DIGITAL DESIGN THROUGH VERILOG

Lecture Notes

B.TECH
(III YEAR – I SEM)
(2018-19)

Prepared by:

Mr. M. ARUN KUMAR, Associate Professor
Mr. K. SURESH, Assistant Professor

Department of Electronics and Communication Engineering

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
(Autonomous Institution – UGC, Govt. of India)
Recognized under 2(f) and 12 (B) of UGC ACT 1956
(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015 Certified)
Maisammaguda, Dhulapally (Post Via. Kompally), Secunderabad – 500100, Telangana State, India

Unit-1

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

The components of the target design can be described at different levels with the help of the constructs in Verilog.

In Verilog HDL a module can be defined using various levels of abstraction. There are four levels of abstraction in verilog.
They are: 1. Circuit Level 2. Gate Level 3. Data Flow Level 4. Behavioral Level

Circuit 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.

The below Figure1 shows the circuit of an inverter suitable for description with the switch level constructs of Verilog.

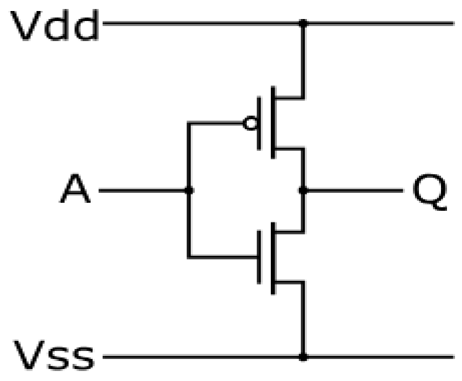


Figure 1 CMOS inverter

Gate Level

At the next higher level of abstraction, design is carried out in terms of basic gates. 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 2 shows an AND gate suitable for description using the gate primitive of Verilog.

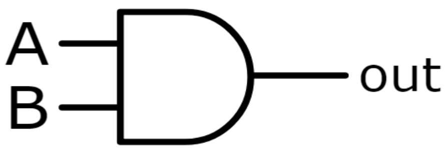


Figure 2 AND gate symbol

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.

Figure 3 shows an A-O-I relationship suitable for description with the Verilog constructs at the data flow level.

$$\boxed{e=\overline{a.b+c.d}}$$

Figure 3 An A-O-I gate represented as a data flow type of relationship.

Behavioral Level

Behavioral level constitutes the highest level of design description; it is essentially at the system level itself. With the assignment possibilities, looping constructs and conditional branching possible, the design description essentially looks like a “C” program.

A module can be implemented in terms of the design algorithm. The designer no need to have any knowledge of hardware implementation.

The statements involved are “dense” in function. Compactness and the comprehensive nature of the design description make the development process fast and efficient.

Figure 4 shows an A-O-I gate expressed in pseudo code suitable for description with the behavioral level constructs of Verilog.

```
If ($a$, $b$, $c$ or $d$ changes)
Compute $e$ as
$e = \overline{a.b + c.d}$
```

Figure . 4 An A-O-I gate in pseudo code at behavioral level.

The Overall Design Structure in Verilog

The possibilities of design description statements and assignments at different levels necessitate their accommodation in a mixed mode. In fact the design statements coexisting in a seamless manner within a design module is a significant characteristic of Verilog. Thus Verilog facilitates the mixing of the above-mentioned levels of design. A design built at data flow level can be instantiated to form a structural mode design. Data flow assignments can be incorporated in designs which are basically at behavioral level.

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.

A number of activities – may be spread over different modules – are to be run concurrently here. Verilog simulators are built to simulate concurrency. (This is in contrast to programs in the normal languages like C where execution is sequential.)

Concurrency is achieved by proceeding with simulation in equal time steps. The time step is kept small enough to be negligible compared with the propagation delay values. All the activities scheduled at one time step are completed and then the simulator advances to the next time step and so on. The time step values refer to simulation time and not real time. One can redefine timescales to suit technology as and when necessary and carry out test runs.

In some cases the circuit itself may demand sequential operation as with data transfer and memory-based operations. Only in such cases sequential operation is ensured by the appropriate usage of sequential constructs from Verilog HDL.

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 tools available for synthesis relate more easily with the gate level and data flow level modules [Smith MJ]. The circuits realized from them are essentially direct translations of functions into circuit elements.

In contrast many of the behavioral level constructs are not directly synthesizable; even if synthesized they are likely to yield relatively redundant or wrong hardware. The way out is to take the behavioral level modules and redo each of them at lower levels. The process is carried out successively with each of the behavioral level modules until practically the full design is available as a pack of modules at gate and data flow levels (more commonly called the “RTL level”).

