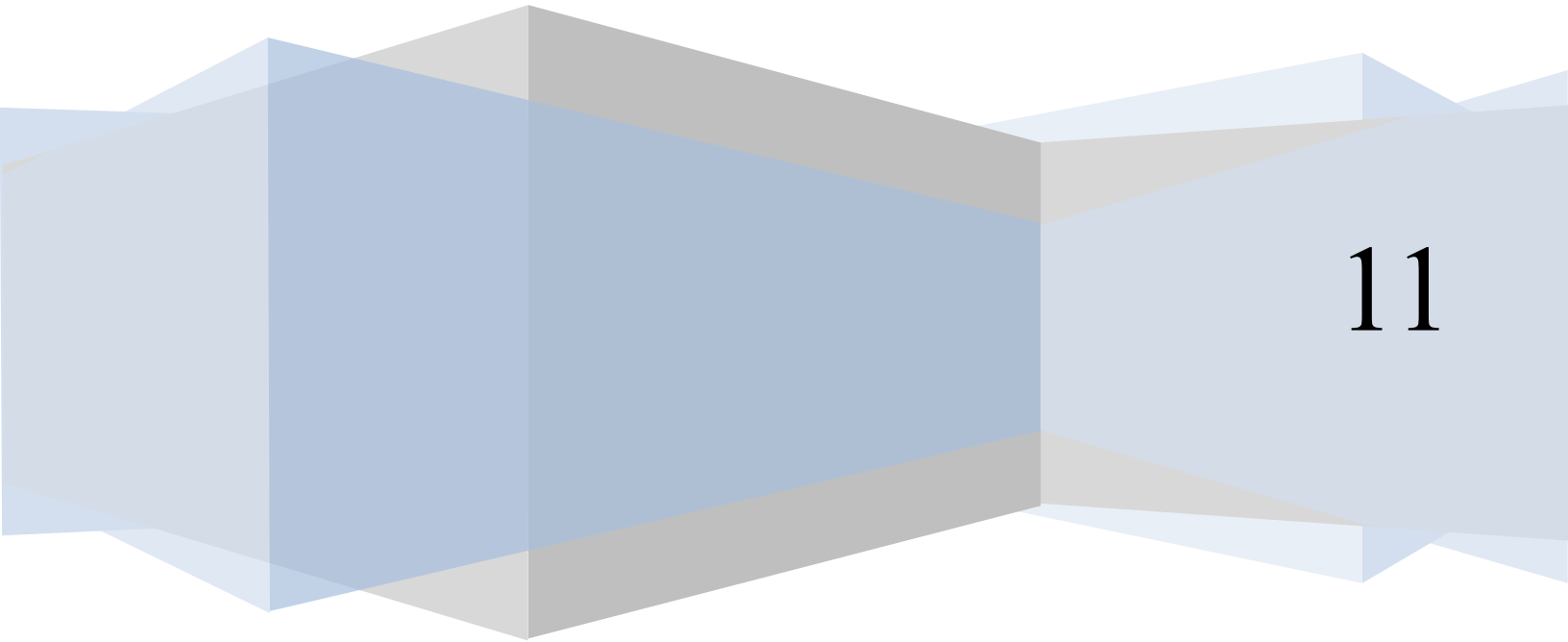
Masamb Electronics Systems Pvt. Ltd.

Verilog HDL for Dummies!

Verilog for Beginners

anyone



11

Verilog HDL for Dummies!

Table of Contents

New Features In Verilog-2001

Lexical Conventions

- White Space Characters
- Comments
- operators
- number Specifications
  - Strings
  - Identifiers and keywords
- Escaped Identifiers

Data Type Declarations

- Value Set
- Nets
- Registers
- Vectors
- Integer,Real ,and Time Registers Data types
- Arrays
- Memories
- Parameters
- Strings

System Tasks and Compiler Directives

- System Tasks
- Compiler Directives

Module Definitions

- Modules
- Ports
- Hierarchical Names

Gate-Level Modeling

- Gate Types
- Gate Delays
- Examples

Dataflow Modeling

- Continuous Assignments
  - Delays
- Expression ,Operators and operands
- Types of operators
- Examples

Behavioral Modeling

- Structured Procedures
  - Procedural Assignments
- Timing Controls

Tasks and Function

Useful Modeling Tech

Simulation Mechanics

Testbench

User Defined Primitives

Guide lines for Verilog coding

Synthesizable verilogCode examples

Introduction to Verilog

Verilog HDL originated circa 1983 at Gateway Design Automation, which was then located in Acton, MA. The company was privately held at that time by Dr. Prabhakar Goel, the inventor of the PODEM test generation algorithm. Verilog HDL was designed by Phil Moorby, who was later to become the Chief Designer for Verilog-XL and the first Corporate Fellow at Cadence Design Systems.

Moorby built a simulator around Verilog-XL in 1984-85, and then went on to make his second major contribution at GDA, viz. the XL algorithm for every fast gate-level simulation, which was first productized in 1986.

Gateway Design Automation grew rapidly with the success of Verilog-XL and was finally acquired by Cadence Design Systems, San Jose, CA in 1989. Up till this time, Verilog HDL was still a proprietary language, being the property of Cadence Design Systems.

Cadence Design Systems decided to open the language to the public in 1990, and thus OVI was born.When OVI was formed in 1991, a number of small companies began working on Verilog simulators. The first of these came to market in 1992, and now there are mature Verilog simulators available from several souces.

As a result, the Verilog market has grown substantially. The market for Verilog-related tools in 1994 was well over $75m, making it the most commercially significant hardware description language on the market.

Verilog is now in the process of being standardized by the IEEE. There is an IEEE working group established under the Design Automation Sub-Committee which was established in 1993 to produce the IEEE Verilog standard 1364. This working group is currently active and expects to produce a draft standard for balloting sometime in 1995.

Advantage of Verilog HDL

- ✓ Compact description.
- ✓ Easy to edit.
- ✓ Highly portable.
- ✓ Supports a higher level of abstraction.
- ✓ Rapid prototyping of design.
- ✓ Availability of extensive vendor libraries.
- ✓ Increasing capability of synthesis tools.

Concurrency

The following Verilog HDL constructs are independent processes that are evaluated concurrently in simulation time:

- ✓ module instances
- ✓ primitive instances
- ✓ continuous assignments
- ✓ procedural blocks

Lexical Conventions

The basic lexical conventions used by Verilog HDL are similar to those in the C programming language. Verilog HDL is a case-sensitive language. All keywords are in lowercase.

White Space Characters

blanks space(\b), tabs(\t), newlines (\n) comprise the white space

These characters are ignored except when they serve to separate other tokens. However, blanks and tabs are significant in strings.

White spaces have no syntactical significance and can be inserted for better readability. White space is not ignored in strings.

Comments

// begins a single line comment, terminated by a newline.

/* begins a multi-line block comment, terminated by a */

Number Specfications

There are two types of number specfications in verilog:sized and unsized

Sized Numbers:-

- ✓ Sized numbers are represented as <size>'<base format><number>
- ✓ Size is only written in decimal and specifies the number of bits in the number.
- ✓ Base formats are decimal('d or 'D),Hexadecimal('h or 'H),Binary('b or 'B) and Octal('o or'O)
- ✓ Number is specified as consecutive digits from 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f.Uppercase letters are legal for number specifications.Example: 4'b1111 //4bit binary number

unsigned numbers:

Numbers that are specified without a <base format>specification are decimal number by default.example: 23456 //this is a 32bit decimal number by default

Numbers that are written without a <size> specification have a default number of bits.that is simulator and machine specific.example: 'hc3 //32bit hexadecimal number

X or Z values:

An unknown value is denoted by x.example 6'hx //6bit hexadecimal number.A high impedance value is denoted by z.example 32'bz //32bit high impedance number

Strings

A string is a sequence of characters that are enclosd by double quotes.
It canot be multiple lines.
Example:"Hello " \\ is a string

Identifiers and keywords

Identifiers

Identifiers are names used to give an object, such as a register or a module, a name so that it can be referenced from other places in a description.

- ✓ Identifiers must begin with an alphabetic character or the underscore character (a-z A-Z)
- ✓ Identifiers may contain alphabetic characters, numeric characters, the underscore, and the dollar sign (a-z A-Z 0-9 _ $)
- ✓ Identifiers can be up to 1024 characters long.

Examples of identifiers:-

reg value ;//reg is a keyword and value is a identifier.
Input clk ;//input is a keyword and clk is an identifier

Escaped Identifiers

- ✓ Verilog HDL allows any character to be used in an identifier by escaping the identifier. Escaped identifiers provide a means of including any of the printable ASCII characters in an identifier (the decimal values 33 through 126, or 21 through 7E in hexadecimal).
- ✓ Escaped identifiers begin with the back slash (\ ).
- ✓ Entire identifier is escaped by the back slash.
- ✓ Escaped identifier is terminated by white space
  - ➢ Characters such as commas, parentheses, and semicolons become part of the escaped identifier unless preceded by a white space.
- ✓ Terminate escaped identifiers with white space, otherwise characters that should follow the identifier are considered as part of it

Data Type declaration

Value Set:

Verilog consists of only four basic values. Almost all Verilog data types store all these values:

0 (logic zero, or false condition)

1 (logic one, or true condition)

x (unknown logic value) x and z have limited use for synthesis.

z (high impedance state)

Wire

A wire represents a physical wire in a circuit and is used to connect gates or modules. The value of a wire can be read, but not assigned to, in a function or block. and Procedures: Always and Initial.

A wire does not store its value but must be driven by a continuous assignment statement or by connecting it to the output of a gate or module.

Other specific types of wires include: wand (wired-AND);the value of a wand depend on logical AND of all the drivers connected to it.

wor (wired-OR);: the value of a wor depend on logical OR of all the drivers connected to it.

tri (three-state;): all drivers connected to a tri must be z, except one (which determines the value of the tri).

Syntax

Example

[msb:lsb] wire_variable_list;

wand [msb:lsb] wand_variable_list;

the logical AND of

wor [msb:lsb] wor_variable_list;

wire c // simple wire wire

wand d;

assign d = a; // value of d is

assign d = b; // a and b

tri [msb:lsb] tri_variable_list;

wire [9:0] A; // a cable (vector)

of 10 wires.

Register Data Types

- ✓ Registers store the last value assigned to them until another assignment statement changes their value.

- ✓ Registers represent data storage constructs.
- ✓ You can create arrays of the regs called memories.
  - ➢ register data types are used as variables in procedural blocks.
  - ➢ A register data type is required if a signal is assigned a value within a procedural block
  - ➢ Procedural blocks begin with keyword initial and always.

