DIGITAL DESIGN THROUGH VERILOG

LECTURE NOTES

B.TECH
(III YEAR – I SEM)
(2019-2020)

Prepared by:
Mrs. P. ANITHA, 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

MALLA REDDY COLLEGE OF ENGINEERING AND TECHNOLOGY

III Year B.Tech. ECE-I Sem L T/P/D C 4 1/ - /- 3 (R15A0410)

DIGITAL DESIGN THROUGH VERILOG

OBJECTIVE:
This course teaches designing digital circuits, behavior and RTL modeling of digital circuits using Verilog HDL, verifying these Models and synthesizing RTL models to standard cell libraries and FPGAs.
Students aim practical experience by designing, modeling, implementing and verifying several digital circuits.
This course aims to provide students with the understanding of the different technologies related to HDLs, construct, compile and execute Verilog HDL programs using provided

Software Tools: Design digital components and circuits that is testable, reusable, and synthesizable.

UNIT - I: Introduction to Verilog HDL: Verilog as HDL, Levels of Design Description, Concurrency, Simulation and Synthesis, Programming Language Interface, Module.
Language Constructs and Conventions: Introduction, Keywords, Identifiers, White Space, Characters, Comments, Numbers, Strings, Logic Values, Data Types, Scalars and Vectors, Operators.

UNIT - II: Gate Level Modeling: Introduction, AND Gate Primitive, Module Structure, Other Gate Primitives, Illustrative Examples, Tristate Gates, Array of Instances of Primitives, Design of Flip-Flops with Gate Primitives, Gate Delay, Strengths and Contention Resolution, Net Types.
Modeling at Dataflow Level: Introduction, Continuous Assignment Structure, Delays and Continuous Assignments, Assignment to Vector, Operators.

UNIT - III: Behavioral Modeling: Introduction, Operations and Assignments, 'Initial' Construct, Always construct, Assignments with Delays, 'Wait ‘Construct, Design at Behavioral Level, Blocking and Non-Blocking Assignments, The 'Case' Statement, 'If' and 'if-Else' Constructs, 'Assign- De-Assign' Constructs, 'Repeat' Construct, for loop, 'The Disable' Construct, 'While Loop', Forever Loop, sequential and Parallel Blocks.

UNIT - IV: Switch Level Modeling: Basic Transistor Switches, CMOS Switches, Bidirectional Gates, Time Delays with Switch Primitives, instantiation with strengths and delays, Switch level modeling for NAND, NOR and XOR.
UNIT - V: System Tasks, Functions and Compiler Directives: Parameters, Path Delays, Module Parameters, System Tasks and Functions, User Defined Primitives, Compiler directives.

Sequential Circuit Description: Sequential Models - Feedback Model, Capacitive Model, Implicit Model.

TEXT BOOKS:

1. T.R. Padmanabhan, B Bala Tripura Sundari, Design Through Verilog HDL, Wiley 2009.
2. Verilog HDL - Samir Palnitkar, 2nd Edition, Pearson Education, 2009.

REFERENCE BOOKS:

1. Fundamentals of Digital Logic with Verilog Design - Stephen Brown,Zvonkoc Vranesic, TMH, 2nd Edition.
2. Zainalabdien Navabi, Verliog Digital System Design, TMH, 2nd Edition.
3. Advanced Digital Logic Design using Verilog, State Machines & Synthesis for FPGA - Sunggu Lee, Cengage Learning, 2012.
4. Advanced Digital Design with Verilog HDL - Michel D. Ciletti, PHI, 2009.

OUTCOMES: By the end of the course student should be able to:

- Describe Verilog HDL
- Design Digital circuits
- Write behavior model of digital circuits
- Write RTL models of digital circuits
- Verify behavior and RTL models
- Describe standard Cell Libraries and FPGAs
- Synthesize RTL models to standard cell libraries and FPGAs
- Implement RTL models on FPGAs and testing and verification

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.

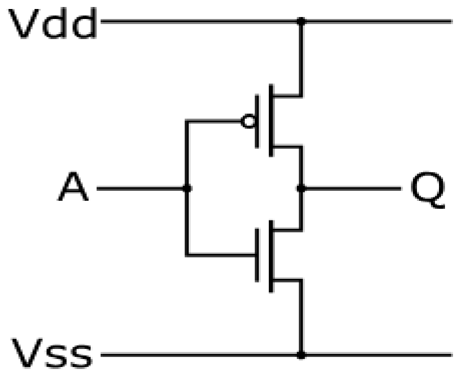


Figure 1 CMOS inverter

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.

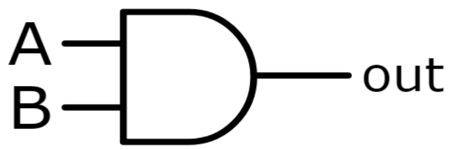


Figure 2 AND gate symbol

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").

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.

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 1. The terminals of the module are referred to as 'ports'. The ports attached to a module can be of three types:

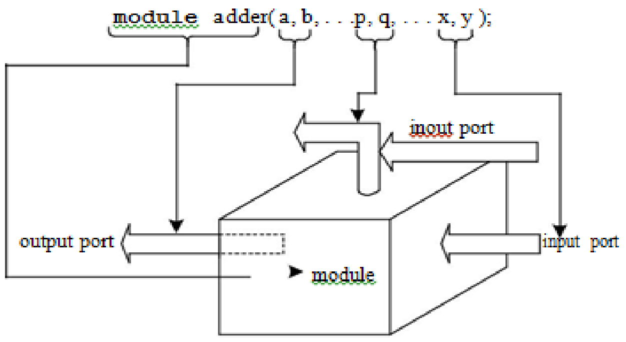


Figure 1 Representation of a module as black box with its ports.

- input ports through which 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; These are terminals through which signals are input to the module sometimes; at some other times signals are output from the module through these.

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.

All the constructs in Verilog are centered on the module. They define ways of building up, accessing, and using modules. The structure of modules and the mode of invoking them in a design are discussed here.

A module comprises a number of “lexical tokens” arranged according to some predefined order. The possible tokens are of seven categories:

- White spaces
- Comments
- Operators
- Numbers
- Strings
- Identifiers
- Keywords

The rules constraining the tokens and their sequencing will be dealt with as we progress. For the present let us consider modules. In Verilog any program which forms a design description is a “module.” Any program written to test a design description is also a “module.” The latter are often called as “stimulus modules” or “test benches.” A module used to do simulation has the form shown in Figure 2. 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.

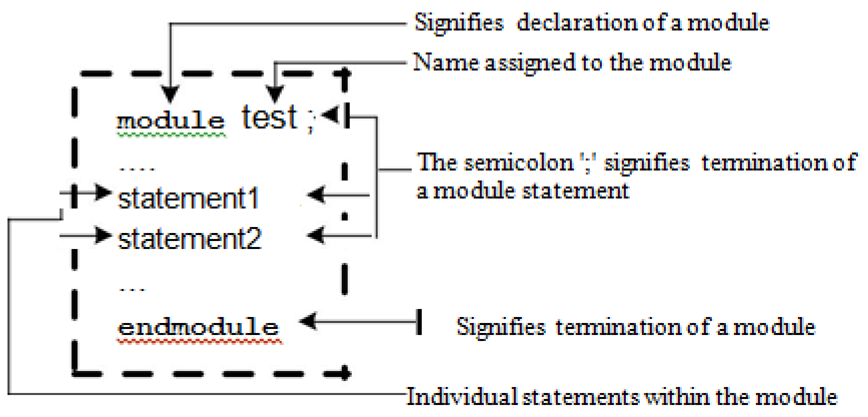

Signifies declaration of a module
Name assigned to the module
module test ;
....
statement1
statement2
...
endmodule
The semicolon ';' signifies termination of a module statement
Signifies termination of a module
Individual statements within the module

Figure 2 Structure of a typical simulation module.