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. Both can be carried out with Verilog. Often testing or functional verification is carried out by setting up a "test bench" for the design. The test bench will have the design instantiated in it; it will generate necessary test signals and apply them to the instantiated design. The outputs from the design are brought back to the test bench for further analysis. The input signal combinations, waveforms and sequences required for testing are all to be decided in advance and the test bench configured based on the same. The test benches are mostly done at the behavioral level. The constructs there are flexible enough to allow all types of test signals to be generated.

In the process of testing a module, one may have to access variables buried inside other modules instantiated within the master module. Such variables can be accessed through suitable hierarchical addressing.

Test Inputs for Test Benches

Any digital system has to carry out a number of activities in a defined manner. Once a proper design is done, it has to be tested for all its functional aspects. The system has to carry out all the expected activities and not falter. Further, it should not malfunction under any set of input conditions. Functional testing is carried out to check for such requirements. Test inputs can be purely combinational, periodic, numeric sequences, random inputs, conditional inputs, or combinations of these. With such requirements, definition and design of test benches is often as challenging as the design itself. As the circuit design proceeds, one develops smaller blocks and groups them together to form bigger circuit units. The process is repeated until the whole system is fully built up. Every stage calls for tests to see whether the subsystem at that layer behaves in the manner expected. Such testing calls for two types of observations:

- Study of signals within a small unit when test inputs are given to the whole unit.
- Isolation of a small element and doing local test to facilitate debugging.

Verilog has constructs to accommodate both types of observation through a hierarchical description of variables within.

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.

Programming Language Interface (PLI)

PLI provides an active interface to a compiled Verilog module. The interface adds a new dimension to working with Verilog routines from a C platform. 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.

LANGUAGE CONSTRUCTS AND CONVENTIONS IN VERILOG

Introduction

The constructs and conventions make up a software language. A clear understanding and familiarity of these is essential for the mastery of the language. Verilog has its own constructs and conventions [IEEE, Sutherland]. In many respects they resemble those of C language [Gottfried].

Any source file in Verilog (as with any file in any other programming language) is made up of a number of ASCII characters. The characters are grouped into sets — referred to as "lexical tokens." A lexical token in Verilog can be a single character or a group of characters. Verilog has 7 types of lexical tokens- operators, keywords, identifiers, white spaces, comments, numbers, and strings.

Case Sensitivity

Verilog is a case-sensitive language like C. Thus sense, Sense, SENSE, sENse,… etc., are all related 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. 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.

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 m not allowed as an identifier because of the presence of the character “+”.

White Space Characters

Blanks (\b), tabs (\t), newlines (\n), and formfeed form the white space characters in Verilog. In any design description the white space characters are included to improve readability. 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

Comments can be inserted in the code for readability and documentation. There are two ways to write comments. A one-line comment starts with "//". Verilog skips from that point to the end of line. A multiple-line comment starts with "/*" and ends with "*/". Multiple-line comments cannot be nested. However, one-line comments can be embedded in multiple-line comments.

```
a = b && c; // This is a one-line comment

/* This is a multiple line

comment */

/* This is /* an illegal */ comment */

/* This is //a legal comment */
```

Operators

Operators are of three types: unary, binary, and ternary. Unary operators precede the operand. Binary operators appear between two operands. Ternary operators have two separate operators that separate three operands.

a = ~ b; // ~ is a unary operator. b is the operand

a = b && c; // && is a binary operator. b and c are operands

a = b ? c : d; // ?: is a ternary operator. b, c and d are operands

Number Specification

There are two types of number specification in Verilog: sized and unsized.
Sized numbers
Sized numbers are represented as <size> '<base format> <number>.

<size> is written only in decimal and specifies the number of bits in the number. Legal base formats are decimal ('d or 'D), hexadecimal ('h or 'H), binary ('b or 'B) and octal ('o or 'O). The number is specified as consecutive digits from 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f. Only a subset of these digits is legal for a particular base. Uppercase letters are legal for number specification.

4'b1111 // This is a 4-bit binary number
12'habc // This is a 12-bit hexadecimal number
16'd255 // This is a 16-bit decimal number.

Unsized numbers

Numbers that are specified without a <base format> specification are decimal numbers by default. Numbers that are written without a <size> specification have a default number of bits that is simulator- and machine-specific (must be at least 32).

23456 // This is a 32-bit 'hc3 // This is a 32-bit 'o21 // This is a 32-bit

decimal number by default hexadecimal number octal number

X or Z values

Verilog has two symbols for unknown and high impedance values. These values are very important for modeling real circuits. An unknown value is denoted by an x. A high impedance value is denoted by z.

12'h13x // This is a 12-bit hex number; 4 least significant bits unknown

6'hx // This is a 6-bit hex number

32'bz // This is a 32-bit high impedance number

An x or z sets four bits for a number in the hexadecimal base, three bits for a number in the octal base, and one bit for a number in the binary base. If the most significant bit of a number is 0, x, or z, the number is automatically extended to fill the most significant bits, respectively, with 0, x, or z. This makes it easy to assign x or z to whole vector. If the most significant digit is 1, then it is also zero extended.