Data Types	Functionality
reg	Unsigned variable
integer	Signed variable - 32 bits
time	Unsigned integer - 64 bits
real	Double precision floating point variable

Vector

Nets or reg data types can be declared as vectors (multiple bit widths).If bit width is not specified, and then default is scalar (1 bit).

Example:-

Wire a;

Wire [7:0] //8bit bus

Reg clk;

Bus [2:0] //three least significant bits of vector bus, using bus [0:2] is illegal because the significant bit should always be left of a range specification.

*Vectors can be declared at [high# :low#] or [low# : high#] ,but the left number in the squared brackets is always the most significant bit of the vector

Integer, Real ,and Time Registers Data types

Integer

An integer is a general purpose register data types used for manipulating quantities. Integers are declared by the keyword **Integer.** The default width for an integer is the host machine word size ,which is implementation specific but is at least 32 bits. Registers declared as data type reg store values as unsigned quantities, where as integers store values as signed quantities.

Example:-

```
Integer counter;


 Initial
 Counter = -1;
```

Real

Real number constants and real register data types are declared with the keyword real. They can be specified in decimal notation (3.14) or in scientific notation Real numbers cannot have a range declaration and there default value is 0.When a real value is assigned to an integer ,the real number is round off to the nearest integer.

Example

```
Real delta
Initial begin
delta = 4e10;
delta = 2.13;
end
integer i  //define an integer i
initial
i =  delta; //I gets the value 2
```

Time

Time is a 64-bit quantity that can be used in conjunction with the $time system task to hold simulation time. Time is not supported for synthesis and hence is used only for simulation purposes.

Syntax

time time_variable_list;

Example

time c;

c = $time; //c = current simulation time

Arrays

Arrays are allowed in Verilog for reg, integer, and time data types. Arrays are not allowed for real variables. Multidimensional arrays are also not permitted in Verilog

```
integer count [1:15] ;  //  15 integers
reg var [-15:16] ;   //   32 1-bit regs
reg [7:0] mem [0:1023] ;   //  1024  8-bit regs


reg mda[1:100] [1:100] ;   //  INVALID
```

Array are allowed by <array_name>[<subscript name>]

Array element can be accessed by using subscript. Array part-select or entire array cannot be accessed at once.

```
mem[10] = 8'b10101010 ;
var[2:9] = 8'b11011101 ;   //   INVALID
var = 32'd567 ;   //   INVALID
```

Array is multiple elements that are 1bit or nbits wide where as vector is a single element that is nbit wide.

Memories

Memories are modeled in Verilog simply as a one-dimensional array of registers. Each element of the array is known as an element or word and is addressed by a single array index. Each word can be one or more bits. A particular word in memory is obtained by using the address as a memory array subscript.

Example:

```
Reg mem1bit[0:1023]  //memory mem1bit with 1k 1-bit words
Reg [7:0] membyte[0:1023]   //memory membyte with 1k 8bit words(bytes)
```

Parameters

A parameter defines a constant that can be set when you instantiate a **module**. This allows customization of a module during instantiation. A constant in Verilog is declared with the keyword parameter, which declares and assigns values to the constant.

Syntax

parameter par_1 = *value*,
par_2 = *value*, ...;

parameter [range] parm_3 = value

Example

parameter add = 2'b00, sub = 3'b111;
parameter n = 4;
parameter n = 4;
parameter [3:0] param2 = 4'b1010;
. . .
reg [n-1:0] harry; /* *A 4-bit register whose length is* set by parameter n above. */
always @(x)
y = {{(add - sub){x}};
if (x) begin

state = param2[1]; else state = param2[2];
end

Strings

Strings can be stored in reg. The width of the register variables must be large enough to hold the string. Each character in the string takes up 8bits(1byte).If the width of the register is greater than the size of the string ,Verilog fills bits to the left of the string with zero. Where as if the register width is smaller than the string width ,Verilog truncates the leftmost bits of the string.

Example

```
Reg [8*18:1] string value;
Initial
String_value = "Hello Verilog World";  //string can be stored in variable
```

Special characters serve a special purpose in displaying strings, such as newline ,tabs and displaying arguments values.

Special character can be displayed in strings only when they are preceded by escape characters, as shown in below

Escaped Character	Displayed
\n	newline
\t	tab

%% %

System Tasks and Compiler Directive

System Tasks

All System Tasks appear in the form $<keyword>.Operation such as displaying on the screen, monitoring values of nets, stopping and finishing are done by system tasks.

>Displaying information:

$display is the main system task for displaying values of variables or strings or expression. most useful tasks in Verilog .

Usage: $display(p1 ,p2,p3……pn);

P1,p2,p3….pn can be quoted strings or variables or expressions. $display inserts a newline at the end of string by default

Format	Display
%d or %D	Display variable in decimal
%b or %B	Display variable in binary
%s or %S	Display string
%h or %H	Display variable in hex
%c or %C	Display ASCII character
%o or %O	Display variable in octal

This task simply prints output as specified by the optional format string and the argument expressions. It adds a newline character to the output string. There are several variations of the $display system task which are used for output. They all take the same arguments, and differ only in their default radix.

The general format is: $display(arg1, arg2, ...);

where argi is either: a format string or an expression.

If there is no format string given, then each expression is converted to the default radix and printed. If an expression is given as an argument with no corresponding format specifier, then it is converted according to the default radix and inserted into the output string. If an argument is omitted, as indicated by consecutive ",", then a space is inserted into the output.

>Example of the $display task

If variables contain x or z values they are printed in the displayed string as “x” or “z”.

```
//Display the string in quotes
$display(“Hello Verilog World”);
```

```
--Hello Verilog world
//Display value of current simulation time 230
$display($time);
```

$Write

The $write system task is just like $display, except that it does not add a newline character to the output string.

Example

```
$write ($time," array:");
for (i=0; i<4; i=i+1)
      write(" %h", array[i]);
$write("\n");
```

This would produce the following output:

```
110 array: 5a5114b3 0870261a 0678448d 4e8a7776
```

>Monitoring Information

Verilog provide a mechanism to monitor a signal when its value changes. This facility is provided by the $monitor task
Usage: $monitor (p1,p2,p3……pn);
The parameter p1,p2…pn can be variables, signal names or quoted strings.

$monitor continuously monitors the values of the variables or signals specified in the parameter list and display all parameters in the list whenever the value of anyone variable or signal changes.
Two tasks are used to switch monitoring on and off.

Usage: $monitoron; --tasks enables monitoring during simulation
$monitoroff; --tasks disable monitoring during simulation

>Stopping and Finishing in a simulation

$stop

The task $stop is provided to stop during a simulation

Usage:$stop:

This system task is used to suspend simulation and enter whatever interactive environment the simulator may provide. Note that it is not useful for other, non-simulation activities, like synthesis or timing analysis.

Because each simulator provides a somewhat different interactive environment, what happens after $stop is executed is implementation-dependent. However, when the `$stop` task is executed, the simulation suspends at that point. It is not necessarily the case that simulation is at the end of a time step. It is the case that simulation is resumable as if the suspension had not happened.

$stop task puts the simulation in an interactive mode. The designer can then debug the design from the interactive mode.

The $stop task is used whenever the design wants to suspend the simulation and

examine the values of signals in the design

Example:

```
initial begin
   repeat (5)
#1000   $stop;
   $finish;
end
```

This example would allow the user to inspect the state of the simulation every 1000 time units 5 times before the simulation terminated.

$finish(Simulation Termination)

The $finish task terminates the simulation.

Usage:$finish

This system task is used to terminate simulation. It can take an optional argument which indicates how much information the simulator should print out about the simulation execution. Typically this information will include the number of events, CPU time and amount of memory the model has consumed.

Example:

```
case (data)
   `none:    $finish;
   `some:    $finish(1);
   `all:     $finish(2);
endcase
```

This contrived example shows how you might dynamically select how much data you want about the simulation run. Any statement after this case statement would not get executed.

Example:

```
Initial
begin
clock = 0;
reset = 1;
#100 $stop;
#900 $finish;
End
```

Simulation terminates when `$finish` executes. It also terminates when there are no more events on the event list. This makes sense, since there is nothing more for it to do.

Simulation terminating because the event list is exhausted is nearly always an error. For this to be the case, the clock loop is not running. Since most models have a clock, this usually means the clock has gone to x, and probably everything else, too.

Opening a file:

A file can be opened with the system task $fopen
Usage:$fopen (“<name_of_file>”);
Usage: <file_handle> = $fopen(“<name_of_file>”);
The tasks $fopen returns a 32bit value called a multichannel descriptor.
Example:

```
Initial
begin
 handle1 = $fopen(“file1.out”);
```

Writing to files:-

The system tasks $fdisplay ,$fmonitor ,$fwrite , $fstrobe are used to write to files.
Usage: $fdisplay (<file_descriptor>, p1, p2 …, pn);
$fmonitor(<file_descriptor>, p1,p2,…, pn);

Closing files:

Files can be closed with the system task $fclose.
Usage: $close(<file_handle>);

Strobing:

It is done with the system task keyword $strobe. When $strobe is used ,it is always executed after all other assignment statements in the same time unit have executed. It provides a synchronization mechanism to ensure that the data is displayed only after all other assignment statements, which change the data in that time step ,have executed. Where as if many other statements are executing the same time unit as the $display task , the order in which the statements and the $display task are executed is nondeterministic.

The $strobe system task is just like $display, except that it waits until the end of the time step before printing. This can be useful to be sure that the output comes after all value changes.

Example:

```
 always @(posedge clock)
       state1 = newstate1;
  always @(posedge clock)
       state2 = newstate2;
   always @(posedge clock)
       $strobe($stime,,state1,,state2);
```

This code would print out the new values of state1 andstate2 at each rising clock edge, after all other processing had been completed. If you simply used $display here, you could not be sure that the displayed values were the new ones.

Example:

```
always @(posedge clock)
begin
```

```
  a = b;
  c = d;
end
always @(posedge clock)
 $strobe("Displaying a = %b, c = 5b" ,a,c);
```

Here the value at positive edge of clock will be displayed only after statements a = b and c = d execute. If $display was used $display might execute before statements a = b and c = d thus displaying different values.

Compiler Directives

All Compiler directives are defined by using the `<keyword>construct.
Two most useful compiler directives.

`define

--'define directive are used to define text macros in Verilog.

Example:

```
`define S $stop;
`define WORD_SIZE 32 //used as 'WORDSIZE in the code
```

`include

This directives allows us to include entire contents of a Verilog source file in another Verilog file during compilation

Example

```
`include header.v
```

`timescale:--

Simulation time is simply a 64-bit unsigned number. However, in many cases, it is convenient to give an interpretation to the time, as nanoseconds, microseconds, or some other time unit.