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.

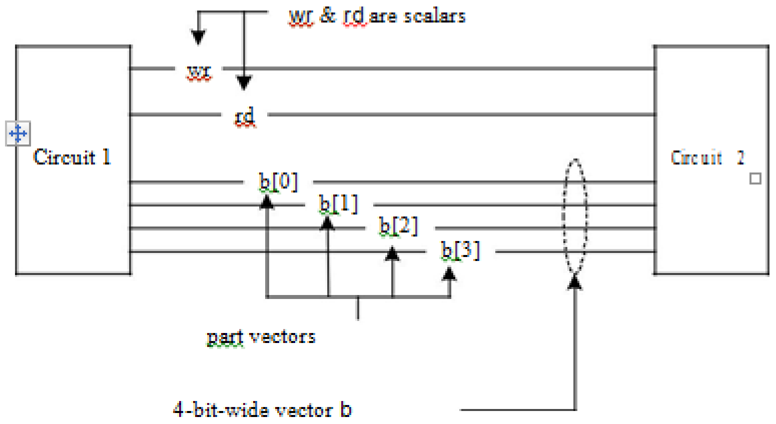


Figure Illustration of scalars and vectors.

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.

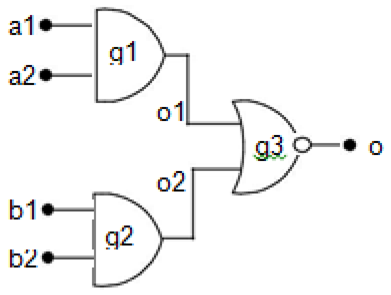


Figure for a typical A-O-I gate circuit.

```
/*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.

  When
  control = 1,
  out = in.

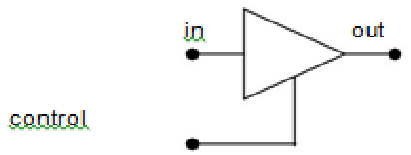
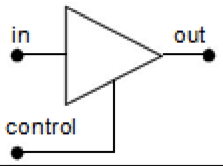
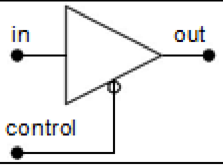
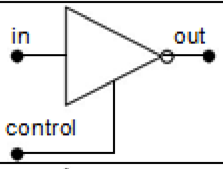
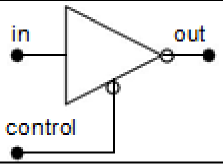


Figure 1 A tri-state buffer.

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

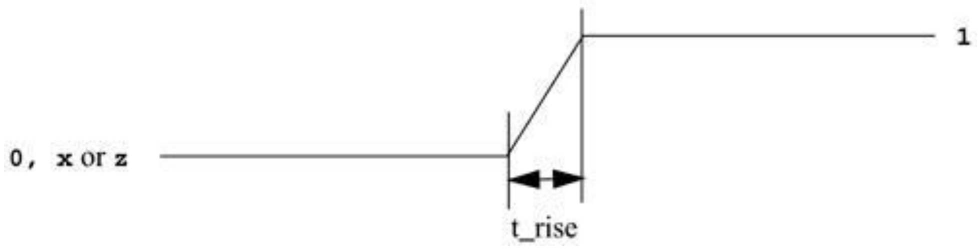
Until now, we described circuits without any delays (i.e., zero delay). In real circuits, logic gates have delays associated with them. Gate delays allow the Verilog user to specify delays through the logic circuits. Pin-to-pin delays can also be specified in Verilog.

Rise, Fall, and Turn-off Delays

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

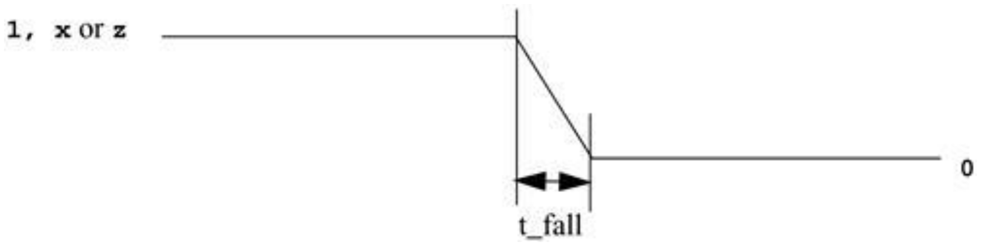
Rise delay

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.

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
```

Dataflow Modeling

Introduction

For small circuits, the gate-level modeling approach works very well because the number of gates is limited and the designer can instantiate and connect every gate individually. Also, gate-level modeling is very intuitive to a designer with a basic knowledge of digital logic design. However, in complex designs the number of gates is very large. Thus, designers can design more effectively if they concentrate on implementing the function at a level of abstraction higher than gate level. Dataflow modeling provides a powerful way to implement a design. Verilog allows a circuit to be designed in terms of the data flow between registers and how a design processes data rather than instantiation of individual gates. Later in this chapter, the benefits of dataflow modeling will become more apparent.

With gate densities on chips increasing rapidly, dataflow modeling has assumed great importance. No longer can companies devote engineering resources to handcrafting entire designs with gates. Currently, automated tools are used to create a gate-level circuit from a dataflow design description. This process is called logic synthesis. Dataflow modeling has become a popular design approach as logic synthesis tools have become sophisticated. This approach allows the designer to concentrate on optimizing the circuit in terms of data flow. For maximum flexibility in the design process, designers typically use a Verilog description style that combines the concepts of gate-level, data flow, and behavioral design. In the digital design community, the term RTL (Register Transfer Level) design is commonly used for a combination of dataflow modeling and behavioral modeling.

Continuous Assignments

A continuous assignment is the most basic statement in dataflow modeling, used to drive a value onto a net. This assignment replaces gates in the description of the circuit and describes the circuit at a higher level of abstraction. The assignment statement starts with the keyword assign. The syntax of an assign statement is as follows.

```
continuous_assign ::= assign [ drive_strength ] [ delay3 ]
                              list_of_net_assignments ;

list_of_net_assignments ::= net_assignment { , net_assignment }
net_assignment ::= net_lvalue = expression
```

Notice that drive strength is optional and can be specified in terms of strength levels The default value for drive strength is strong1 and strong0. The delay value is also optional and can be used to specify delay on the assign statement. This is like specifying delays for gates. Delay specification is discussed in this chapter. Continuous assignments have the following characteristics:

1. The left hand side of an assignment must always be a scalar or vector net or a concatenation of scalar and vector nets. It cannot be a scalar or vector register.
2. Continuous assignments are always active. The assignment expression is evaluated as soon as one of the right-hand-side operands changes and the value is assigned to the left-hand-side net.
3. The operands on the right-hand side can be registers or nets or function calls. Registers or nets can be scalars or vectors.

4. Delay values can be specified for assignments in terms of time units. Delay values are used to control the time when a net is assigned the evaluated value. This feature is similar to specifying delays for gates. It is very useful in modeling timing behavior in real circuits.

Examples of Continuous Assignment

Continuous assign. out is a net. i1 and i2 are nets. assign out = i1

& i2;

Continuous assign for vector nets. addr is a 16-bit vector net

addr1 and addr2 are 16-bit vector registers.

assign addr[15:0] = addr1_bits[15:0] ^ addr2_bits[15:0];

Concatenation. Left-hand side is a concatenation of a scalar

net and a vector net.

```
assign {c_out, sum[3:0]} = a[3:0] + b[3:0] + c_in;
```

Implicit Continuous Assignment

Instead of declaring a net and then writing a continuous assignment on the net, Verilog provides a shortcut by which a continuous assignment can be placed on a net when it is declared. There can be only one implicit declaration assignment per net because a net is declared only once.