Negative numbers

Negative numbers can be specified by putting a minus sign before the size for a constant number. Size constants are always positive. It is illegal to have a minus sign between <base format> and <number>. An optional signed specifier can be added for signed arithmetic.

-6'd3 // 8-bit negative number stored as 2's complement of 3 -6'sd3 // Used for performing signed integer math 4'd-2 // Illegal specification

Underscore characters and question marks

An underscore character "_" is allowed anywhere in a number except the first character. Underscore characters are allowed only to improve readability of numbers and are ignored by Verilog.
A question mark "?" is the Verilog HDL alternative for z in the context of numbers.
12'b1111_0000_1010 // Use of underline characters for readability

4'b10?? // Equivalent of a 4'b10zz

Strings

A string is a sequence of characters that are enclosed by double quotes. The restriction on a string is that it must be contained on a single line, that is, without a carriage return. It cannot be on multiple lines. Strings are treated as a sequence of one-byte ASCII values.

"Hello Verilog World" // is a string

"a / b" // is a string

Value Set or Logic Values

Verilog supports four values and eight strengths to model the functionality of real hardware. The four value levels are listed in Table below.

Value Level	Condition in Hardware Circuits
0	Logic zero, false condition
1	Logic one, true condition
x	Unknown logic value
Z	High impedance, floating state

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.

In addition to logic values, strength levels are often used to resolve conflicts between drivers of different strengths in digital circuits. Value levels 0 and 1 can have the strength levels listed in Table below

Strength Level	**Type**	**Degree**
supply	Driving	strongest
strong	Driving	
pull	riving	
large	Storage	↑
weak	Driving	
medium	Storage	
small	Storage	
highz	High Impedance	weakest

If two signals of unequal strengths are driven on a wire, the stronger signal prevails.
For example, if two signals of strength strong1 and weak0 contend, the result is resolved as a strong1. If two signals of equal strengths are driven on a wire, the result is unknown. If two signals of strength strong1 and strong0 conflict, the result is an x. Strength levels are particularly useful for accurate modeling of signal contention, MOS devices, dynamic MOS, and other low-level devices.

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.

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.
Nets are one-bit values by default unless they are declared explicitly as vectors. The terms wire and net are often used interchangeably.

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.

Time
Verilog simulation is done with respect to simulation time. A special time register data type is used in Verilog to store simulation time. A time variable is declared with the keyword time. The width for time register data types is implementation-specific but is at least 64 bits. The system function $time is invoked to get the current simulation time.

```
time save_sim_time; // Define a time variable save_sim_time initial

save_sim_time = $time; // Save the current simulation time
```

Scalars and Vectors

Entities representing single bits — whether the bit is stored, changed, or transferred — are called "scalars." Often multiple lines carry signals in a cluster – like data bus, address bus, and so on. Similarly, a group of regs stores a value, which may be assigned, changed, and handled together. The collection here is treated as a "vector."
Figure below illustrates the difference between a scalar and a vector. wr and rd are two scalar nets connecting two circuit blocks circuit1 and circuit2. b is a 4-bit-wide vector net connecting the same two blocks. b[0], b[1], b[2], and b[3] are the individual bits of vector b. They are "part vectors."

A vector reg or net is declared at the outset in a Verilog program and hence treated as such. The range of a vector is specified by a set of 2 digits (or expressions evaluating to a digit) with a colon in between the two. The combination is enclosed within square brackets.

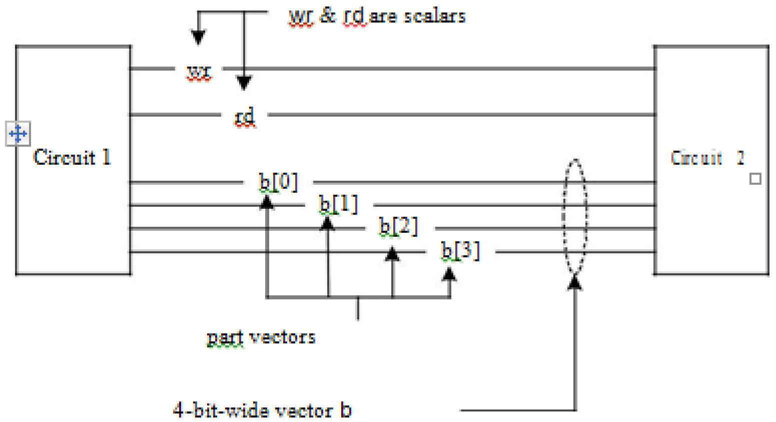


Figure Illustration of scalars and vectors.