The way to do that in Verilog is by using the: `timescale time_unit/time_precision

This compiler directive tells the compiler how to interpret delays that it sees in the source. By using different values with `timescale, different modules or parts of the model can use delays with different time scales, and the compiler can scale them all correctly.

For example, one module could specify its delays in microseconds and another module could specify its delays in nanoseconds, and the simulator could handle them consistently.

The time_unit argument specifies the unit for all delays which appear after the directive. That is, if the time_unit is milliseconds, all delays will be scaled as if they are milliseconds.

The time_precision argument tells what the minimum unit of accuracy is for the delays. That is, if the precision is microseconds, and the time unit is milliseconds, then any delay

with more than 3 decimal digits would be rounded. For example, #4.12345 would be converted to 4123 microseconds.

The valid time_unit and time_precision arguments are:

argument	unit
1s,10s,100s	seconds
1ms,10ms,100ms	milliseconds
1us,10us,100us	microseconds
1ns,10ns,100ns	nanoseconds
1ps,10ps,100ps	picoseconds
1fs,10fs,100fs	femtoseconds

The time_precision argument must be at least as small as the time_unit argument.

Because there can be more than one `timescale directive in a model, the compiler must take the smallest time_precision and use that as a global precision for the simulation. If the smallest precision is 100ns, then 1 time unit of the simulation clock would correspond to 100ns.

Modules and ports

Modules:

Module Declaration

A module is the principal design entity in Verilog. The first line of a module declaration specifies the name and portlist (arguments). The next few lines specify the i/o type (input, output or inout) and width of each port. The default port width is 1 bit. Then the port variables must be declared wire, wand,. . ., reg . The default is wire. Typically inputs are wire since their data is latched outside the module. Outputs are type reg if their signals were stored inside an always or initial block.

Syntax

```
input add;                    // defaults to wire

 input [7:0] in1, in2; wire in1, in2;
```

```
output [7:0] oot; reg oot;

... statements ...

endmodule
```

- ✓ Ports allow communication between a module and its environment.
- ✓ All but the top-level modules in a hierarchy have ports.
- ✓ Ports can be associated by order or by name.

You declare ports to be input, output or inout. The port declaration syntax is :

```
input [range_val:range_var] list_of_identifiers;

output [range_val:range_var] list_of_identifiers;

inout [range_val:range_var] list_of_identifiers;
```

Examples : Port Declaration

```
input         clk         ; // clock input

input [15:0]  data_in     ; // 16 bit data input bus


output [7:0]  count       ; // 8 bit counter output

inout         data_bi     ; // Bi-Directional data bus
```

Input, Output, Inout

These keywords declare input, output and bidirectional ports of a module . Input and inout ports are of type wire. An output port can be configured to be of type wire, reg, wand, wor or tri. The default is wire.

Gate-Level Modeling

Primitive logic gates are part of the Verilog language. Two properties can be specified, drive_strength and delay.Drive_strength specifies the strength at the gate outputs. The strongest output is a direct connection to a source, next comes a connection through a conducting transistor, then a resistive pull-up/down. The drive strength is usually not specified, in which case the strengths defaults to strong1 and strong0. Delays: If no delay

is specified, then the gate has no propagation delay; if two delays are specified, the first represent the rise delay, the second the fall delay; if only one delay is specified, then rise and fall are equal. Delays are ignored
in synthesis The parameters for the primitive gates have been predefined as delays.

Basic Gates

These implement the basic logic gates. They have one output and one or more inputs. In the gate instantiation syntax shown below, GATE stands for one of the keywords and, nand, or, nor, xor, xnor.

Syntax

```
GATE (drive_strength) # (delays)
 instance_name1(output, input_1, input_2,..., input_N),
 instance_name2(outp,in1, in2,..., inN);
```

Delays is

#(rise, fall) or

rise_and_fall or

#(rise_and_fall)

Example

```
and c1 (o, a, b, c, d);      // 4-input AND called c1 and

c2 (p, f g);                 // a 2-input AND called c2.

or #(4, 3) ig (o, a, b);     /* or gate called ig (instance name);   rise time = 4, fall time = 3 */

xor #(5) xor1 (a, b, c);     // a = b XOR c after 5 time units

xor (pull1, strong0) #5 (a,b,c);  /* Identical gate with pull-up  strength pull1 and pull-down
                                     strength  strong0. */
```

buf , not Gates

These implement buffers and inverters, respectively. They have one input and one or more outputs. In the gate instantiation syntax shown below, GATE stands for either the keyword buf or not

Syntax Example

```
not #(5) not_1 (a, c);   // a = NOT c after 5

        time units
GATE (drive_strength) # (delays)

                                    buf c1 (o, p, q, r, in); // 5-output and 2-

output buffers
instance_name1(output_1, output_2,

                                c2 (p, f g);

                        ..., output_n, input),

 instance_name2(out1, out2, ..., outN, in);
```

Three-State Gates; bufif1, bufif0, notif1, notif0

These implement 3-state buffers and inverters. They propagate z (3-state or high-impedance) if their control signal is deasserted. These can have three delay specifications: a rise time, a fall time, and a time to go into 3-state.

Gate Delays:

Delays are specified in several ways in Verilog. Delays are introduced with the "#" character. A delay can be assigned to a net driver (either a gate, primitive, or continuous assignment), or it can be assigned to the net directly.

example

following are all legal delay specifications:

```
assign #10 net1 = ra + rb;
xor #3 xo1(a, b, c);
wire #(4,2) control_line;
```

A delay can also be specified in procedural statements, causing time to pass before the execution of the next statement.

```
always
 begin
#period/2      clk = ~clk;
 end
```

In a procedural delay, the delay value may be an arbitrary expression. By contrast, a delay in a declarative statement must be able to be evaluated at compile time, which is another way of saying it must be a constant or constant-valued expression.

Delays that are associated with nets in declarations or continuous assignments can have multiple values. There can be one, two, or three delay values specified: #(rise_delay, fall_delay, turnoff_delay).

These are used when the value of the net makes the corresponding change:

rise_delay 0, x, or z -> 1
fall_delay 1, x, or z -> 0
turnoff_delay 0, 1, or x -> z

The delay values are interpreted positionally. That is, the first one is the rise delay, the second is the fall delay, and the third is the turnoff delay. When the net value becomes x, then the smallest of the three delay values is used.

Some nets are not expected to take a z value, so it is not necessary to specify the turnoff delay. For these nets, you can simply use the rise and fall delays. If, by chance, the net does make a transition to z (or to x), the delay used will be the smaller of the two which are specified.

Similarly, most of the gate primitives cannot drive a z value, so it does not make sense to supply a turnoff delay for them. For these gates, the syntax specifies that you can only provide two delay values (the syntax description uses the term delay2, instead of delay3). The following gates take only two delay values:
and, nand, or, nor, xor, xnor, buf, not

If only one delay is specified, then it is used for all value transitions. Any place where a multiple delay value can be used it is also permissible to use a single delay value.

Single delay values are always used in procedural code.

Rise Delay

The rise delay is associated with a gate output transition to 1 from another value (0,x,z).

Fall Delay

The fall delay is associated with a gate output transition to 0 from another value (1,x,z).

Turn-off Delay

The fall delay is associated with a gate output transition to z from another value (0,1,x).

Min Value

The min value is the minimum delay value that the gate is expected to have.

Typ Value

The typ value is the typical delay value that the gate is expected to have.

Max Value

The max value is the maximum delay value that the gate is expected to have.

Examples

```
// Delay for all transitions

or #5 u_or (a,b,c);

// Rise and fall delay

and #(1,2) u_and (a,b,c);

// Rise, fall and turn off delay

nor # (1,2,3) u_nor (a,b,c);

//One Delay, min, typ and max

nand #(1:2:3) u_nand (a,b,c);

//Two delays, min,typ and max

buf #(1:4:8,4:5:6) u_buf (a,b);

//Three delays, min, typ, and max

notif1 #(1:2:3,4:5:6,7:8:9) u_notif1 (a,b,c);
```

Gate Delay Code Example

```
module not_gate (in,out);
input in;
output out;
not #(5) (out,in);
endmodule
```

Normally we can have three models of delays, typical, minimum and maximum delay. During compilation of a modules one needs to specify the delay models to use, else Simulator will use the typical model.

DataFlow Modeling

continuous assignment statement

Describes the design in terms of expressions instead of primitive gates. A continuous assignment statement is the most basic statement in data flow modeling ,used to drive a value onto a net. This assignment replaces gates in the description of the ckt and describe the circuit at a higher level of abstraction.

Syntax:

continuous_assign ::= assign [drive_strength][delays] list-of-net-assignment;

List_of_net_assignments:= net_assignment{,net_assignment}

net_assignment := net_lvalue = expression

Example

```
assign A = x | (Y & ~Z) ;
assign B[3:0] = 4'b10xx ;
assign C[15:0] = F[15:0] ^ E[15:0] ;
```

Left hand side of the assignment must be nets (scalar or vector).Right hand side expression can have registers, nets or function calls as operands.

The continuous assignment statement are continuously active and they all execute in parallel. Whenever value of any operand on right side changes expression is reevaluated and new value is assigned to the corresponding net.

Implict Continuous Assignments

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.

```
//regular continuous assignment
wire out;
assign out = in1 & in2;
//same effect is achived by implict continuous assignment statements
wire out = in1 & in2;
```

Delays

Delay values control the timing 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 assignments

Regular Assignment Delays

The delay value is specified after the keyword assign

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

Any change in values of in1 and in2 will result in a delay of 10times units before recomputation of the expression in1 & in2 and the result will assigned to out.if in1 and in2 changes value again before 10time unit . 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.

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.

Example:

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

Implicit Continuous Assignment Delay

We can use both a delay and an assignment on the net.

Example

```
//implict continuous assignment delay
wire #10 out = in1 &in2;
//same as
wire out;
assign #10 out = in1 &in2;
```

Expression,Operators,and Operands

Expression:

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

```
//Example of expressions combines operands and operator
 a ^ b
 in1 | in2
```

Operands:

Operands can be any one of the data types

It can be constants,integers,real numbers,nets,registerd,times,bit-select.

Example:

```
integer count,final_count
final_count = count + 1;//count is a integer operand
real a,b,c;
c = a-b; // a and b are real operands
```

Operators:

Operators act on the oprands to produce desired results.

Example

d1 && d2 //&& is an operator on operands d1 and d2

Types Of Verilog Operators

Arithmetic Operators

- ✓ Binary: +, -, *, /, % (the modulus operator)
- ✓ Unary: +, -
- ✓ Integer division truncates any fractional part
- ✓ The result of a modulus operation takes the sign of the first operand
- ✓ If any operand bit value is the unknown value x, then the entire result
- ✓ value is x
- ✓ Register data types are used as unsigned values
  - >negative numbers are stored in two' complement form

Relational Operators

a<b	a less than b
a>b	a greater than b
a<=b	a less than or equal to b
a>=b	a greater than or equal to b

- ✓ The result is a scalar value:
  - 0 if the relation is false
  - 1 if the relation is true
- ✓ x if any of the operands has unknown x bits
- ✓ Note: If a value is x or z, then the result of that test is false

Equality Operators

a === b a equal to b, including x and z
a !== b a not equal to b, including x and z
a == b a equal to b, resulting may be unknown
a != b a not equal to b, result may be unknown

- ✓ Operands are compared bit by bit, with zero filling if the two operands do not have the same length.
  - Result is 0 (false) or 1 (true)
- ✓ For the == and != operators the result is x, if either operand contains an x or a z
- ✓ For the === and !== operators
  - bits with x and z are included in the comparison and must match for the result to be true
    - the result is always 0 or 1

Logical Operators

! logic negation

&& logical and

|| logical or

- ✓ Expressions connected by && and || are evaluated from left to right
- ✓ Evaluation stops as soon as the result is known
- ✓ The result is a scalar value:
  - 0 if the relation is false
  - 1 if the relation is true
  - x if any of the operands has unknown x bits

Bit-wise Operators

~ negation
& and
| inclusive or
^ exclusive or

^~ or ~^ exclusive nor (equivalence)

Computations include unknown bits, in the following way:

~x = x
0&x = 0
1&x = x&x = x
1|x = 1
0|x = x|x = x
0^x = 1^x = x^x = x
0^~x = 1^~x = x^~x = x

When operands are of unequal bit length, the shorter operand is zero-filled in the most significant bit positions

Reduction Operators

&	and
~&	nand
|	or
~|	nor
^	xor
^~ or ~^	xnor

- ✓ Reduction operators are unary.
- ✓ They perform a bit-wise operation on a single operand to produce a single bit result.
- ✓ Reduction unary NAND and NOR operators operate as AND and OR respectively, but with their outputs negated.
  - > Unknown bits are treated as described before.

Shift Operators

<<	left shift
>>	right shift

- ✓ The left operand is shifted by the number of bit positions given by the right operand.
- ✓ The vacated bit positions are filled with zeroes.

Concatenation Operator

- ✓ Concatenations are expressed using the brace characters { and }, with commas separating the expressions within.
  Examples
  {a, b[3:0], c, 4'b1001} // if a and c are 8-bit numbers,the results has 24 bits

- ✓ Unsized constant numbers are not allowed in concatenations
- ✓ Repetition multipliers that must be constants can be used:
  - {3{a}} // this is equivalent to {a, a, a}
- ✓ Nested concatenations are possible:
  - {b, {3{c, d}}} // this is equivalent to {b, c, d, c, d, c, d}

Conditional Operator

- ✓ The conditional operator has the following C-like format:
  - cond_expr ? true_expr : false_expr
- ✓ The true_expr or the false_expr is evaluated and used as a result depending on whether cond_expr evaluates to true or false.
- ✓ Example
- ✓ out = (enable) ? data : 8'bz; // Tri state buffer

Operator Precedence

Operator	Symbols
Unary, Multiply, Divide, Modulus	+-!~*/%
Add, Subtract, Shift.	+, - , <<, >>
Relation, Equality	<,>,<=,>=,==,!=,===,!===
Reduction	&, !&,^,^~,|,~|
Logic	&&, ||

Conditional ?:

Behavioral Modeling

Behavioral Models : Higher level of modeling where behavior of logic is modeled.

Structured Procedure

There are two structured procedure statements in verilog:always and initial.
>initial : initial blocks execute only once at time zero (start execution at time zero).
> always : always blocks loop to execute over and over again, in other words as name means, it executes always.

initial and always Constructs

In behavioral modeling, we write Verilog code for desired functionality (behavior) of the circuit and leave the issue of hardware implementation to synthesis tools.

The two basic language constructs used are always and initial procedure blocks. Each always and initial block represents a separate activity flow (or hardware process) in Verilog starting at simulation time 0.

The initial block starts execution at tsim = 0 and stops after a single execution while always block also starts execution at tsim = 0, but it keeps on executing continuously in a loop.

If there are multiple initial and always blocks, each of them starts to execute concurrently at the time and finishes execution independent of other blocks. A module may contain any number of blocks but these cannot be nested.

Always Block

Syntax:

```
always @ (signal1 or signal2 or signal3) begin
 Block
end
```

This block implies, the processor should schedule Block whenever there is a change/transition in any of the three signals – signal1, signal2 or signal3.

Here the sensitivity list of this always block consists of these three signals.

We usually do not use logical operations inside the sensitivity list. Instead, condition checking is done inside the Block.

Here, the verilog scheduler monitors all the three signals individually. Whenever there is a change it enters the Block.

An example of the condition checking is seen here:

```
always @ (signal1 or signal2 or signal3) begin
if (signal1 == 1 and signal2 == 0)begin
  Block1
 end
else if (signal3 == 0) begin
 Block2
 end
 else begin
 Block3
end
```

Example : initial and always

initial	**always @ (posedge clk)**
begin	begin : D_FF
clk = 0;	if (reset == 1)
reset = 0;	q <= 0;
enable = 0;	else
data = 0;	q <=d;
end	end

example:

```
 module clk_gen ;
reg clk ;
initial                 //starts at tsim = 0 and
clk = 1'b0 ;            //execute once only
always                  //starts at tsim = 0 and
#5 clk = ~clk ;         //execute repeatedly
 initial                   //starts at tsim = 0 and
#1000 $finish ;       //scheduled to execute
endmodule
```

Procedural Assignment:

Procedural assignments update values of reg,integer,real or time variables.The value placed on a variable will remain unchanged until another procedural assignment updates the variable with different value.There are two types of Procedural assignments:blocking and nonblocking.

Blocking Procedural Assignments

Blocking assignments written inside an always or initial block execute in the same order in which they are written. A statement that follows a blocked statement in sequential flow cannot be executed until the preceding statement has completed execution. These assignment use the = operator.

target1 = expression1 ; executed 1st
target2 = expression2 ; executed 2nd
target3 = expression3 ; executed 3rd

Blocking

Syntax

Blocking

variable = expression;

variable = #Δt expression;

grab inputs now, deliver ans.later

#Δt variable = expression;

grab inputs later, deliver ans. later

Example For simulation

```
initial
begin

a=1; b=2; c=3;
```

#5 a = b + c; // wait for 5 units, and execute a= b + c =5.

d = a; // Time continues from lastline, d=5 = b+c at t=5.

Example For synthesis

always @(posedge clk)

begin

Z=Y; Y=X; // shift register

y=x; z=y; //parallel ff.

end

Nonblocking Procedural Assignment

- ✓ Non-blocking assignment behave differently than blocked assignments, it does not block the execution of the statements that follows it in listed code.
- ✓ Non-blocking are made using <= operator.
- ✓ The RHS expression of all non-blocking assignments in a block are evaluated concurrently and independent of their order with respect to each other, later the assignments are made to LHS.

target1 <= expression1 ;

target2 <= expression2 ; all starts executing at tsim = 0

target3 <= expression3 ;

Syntax

variable <= expression;
variable <= #Δt expression;
#Δt variable <= expression;

Example For simulation

```
initial
begin
#3 b <= a;       /* grab a at t=0 Deliver b at t=3.
#6 x <= b + c;  // grab b+c at t=0, wait and assign x at t=6.
end                          x is unaffected by b's change. */
```

Example For synthesis

```
always @( posedge clk)
begin
   Z<=Y; Y<=X; // shift register
   y<=x; z<=y; //also a shift register.
End
```

- ✓ Use <= to transform a variable into itself.

  reg G[7:0];

  always @(posedge clk)

  G <= { G[6:0], G[7]}; // End around rotate 8-bit register.

- ✓ The following example shows interactions between blocking and non-blocking for simulation.
- ✓ Do not mix the two types in one procedure for synthesis.

Example for simulation only

```
initial begin

a=1; b=2; c=3; x=4;

#5 a = b + c;          // wait for 5 units, then grab b,c and execute a=2+3.

d = a;                 // Time continues from last line, d=5 = b+c at t=5.

x <= #6 b + c;         // grab b+c now at t=5, don't stop, make x=5 at t=11.

b <= #2 a;             /* grab a at t=5 (end of last blocking statement).

                          Deliver b=5 at t=7. previous x is unaffected by b change. */

y <= #1 b + c;         // grab b+c at t=5, don't stop, make x=5 at t=6.

#3 z = b + c;          // grab b+c at t=8 (#5+#3), make z=5 at t=8.

w <= x                 // make w=4 at t=8. Starting at last blocking assignment.
```

Blocking and Nonblocking Examples

```
 initial
 begin
  A= 1 ;
  B = 0 ;
   . . .
   A <= B ; // uses B = 0
   B <= A ; // uses A = 1
end
----------------------------
 initial
 begin
  A= 1 ;
 B = 0 ;
  . . .
A = B ; // uses B = 0
B = A ; // uses A = 0
end
-----------------
initial  begin
 A= 1 ;
B = 0 ;
 . . .
 B <= A ; // uses A = 1
 A <= B ; // uses B = 0
end
-------------------------------
initial  begin
 A= 1 ;
 B = 0 ;
 . . .
 B = A ; // uses A = 1
A = B ; // uses B = 1
End
```

BLOCK STATEMENTS

The block statements are a means of grouping two or more statements together so that they act syntactically like a single statement. There are two types of blocks in the Verilog HDL: Sequential block, also called begin- end block .The sequential block shall be

delimited by the keywords begin and end. The procedural statements in sequential block shall be executed sequentially in the given order.Parallel block, also called fork-join block The parallel block shall be delimited by the keywords fork and join. The procedural statements in parallel block shall be executed concurrently.

Sequential Blocks

- ✓ The **begin** - **end** keywords:
- ✓ Group several statements together.
- ✓ Cause the statements to be evaluated in sequentially (one at a time).
  - Any timing within the sequential groups is relative to the previous statement.
  - Delays in the sequence accumulate (each delay is added to the previous delay)
- ✓ Block finishes after the last statement in the block.

A sequential block shall have the following characteristics:

Statements shall be executed in sequence, one after another

Delay values for each statement shall be treated relative to the simulation time of the execution of the previous statement

Control shall pass out of the block after the last statement executes

EXAMPLE:

```
begin
areg = breg;
creg = areg; // creg stores the value of breg
end
```

Parallel Blocks

The **fork - join** keywords:

A parallel block shall have the following characteristics:

- ✓ Statements shall execute concurrently.
- ✓ Delay values for each statement shall be considered relative to the simulation time of entering the block.
- ✓ Delay control can be used to provide time-ordering for assignments.
- ✓ Control shall pass out of the block when the last time-ordered statement executes
- ✓ Group several statements together.
- ✓ Cause the statements to be evaluated in parallel (all at the same time).
  - Timing within parallel group is absolute to the beginning of the group.
  - Block finishes after the last statement completes(Statement with high delay,

it can be the first statement in the block).

EXAMPLE:

```
fork
@enable_a
begin
#ta wa = 0;
#ta wa = 1;
#ta wa = 0;
end
@enable_b
begin
#tb wb = 1;
#tb wb = 0;
#tb wb = 1;
end
join
```

Timing Controls

Delay Control, Not synthesizable

This specifies the delay time units before a statement is executed during simulation. A delay time of zero can also be specified to force the statement to the end of the list of statements to be evaluated at the current simulation time.

Syntax

#delay statement;

Example

```
#5 a = b + c;     // evaluated and assigned after 5 time units

#0 a = b + c;     // very last statement to be evaluated
```

Three types of delay specification are allowed for procedural assignments.

Regular Delay

It is specified by simply placing a non-zero number with # to the left of a procedural statements. It delays the execution of statement by specified number of time units relative to the time when statement is encountered in the activity flow.

#3 tran = pas1 & pas2 ;

Inter Assignmnet Delay

This is the most common delay used - sometimes also referred to as inter-assignment delay control.

EXAMPLE

#10 q = x + y;
It simply waits for the appropriate number of timesteps before executing the command.

Intra-Assignment Delay Control

With this kind of delay, the value of x + y is stored at the time that the assignment is executed,but this value is not assigned to q until after the delay period, regardless of whether or not x or y have changed during that time.

EXAMPLE

q = #10 x + y;

Zero Delay

This delay specification is used to ensure that a statement is executed in the last,after all other statements are executed in that simulation time. It is specified by placing #0 before the assignment (to left most)

```
tran1 = pas11 & pas12 ;#0 tran2 = pas21 & pas22 ; // executed last
 tran3 = pas31 & pas32 ;
```

Event Based Timing Control:

An event is the change in the value on register or a net.events can be utilized to trigger execution of a statement or a block of statements.These are 4 types of event based timing control:regular event control,named event control,event OR control and level sensitive timing control.

Regular event control

The @symbol is used to specify an event control.Statements can be executed on changes in signal value or at a positive or negative transition of the signal.The keyword posedge is used for a positive transition.

Example

@(clock) q = d; //q = d is executed whenever signal clock changes value

@(posedge clock) q = d;//q = d is executed whenever signal clock does a positive transition (0 to 1,x or z,x to 1,z to 1)

Named event Control

Verilog provides the capability to declare an event and then trigger and recongnize the occurrence of that event .The event does not hold any data.A named event is declared by the keyword event.An event is triggered by the symbol →.The triggering of the event is recognized by the symbol @.

Example

```
//example of a data buffer storing data after the last packet of data has arrived.
Event received_data;     //define a event called received_data
always@(posedge clock) //check at each positive clock edge
begin
if(last_data_packet)    //if this is the last data packet
          -->received_data; //trigger the event received_data
end
always@(received_data) //Await triggering of event received_data when event is triggered ,store all   four packets of received data
data_buf = {data_pkt[0],data_pkt[1],data_pkt[2],data_pkt[3};]
```

Event OR Control

Sometimes a transition on any one of multiple signal or events can trigger the execution of a statement or a block of statements.This is expressed as an OR of events or signals.The list of events or signals expressed as an OR is also know as a sensitivity list.The keyword **or** is used to specify multiple triggers ,as shown in below example.

```
Example
//A level sensitive latch with asynchronous reset
always@(reset or clock or d)
begin                          //wait for reset or clock or d to change
if(reset)
q = 1'b0;                       //if reset signal is high ,set q to 0
else if(clock)
q = d;
end
```

Level Triggered

Activity flow is suspended(but not terminated) until a condition is “TRUE”. Keyword *wait* is used for this event control mechanism

```
wait (condition)
proc_statement
```

Procedural Statement is executed if condition is true

```
wait (enable) reg_a = reg_b ;
```

syntax

```
wait (<expression>) <statement>;
```

Example

```
while (mem_read == 1'b1) begin
 wait (data_ready) data = data_bus;
 read_ack = 1;
end
```

Verilog Loop Statements

Loop statements are used to control repeated execution of one or more statements. There are 4 types of looping statements in Verilog:

```
forever statement;
repeat(expression) statement;
while(expression) statement;
for(initial_assignment; expression; step_assignment) statement;
```

We can combine more than one statements using begin -- end block in an looping instruction. Looping statements should be used within a procedural block.

Forever Loop:

The forever instruction continuously repeats the statement that follows it. Therefore, it should be used with procedural timing controls (otherwise it hangs the simulation). Consider this example:

```
initial
begin
   clk = 0;
   forever #5 clk = ~clk;
end
```

Repeat Loop:

Repeats the following instruction for specified times. The number of executions is set by the expression or constant value. If expression evaluates to high impedance or unknown, then statement will not be executed.

```
initial
begin
  x = 0;
  repeat( 16 )
  begin
     #2 $display("y= ", y);
     x = x + 1;
  end
end
```

while Loop:

while loop repeats the statement until the expression returns true. If starts with false value, high impedance or unknown value, statement will not be executed.

```
initial
begin
  x = 0;
  while( x <= 10 )
  begin
    #2 $display("y= ", y);
    x = x + 1;


  end
end
```

for Loop:

Executes initial_assignment once when the loop starts, Executes the statement or statement group as long as the expression evaluates as true and executes the step_assignment at the end of each pass through the loop.

for(initial_assignment; expression; step_assignment) statement;

Syntax is similar to C language except that **`begin--end`** is used instead of **`{--}`** to combine more than one statements. Remember that we don't have ++ operator in Verilog.

for(I = 0;i<=10;i++);

case

The case statement allows a multipath branch based on comparing the expression with a list of case choices.Statements in the default block executes when none of the case choice comparisons are true (similar to the else block in the if ... else if ... else). If no comparisons , including default, are true, synthesizers will generate unwanted latches.

Good practice says to make a habit of puting in a default whether you need it or not.

If the defaults are dont cares, define them as ‘x’ and the logic minimizer will treat them as don’t cares.

Case choices may be a simple constant or expression, or a comma-separated list of same.

Syntax

```
 case (expression)
 case_choice1:
begin
... statements ...
end
case_choice2:
begin
... statements ...
end
... more case choices blocks ...
```

```
default:
begin
... statements ...
end
endcase
```

Example

```
case (alu_ctr)

2’b00: aluout = a + b;

2’b01: aluout = a - b;

2’b10: aluout = a & b;

default: aluout = 1’bx; // Treated as don’t cares for

end
```

Example

```
case (x, y, z)
2’b00: aluout = a + b; //case if x or y or z is 2’b00.
2’b01: aluout = a - b;
2’b10: aluout = a & b;
default: aluout = a | b;
endcase
```

casex

In casex(a) the case choices constant “a” may contain z, x or ? which are used as don’t cares for comparison. With case the corresponding simulation variable would have to match a tri-state, unknown, or either signal. In short, case uses x to compare with an unknown signal. Casex uses x as a don’t care which can be used to minimize logic.

Syntax :

same as for case statement

Example

```
casex (a)
2'b1x: msb = 1;              // msb = 1 if a = 10 or a = 11
                             // If this were case(a) then only a=1x would match.

default: msb = 0;

endcase
```

casez

Casez is the same as casex except only ? and z (not x) are used in the case choice constants as don't cares. Casez is favored over casex since in simulation, an inadvertent x signal, will not be matched by a 0 or 1 in the case choice.

Syntax :

same as for case statement

Example

```
casez (d)
3'b1??: b = 2'b11; // b = 11 if d = 100 or greater
3'b01?: b = 2'b10; // b = 10 if d = 010 or 011
default: b = 2'b00;
endcase
```

Task :Not Synthesizable

Tasks are used in all programming languages, generally known as procedures or sub routines. Many lines of code are enclosed in task....end task brackets. Data is passed to the task, the processing done, and the result returned to a specified value. They have to be specifically called,

with data in and outs, rather than just " wired in" to the general netlist. Included in the main body of code they can be called many times, reducing code repetition.

- ✓ task are defined in the module in which they are used. it is possible to define task in separate file and use compile directive "include to include the task in the file which instantiates the task.

- ✓ task can include timing delays, like posedge, negedge, # delay.

- ✓ task can have any number of inputs and outputs.
- ✓ The variables declared within the task are local to that task. The order of declaration within the task defines how the variables passed to the task by the caller are used.
- ✓ task can take drive and source global variables, when no local variables are used.
- ✓ When local variables are used, it basically assigned output only at the end of task execution.
- ✓ task can call another task or function.
- ✓ task can be used for modeling both combinational and sequential logic.
- ✓ A task must be specifically called with a statement, it cannot be used within an expression as a function can.

Syntax

- ✓ task begins with keyword task and end's with keyword endtask
- ✓ input and output are declared after the keyword task.
- ✓ local variables are declared after input and output declaration.

Syntax

task task_name;
input [msb:lsb] input_port_list;
output [msb:lsb] output_port_list;
reg [msb:lsb] reg_variable_list;
parameter [msb:lsb] parameter_list;
integer [msb:lsb] integer_list;
... statements ...

Endtask

Example

module alu (func, a, b, c);
input [1:0] func;

```
input [3:0] a, b;
output [3:0] c;
reg [3:0] c; // so it can be assigned in always block
task my_and;
input[3:0] a, b;
output [3:0] andout;
integer i;
begin
for (i = 3; i >= 0; i = i - 1)
andout[i] = a[i] & b[i];
end
endtask
always @(func or a or b) begin
case (func)
2'b00: my_and (a, b, c);
2'b01: c = a | b;
2'b10: c = a - b;
default: c = a + b;
endcase
end
endmodule
```

Automatic(Re-entrant)Tasks

Tasks are normally static in nature.All declared items are statically allocated and they are shared across all uses of the task executing concurrently.Therefore if a task is called concurrently from two places in the code ,these task calls will operate on the same task variables.It is highly likely that the result of such an operation will be incorrect.To avoid this problem ,a keyword **automatic** is added in front of the task keyword to make the task re-entrant.Such tasks are called automatic tasks.

All items declared inside automatic tasks are allocated dynamically for each invocation.Eachtask call operations in an independent space.Thus ,the task calls operate on independent copies of the task variables.This result is correct operation .It is recommended that automatic tasks be used if there is a chance that a task might be called concurrently from two locations in the code.

Example

```
module top
reg [15:0] cd_xor,ef_xor;
 reg [15:0] c,d,e,f;
….
task automatic bitwise_xor;
output [15:0] ab_xor;  //output from the task
input [15:0] a,b;   //inputs to the task
begin
  #delay ab_and = a & b;
```

```
  ab_or = a | b;
  ab_xor = a ^ b;
end
endtask
always @(posedge clk) //these two always blocks will call the bitwise _xor task
                      //concurrently at each positive edge of clock.However ,since the
                      //task is re-entrant these concurrent calls will work correctly.
 bitwise_xor(ef_xor, e,f);
….
always @ (posedge clk2)
    bitwise_xor(cd_xor,c,d);
…
…
endmodule
```

Function

Functions are declared within a module, and can be called from continuous assignments, always blocks, or other functions. In a continuous assignment, they are evaluated when any of its declared inputs change. In a procedure, they are evaluated when invoked.

Functions describe combinational logic, and by do not generate latches. Thus an **if** without an **else** will simulate as though it had a latch but synthesize without one. This is a particularly bad case of synthesis not following the simulation.

It is a good idea to code functions so they would not generate latches if the code were used in a procedure.

Functions are a good way to reuse procedural code, since modules cannot be invoked from within a procedure.

Function Declaration

A function declaration specifies the name of the function, the width of the function return value, the function input arguments, the variables (reg) used within the function, and the function local parameters and integers.

Syntax, Function Declaration

```
function [msb:lsb] function_name;
input [msb:lsb] input_arguments;
reg [msb:lsb] reg_variable_list;
parameter [msb:lsb] parameter_list;
integer [msb:lsb] integer_list;
... statements ...
Endfunction
```

Example

```
Module foo2(cs,in1,in2,ns)
Input [1:0]cs;
Input in1,in2;
Outrput ns;
Function [1:0] generate_next_state;
Input [1:0] current_state;
Input input1,input2;
Reg[1:0] current_state;
//input1 causes  0→1 transition
//input2 causes  1→2 transition
//2→0 illegal and unknow state go to 0;
Begin
Case(current state)
2'h0:next-state = input1 ? 2'h1 : 2'h0;
2'h1:next -state = input2 ? 2'h2 : 2h1;
2'h2:next-state = 2'h0;
Default:next-state = 2'ho;
Endcase
Generate-next-state = next-state;
End
Endfunction//generate next-state
Assign ns = generate-next-state(cs,in1,in2);
Endmodule
```

example

```
function [7:0] my_func; // function return 8-bit value
input [7:0] i;
reg [4:0] temp;
integer n;
temp= i[7:4] | ( i[3:0]);
my_func = {temp, i[[1:0]};
endfunction
```

Function Return Value

When you declare a function, a variable is also implicitly declared with the same name as the function name, and with the width specified for the function name (The default width is 1-bit). At least one statement in the function must assign the function return value to this variable.

Function Call

A function call is an operand in an expression. A function call must specify in its terminal list all the input parameters.

Function Rules

The following are some of the general rules for functions:

- ✓ Functions must contain at least one input argument.
- ✓ Functions cannot contain an inout or output declaration.
- ✓ Functions cannot contain time controlled statements (#, @, wait).
- ✓ Functions cannot enable tasks.
- ✓ Functions must contain a statement that assigns the return value to the implicit function name register.

Automatic (Recursive)Functions

Function are normally used non-recursively.If a function is called concurrently from two locations,the results are non-determinstic because both calls operate on the same variable space.

However ,the keyword automatic can be used to declare a recursive (automatic)function where all function declarations are allocated dynamically for each recursive call.Each call to an automatic function operates in an independent variable space.Automatic function items cannot be accessed by hierarchical references.Automatic functions can be invoked through the use of their hierarchical name.

Example

```
//define a factorial with a recursive function
Module top;
…
//define the function
Function automatic integer factorial;
Input [31:0] oper;
Integer I;


Begin
If (operand >= 2)
  Factorial = factorial (oper -1) * oper;//recursive call
Else
 Factorial = 1;
End
Endfunction
//call the function
Integer result;
Initial
Begin
Result = factorial(4);
$display(“factorial of 4 is %0d”,result);
End
…
…
endmodule
```

Procedural Continuous Assignments

Procedural Continuous assignments behave differently.They are procedural statements which allow values of expressions to be driven continuously onto registers or nets for limited periods.

Procedural continuous assignments override existing assignments to a register or net.They provide an useful extension to regular procedural assignment statement.

Assign and deassign

The keywords assign and deassign are used to express the first type of procedural continuous assignments.The left hand side of procedural continuous assignments can only be a register or a concatenation of registers.It canot be a part or bit select of a net or an array of registers.Procedural continuous assignments override the effect of regular procedural assignments.Procedural continuous assignments are normally used for controlled periods of time.

Example

```
module edge_dff(q,qbar,d,clk,reset)
output q,qbar ;
input d,clk,reset;
reg q,qbar;
always@(negedge clk)
begin
    q = d;
    qbar = ~d;
end
always@(reset)
  if(reset)
begin                       //if reset is high ,override regular assignments to q with the new values,
                            using procedural continuous assignment.


   assign q = 1'b0;
   assign qbar = 1'b1;
end
else begin          //If reset goes low ,remove the overriding  values by  deassign the registered.
                    After this the regular assignments q = d and qbar = ~d  will be able to
                     change the registers on   the next negative edge of clock
     deassign q;
     deassign qbar;
end
endmodule
```

force and release

force and release are used to express the second from of the procedural continuous assignments.They can be used to override assignments on both the registers and nets.

Force and release statements are typically used in the interactive debugging process.where certain registers or nets are forced to a value and the effect on other registers and nets is noted.force and release statements not be used inside design blocks.

Force and release on register

Force on a register overrides any procedural assignments or procedural continuous

assignments on the register until the register is released.The register variables will continue to store the forced value after being released but can then be changed by a future procedural assignment.

Example

```
Module stimulus;
….
Edge_dff dff (q,qbar,d,clk,reset)
….
Initial                    //these statements force value of 1 on dff.q between time 50 and 100 ,regardless of the actual output of the edge_dff.
begin
  #50 force dff .q = 1'b1;     //force value of q to 1 at time 50
#50 release dff.q;                //release the value of q at time 100
end
…
Endmodule
```

Force and release on nets

Force on nets overrides any continuous assignments until the net is release.The net will immediately return to its normal driven value when it is released.A net can be forced to an expression or a value .

Example

```
Module top;
…
assign out = a & b &c;
…..
Initial
   #50 force out = a | b & c;
  #50 release out;
end
…
endmodule
```

Value Change Dump:

A value change dump (VCD) is an ASICC file that contains information about simulation time, scope and signal definitions and signal value changes in the simulation run. All signals or a selected set of signals in a design can be written to a VCD file during simulation.

System tasks are provide for selecting module instances or module instance signals to dump($dumpvars),name of VCD file ($dumpfile),starting and stopping the dump process ($dumpon , $dumpoff).

$dumpvars:

A variation on $monitor is the $dumpvars system task. This language feature was added as a means of dumping the state changes of a large number of signals in the model. It always dumps the specified signals to a file in a predefined format which was chosen to minimize the space requirements of the file while still using an ASCII encoding. The file format is called VCD, for Value Change Dump. Using the dumpfile, you have (nearly) all the information available about the changes in signal values. However, you need an external program to interpret the VCD file, which can be quite large.

This task causes every value change of its arguments to be logged into a Value-Change-Dump file. Each entry includes the time of the value change and the new value. From this data, a complete history of the simulation can be obtained for the arguments dumped.

The general format is:

$dumpvars(levels, arg1, arg2, ...);

where: levels is a number

argi is a module instance identifier

a net or register identifier

If the arguments are omitted, then all nets and registers in the model are dumped. Otherwise, the levels argument indicates how many levels of the hierarchy are to be dumped.

0 - dump all levels below any given module instance

1 - dump just the nets and registers in the given module instance

2 - dump nets and registers in the module instance and in the modules it instantiates

```
$dumpvars;                    // dumps all signals in the model
$dumpvars(0,top.m1);          // dump all signals in top.m1 and all module  // instances below it
$dumpvars(1, r1, r2, n1);     //dump only signals r1, r2, and n1
```

Simulation Mechanics

Verilog is a hardware description language but it is also a simulation language. The behavior which is described using Verilog can be reproduced by a Verilog simulator. The principal difference between the behavior exhibited by the model under simulation and the real hardware that the model represents is the time that the model's operation takes is much different from the time the real hardware will take. This difference can be orders of magnitude.

A simulation model, which is what a Verilog hardware description is, is made up of a collection of simultaneous activities, or processes. These processes exhibit some behavior over time, possibly interacting with each other. (In formal modeling terminology, the processes each have a set of states which they transition between. The trail of states is called a state trajectory. The union of all the sets of states is called the state space. Any given simulation run produces a single state trajectory out of the multitude of possible ones.)

A process is composed of a set of discrete actions, each one taking place at a single point in time. These actions are called events, and this whole method is called discrete event simulation. So a process goes from event to event, each event takes place instantaneously, and time may pass in between events.

1)Simulation Mechanics-->Time

Time is kept as a relative value, and is global to the entire set of processes (the model). We usually call this simulation time. In Verilog, simulation time is a 64-bit unsigned integer (time is always positive), and it can be obtained by the $time system function. The amount of time which passes between different events in a process affects the interaction with other processes. The total of all the delays in a given path through a process will determine how long that process takes.

Example:

The following Verilog code takes 90 time units to execute:

```
initial begin
      x = 0;
   #10 x = x + 1;
   #20 x = x + 2;
   #30 x = x + 3;
   #20 x = x + 2;
   #10 x = x + 1;
End
```

When simulated, x takes on different values at the different time instants:

Time	Value of x
0	0
10	1
30	3
60	6
80	8
90	9

2) Time--->starts at 0

When a simulation begins, the value of simulation time is 0. There is nothing particularly important about this choice of starting value, and 0 is the obvious choice.

3) Time-->Delays are relative

Time always advances. Like in real life, it can't back up. Also as in real life, time is relative. Though there is an absolute time kept in the simulator, in a sense it is irrelevant to the model. What counts in the definition of a process is how long something takes to happen (a delay). Consequently, delays are always specified as relative values, to be interpreted as "now + delay_value".

4) Time--->Dimensionless

It is important to emphasize that simulation time is an abstract concept. It really is just a mapping of integers to events in a monotonically increasing order. That is to say, a delay value of "10" in Verilog is not 10 picoseconds, or 10 nanoseconds, or 10 seconds, or 10 years. It could be any of those, or none of them -- it all depends on an interpretation external to the model.

There is a way in Verilog to associate time units to the numbers which are used for delay values (see timescale in Chapter 6), but that is simply an interpretation placed on the abstract notion of simulation time. You may like to think of time delays in terms of familiar units, but to the simulator, time is just a set of integers, starting at 0.

5) Time--->64bits

In Verilog, the simulation time is a 64-bit quantity. When the simulation time is interpreted as very small real time units, like femtoseconds, 64 bits is necessary to represent a reasonable amount of real time. With 64 bits, 18,447 seconds can be represented if the smallest unit is femtoseconds. If a 32 bit quantity was used for time, then only 4.3 microseconds could be represented.

1)Simulation Mechanics-->Delays

Delays are specified in several ways in Verilog. Delays are introduced with the "#" character. A delay can be assigned to a net driver (either a gate, primitive, or continuous assignment), or it can be assigned to the net directly.

For example, the following are all legal delay specifications:

```
assign #10 net1 = ra + rb;
xor #3 xo1(a, b, c);
wire #(4,2) control_line;
```

A delay can also be specified in procedural statements, causing time to pass before the execution of the next statement.

```
always
        begin
#period/2       clk = ~clk;
        end
```

In a procedural delay, the delay value may be an arbitrary expression. By contrast, a delay in a declarative statement must be able to be evaluated at compile time, which is another way of saying it must be a constant or constant-valued expression.

2) Delays(multiple)-->3delays(#(rise_delay, fall_delay, turnoff_delay)

Delays that are associated with nets in declarations or continuous assignments can have multiple values. There can be one, two, or three delay values specified:

```
#(rise_delay, fall_delay, turnoff_delay)
```

These are used when the value of the net makes the corresponding change:

rise_delay 0, x, or z -> 1

fall_delay 1, x, or z -> 0

turnoff_delay 0, 1, or x -> z

The delay values are interpreted positionally. That is, the first one is the rise delay, the second is the fall delay, and the third is the turnoff delay. When the net value becomes x, then the smallest of the three delay values is used.

3) Delays(multiple)--2delays(#(rise_delay, fall_delay)

Some nets are not expected to take a z value, so it is not necessary to specify the turnoff delay. For these nets, you can simply use the rise and fall delays. If, by chance, the net does make a transition to z (or to x), the delay used will be the smaller of the two which are specified.

Similarly, most of the gate primitives cannot drive a z value, so it does not make sense to supply a turnoff delay for them. For these gates, the syntax specifies that you can only provide two delay values (the syntax description uses the term `delay2`, instead of `delay3`). The following gates take only two delay values:

```
and, nand, or, nor, xor, xnor, buf, not
```

4) Delays-->1delay

If only one delay is specified, then it is used for all value transitions. Any place where a multiple delay value can be used it is also permissible to use a single delay value.

Single delay values are always used in procedural code.

1)Simulation Mechanics-->Events

Verilog falls in the class of discrete event simulation languages. Other languages in this class are GPSS, Simulate, and Simscript, as well as VHDL and ISP'. All of these languages share a common simulation paradigm, which is based on instantaneous events with time elapsing between them.

The fundamental data structure in simulators for these languages is the Future Event List, or just the event list. This is a list which contains pointers to events and associates a time with them. The time values in this list are absolute time values. I.e. in the list, an entry will indicate that event A should happen at time t. With an ordered list like this, the simulation operation is simply

Take the first event off the list

If the event's time is greater than the current time, advance the current time to the event's time.

Do the event.

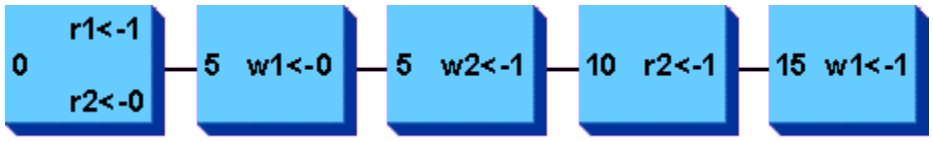
The following code

```
module top;
    wire w1, w2;
    reg r1, r2;

    assign #5 w1 = r1 & r2,
        w2 = r1 | r2;
```

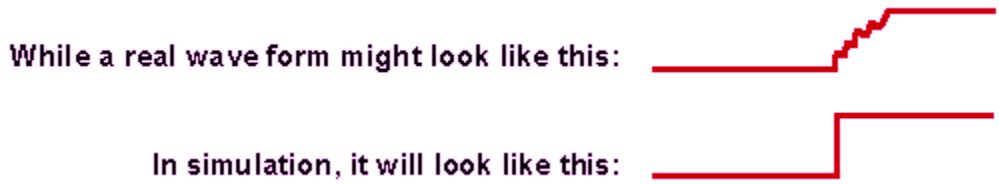
```
        initial begin
            r1 = 1; r2 = 0;
        #10   r2 = 1;
        end
    endmodule
```

would result in an event list which would look like this



2) Events-->Action occurs at time of instants

It is important to keep in mind that actions only actually happen at discrete time instants. When one of the above assignments takes place, time is stopped. It will move on after all assignments (or events in general) at that time have been accomplished. An event which in real life takes some amount of time to occur is idealized to occur only at a single instant. If the process of occurring is important to the model, then more than one event must be used to delineate it (usually a start and a stop event, though you could have mid-process events as well).



3)Simulation Mechanics-->Events-->Specific

Events can be identified explicitly in the source of a model. The most common use of events is to cause a process to wait for an event to occur before proceeding. An event is identified by the "@" character. For example,

```
initial begin
  ...
  @(posedge clk) x = f(y);
  ...
end
```

The `@(posedge clk)` is called an event control. When execution gets to that statement, it will wait until the clock makes a transition from 0 to 1 before the assignment statement will be executed.

1)Simulation Mechanics-->Concurrency and Parallelism

Hardware is parallel by nature. The Verilog Hardware Description Language captures that behavior, as it is a parallel language. The language allows models to be constructed of concurrent, asynchronous processes. These processes are parallel, in the sense that they execute at the same time, and have no inherent ordering which is defined by the language semantics.

However, an implementation of Verilog, and here we mean a simulator, does not necessarily reflect truly parallel operation. Simulation on a single processor computer, which is the norm, must emulate parallel behavior. At best, a unit-processor simulator can reproduce just one of many possible trajectories through the simulation space of the model.

2) Concurrency and Parallelism-->Processes-->Concurrent and Asynchronous

A typical Verilog model contains many separate processes. These include procedural blocks -- always and initial blocks -- as well as continuous assigns and primitive instances. Each of these can be thought of as an independent process.

Processes are concurrent and asynchronous, but there are features in the language with which processes can be synchronized. These include time delays, event controls, and value propagation.

Here is an example of two processes which are concurrent and independent.

```
 initial begin                  initial begin
   r1 = f(x);                     s1 = f(a);
#d1 r2 = f(y);                    #d2 s2 = f(b);
end                             end
```

These two processes both begin at time 0, but because the delays d1 and d2 may be variable, you cannot tell which one will finish first without knowing the values of d1 and d2 at the time of execution. They are independent because neither one affects the execution of the other.

We could decouple these two processes even more strongly by using a wait event control, as follows.

```
 initial begin                  initial begin
   r1 = f(x);                     s1 = f(a);
wait(x!=y);                      #d2 ;
   r2 = f(y);                     s2 = f(b);
end                             end
```

Here, you cannot determine when the r2 assignment will happen without determining when x and y will have different values.

3) Concurrency and Parallelism-->Event in Processes:Partially Ordered

Processes consist of a collection of events. In most cases, the events within a single process are completely ordered (this is not true when using the fork statement, however!). That is, each event can be said to be before or after every other event within that process.

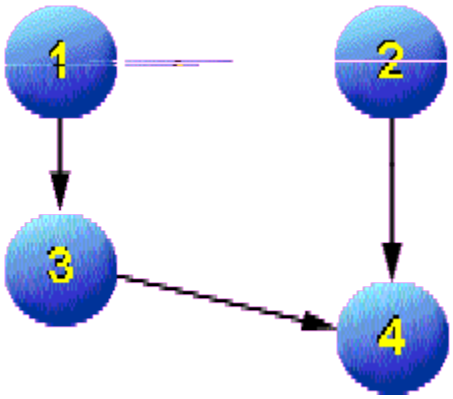
However, events in different processes are only partially ordered. That is, there may be some events for which you cannot say whether one precedes the other or not. Here is an example of two processes which are concurrent and have events which are only partially ordered:

```
always @(posedge clock) begin
state1 = newstate1;
    #10 newstate1 = func(state1, in1);
     end
```

```
always @(posedge clock) begin
  state2 = newstate2;
    #11 newstate2 = func(state2, in2);
        end
```

In this example, each always block is a separate process. Both begin at the same instant of simulated time, namely when the clock rises. However, it is not specified which one will begin first. What is guaranteed is that both state1 and state2 will have their new values before simulation proceeds past the time instant of the rising clock edge. It is also guaranteed that the two assignments to newstate1 and newstate2 will be executed at different time instants. If we call the clock edge tr, then newstate1 will be assigned at tr+10 and newstate2 will be assigned at tr+11. Thus, there is no guaranteed ordering in the first part of these two processes, but there is a guaranteed ordering in the second part.

Graphically, the event ordering looks like this:



Events 1 and 2 are unordered, while events 3 and 4 are ordered with respect to each other.

4) Concurrency and Parallelism->Defined ve.UndefinedBehavior(races)

Because some events are unordered, when they are executed (at the same instant of simulated time), it is undefined which one will execute first. What this means is that it is permissable for a simulator to execute them in either order it chooses. Two different simulators may execute the events in a different order, or the same simulator may execute them in a different order at different times during the simulation. Or, going further, a truly parallel simulator could execute them both at the same time, with either one finishing first, due solely to chance.

The implication of all this is that you had better not write Verilog code which has a different result depending on the order of execution of simultaneous, unordered events. This is known generally as a race condition, and it occurs when one event samples a data value, another event changes the data value, and the two events are unordered with respect to each other.

A classic example of a race is as follows:

```
always @(posedge clock)        always @(posedge clock)
  dff1 = f(x);                   dff2 = dff1;
```

This attempt at a pipeline doesn't work, because the value of dff2 may be either the old or the new value of dff1.

1)Simulation Mechanics--> Initializations

When simulation starts, all of the nets and registers in the model must have some initial value. Since all nets and registers have a minimum of 4 data values, and one of them is the unknown value x, it is natural for all initial values to be x.

There is a distinction between initialization time and time 0, Initialization time is when the time is 0 and no events have occurred. The first events will occur at time 0, at least one for each procedural block in the model. All nets and registers have their initial values before the first event executes.

2) Initilization-->Nets and Register

While we normally think of a net as having a value, the value is derived from the values of whatever drivers there are on that net. For initialization, this actually makes a

difference, since the drivers are the things which have x as their initial values, and the net resolution function works normally. So, at initialization, if a net has any drivers attached to it, the net will take on the value of the driver (x), unless there is a resolution function which overrides it.

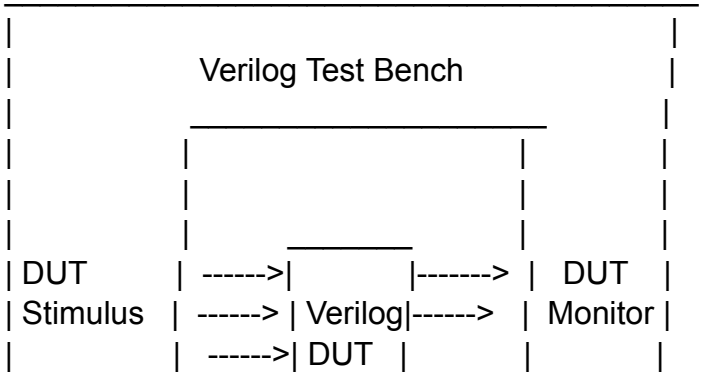
Possible initial values for a net:

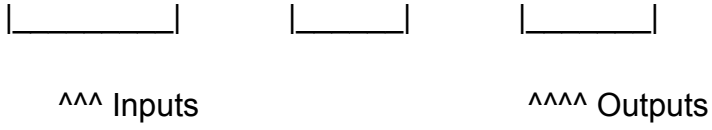
x	the net has 1 or more drivers
z	the net has no drivers
0 or 1	the net is supply0 or supply1

Registers are simpler than nets. They always start out x and will keep that value until the first procedural assignment yields a non-x value.

The Basic Testbench

The most basic test bench is comprised of the following set of items:





1. A device under test, called a DUT. This is what our testbench is testing.
2. A set of stimulus for your DUT. This can be simple, or complex.
3. A monitor, which captures or analyzes the output of your DUT.
4. You need to connect the inputs of the DUT to the testbench.
5. You need to connect the outputs of the DUT to the testbench.

TestBench

For writing testbench it is important to have the design specification of "design under test" or simply DUT. Specs need to be understood clearly and test plan is made, which basically documents the test bench architecture and the test scenarios (test cases) in detail.

Example : Counter

Let's assume that we have to verify a simple 4-bit up counter, which increments its count when ever enable is high and resets to zero, when reset is asserted high. Reset is synchronous to clock.

Code for Counter

```
module counter (clk, reset, enable, count);
  input clk, reset, enable;
  output [3:0] count;
  reg [3:0] count;
  always @ (posedge clk)
  if (reset == 1'b1)
     count <= 0;
  else if ( enable == 1'b1)
     count <= count + 1;
endmodule
```

Test Plan

We will write self checking test bench, but we will do this in steps to help you understand the concept of writing automated test benches.DUT is instantiated in testbench, and testbench will contain a clock generator, reset generator, enable logic generator, compare logic, which basically calculate the expected count value of counter and compare the output of counter with calculated value.

Test Cases

- ✓ Reset Test : We can start with reset deasserted, followed by asserting reset for few clock ticks and deasserting the reset, See if counter sets its output to zero.
- ✓ Enable Test : Assert/ deassert enable after reset is applied.
- ✓ Random Assert/ deassert of enable and reset.

We can add some more test cases, but then we are not here to test the counter, but to learn how to write test bench.

Writing TestBench

First step of any testbench creation is to creating a dummy template which basically declares inputs to DUT as reg and outputs from DUT as wire, instantiate the DUT as shown in code below. Note there is no port list for the test bench.

```
module counter_tb;
  reg clk, reset, enable;
  wire [3:0] count;
  counter U0 ( .clk (clk), .reset (reset), .enable (enable),  .count (count) );
endmodule
```

Next step would be to add clock generator logic, this is straight forward, as we know how to generate clock. Before we add clock generator we need to drive all the inputs to DUT to some know state as shown in code below.

Test Bench with Clock gen

```
module counter_tb;
 reg clk, reset, enable;
 wire [3:0] count;
 counter U0 (.clk (clk),  .reset (reset), .enable (enable),  .count (count)  );
 initial
 begin
 clk = 0;
 reset = 0;
 enable = 0;
 end
 always
    #5 clk = !clk;
endmodule
```

Initial block in verilog is executed only once, thus simulator sets the value of clk, reset and enable to 0, which by looking at the counter code (of course you will be referring to the the DUT specs) could be found that driving 0 makes all this signals disabled. There are many ways to generate clock, one could use forever loop inside a initial block as an alternate to above code. You could add parameter or use `define to control the clock frequency. You may writing complex clock generator, where we could introduce PPM (Parts per million, clock width drift), control the duty cycle. All the above

depends on the specs of the DUT and creativity of a "Test Bench Designer".
Test Bench continues...

```
module counter_tb;
 reg clk, reset, enable;
 wire [3:0] count;
 counter U0 ( .clk (clk),  .reset (reset),  .enable (enable),  .count (count)  );
 initial

 begin
    clk = 0;
    reset = 0;
    enable = 0;
 end
 always
    #5 clk = !clk;




 initial
 begin
    $dumpfile ("counter.vcd");
    $dumpvars;
 end
 initial
 begin
   $display("\t\ttime,\tclk,\treset,\tenable,\tcount");
   $monitor("%d,\t%b,\t%b,\t%b,\t%d",$time,
   clk,reset,enable,count);
 end
 initial
 #100 $finish;



 //Rest of testbench code after this line
endmodule
```

$dumpfile:

It is used for specifying the file that simulator will use to store the waveform, that can be used later to view using waveform viewer. (Please refer to tools section for freeware version of viewers.) $dumpvars basically instructs the Verilog compiler to start dumping all the signals to "counter.vcd".

$display :

It is used for printing text or variables to std out (screen), \t is for inserting tab. Syntax is same as printf. Second line $monitor is bit different, $monitor keeps track of changes to the variables that are in the list (clk, reset, enable, count). When ever anyone of them changes, it prints their value, in the respective radix specified.

$finish:

It is used for terminating simulation after #100 time units (note, all the initial, always blocks start execution at time 0)

Adding Reset Logic

Once we have the basic logic to allow us to see what our testbench is doing, we can next add the reset logic, If we look at the testcases, we see that we had added a constraint that it should be possible to activate reset anytime during simulation. To achieve this we have many approaches, but I am going to teach something that will go long way. There is something called 'events' in Verilog, events can be triggered, and also monitored to see, if a event has occurred.

Lets code our reset logic in such a way that it waits for the trigger event "reset_trigger" to happen, when this event happens, reset logic asserts reset at negative edge of clock and de-asserts on next negative edge as shown in code below. Also after de-asserting

the reset, reset logic triggers another event called "reset_done_trigger". This trigger event can then be used at some where else in test bench to sync up.

Code of reset logic

```
event reset_trigger;
 event reset_done_trigger;
 initial begin
   forever begin
     @ (reset_trigger);
     @ (negedge clk);
     reset = 1;
     @ (negedge clk);
     reset = 0;
     -> reset_done_trigger;
   end
 end
```

Adding test case logic

Moving forward, lets add logic to generate the test cases, ok we have three testcases as in the first part of this tutorial. Lets list them again .

>> Reset Test : We can start with reset deasserted, followed by asserting reset for few clock ticks and deasserting the reset, See if counter sets its output to zero.

>> Enable Test : Assert/deassert enable after reset is applied.

>> Random Assert/deassert of enable and reset.

Repeating it again "There are many ways" to code a test case, it all depends on the creativity of the Test bench designer. Lets take a simple approach and then slowly build upon it.

Test Case # 1 : Asserting/ Deasserting reset

In this test case, we will just trigger the event reset_trigger after 10 simulation units.

```
initial
begin: TEST_CASE
  #10 -> reset_trigger;
end
```

Test Case # 2 : Asserting/ Deasserting enable after reset is applied.

In this test case, we will trigger the reset logic and wait for the reset logic to complete its operation, before we start driving enable signal to logic 1.

```
initial

begin: TEST_CASE
  #10 -> reset_trigger;
  @ (reset_done_trigger);
  @ (negedge clk);
  enable = 1;
  repeat (10) begin
    @ (negedge clk);
  end
  enable = 0;
end
```

Test Case # 3 : Asserting/Deasserting enable and reset randomly.

In this testcase we assert the reset, and then randomly drive values on to enable and reset signal.

```
initial
begin : TEST_CASE
  #10 -> reset_trigger;
  @ (reset_done_trigger);
  fork begin
    repeat (10) begin
      @ (negedge clk);
      enable = $random;
    repeat (10) begin
      @ (negedge clk);
      reset = $random;
    end
  end
end
```

Well you might ask, are all this three test case exist in same file, well the answer is no. If we try to have all three test cases on one file, then we end up having race condition due to three initial blocks driving reset and enable signal. So normally, once test bench coding is done, test cases are coded separately and included in testbench as `include directive as shown below.

(There are better ways to do this, but you have to think how you want to do it).

If you look closely all the three test cases, you will find that, even through test case execution is not complete, simulation terminates. To have better control, what we can do is, add a event like "terminate_sim" and execute $finish only when this event is triggered. We can trigger this event at the end of test case execution. The code for $finish now could look as below.

```
event terminate_sim;
initial begin
@ (terminate_sim);
  #5 $finish;
end
```

and the modified test case #2 would like.

```
initial
begin: TEST_CASE
  #10 -> reset_trigger;
  @ (reset_done_trigger);
  @ (negedge clk);
```

```
    enable = 1;
    repeat (10) begin
      @ (negedge clk);
    end
    enable = 0;
    #5 -> terminate_sim;
  end
```

Second problem with the approach that we have taken till now it that, we need to manually check the waveform and also the output of simulator on the screen to see if the DUT is working correctly. Part IV shows how to automate this.

Adding compare Logic

To make any testbench self checking/automated, first we need to develop model that mimics the DUT in functionality. In our example, to mimic DUT,it going to be very easy, but at times if DUT is complex, then to mimic the DUT will be a very complex and requires lot of innovative techniques tomake self checking work.

```
  reg [3:0] count_compare;
  always @ (posedge clk)
  if (reset == 1'b1)
    count_compare <= 0;
  else if ( enable == 1'b1)
    count_compare <= count_compare + 1;
```

Once we have the logic to mimic the DUT functionality, we need to add the checker logic, which at any given point keeps checking the expected value with the actual value. Whenever there is any error, it print's out the expected and actual value, and also terminates the simulation by triggering .
the event "terminate_sim".

```
  always @ (posedge clk)
 if (count_compare != count) begin
    $display ("DUT Error at time %d", $time);
    $display (" Expected value %d, