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.

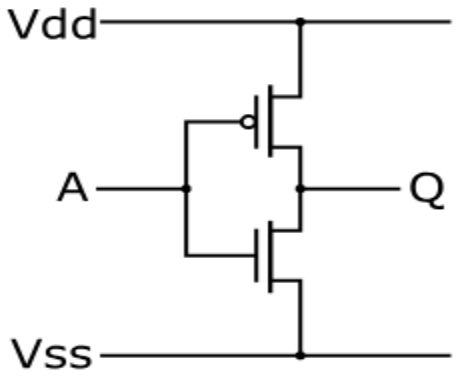


Figure 1 CMOS inverter

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.

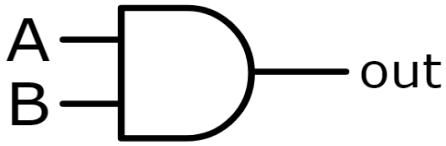


Figure 2 AND gate symbol

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.

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

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:

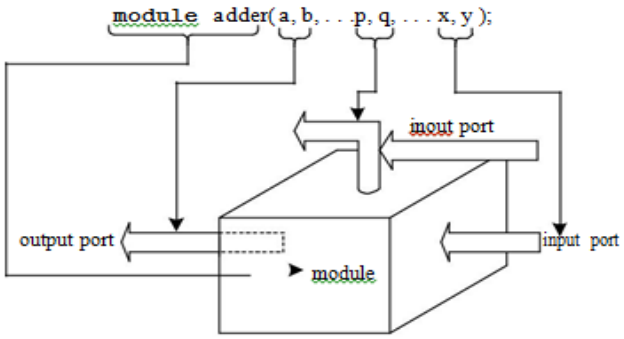


Figure 1 Representation of a module as black box with its ports.

- 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.

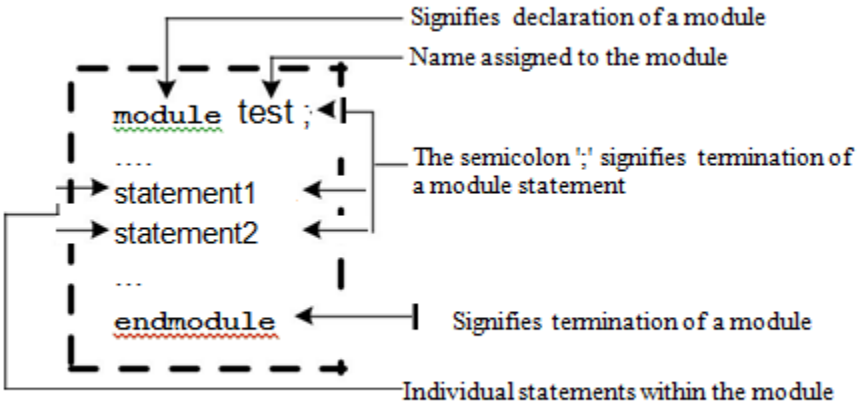


Figure 2 Structure of a typical simulation module.

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.

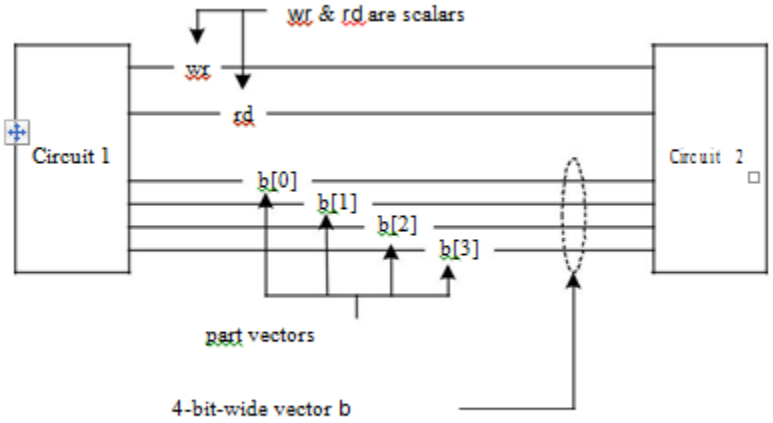


Figure Illustration of scalars and vectors.

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.
```

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>;

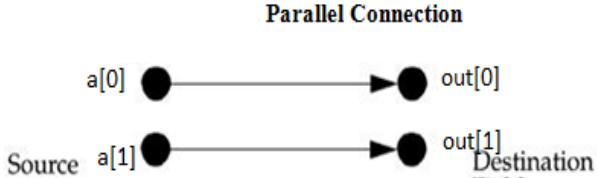
In a parallel connection, each bit in source field connects to its corresponding bit in the destination field.

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

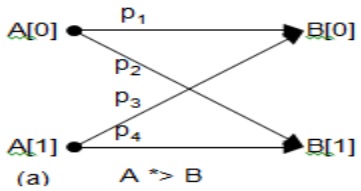
A full connection is specified by the symbol *> and is used as shown below.

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
```

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 Left shift Arithmetic right shift Arithmetic left shift	Two Two Two 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

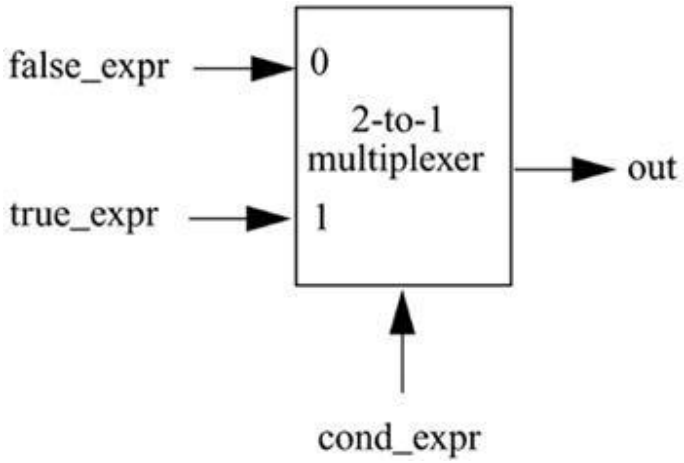
Conditional Operator

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

Testbench

- Test benches are used to simulate your design without the need of any physical hardware.
- A test bench is actually just another Verilog file! However, the Verilog you write in a test bench is not quite the same as the Verilog you write in your designs
- If the number of input signals are very large and/or we have to perform simulation several times, then this process can be quite complex, time consuming and irritating.
- with the help of testbenches, we can generate results in the form of csv (comma separated file), which can be used by other softwares for further analysis e.g. Python, Excel and Matlab etc.

Procedure

- Testbenches are written in separate Verilog files
- A test bench starts off with a module declaration
- A testbench with name ‘half_adder_tb
- Ports of the testbench is always empty i.e. no inputs or outputs are defined
- After we declare our variables, we instantiate the module we will be testing
- ‘Initial block’ is used , which is executed only once, and terminated when the last line of the block executed
- DUT is a very common name for the module to be tested in a test bench

Half adder

```
Module half_adder( input wire a, b,
Output wire sum, carry);

assign sum = a ^ b;
assign carry = a & b;

endmodule
```

Half adder test bench

```
module half_adder_tb;
reg a, b;
wire sum, carry;
localparam period = 20;

 half_adder UUT (.a(a), .b(b), .sum(sum), .carry(carry));

 initial // initial block executes only once
begin // values for a and b
a = 0; b = 0;
#period; // wait for period
a = 0; b = 1;
 #period;
 a = 1; b = 0;
#period;
 a = 1; b = 1;
 #period;
 end
 endmodule
```

Jk flipflop

```
module jkff_behave(clk,j,knq,qbar);
input clk,j,k;
output reg q,qbar;
always@(posedge clk)
begin
if(k = 0)
begin
q <= 0;
qbar <= 1;
end
```

```
always@(posedge clk)
begin
  if(k = 0)
   begin
    q <= 0;
    qbar <= 1;
   end
  else if(j = 1)
   begin
    q <= 0;
    qbar <= 0;
   end
```

```
Else if(j = 0 & k = 0)
   begin
    q <= q;
    qbar <= qbar;
   end
  else if(j = 1 & k = 1)
   begin
    q <= ~q;
    qbar <= ~qbar;
   end
end
endmodule
```

Using case statement

```
module JKFlipFlop( input J,input K,input clk,output Q, output Qbar  );
reg Q,Qbar;
always@(posedge clk)
begin
   case({J,K})
   2’b00:Q<=Q;
   2’b01:Q<=1’b0;
   2’b10:Q<=1’b1;
   2’b11:Q<=Qbar;
endcase
end
endmodule
```

Test Bench

```
module JK_FlipFlop_TB;
    // Inputs
    reg J;
    reg K;
    // Outputs
    wire Q;
    wire Qbar;
    // Instantiate the Unit Under Test (UUT)
    JKFlipFlop uut ( .J(J), .K(K), .Q(Q),.Qbar(Qbar)  );
```

```
    initial begin
        // Initialize Inputs
clk=0;
forever #5 clk=~clk
          #100  J = 0; K = 0;
        #100  J=0; K=1;
        #100  J=1;k=0;
      #100  J=1;  K=1;
    end
endmodule
```

Up counter design

```
Module up_counter(input clk, reset, output[3:0] counter
   );
reg [3:0] counter_up;

// up counter
always @(posedge clk or posedge reset)
begin
if(reset)
 counter_up <= 4'd0;
else
 counter_up <= counter_up + 4'd1;
end
assign counter = counter_up;
endmodule
```

Test bench

```
Module upcounter_testbench();
reg clk, reset;
wire [3:0] counter;
up_counter dut(clk, reset, counter);
initial begin
clk=0;
forever #5 clk=~clk;
end
initial begin
reset=1;
#20;
reset=0;
end
endmodule
```