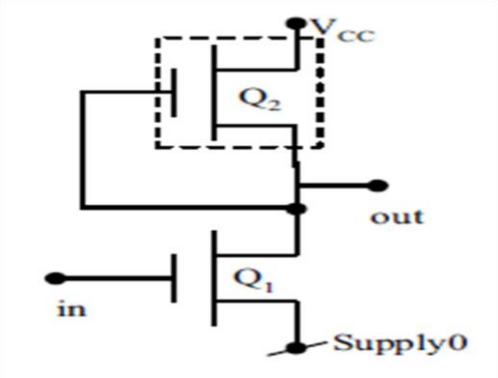
- › Define basic terms in HDL
- Knows Syntax and lexical conventions
- Remembers Data types, operators
- Remember test benches for simulation and verification

VERILOG AS HDL: Verilog has a variety of constructs as part of it. All are aimed at providing a functionally tested and a verified design description for the target FPGA or ASIC. The language has a dual function – one fulfilling the need for a design description and the other fulfilling the need for verifying the design for functionality and timing constraints like propagation delay, critical path delay, slack, setup, and hold times.

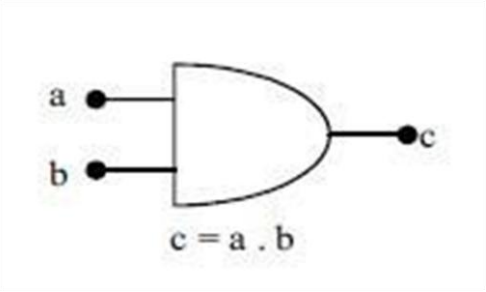
LEVELS OF DESIGN DESCRIPTION:

1. Gate Level
2. Data Flow
3. Behavioral Level
4. Circuit Level or switch level

Circuit Level or switch level: At the circuit level, a switch is the basic element with which digital circuits are built. Switches can be combined to form inverters and other gates at the next higher level of abstraction. Verilog has the basic MOS switches built into its constructs, which can be used to build basic circuits like inverters, basic logic gates, simple 1-bit dynamic and static memories. They can be used to build up larger designs to simulate at the circuit level, to design performance critical circuits.



Gate Level: All the basic gates are available as ready modules called "Primitives." Each such primitive is defined in terms of its inputs and outputs. Primitives can be incorporated into design descriptions directly. Just as full physical hardware can be built using gates, the primitives can be used repeatedly and judiciously to build larger systems. Figure shows an AND gate suitable for description using the gate primitive of Verilog. The gate level modeling or structural modeling as it is sometimes called is akin to building a digital circuit on a bread board, or on a PCB. One should know the structure of the design to build the model here. One can also build hierarchical circuits at this level. However, beyond 20 to 30 of such gate primitives in a circuit, the design description becomes unwieldy; testing and debugging become laborious.



Data Flow: Data flow is the next higher level of abstraction. All possible operations on signals and variables are represented here in terms of assignments. All logic and algebraic operations are accommodated. The assignments define the continuous functioning of the concerned block. At the data flow level, signals are assigned through the data manipulating equations. All such assignments are concurrent in nature. The design descriptions are more compact than those at the gate level. y = (ab+cd)

Behavioral Level: Behavioral level constitutes the highest level of design description; it is essentially at the system level itself [Bhaskar]. With the assignment possibilities, looping constructs and conditional branching possible, the design description essentially looks like a "C" program. The statements involved are "dense" in function. Compactness and the comprehensive nature of the design description make the development process fast and efficient.

Verilog Language Concepts

Concurrency: In an electronic circuit all the units are to be active and functioning concurrently. The voltages and currents in the different elements in the circuit can change simultaneously. In turn the logic levels too can change. Simulation of such a circuit in an HDL calls for concurrency of operation. All the activities scheduled at one time step are completed and then the simulator.