In the example below, an implicit continuous assignment is contrasted with a regular continuous assignment.

```
//Regular continuous assignment

wire out;

assign out = in1 & in2;


//Same effect is achieved by an implicit continuous assignment wire out =
in1 & in2;
```

Implicit Net Declaration

If a signal name is used to the left of the continuous assignment, an implicit net declaration will be inferred for that signal name. If the net is connected to a module port, the width of the inferred net is equal to the width of the module port.

```
wire i1, i2;

assign out = i1 & i2; //Note that out was not declared as a wire

                          //but an implicit wire declaration for out //is done
                          by the simulator
```

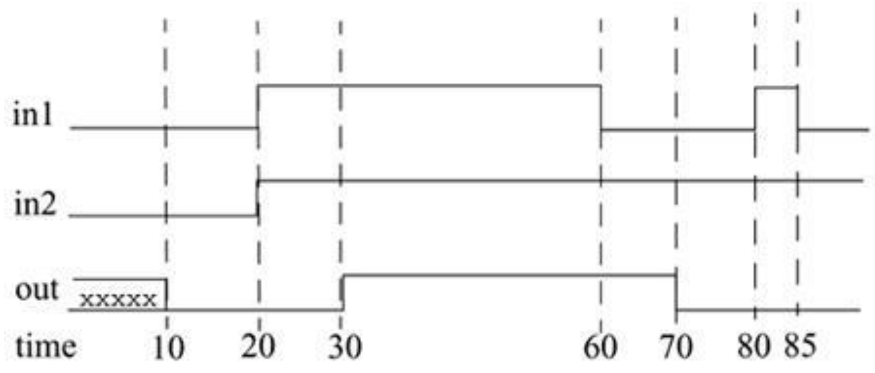
Delays

Delay values control the time between the change in a right-hand-side operand and when the new value is assigned to the left-hand side. Three ways of specifying delays in continuous assignment statements are regular assignment delay, implicit continuous assignment delay, and net declaration delay.

Regular Assignment Delay

The first method is to assign a delay value in a continuous assignment statement. The delay value is specified after the keyword assign. Any change in values of in1 or in2 will result in a delay of 10 time units before recomputation of the expression in1 & in2, and the result will be assigned to out. If in1 or in2 changes value again before 10 time units when the result propagates to out, the values of in1 and in2 at the time of recomputation are considered. This property is called inertial delay. An input pulse that is shorter than the delay of the assignment statement does not propagate to the output.

assign #10 out = in1 & in2; // Delay in a continuous assign



\

Figure: Delays

The above waveform is generated by simulating the above assign statement. It shows the delay on signal out. Note the following change:

When signals in1 and in2 go high at time 20, out goes to a high 10 time units later (time = 30).

When in1 goes low at 60, out changes to low at 70.

However, in1 changes to high at 80, but it goes down to low before 10 time units have elapsed.

Hence, at the time of recomputation, 10 units after time 80, in1 is 0. Thus, out gets the value 0. A pulse of width less than the specified assignment delay is not propagated to the output.

Implicit Continuous Assignment Delay

An equivalent method is to use an implicit continuous assignment to specify both a delay and an assignment on the net.

```
//implicit continuous assignment delay

wire #10 out = in1 & in2;


//same as

wire out;

assign #10 out = in1 & in2;
```

The declaration above has the same effect as defining a wire out and declaring a continuous assignment on out.

Net Declaration Delay

A delay can be specified on a net when it is declared without putting a continuous assignment on the net. If a delay is specified on a net out, then any value change applied to the net out is delayed accordingly. Net declaration delays can also be used in gate-level modeling.

```
//Net Delays

wire # 10 out;

assign out = in1 & in2;


//The above statement has the same effect as the following.
wire out;

assign #10 out = in1 & in2;
```

Expressions, Operators, and Operands

Dataflow modeling describes the design in terms of expressions instead of primitive gates. Expressions, operators, and operands form the basis of dataflow modeling.

Expressions

Expressions are constructs that combine operators and operands to produce a result.

Examples of expressions. Combines operands and operators a ^ b

```
addr1[20:17] + addr2[20:17] in1 | in2 ;
```

Operands

Some constructs will take only certain types of operands. Operands can be constants, integers, real numbers, nets, registers, times, bit-select (one bit of vector net or a vector register), part-select (selected bits of the vector net or register vector), and memories or function calls (functions are discussed later).

```
integer count, final_count;

final_count = count + 1;//count is an integer operand

real a, b, c;

c = a - b; //a and b are real operands

reg [15:0] reg1, reg2;

reg [3:0] reg_out;

reg_out = reg1[3:0] ^ reg2[3:0];//reg1[3:0] and reg2[3:0] are //part-select
                                             register operands

reg ret_value;

ret_value = calculate_parity(A, B);//calculate_parity is a //function
                                             type operand
```

Operators

Operators act on the operands to produce desired results. Verilog provides various types of operators.

```
d1 && d2 // && is an operator on operands d1 and d2 !a[0]
// ! is an operator on operand a[0]

B >> 1 // >> is an operator on operands B and 1
```

Operator Types

Verilog provides many different operator types. Operators can be arithmetic, logical, relational, equality, bitwise, reduction, shift, concatenation, or conditional. Some of these operators are similar to the operators used in the C programming language. Each operator type is denoted by a symbol. The following table shows the complete listing of operator symbols classified by category.

Table: Operator Types and Symbols

Operator Type	**Operator Symbol**	**Operation Performed**	**Number of Operands**
Arithmetic	*	multiply	two
	/	divide	two
	+	add	two
	-	subtract	two
	%	modulus	two
	**	power (exponent)	two
Logical	!	logical negation	one
	&&	logical and	two
	||	logical or	two
	>	greater than	two

Relational	<	less than	two
	>=	greater than or equal	two
	<=	less than or equal	two
Equality	==	equality	two
	!=	inequality	two
	===	case equality	two
	!==	case inequality	two

Bitwise	~	bitwise negation	one
	&	bitwise and	two
	|	bitwise or	two
	^	bitwise xor	two
	^~ or ~^	bitwise xnor	two
Reduction	&	reduction and	one
	~&	reduction nand	one
	|	reduction or	one
	~|	reduction nor	one
	^	reduction xor	one
	^~ or ~^	reduction xnor	one

Shift	>>	Right shift	Two
	<<	Left shift	Two
	>>>	Arithmetic right shift	Two
	<<<	Arithmetic left shift	Two
Concatenation	{ }	Concatenation	Any number
Replication	{ { } }	Replication	Any number
Conditional	?:	Conditional	Three

Let us now discuss each operator type in detail.

Arithmetic Operators

There are two types of arithmetic operators: binary and unary.

Binary operators

Binary arithmetic operators are multiply (*), divide (/), add (+), subtract (-), power (**), and modulus (%). Binary operators take two operands.

A = 4'b0011; B = 4'b0100; // A and B are register vectors D = 6; E =

4; F=2// D and E are integers

A * B // Multiply A and B. Evaluates to 4'b1100

D / E // Divide D by E. Evaluates to 1. Truncates any fractional part. A + B // Add A and B. Evaluates to 4'b0111

B - A // Subtract A from B. Evaluates to 4'b0001 F = E **
F; //E to the power F, yields 16

If any operand bit has a value x, then the result of the entire expression is x. This seems intuitive because if an operand value is not known precisely, the result should be an unknown.

in1 = 4'b101x;

in2 = 4'b1010;

sum = in1 + in2; // sum will be evaluated to the value 4'bx

Modulus operators produce the remainder from the division of two numbers. They operate similarly to the modulus operator in the C programming language.

```
 13  % 3 // Evaluates to 1

 16  % 4 // Evaluates to 0

  -7  % 2 // Evaluates      to   -1, takes sign of the first operand

   7 % -2 // Evaluates      to   +1, takes sign of the first operand
```

Unary operators

The operators + and - can also work as unary operators. They are used to specify the positive or negative sign of the operand. Unary + or ? operators have higher precedence than the binary + or ? operators.

-4 // Negative 4

+5 // Positive 5

Negative numbers are represented as 2's complement internally in Verilog. It is advisable to use negative numbers only of the type integer or real in expressions. Designers should avoid negative numbers of the type <sss> '<base> <nnn> in expressions because they are converted to unsigned 2's complement numbers and hence yield unexpected results.

//Advisable to use integer or real numbers -10 /

5// Evaluates to -2

//Do not use numbers of type <sss> '<base> <nnn>

-'d10 / 5// Is equivalent (2's complement of 10)/5 = (232 - 10)/5

where 32 is the default machine word width.

This evaluates to an incorrect and unexpected result

Logical Operators

Logical operators are logical-and (&&), logical-or (||) and logical- not (!). Operators && and || are binary operators. Operator ! is a unary operator. Logical operators follow these conditions:

Logical operators always evaluate to a 1-bit value, 0 (false), 1 (true), or x ambiguous).If an operand is not equal to zero, it is equivalent to a logical 1 (true condition). If it is 01equal to zero, it is equivalent to a logical 0 (false condition). If any operand bit is x or z, it is equivalent to x (ambiguous condition) and is normally treated by simulators as a false condition.Logical operators take variables or expressions as operands.Use of parentheses to group logical operations is highly recommended to improve readability. Also, the user does not have to remember the precedence of operators.

Logical operations A = 3;
B = 0;

A && B // Evaluates to 0. Equivalent to (logical-1 && logical-0) A || B // Evaluates to 1. Equivalent to (logical-1 || logical-0) !A// Evaluates to 0. Equivalent to not(logical-1)

!B// Evaluates to 1. Equivalent to not(logical-0)

Unknowns

A = 2'b0x; B = 2'b10;

A && B // Evaluates to x. Equivalent to (x && logical 1)

```
// Expressions

(a == 2) && (b == 3) // Evaluates to 1 if both a == 2 and b == 3 are true.
 // Evaluates to 0 if either is false.
```

Relational Operators

Relational operators are greater-than (>), less-than (<), greater-than-or-equal-to (>=), and less-than-or-equal-to (<=). If relational operators are used in an expression, the expression returns a logical value of 1 if the expression is true and 0 if the expression is false. If there are any unknown or z bits in the operands, the expression takes a value x. These operators function exactly as the corresponding operators in the C programming language.

```
    A = 4, B = 3

    X = 4'b1010, Y = 4'b1101, Z = 4'b1xxx

A <= B // Evaluates to a logical 0

A > B // Evaluates to a logical 1

Y >= X // Evaluates to a logical 1

Y < Z // Evaluates to an x
```

Equality Operators

Equality operators are logical equality (==), logical inequality (!=), case equality (===), and case inequality (!==) . When used in an expression, equality operators return logical value 1 if true, 0 if false. These operators compare the two operands bit by bit, with zero filling if the operands are of unequal length. Table below lists the operators.

It is important to note the difference between the logical equality operators (==, !=) and case equality operators (===, !==). The logical equality operators (==, !=) will yield an x if either operand has x or z in its bits. However, the case equality operators (===, !==) compare both operands bit by bit and compare all bits, including x and z. The result is 1 if the operands match exactly, including x and z bits. The result is 0 if the operands do not match exactly. Case equality operators never result in an x.

Table: Equality Operators

Expression	**Description**	**Possible Logical Value**
a == b	a equal to b, result unknown if x or z in a or b	0, 1, x
a != b	a not equal to b, result unknown if x or z in a or B	0, 1, x
a === b	a equal to b, including x and z	0, 1
a !== b	a not equal to b, including x and z	0, 1

A = 4, B = 3

X = 4'b1010, Y = 4'b1101

Z = 4'b1xxz, M = 4'b1xxz, N = 4'b1xxx

A == B // Results in logical 0

X != Y // Results in logical 1

X == Z // Results in x

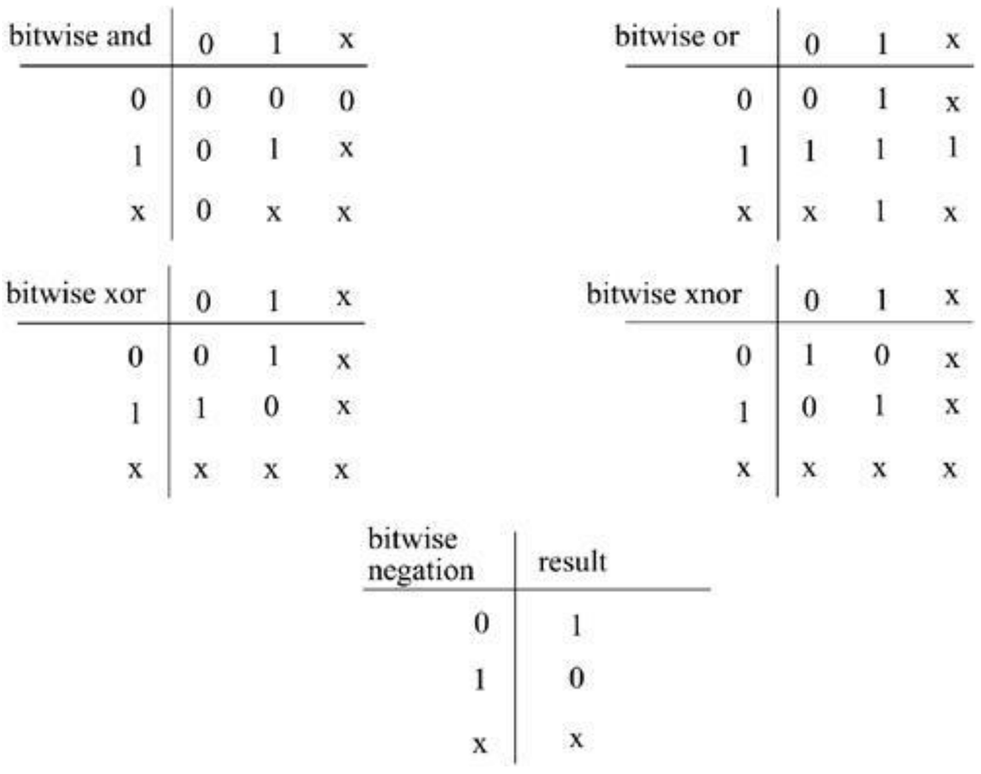
Z === M // Results in logical 1 (all bits match, including x and z)

Z === N // Results in logical 0 (least significant bit does not match) M !== N // Results in logical 1

Bitwise Operators

Bitwise operators are negation (~), and(&), or (|), xor (^), xnor (^~, ~^). Bitwise operators perform a bit-by-bit operation on two operands. They take each bit in one operand and perform the operation with the corresponding bit in the other operand. If one operand is shorter than the other, it will be bit-extended with zeros to match the length of the longer operand. Logic tables for the bit-by-bit computation are shown in Table. A z is treated as an x in a bitwise operation. The exception is the unary negation operator (~), which takes only one operand and operates on the bits of the single operand.