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]. */

Whenever a range is not specified for a net or a reg, the same is treated as a scalar – a single bit quantity. In the range specification of a vector the most significant bit and the least significant bit can be assigned specific integer values. These can also be expressions evaluating to integer constants – positive or negative.

Normally vectors – nets or regs – are treated as unsigned quantities. They have to be specifically declared as “signed” if so desired.

Examples

```
wire signed[4:0] num; // num is a vector in the range -16 to +15.

reg signed [3:0] num_1;         // num_1 is a vector in the range -8 to +7.
```

Unit-2

Gate Level Modeling

Introduction

Digital designers are normally familiar with all the common logic gates, their symbols, and their working. Flip-flops are built from the logic gates. All other functionally complex and more involved circuits can also be built using the basic gates. All the basic gates are available as “Primitives” in Verilog. Primitives are generalized modules that already exist in Verilog [IEEE]. They can be instantiated directly in other modules.

And Gate Primitive

The AND gate primitive in Verilog is instantiated with the following statement:

and g1 (O, I1, I2, . . ., In);

Here ‘and’ is the keyword signifying an AND gate. g1 is the name assigned to the specific instantiation. O is the gate output; I1, I2, etc., are the gate inputs. The following are noteworthy:

- The AND module has only one output. The first port in the argument list is the output port.
- An AND gate instantiation can take any number of inputs — the upper limit is compiler-specific.
- A name need not be necessarily assigned to the AND gate instantiation; this is true of all the gate primitives available in Verilog.

Truth Table of AND Gate Primitive

The truth table for a two-input AND gate is shown in Table below It can be directly extended to AND gate instantiations with multiple inputs. The following observations are in order here:

Truth table of AND gate primitive

		Input 1			
		0	1	x	z
Input 2	0	0	0	0	0
	1	0	1	x	x
	x	0	x	x	x
	z	0	x	x	x

- If any one of the inputs to the AND gate instantiation is in the 0 state, its output is also in the 0 state. It is irrespective of whether the other inputs are at the 0, 1, x or z state.

- The output is at 1 state if and only if every one of the inputs is at 1 state.

- For all other cases the output is at the x state.

- Note that the output is never at the z state – the high impedance state. This is true of all other gate primitives as well.

Module Structure

In a general case a module can be more elaborate. A lot of flexibility is available in the definition of the body of the module. However, a few rules need to be followed:

- The first statement of a module starts with the keyword module; it may be followed by the name of the module and the port list if any.

- All the variables in the ports-list are to be identified as inputs, outputs, or inouts. The corresponding declarations have the form shown below:

▯ Input a1, a2;
▯ Output b1, b2;
▯ Inout c1, c2;

The port-type declarations here follow the module declaration mentioned above.

- The ports and the other variables used within the body of the module are to be identified as nets or registers with specific types in each case. The respective declaration statements follow the port-type declaration statements.

Examples:

wire a1, a2, c;
reg b1, b2;

The type declaration must necessarily precede the first use of any variable or signal in the module.

- The executable body of the module follows the declaration indicated above.

- The last statement in any module definition is the keyword “endmodule”.

- Comments can appear anywhere in the module definition.

Other Gate Primitives

All other basic gates are also available as primitives in Verilog. Details of the facilities and instantiations in each case are given in Table below. The following points are noteworthy here:

- In all cases of instantiations, one need not necessarily assign a name to the instantiation. It need be done only when felt necessary – say for clarity of circuit description.
- In all the cases the output port(s) is (are) declared first and the input port(s) is (are) declared subsequently.
- The buffer and the inverter have only one input each. They can have any number of outputs; the upper limit is compiler-specific. All other gates have one output each but can have any number of inputs; the upper limit is again compiler-specific.

Table for Basic gate primitives in Verilog with details

Gate	Mode of instantiation	Output port(s)	Input port(s)
AND	**and** ga (o, i1, i2, . . . i8);	o	i1, i2, . .
OR	**or** gr (o, i1, i2, . . . i8);	o	i1, i2, . .
NAND	**nand** gna (o, i1, i2, . . . i8);	o	i1, i2, . .
NOR	**no**r gnr (o, i1, i2, . . . i8);	o	i1, i2, . .
XOR	**xor** gxr (o, i1, i2, . . . i8);	o	i1, i2, . .
XNOR	**xnor** gxn (o, i1, i2, . . . i8);	o	i1, i2, . .
BUF	**buf** gb (o1, o2, …. i);	o1, o2, o3, . .	i
NOT	**not** gn (o1, o2, o3, . . . i);	o1, o2, o3, . .	i

Example for a typical A-O-I gate circuit

The commonly used A-O-I gate is shown in Figure 1 for a simple case. The module and the test bench for the same are given in Figure 2. The circuit has been realized here by instantiating the AND and NOR gate primitives. The names of signals and gates used in the instantiations in the module of Figure 2 remain the same as those in the circuit of Figure 1. The module aoi_gate in the figure has input and output ports since it describes a circuit with signal inputs and an output. The module aoi_st is a stimulus module. It generates inputs to the aoi_gate module and gets its output. It has no input or output ports.

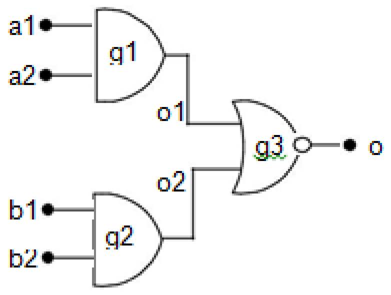


Figure for a typical A-O-I gate circuit.

```
/*module for the aoi-gate of figure 1 instantiating the gate primitives – fig 2*/

module aoi_gate(o,a1,a2,b1,b2);

input a1,a2,b1,b2;     // a1,a2,b1,b2 form the input //ports of the module

output o;              //o is the single output port of the module

wire o1,o2;            //o1 and o2 are intermediate signals //within the module

and g1(o1,a1,a2);   //The AND gate primitive has two and g2(o2,b1,b2);

                    // instantiations with assigned //names g1 & g2.

nor g3(o,o1,o2);    //The nor gate has one instantiation with assigned name g3.

endmodule

//Test-bench for the aoi_gate above
module aoi_st;
reg a1,a2,b1,b2;

//specific values will be assigned to a1,a2,b1, // and b2 and these connected
//to input ports of the gate insatntiations;
```

```
//hence these variables are declared as reg
wire o;
initial
begin
a1 = 0;
a2 = 0;
b1 = 0;
b2 = 0;
#3 a1 = 1;
#3 a2 = 1;
#3 b1 = 1;
#3 b2 = 0;
#3 a1 = 1;
#3 a2 = 0;
#3 b1 = 0;
end
initial #100 $stop;//the simulation ends after //running for 100 tu's.
initial $monitor($time , " o = %b , a1 = %b , a2 = %b , b1 = %b ,b2 = %b ",o,a1,a2,b1,b2);
aoi_gate gg(o,a1,a2,b1,b2);
endmodule
```

Tri-State Gates

Four types of tri-state buffers are available in Verilog as primitives. Their outputs can be turned ON or OFF by a control signal. The direct buffer is instantiated as
Bufif1 nn (out, in, control);

The symbol of the buffer is shown in Figure 1. We have

- out as the single output variable
- in as the single input variable and
- control as the single control signal variable.

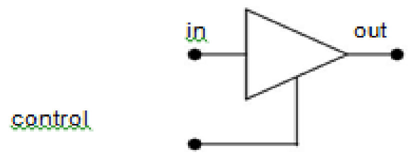


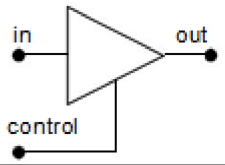
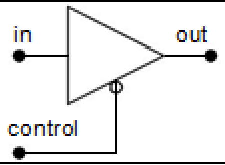
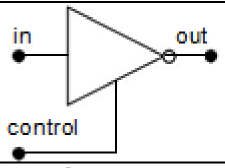
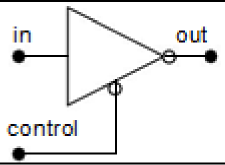
Figure 1 A tri-state buffer.

When
control = 1,
out = in.

When
control = 0,
out=tri-stated

out is cut off from the input and tri-stated. The output, input and control signals should appear in the instantiation in the same order as above. Details of bufif1 as well as the other tri-state type primitives are shown in Table 1.
In all the cases shown in Table 1, out is the output; in is the input, and control, the control variable.

Table 1 Instantiation and functional details of tri-state buffer primitives

Typical instantiation	Functional representation	Functional description
`bufif1` (out, in, control);		Out = in if control = 1; else out = z
`bufif0` (out, in, control);		Out = in if control = 0; else out = z
`notif1` (out, in, control);		Out = complement of in if control = 1; else out = z
`notif0` (out, in, control);		Out = complement of in if control = 0; else out = z

Array of Instances of Primitives

The primitives available in Verilog can also be instantiated as arrays. A judicious use of such array instantiations often leads to compact design descriptions. A typical array instantiation has the form

```
and gate [7 : 4 ] (a, b, c);
```

where a, b, and c are to be 4 bit vectors. The above instantiation is equivalent to combining the following 4 instantiations:

```
and gate [7] (a[3], b[3], c[3]), gate [6] (a[2], b[2], c[2]), gate [5] (a[1], b[1], c[1]), gate [4] (a[0], b[0], c[0]);
```

The assignment of different bits of input vectors to respective gates is implicit in the basic declaration itself. A more general instantiation of array type has the form

```
and gate[mm : nn](a, b, c);
```

where mm and nn can be expressions involving previously defined parameters, integers and algebra with them. The range for the gate is 1+ (mm-nn); mm and nn do not have restrictions of sign; either can be larger than the other.

Gate Delays

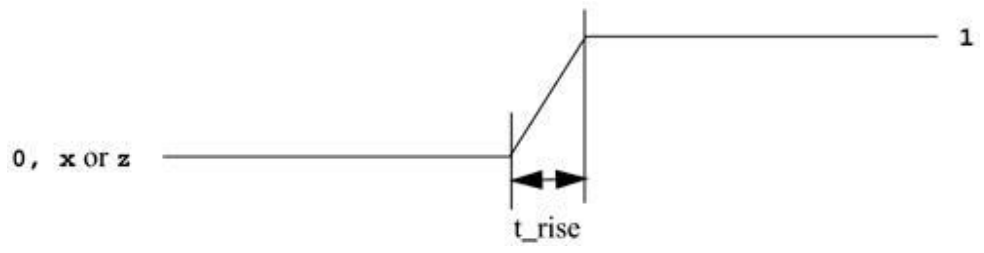
Until now, we described circuits without any delays (i.e., zero delay). In real circuits, logic gates have delays associated with them. Gate delays allow the Verilog user to specify delays through the logic circuits. Pin-to-pin delays can also be specified in Verilog.

Rise, Fall, and Turn-off Delays

There are three types of delays from the inputs to the output of a primitive gate.

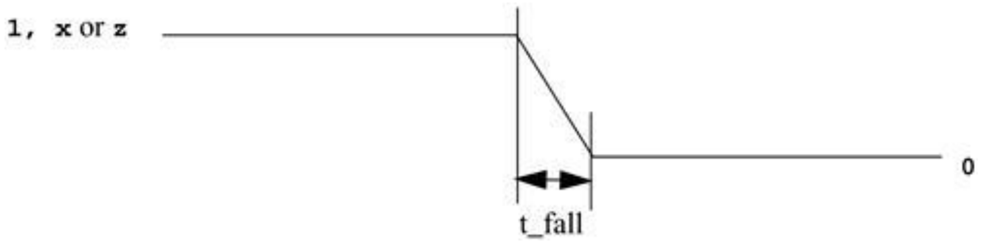
Rise delay

The rise delay is associated with a gate output transition to a 1 from another value.



Fall delay

The fall delay is associated with a gate output transition to a 0 from another value.



Turn-off delay

The turn-off delay is associated with a gate output transition to the high impedance value (z) from another value.

If the value changes to x, the minimum of the three delays is considered.

Three types of delay specifications are allowed. If only one delay is specified, this value is used for all transitions. If two delays are specified, they refer to the rise and fall delay values. The turn-off delay is the minimum of the two delays. If all three delays are specified, they refer to rise, fall, and turn-off delay values. If no delays are specified, the default value is zero. Examples of delay specification are shown in below

Example--Types of Delay Specification

```
    //Delay of delay_time for all transitions
and #(delay_time) a1(out, i1, i2);

    // Rise and Fall Delay Specification.

and #(rise_val, fall_val) a2(out, i1, i2);

// Rise, Fall, and Turn-off Delay Specification

bufif0 #(rise_val, fall_val, turnoff_val) b1 (out, in, control);
```

Examples of delay specification are shown below.

```
and #(5) a1(out, i1, i2); //Delay of 5 for all transitions and #(4,6) a2(out, i1, i2); // Rise =
4, Fall = 6

bufif0 #(3,4,5) b1 (out, in, control); // Rise = 3, Fall = 4, Turn-off = 5
```

Unit-3
Behavioral Modeling

Introduction

Behavioral modeling is the highest level of abstraction in the Verilog HDL. The other modeling techniques are relatively detailed. They require some knowledge of how hardware or hardware signals work. The abstraction in this modeling is as simple as writing the logic in C language. This is a very powerful abstraction technique. All that designer needs is the algorithm of the design, which is the basic information for any design.
Most of the behavioral modeling is done using two important constructs: initial and always. All the other behavioral statements appear only inside these two structured procedure constructs.

The Initial Construct

The statements which come under the *initial* construct constitute the initial block. The initial block is executed only once in the simulation, at time 0. If there is more than one initial block, then all the initial blocks are executed concurrently. The initial construct is used as follows:

```
initial
begin
reset=1'b0;
clk=1'b1;
end
or
initial
clk = 1'b1;
```

In the first initial block there are more than one statements hence they are written between begin and end. If there is only one statement then there is no need to put begin and end.

The always construct

The statements which come under the *always* construct constitute the always block. The always block starts at time 0, and keeps on executing all the simulation time. It works like a infinite loop. It is generally used to model a functionality that is continuously repeated.

```
always
#5clk=~clk;
initial
clk = 1'b0;
```

The above code generates a clock signal clk, with a time period of 10 units. The initial blocks initiates the clk value to 0 at time 0. Then after every 5 units of time it toggled, hence we get a time period of 10 units. This is the way in general used to generate a clock signal for use in test benches.

```
always@(posedge clk, negedge reset)
begin
a = b + c;
```

```
  d = 1'b1;
end
```

In the above example, the always block will be executed whenever there is a positive edge in the clk signal, or there is negative edge in the reset signal. This type of always is generally used in implement a FSM, which has a reset signal.

```
always @(b,c,d)
begin
  a = ( b + c )*d;
  e = b | c;
end
```

In the above example, whenever there is a change in b, c, or d the always block will be executed. Here the list b, c, and d is called the *sensitivity list*.

In the Verilog 2000, we can replace always @(b,c,d) with always @(*), it is equivalent to include all input signals, used in the always block. This is very useful when always blocks is used for implementing the combination logic.

Procedural Assignments

Procedural assignments are used for updating *reg*, *integer*, *time*, *real*, *realtime*, and *memory* data types. The variables will retain their values until updated by another procedural assignment. There is a significant difference between procedural assignments and continuous assignments.
Continuous assignments drive nets and are evaluated and updated whenever an input operand changes value. Where as procedural assignments update the value of variables under the control of the procedural flow constructs that surround them.

The LHS of a procedural assignment could be:

- *reg*, *integer*, *real*, *realtime*, or *time* data type.
- Bit-select of a *reg*, *integer*, or *time* data type, rest of the bits are untouched.
- Part-select of a *reg*, *integer*, or *time* data type, rest of the bits are untouched.
- Memory word.

Concatenation of any of the previous four forms can be specified.
When the RHS evaluates to fewer bits than the LHS, then if the right-hand side is signed, it will be sign-extended to the size of the left-hand side.

There are two types of procedural assignments: **blocking and non-blocking assignments.**

Blocking assignments: A blocking assignment statements are executed in the order they are specified in a sequential block. The execution of next statement begins only after the completion of the present blocking assignments. A blocking assignment will not block the execution of the next statement in a parallel block. The blocking assignments are made using the operator =.

```
initial
begin
   a = 1;
   b = #5 2;
   c = #2 3;
end
```

In the above example, a is assigned value 1 at time 0, and b is assigned value 2 at time 5, and c is assigned value 3 at time 7.

Non-blocking assignments: The nonblocking assignment allows assignment scheduling without blocking the procedural flow. The nonblocking assignment statement can be used whenever several variable assignments within the same time step can be made without regard to order or dependence upon each other. Non-blocking assignments are made using the operator <=.
Note: <= is same for less than or equal to operator, so whenever it appears in a expression it is considered to be comparison operator and not as non-blocking assignment.

```
initial
begin
   a <= 1;
   b <= #5 2;
   c <= #2 3;
end
```

In the above example, a is assigned value 1 at time 0, and b is assigned value 2 at time 5, and c is assigned value 3 at time 2 (because all the statements execution starts at time 0, as they are non-blocking assignments.

Block Statements

Block statements are used to group two or more statements together, so that they act as one statement. There are two types of blocks:

- Sequential block.
- Parallel block.

Sequential block: The sequential block is defined using the keywords *begin* and *end*. The procedural statements in sequential block will be executed sequentially in the given order. In sequential block delay values for each statement shall be treated relative to the simulation time of the execution of the previous statement. The control will pass out of the block after the execution of last statement.

Parallel block: The parallel block is defined using the keywords *fork* and *join*. The procedural statements in parallel block will be executed concurrently. In parallel block delay values for each statement are considered to be relative to the simulation time of entering the block. The delay control can be used to provide time-ordering for procedural assignments. The control shall pass out of the block after the execution of the last time-ordered statement.

Note that blocks can be nested. The sequential and parallel blocks can be mixed.

Block names: All the blocks can be named, by adding *: block_name* after the keyword *begin* or *fork*. The advantages of naming a block are:

It allows to declare local variables, which can be accessed by using hierarchical name referencing.
They can be disabled using the *disable* statement (*disable block_name;*).

Conditional (if-else) Statement

The condition (if-else) statement is used to make a decision whether a statement is executed or not. The keywords *if* and *else* are used to make conditional statement. The conditional statement can appear in the following forms.

```
if ( condition_1 )
   statement_1;

if ( condition_2 )
   statement_2;
else
   statement_3;

if ( condition_3 )
   statement_4;
else if ( condition_4 )
   statement_5;
else
   statement_6;

if ( condition_5 )
begin
   statement_7;
   statement_8;
end
else
begin
   statement_9;
   statement_10;
end
```

Conditional (if-else) statement usage is similar to that if-else statement of C programming language, except that parenthesis are replaced by *begin* and *end*.

Case Statement

The case statement is a multi-way decision statement that tests whether an expression matches one of the expressions and branches accordingly. Keywords *case* and *endcase* are used to make a case statement.

The case statement syntax is as follows.

```
case (expression)
   case_item_1: statement_1;
   case_item_2: statement_2;
   case_item_3: statement_3;
   ...
   ...
   default: default_statement;
endcase
```

If there are multiple statements under a single match, then they are grouped using begin, and end keywords. The default item is optional.

Case statement with don't cares: casez and casex

casez treats high-impedance values (z) as don't cares. *casex* treats both high-impedance (z) and unknown (x) values as don't cares. Don't-care values (z values for *casez*, z and x values for *casex*) in any bit of either the case expression or the case items shall be treated as don't-care conditions during the comparison, and that bit position shall not be considered. The don't cares are represented using the ? mark.

Loop Statements

There are four types of looping statements in Verilog:

forever
repeat
while
for

Forever Loop

Forever loop is defined using the keyword forever, which Continuously executes a statement. It terminates when the system task $finish is called. A forever loop can also be ended by using the disable statement.

```
initial
begin
   clk = 1'b0;
   forever #5 clk = ~clk;
end
```

In the above example, a clock signal with time period 10 units of time is obtained.

Repeat Loop

Repeat loop is defined using the keyword repeat. The repeat loop block continuously executes the block for a given number of times. The number of times the loop executes can be mention using a constant or an expression. The expression is calculated only once, before the start of loop and not during the execution of the loop. If the expression value turns out to be z or x, then it is treated as zero, and hence loop block is not executed at all.

```
initial
begin
   a = 10;
   b = 5;
   b <= #10 10;
   i = 0;
   repeat(a*b)
   begin
      $display("repeat in progress");
      #1 i = i + 1;
   end
end
```

In the above example the loop block is executed only 50 times, and not 100 times. It calculates (a*b) at the beginning, and uses that value only.

While Loop

The while loop is defined using the keyword while. The while loop contains an expression. The loop continues until the expression is true. It terminates when the expression is false. If the calculated value of expression is z or x, it is treated as a false. The value of expression is calculated each time before starting the loop. All the statements (if more than one) are mentioned in blocks which begins and ends with keyword begin and end keywords.

```
initial
begin
   a = 20;
   i = 0;
   while (i < a)
   begin
   $display("%d",i);
   i = i + 1;
   a = a - 1;
   end
end
```

In the above example the loop executes for 10 times. (Observe that a is decrementing by one and i is incrementing by one, so loop terminated when both i and a become 10).

For Loop

The For loop is defined using the keyword for. The execution of for loop block is controlled by a three step process, as follows:

Executes an assignment, normally used to initialize a variable that controls the number of times the for block is executed.
Evaluates an expression, if the result is false or z or x, the for-loop shall terminate, and if it is true, the for-loop shall execute its block.
Executes an assignment normally used to modify the value of the loop-control variable and then repeats with second step.
Note that the first step is executed only once.

```
initial
begin
   a = 20;
   for (i = 0; i < a; i = i + 1, a = a - 1)
   $display("%d",i);
end
```

The above example produces the same result as the example used to illustrate the functionality of the while loop.

Examples:

1. Implementation of a 4x1 multiplexer.

```
module  mux4_1 (out, in0, in1, in2, in3, s0, s1);

output out;

// out is declared as reg, as default is wire

reg out;

// out is declared as reg, because we will
// do a procedural assignment to it.

input in0, in1, in2, in3, s0, s1;

// always @(*) is equivalent to
// always @( in0, in1, in2, in3, s0, s1 )

always @(*)
begin
  case ({s1,s0})
     2'b00: out = in0;
```

```
      2'b01: out = in1;
      2'b10: out = in2;
      2'b11: out = in3;
      default: out = 1'bx;
  endcase
end
endmodule
```

2. Implementation of a full adder.

```
module full_adder (sum, c_out, in0, in1, c_in);

output sum, c_out;
reg sum, c_out

input in0, in1, c_in;

always @(*)
  {c_out, sum} = in0 + in1 + c_in;

endmodule
```

3. Implementation of a 8-bit binary counter.

```
module ( count, reset, clk );

output [7:0] count;
reg [7:0] count;

input reset, clk;

// consider reset as active low signal

always @( posedge clk, negedge reset)
begin
  if(reset == 1'b0)
      count <= 8'h00;
  else
      count <= count + 8'h01;
end

endmodule
```

Implementation of a 8-bit counter is a very good example, which explains the advantage of behavioral modeling. Just imagine how difficult it will be implementing a 8-bit counter using gate-level modeling. In the above example the incrementation occurs on every positive edge of the clock. When count becomes 8'hFF, the next increment will make it 8'h00, hence there is no need of any modulus operator. Reset signal is active low.