Simulation and Synthesis: The design that is specified and entered as described earlier is simulated for functionality and fully debugged. Translation of the debugged design into the corresponding hardware circuit (using an FPGA or an ASIC) is called "synthesis." The circuits realized from them are essentially direct translations of functions into circuit elements.

Functional Verification: Testing is an essential ingredient of the VLSI design process as with any hardware circuit. It has two dimensions to it – *functional tests* and *timing tests*. Testing or functional verification is carried out by setting up a "test bench" for the design.

Constructs for Modeling Timing Delays

Any basic gate has propagation delays and transmission delays associated with it. As the elements in the circuit increase in number, the type and variety of such delays increase rapidly; often one reaches a stage where the expected function is not realized thanks to an unduly large time delay. Thus there is a need to test every digital design for its performance with respect to time. Verilog has constructs for modeling the following delays:

- Gate delay
- Net delay
- Path delay
- Pin-to-pin delay

In addition, a design can be tested for setup time, hold time, clock-width time specifications, *etc*. Such constructs or delay models are akin to the finite delay time, rise time, fall time, path or propagation delays, *etc*., associated with real digital circuits or systems. The use of such constructs in the design helps simulate realistic conditions in a digital circuit. Further, one can change the values of delays in different ways. If a buffer capacity is increased, its associated delays can be reduced. If a design is to migrate to a better technology, the delay values can be rescaled. With such testing, one can estimate the minimum frequency of operation, the maximum speed of response, or typical response times.

System Tasks: A number of system tasks are available in Verilog. Though used in a design description, they are not part of it. Some tasks facilitate control and flow of the testing process. A set of system functions add to the flexibility of test benches:

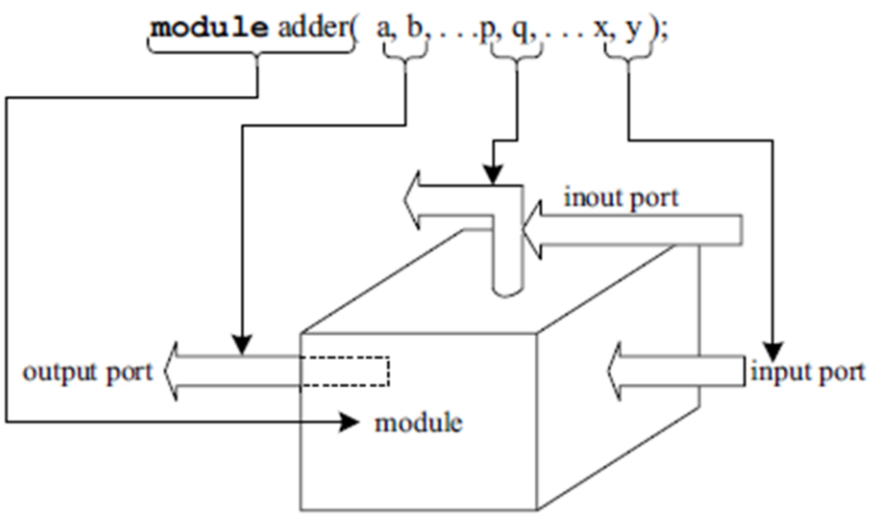
They are of three categories:

- Functions that keep track of the *progress of simulation time*
- Functions to *convert data or values of variables from one format to another*
- Functions *to generate random numbers with specific distributions.*

Programming Language Interface (PLI): Programming Language Interface (PLI) is a way to provide Application Program Interface (API) to Verilog HDL. Essentially it is a mechanism to invoke a C function from a Verilog code. PLI is primarily used for doing the things which would not have been possible otherwise using Verilog syntax. The key functions of the interface are as follows:

- One can read data from a file and pass it to a Verilog module as input. Such data can be test vectors or other input data to the module. Similarly, variables in Verilog modules can be accessed and their values written to output devices.
- Delay values, logic values, *etc*., within a module can be accessed and altered.
- Blocks written in C language can be linked to Verilog modules.

MODULE: 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.› The terminals of the module are referred to as 'ports'.



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

- **input** ports through which one gets entry into the module
- **output** ports through which one exits the module.
- **inout** ports: These represent ports through which one gets entry into the module or exits the module. All the constructs in Verilog are centred on the module.

MODULE SYNTAX

module module_name (port_list);
Input, output, inout declaration Intermediate variable declarations
Functional Description
(gate / switch / data flow / Behav.)
endmodule

SIMULATION AND SYNTHESIS TOOLS

A variety of Software tools related to VLSI design is available**.** Two of them are

- Modelsim: has been used to simulate the designs.
- Leonardo Spectrum of Mentor Graphics: has been used to obtain the synthesized circuits

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.

Verilog has the provision for all the above. One sets up a "test bench" in software and caries out a simulated test. The facilities required to set up test benches are discussed in detailed as

- Simulated testing is a time-based activity. It is usually carried out in simulated time. With any simulation tool the simulation progresses through equal simulation time steps. The time step can be 1 fs, 1 ps, 1 ns and so on.
- In the text the default value is taken as 1 ns. In some cases it is mentioned explicitly; in other situations it is implicit, *that is,* whenever 'time step' is mentioned, it implies 1ns of simulation time.

```
Initial
Begin
        a1 = 0;
        a2 = 0;
 #3     a1 = 1;
 #1     a1 = 0;
 #2     a2 = 1;
 #4     a1 = 1;
```

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. 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.
- 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.

Summarizing testing constitutes three activities:

- 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

TEST BENCH SYNTAX: A test bench is HDL code that allows you to provide a documented, repeatable set of stimuli.

module tb_module_name ;
Input, output, inout declaration
Intermediate variable declarations
Stimulus (initial / always)
endmodule

LANGUAGE CONSTRUCTS AND CONVENTIONS IN VERILOG

CASE SENSITIVITY: Verilog is a case-sensitive language like C. *Example:* sense, Sense, SENSE, sENse…etc., are all treated as different entities/quantities in verilog

KEYWORDS: The keywords define the language constructs. A keyword signifies an activity to be carried out, initiated, or terminated. All keywords in Verilog are in small letters. *Example:* module, endmodule, begin, end, if, while

IDENTIFIERS: Any program requires blocks of statements, signals, *etc., to be identified with an* attached nametag. Such nametags are identifiers

› 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

Valid Identifiers	Invalid Identifiers
name,	name aa,
_name,	$name,
name1,	1_name,
name_$,	@name,
\abc (escaped identifier)	a+b

WHITE SPACE CHARACTERS: Blanks (\b), tabs (\t), newlines (\n), and form feed form the white space characters in Verilog. Functionally, they separate legal tokens. They are introduced between keywords, keyword and an identifier, between two identifiers, between identifiers and operator symbols, and so on. White space characters have significance only when they appear inside strings

COMMENTS: A single line comment begins with "//" and ends with a new line. Multiline comments "/*" signifies the beginning of a comment and "*/" its end.

NUMBERS

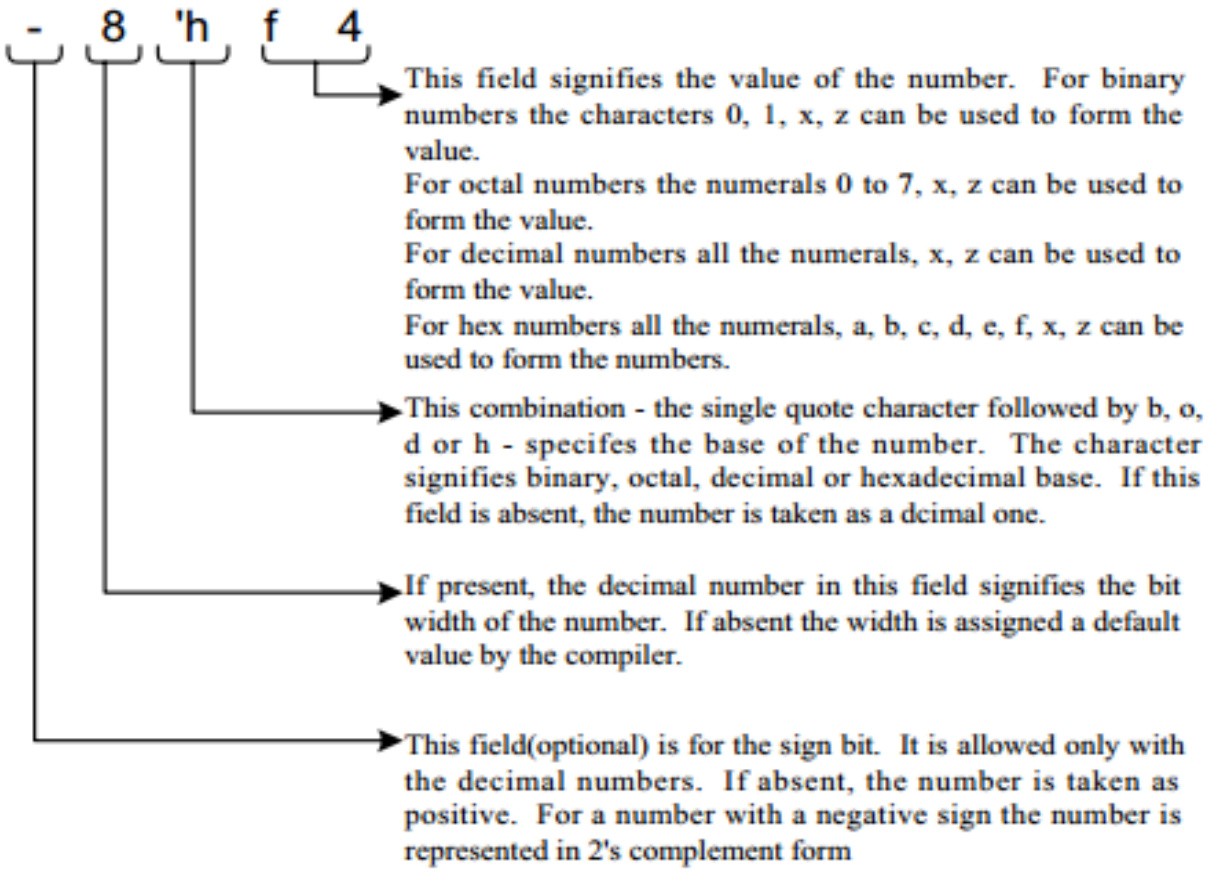
Integer Numbers: the number is taken as 32 bits wide. Ex: 25, 253, –253, - 8 'h f 4

Real Numbers: Real numbers can be specified in decimal or scientific notation.

Ex: 4.3, 4.3e2

STRINGS: A string is a sequence of characters enclosed within double quotes.

"This is a string"



LOGIC VALUES: signifies the 1 or high or true level and 0 signify the 0 or low or false level. Two additional levels are also possible designated as **x and z.** Here **x** represents an unknown or an uninitialized value. This corresponds to don't care case in logic circuits. **z** represents or signifies a high impedance state. 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 i.e., four of these are of the driving type, three are of capacitive type and one of the hi-Z type.

STRENGTHS

Strength Level	Strength Name	Element Modelled	Declaration Abbreviation
7	Supply Drive	Power supply connections.	supply
6	Strong Drive	Default gate & assign output strength.	strong
5	Pull Drive	Gate & assign output strength.	pull
4	Large Capacitor	Size of trireg net capacitor.	large
3	Weak Drive	Gate & assign output strength.	weak
2	Medium Capacitor	Size of trireg net capacitor.	medium
1	Small Capacitor	Size of trireg net capacitor.	small
0	High Impedence	Not Applicable.	highz

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.

Net data type: A net signifies a connection from one circuit unit to another, which 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.

Various nets supported in Verilog

- WIRE / TRI
- WOR / TRIOR
- TRI0
- SUPPLY1 -- For Vdd
- WAND / TRIAND
- TRI1
- TRIREG -- Infers a capacitance
- SUPPLY0 -- For Vss

DIFFERENCES BETWEEN *WIRE* AND *TRI*

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.

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

MEMORY

- Different types and sizes of memory, register file, stack, *etc*., can be formed by extending the vector concept.
- Thus the declaration **Reg [15:0] memory[511:0];**
- declares an array called “memory”; it has 512 locations. Each location is 16 bits wide. The value of any chosen location can be assigned to a selected register or *vice versa*;.

- As an example, consider the assignment statement

B = mem[(p-q)/2];

- The simulator first evaluates (p - q)/2 (which should be an integer) Then the data stored at mem is assigned to B.
- Different types of memory addressing like indirect, indexed, *etc*., can also be accommodated.

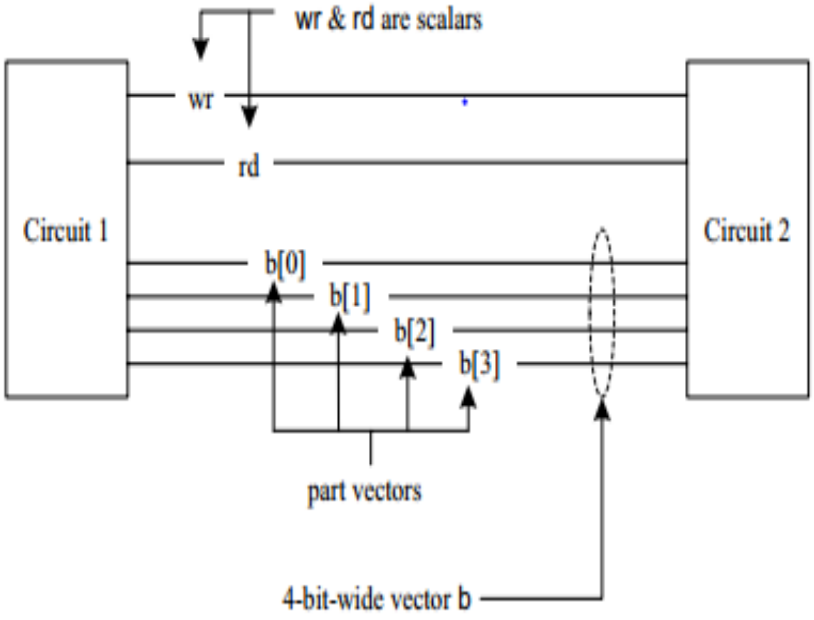
Scalars and Vectors: Entities representing single bits — whether the bit is stored, changed, or transferred — are called "scalars." Multiple lines carry signals in a cluster treated as a "vector."

reg[2:0] b;

reg[4:2] c;

wire[-2:2] d ;

All the above declarations are vectors. If range is not specifies it is treated as scalars



Examples:

```
wire[3:0] a;    /* a is a four bit vector of net type; the bits are designated as
                   a[3], a[2], a[1] and a[0]. */
reg[2:0] b;     /* b is a three bit vector of reg type; the bits are designated as
                   b[2], b[1] and b[0]. */
reg[4:2] c;     /* c is a three bit vector of reg type; the bits are designated as
                   c[4], c[3] and c[2]. */
wire[-2:2] d ;  /* d is a 5 bit vector with individual bits designated as d[-2],
                   d[-1], d[0], d[1] and d[2]. */
```

PARAMETERS: All constants can be declared as parameters at the outset in a Verilog module.

Example: parameter word_size = 16;

parameter word_size = 16, mem_size = 256;

OPERATORS

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.

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.
- The "$" symbol identifies a system task. A task has the format $<keyword>

$display: When the system encounters this task, the specified items are displayed in the formats specified and the system advances to a new line.

Example: **$display ("The value of a is : a = , %d", a);**

$monitor: The $monitor task monitors the variables specified whenever any one of those specified changes. A monitor statement need appear only once in a simulation program.

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

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.

›