Table: Truth Tables for Bitwise Operators



bitwise and	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

bitwise or	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

bitwise xor	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

bitwise xnor	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

bitwise negation	result
0	1
1	0
x	x

Examples of bitwise operators are shown below.

```
   X = 4'b1010, Y = 4'b1101

   Z = 4'b10x1

~X       // Negation. Result is 4'b0101

X & Y    // Bitwise and. Result is 4'b1000

X | Y    // Bitwise or. Result is 4'b1111
```

```
X ^ Y     // Bitwise xor. Result is 4'b0111

X ^~ Y // Bitwise xnor. Result is 4'b1000

X & Z     // Result is 4'b10x0
```

It is important to distinguish bitwise operators ~, &, and | from logical operators !, &&, ||. Logical operators always yield a logical value 0, 1, x, whereas bitwise operators yield a bit-by-bit value. Logical operators perform a logical operation, not a bit-by-bit operation.

```
// X = 4'b1010, Y = 4'b0000

X | Y // bitwise operation. Result is 4'b1010

X || Y // logical operation. Equivalent to 1 || 0. Result is 1.
```

Reduction Operators

Reduction operators are and (&), nand (~&), or (|), nor (~|), xor (^), and xnor (~^, ^~). Reduction operators take only one operand. Reduction operators perform a bitwise operation on a single vector operand and yield a 1-bit result. The difference is that bitwise operations are on bits from two different operands, whereas reduction operations are on the bits of the same operand. Reduction operators work bit by bit from right to left. Reduction nand, reduction nor, and reduction xnor are computed by inverting the result of the reduction and, reduction or, and reduction xor, respectively.

```
// X = 4'b1010

&X //Equivalent to 1 & 0 & 1 & 0. Results in 1'b0
|X//Equivalent to 1 | 0 | 1 | 0. Results in 1'b1
^X//Equivalent to 1 ^ 0 ^ 1 ^ 0. Results in 1'b0

//A reduction xor or xnor can be used for even or odd parity
//generation of a vector.
```

The use of a similar set of symbols for logical (!, &&, ||), bitwise (~, &, |, ^), and reduction operators (&, |, ^) is somewhat confusing initially. The difference lies in the number of operands each operator takes and also the value of results computed.

Shift Operators

Shift operators are right shift (>>), left shift (<<), arithmetic right shift (>>>), and arithmetic left shift (<<<). Regular shift operators shift a vector operand to the right or the left by a specified number of bits. The operands are the vector and the number of bits to shift. When the bits are shifted, the vacant bit positions are filled with zeros. Shift operations do not wrap around. Arithmetic shift operators use the context of the expression to determine the value with which to fill the vacated bits.

```
// X = 4'b1100

Y = X >> 1; //Y is 4'b0110. Shift right 1 bit. 0 filled in MSB position.

Y = X << 1; //Y is 4'b1000. Shift left 1 bit. 0 filled in LSB position.

Y = X << 2; //Y is 4'b0000. Shift left 2 bits.

integer a, b, c; //Signed data types

a = 0;

b = -10; // 00111...10110 binary

c = a + (b >>> 3); //Results in -2 decimal, due to arithmetic shift
```

Shift operators are useful because they allow the designer to model shift operations, shift-and-add algorithms for multiplication, and other useful operations.

Concatenation Operator

The concatenation operator ({, }) provides a mechanism to append multiple operands. The operands must be sized. Unsized operands are not allowed because the size of each operand must be known for computation of the size of the result. Concatenations are expressed as operands within braces, with commas separating the operands. Operands can be scalar nets or registers, vector nets or registers, bit-select, part-select, or sized constants.

```
// A =    1'b1, B     = 2'b00, C = 2'b10, D =           3'b110
Y = {B    , C} //       Result Y     is 4'b0010
Y =   {A  , B , C       , D , 3'b001} // Result           Y is 11'b10010110001
Y =   {A  , B[0],      C[1]} //      Result Y is 3'b101
```

Replication Operator

Repetitive concatenation of the same number can be expressed by using a replication constant. A replication constant specifies how many times to replicate the number inside the brackets ({ }).

```
reg A;

reg [1:0] B, C;

reg [2:0] D;

A = 1'b1; B = 2'b00; C = 2'b10; D = 3'b110;


Y = { 4{A} } // Result Y is 4'b1111

Y = { 4{A} , 2{B} } // Result Y is 8'b11110000

Y = { 4{A} , 2{B} , C } // Result Y is 8'b1111000010
```

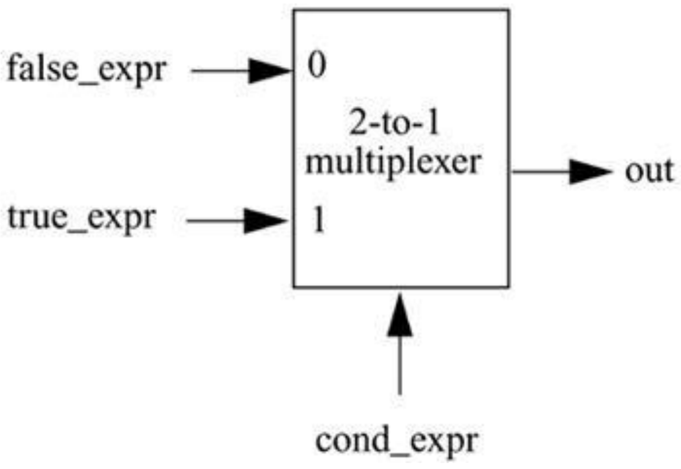
Conditional Operator

The conditional operator(?:) takes three operands.

Usage: condition_expr ? true_expr : false_expr ;

The condition expression (condition_expr) is first evaluated. If the result is true (logical 1), then the true_expr is evaluated. If the result is false (logical 0), then the false_expr is evaluated. If the result is x (ambiguous), then both true_expr and false_ expr are evaluated and their results are compared, bit by bit, to return for each bit position an x if the bits are different and the value of the bits if they are the same.

The action of a conditional operator is similar to a multiplexer. Alternately, it can be compared to the if-else expression.



Conditional operators are frequently used in dataflow modeling to model conditional assignments. The conditional expression acts as a switching control.

```
//model functionality of a tristate buffer

assign addr_bus = drive_enable ? addr_out : 36'bz;


//model functionality of a 2-to-1 mux

assign out = control ? in1 : in0;
```

Conditional operations can be nested. Each true_expr or false_expr can itself be a conditional operation. In the example that follows, convince yourself that (A==3) and control are the two select signals of 4-to-1 multiplexer with n, m, y, x as the inputs and out as the output signal.

```
assign out = (A == 3) ? ( control ? x : y ): ( control ? m : n) ;
```

Operator Precedence

Having discussed the operators, it is now important to discuss operator precedence. If no parentheses are used to separate parts of expressions, Verilog enforces the following precedence. Operators listed in Table are in order from highest precedence to lowest precedence. It is recommended that parentheses be used to separate expressions except in case of unary operators or when there is no ambiguity.

Table: Operator Precedence

Operators	**Operator Symbols**	**Precedence**
Unary	+ - ! ~	Highest precedence
Multiply, Divide, Modulus	* / %	
Add, Subtract	+ -	
Shift	<< >>	
Relational	< <= > >=	
Equality	== != === !==	
Reduction	&, ~& ^ ^~ |, ~|	
Logical	&& ||	
Conditional	?:	Lowest precedence

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 a designer need 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 are used for implementing the combination logic.

OPERATIONS AND ASSIGNMENTS:

The design description at the behavioral level is done through a sequence of assignments. These are called 'procedural assignments' – in contrast to the continuous assignments at the data flow level. Though it appears similar to the assignments at the data flow level discussed in the last chapter, the two are different. The procedure assignment is characterized by the following:

- The assignment is done through the "=" symbol (or the "<=" symbol) as was the case with the continuous assignment earlier.
- An operation is carried out and the result assigned through the "=" operator to an operand specified on the left side of the "=" sign – for example,N = ~N;
- Here the content of reg N is complemented and assigned to the reg N itself. The assignment is essentially an updating activity.
- The operation on the right can involve operands and operators. The operands can be of different types – logical variables, numbers – real or integer and so on.

wait CONSTRUCT

The wait construct makes the simulator wait for the specified expression to be true before proceeding with the following assignment or group of assignments. Its syntax has the form

wait (alpha) assignment1;

alpha can be a variable, the value on a net, or an expression involving them. If alpha is an expression, it is evaluated; if true, assignment1 is carried out. One can also have a group of assignments within a block in place of assignment1.

Example:

```
wait(clk) #2 a = b;
```

The simulator waits for the clock to be high and then assigns b to a with a delay of 2 ns. The assignment will be refreshed as long as the clk remains high.

The below code shows one version of the up-down counter module along with a test bench. It is a modification of the up down counter uses a wait construct. It has an enable input En. The counter is active and counts only when En = 1.

Verilog code:

```
module ctr_wt(a,clk,N,En);
input clk,En;
input[3:0]N;
output[3:0]a;
reg[3:0]a;
initial a=4'b1111;
always
begin
wait(En)
@(negedge clk)
a=(a==N)?4'b0000:a+1'b1;
end
endmodule
```

Test Bench

```
module tst_ctr_wt;
reg clk,En;
reg[3:0]N;
wire[3:0]a;
ctr_wt c1(a,clk,N,En);
initial
begin
clk=0;N=4'b1111;En=1'b0;#5 En=1'b1;#20 En=1'b0;
end
always
#2 clk=~clk;
initial #35 $stop;
initial $monitor($time,"clk=%h,En=%b,N=%b,a=%b",clk,En,N,a,);
 endmodule
```

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. Whereas 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.

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.

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;).

UNIT-4
SWITCH LEVEL MODELING

Introduction

In today's environment the MOS transistor is the basic element around which a VLSI is built. Designers familiar with logic gates and their configurations at the circuit level may choose to do their designs using MOS transistors.
Verilog has the provision to do the design description at the switch level using such MOS transistors.
Switch level modeling forms the basic level of modeling digital circuits.
The switches are available as primitives in Verilog; they are central to design description at this level. Basic gates can be defined in terms of such switches. By repeated and successive instantiation of such switches, more involved circuits can be modeled – on the same lines as was done with the gate level primitives.

BASIC TRANSISTOR SWITCHES

Consider an NMOS transistor of the depletion type. When used in a digital circuit, it can be in one of three modes:

- VG < VS where VG and VS are the gate and source voltages with respect to the drain: The transistor is OFF and offers very high impedance across the source and the drain. It is in the z state.
- VG = VS: The transistor is in the active region. It presents a resistance between the source and the drain. The value depends on the technology. Such a resistive state of the transistor can be modeled in Verilog. A transistor in this mode can be represented as a resistance in Verilog – as pull1 or pull0 depending on whether the drain is connected to supply1 or source is connected to supply0.
- VG > VS: The transistor is fully turned on. It presents very low resistance between the source and drain.
- An enhanced-mode NMOS transistor also has the above three modes of operation.
- It is OFF when VG = VS. It is moderately ON or in the active region when VG is slightly greater than VS, representing a resistive (pull1 or pull0) mode of operation. When VG is sufficiently greater than VS, the transistor is in the on state representing very low resistance. Similar modes are possible for the PMOS transistor also.

Basic Switch Primitive

Different switch primitives are available in Verilog.
Consider an nmos switch. A typical instantiation has the form

nmos (out, in, control);

nmos – a keyword – represents an NMOS transistor functioning as a switch.
The switch has three terminals – in, out, and control.
NMOS transistor symbol shown in below figure 1 with three terminals-

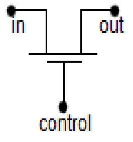


Figure 1 An NMOS switch with terminals

When the control input is at 1 (high) state, the switch is on. It connects the input lead to the output side and offers zero impedance.
When the control input is low, the switch is OFF and output is left floating (z state).
If the control is in the z or the x state, output may take corresponding values.

The keyword pmos represents a PMOS transistor functioning as a switch.
The PMOS switch has three terminals (see Figure 2).
A typical instantiation of the switch has the form

```
pmos (out, in, control);
```

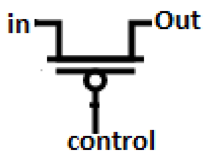


fig. 2 Pmos with 3-terminals

When the control is at 1 (high) state, the switch is off. Output is left floating.
When control is at 0 (low) state, the switch is on, input is connected to output, and output is at the same state as input.

Resistive Switches

nmos and pmos represent switches of low impedance in the on-state. rnmos and rpmos represent the resistive counterparts of these respectively. Typical instantiations have the form

```
rnmos (output1, input1, control1);
rpmos (output2, input2, control2);
```

The rnmos if the control1 input is at 1 (high) state, the switch is ON and functions as a definite resistance. It connects input1 to output1 through a resistance. When control1 is at the 0 (low) state, the switch is OFF and leaves output1 floating.
The rpmos switch is ON when control2 is at 0 (low) state. It inserts a definite resistance between the input and the output signals but retains the signal value.

rpmos and rnmos are resistive switches, they reduce the signal strength when in the on state. The reduced strength is mostly one level below the original strength.
The rpmos and rnmos switches function as unidirectional switches; the signal flow is from the input to the output side.

pullup and pulldown

A MOS transistor functions as a resistive element when in the active state. Realization of resistance in this form takes less silicon area in the IC as compared to a resistance realized directly. pullup and pulldown represent such resistive elements.

A typical instantiation here has the form

```
pullup (x);
```

Here the net x is pulled up to the supply1 through a resistance. Similarly, the instantiation

```
pulldown (y);
```

pulls y down to the supply0 level through a resistance. The pullup and pulldown primitives can be used as loads for switches or to connect the unused input ports to VCC or GND, respectively. They can also form loads of switches in logic circuits.

The default strengths for pullup and pulldown are pull1 and pull0 respectively. One can also specify strength values for the respective nets. For example,

```
pullup (strong1) (x)
```

specifies a resistive pullup of net x to supply1. One can also assign names to the pullup and pulldown primitives. Thus

```
pullup (strong1) rs(x)
```

represents an instantiation of pullup designated rs having strength strong1.

CMOS SWITCH

A CMOS switch is formed by connecting a PMOS and an NMOS switch in parallel – the input leads are connected together on the one side and the output leads are connected together on the other side. Figure 10.15 shows the switch so formed. It has two control inputs:

- N_control turns ON the NMOS transistor and keeps it ON when it is in the 1 state.
- P_control turns ON the PMOS transistor and keeps it ON when it is in the 0 state.

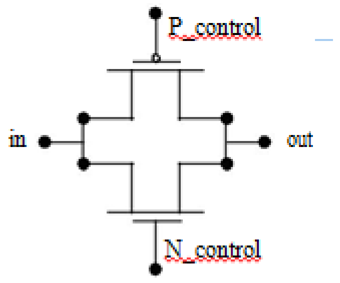


Figure 10.15 A CMOS switch formed by connecting a PMOS transistor and an NMOS transistor in parallel.

The CMOS switch is instantiated as shown below.

```
cmos csw (out, in, N_control, P_control );
```

Significance of the different terms is as follows:
cmos: The keyword for the switch instantiation
csw: Name assigned to the switch in the instantiation
out: Name assigned to the output variable in the instantiation
in: Name assigned to the input variable in the instantiation
N_control: Name assigned to the control variable of the NMOS transistor in the instantiation
P_control: Name assigned to the control variable of the PMOS transistor in the instantiation

BI-DIRECTIONAL GATES

Verilog has a set of primitives for bi-directional switches as well. They connect the nets on either side when ON and isolate them when OFF. The signal flow can be in either direction. None of the continuous-type assignments at higher levels dealt with so far has a functionality equivalent to the bi-directional gates. There are six types of bi-directional gates.

- tran rtran
- tranif1 rtanif1
- tranif0 rtranif0

tran and rtran

The tran gate is a bi-directional gate of two ports. When instantiated, it connects the two ports directly. Thus the instantiation

```
tran (s1, s2);
```

connects the signal lines s1 and s2. Either line can be input, inout or output.
rtran is the resistive counterpart of tran.

tranif1 and rtranif1

tranif1 is a bi-directional switch turned ON/OFF through a control line. It is in the ON-state when the control signal is at 1 (high) state. When the control line is at state 0 (low), the switch is in the OFF state. A typical instantiation has the form

```
tranif1 (s1, s2, c );
```

Here c is the control line. If c=1, s1 and s2 are connected and signal transmission can be in either direction.
rtranif1 is the resistive counterpart of tranif1. It is instantiated in an identical manner.

tranif0 and rtranif0

tranif0 and rtranif0 are again bi-directional switches. The switch is OFF if the control line is in the 1 (high) state, and it is ON when the control line is in the 0 (low) state. A typical instantiation has the form

```
tranif0 (s1, s2, c);
```

With the above instantiation, if c = 0, s1 and s2 are connected and signal transmission can be in either direction. If c = 1, the switch is OFF and s1 and s2 are isolated from each other.
rtranif0 is the resistive counterpart of tranif0.

INSTANTIATIONS WITH STRENGTHS AND DELAYS

Propagation delays can be specified for switch primitives on the same lines as was done with the gate primitives in Chapter 5. For example, an NMOS switch instantiated as

```
nmos g1 (out, in, ctrl );
```

has no delay associated with it. The instantiation

```
nmos (delay1) g2 (out, in, ctrl );
```

has delay1 as the delay for the output to rise, fall, and turn OFF. The instantiation

```
nmos (delay_r, delay_f) g3 (out, in, ctrl );
```

has delay_r as the rise-time for the output. delay_f is the fall-time for the output. The turn-off time is zero. The instantiation

```
nmos (delay_r, delay_f, delay_o) g4 (out, in, ctrl );
```

has delay_r as the rise-time for the output. delay_f is the fall-time for the output delay_o is the time to turn OFF when the control signal ctrl goes from 0 to 1. Delays can be assigned to the other uni-directional gates (rcmos, pmos, rpmos, cmos, and rcmos) in a similar manner. Bi-directional switches do not delay transmission – their rise- and fall-times are zero. They can have only turn-on and turn-off delays associated with them. tran has no delay associated with it.

```
tranif1 (delay_r, delay_f) g5 (out, in, ctrl );
```
represents an instantiation of the controlled bi-directional switch. When control changes from 0 to 1, the switch turns on with a delay of delay_r. When control changes from 1 to 0, the switch turns off with a delay of delay_f.

```
transif1 (delay0) g2 (out, in, ctrl );
```

represents an instantiation with delay0 as the delay for the switch to turn on when control changes from 0 to 1, with the same delay for it to turn off when control changes from 1 to 0. When a delay value is not specified in an instantiation, the turn-on and turn-off are considered to be ideal that is, instantaneous. Delay values similar to the above illustrations can be associated with rtranif1, tranif0, and rtranif0 as well.

INSTANTIATIONS WITH STRENGTHS AND DELAYS

In the most general form of instantiation, strength values and delay values can be combined. For example, the instantiation

```
nmos (strong1, strong0) (delay_r, delay_f, delay_o ) gg (s1, s2, ctrl) ;
```

means the following:

- It has strength strong0 when in the low state and strength strong1when in the high state.
- When output changes state from low to high, it has a delay time of delay_r.
- When the output changes state from high to low, it has a delay time of delay_f.
- When output turns-off it has a turn-off delay time of delay _o.

rnmos, pmos, and rpmos switches too can be instantiated in the general form in the same manner. The general instantiation for the bi-directional gates too can be done similarly.

UNIT-5
SYSTEM TASKS, FUNCTIONS, AND COMPILER DIRECTIVES

A number of facilities in Verilog relate to the management of simulation; starting and stopping of simulation, selectively monitoring the activities, testing the design for timing constraints, etc., are amongst them. Although a variety of such constructs is available in Verilog.

PARAMETERS

Verilog defines parameter as a constant value that is declared within structure of module. The constant value signifies timing values, range of variables, wires e.t.c.
The parameter values can be specified and changed to suit the design environment or test environment. Such changes are effected and frozen at instantiation.
The assigned values cannot change during testing or synthesis.

Two types of parameters are of use in modules: specparam and defparam.
Specparam : Parameters related to timings, time delays, rise and fall times, etc., are technology-specific and used during simulation. Parameter values can be assigned or overridden with the keyword "specparam" preceding the assignments.
Defparam: Parameters related to design, bus width, and register size are of a different category. They are related to the size or dimension of a specific design; they are technology-independent. Assignment or overriding is with assignments following the keyword "defparam".

Timing-Related Parameters

The constructs associated with parameters are discussed here through specific design or test modules.
Example: Module of a half-adder with delays assigned to the output transitions; a test bench is also included in the figure.

```
module ha_1(s,ca,a,b);
 input a,b; output s,ca;
 xor #(1,2) (s,a,b);
and #(3,4) (ca,a,b);
endmodule

//test-bench
module tstha;
 reg a,b; wire s,ca;
ha_1 hh(s,ca,a,b);
initial
begin
```

```
a=0;b=0;
end
always
begin
#5 a=1;b=0;
#5 a=0;b=1;
#5 a=1;b=1;
 #5 a=0;b=0;
 end
initial $monitor($time , " a = %b , b = %b ,out carry = %b , outsum = %b ",a,b,ca,s);
initial #30 $stop;
endmodule
```

Parameter Declarations and Assignments

Declaration of parameters in a design as well as assignments to them can be effected using the keyword “Parameter.” A declaration has the form

```
parameter alpha = a, beta = b;
```

Where

- parameter is the keyword,
- alpha and beta are the names assigned to two parameters and
- a, b are values assigned to alpha and beta, respectively.

In general a and b can be constant expressions. The parameter values can be overridden during instantiation but cannot be changed during the run-time. If a parameter assignment is made through the keyword “localparam,” its value cannot be overridden.

PATH DELAYS

The delay between source pin (input or inout) and destination pin (ouput or inout) of module is called module path delay.

Verilog has the provision to specify and check delays associated with total paths – from any input to any output of a module. Such paths and delays are at the chip or system level. They are referred to as "module path delays".

Constructs available make room for specifying their paths and assigning delay values to them – separately or together.

Specify Blocks

Module paths are specified and values assigned to their delays through specify blocks. They are used to specify rise time, fall time, path delays pulse widths, and the like. A "specify" block can have the form shown in Figure

```
specify
specparam rise_time = 5, fall_time = 6;
(a =>b) = (rise_time, fall_time);
(c => d) = (6, 7);
endspecify
```

The block starts with the keyword "specify" and ends with the keyword "endspecify". Specify blocks can appear anywhere within a module.

Module Paths

Module Path delays are assigned in Verilog within the keywords specify and endspecify. The statements within these keywords constitute a specify block.

Module paths can be specified in different ways inside a specify block.

Parallel connection

Every path delay statement has a source field and a destination field.

A parallel connection is specified by the symbol => and is used as shown below.

Usage: (<source_field> => <destination_field>) = <delay_value>;

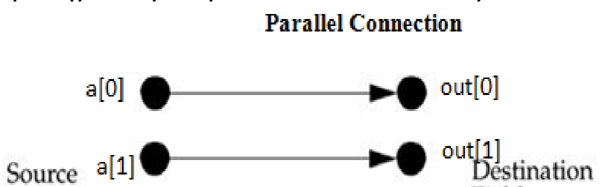
In a parallel connection, each bit in source field connects to its corresponding bit in the destination field.

If the source and the destination fields are vectors, they must have the same number of bits; otherwise, there is a mismatch. Thus, a parallel connection specifies delays from each bit in source to each bit in destination.

Example: Parallel Connection

(a => out) = 9; //bit-to-bit connection. Both a and out are single-bit

// vector connection. Both a and out are 4-bit vectors a[2:0], out[2:0] a is source field, out is destination field.



```
// for three bit-to-bit connection statements.
(a[0] => out[0]) = 9;
(a[1] => out[1]) = 9;
(a[2] => out[2]) = 9;


//illegal connection. a[4:0] is a 5-bit vector, out[3:0] is 4-bit.
 //Mismatch between bit width of source and destination fields
(a => out) = 9;                    //bit width does not match.
```

Full connection

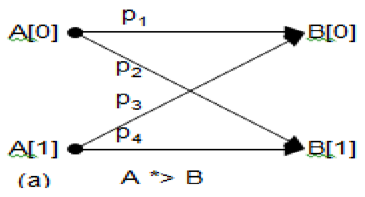
A full connection is specified by the symbol *> and is used as shown below.

Usage: (<source_field> *> <destination_field>) = <delay_value>;

In a full connection, each bit in the source field connects to every bit in the destination field. If the source and the destination are vectors, then they need not have the same number of bits. A full connection describes the delay between each bit of the source and every bit in the destination.

Example:

Figure below illustrates a case of all possible paths from a 2-bit vector A to another 2-bit vector B; the specification implies 4 pa



We can write the module M with pin-to-pin delays, using specify blocks as follows:

```
// Parallel connection

module M (out, a, b, c, d);
output out;
input a, b, c, d;
wire e, f;
  //Specify block with path delay statements
specify
(a => out) = 9;
(b => out) = 9;
(c => out) = 11;
(d => out) = 11;
endspecify

//gate instantiations
and a1(e, a, b);
and a2(f, c, d);
and a3(out, e, f);
endmodule
```

```
//Full Connection

module M (out, a, b, c, d);
output out;
input a, b, c, d;
wire e, f;
specify
(a,b *> out) = 9;
(c,d *> out) = 11;
endspecify
and a1(e, a, b);
and a2(f, c, d);
and a3(out, e, f);
endmodule
```

MODULE PARAMETERS

Module parameters are associated with size of bus, register, memory, ALU, and so on. They can be specified within the concerned module but their value can be altered during instantiation. The alterations can be brought about through assignments made with defparam. Such defparam assignments can appear anywhere in a module.

Example

The parameter msb specifies the ALU size — consistently in the input and the output vectors of the ALU. The size assignment has been made separately through the assignment statement

```
parameter msb = 3;
```

The ALU module with its size declared as a parameter.

```
module alu (d, co, a, b, f,cci);
 parameter msb=3;
output [msb:0] d; output co;
wire[msb:0]d;
input cci;
input [msb : 0 ] a, b;
input [1 : 0] f;
specify
(a,b=>d)=(1,2);
(a,b,cci*>co)=1;
endspecify
assign {co,d}= (f==2'b00)?(a+b+cci):((f==2'b01)?(a-b):((f==2'b10)?{1'bz,a^b}:{1'bz,~a}));
endmodule
```

SYSTEM TASKS AND FUNCTIONS

Verilog has a number of System Tasks and Functions defined in the LRM (language reference manual).

They are for taking output from simulation, control simulation, debugging design modules, testing modules for specifications, etc.

A "$" sign preceding a word or a word group signifies a system task or a system function.

Output Tasks

A number of system tasks are available to output values of variables and selected messages, etc., on the monitor. Out of these $monitor and $display tasks have been extensively used.

Display Tasks

The **$display** task, whenever encountered, displays the arguments in the desired format; and the display advances to a new line.

$strobe Task:

When a variable or a set of variables is sampled and its value displayed, the $strobe task can be used; it senses the value of the specified variables and displays them.

The $strobe task is executed as the last activity in the concerned time step. It is useful to check for specific activities and debug modules.

Example:

```
initial #9 $strobe ("at time %t, di=%b, do=%b", $time, di, do);
```

$monitor Task:

$monitor task is activated and displays the arguments specified whenever any of the arguments changes.

$stop and $finish Tasks:

The $stop task suspends simulation. The compiled design remains active; simulation can be resumed through commands available in the simulator.

In contrast $finish stops simulation, closes the simulation environment, and reverts to the operating system.

$random Function:

A set of random number generator functions are available as system functions.

One can start with a seed number (optional) and generate a random number repeatedly. Such random number sequences can be fruitfully used for testing.

Compiler directives

Compiler directives are special commands, beginning with ‘, that affect the operation of the Verilog simulator.

Time Scale

`timescale specifies the time unit and time precision. A time unit of 10 ns means a time expressed as say #2.3 will have a delay of 23.0 ns. Time precision specifies how delay values are to be rounded off during simulation. Valid time units include s, ms, us (µs), ns, ps, fs.

Only 1, 10 or 100 are valid integers for specifying time units or precision. It also determines the displayed time units in display commands like $display.

Syntax

`timescale time_unit / time_precision;

Examples

`timescale 1 ns/1 ps // unit =1ns, precision=1/1000ns

`timescale 1 ns /100 ps // time unit = 1ns; precision = 1/10ns;

`define

A macro is an identifier that represents a string of text. Macros are defined with the directive **`define**, and are invoked with the quoted macro name as shown in the example. Verilog compiliers will substitute the string for the macro name before starting compilation. Many people prefer to use macros instead of parameters.

The define directive in Verilog is similar to #define in c-language.

Syntax

`define macro_name text_string;

. . . `macro_name . . .

Example

`define add_lsb a[7:0] + b[7:0]

`define N 8 // Word length

wire [`N -1:0] S;

assign S = 'add_lsb; // assign S = a[7:0] + b[7:0];

Include Directive

Include is used to include the contents of a text file at the point in the current file where the include directive is. The include directive is similar to the C/C++ include directive.

Syntax

`include file_name;

Example

module x;

`include “dclr.v”; // contents of file “dclr,v” are put here

USER-DEFINED PRIMITIVES (UDP):

The primitives available in Verilog are all of the gate or switch types. Verilog has the provision for the user to define primitives –called "user defined primitive (UDP)" and use them.

The designers occasionally like to use their own custom-built primitives when developing a design. Verilog provides the ability to define User- Defined Primitives (UDP). These primitives are self-contained and do not instantiate other modules or primitives. UDPs are instantiated exactly like gate-level primitives.

UDPs are basically of two types –combinational and sequential. A combinational UDP is used to define a combinational scalar function and a sequential UDP for a sequential function.

Combinational UDPs:

A combinational UDP accepts a set of scalar inputs and gives a scalar output. An inout declaration is not supported by a UDP. The UDP definition is on par with that of a module; that is, it is defined independently like a module and can be used in any other module.

```
primitive udp_and(out, a, b);
output out;
input a, b;
table
    //   a b: Out;
        0 0: 0;
        0 1: 0;
        1 0: 0;
        1 1: 1;
endtable
endprimitive
```

Sequential UDPs:

Any sequential circuit has a set of possible states. When it is in one of the specified states, the next state to be taken is described as a function of the input logic variables and the present state. A sequential UDP can accommodate all these.

```
    primitive latch(q, d, clock, clear);   // d-latch
    output q;
    reg q; //q declared as reg to create internal storage input d, clock, clear;
    initial q = 0; //initialize output to value 0
    table                               //state table

  //d clock clear: q : q+ ;
    ? ? 1 : ? : 0 ;     //clear condition;
    1 1 0 : ? : 1;    //latchq =data=1
    0 1 0 : ? : 0;    //latchq =data=0
    ? 0 0 : ? : - ;    //retain original state if clock = 0
    endtable
    endprimitive
```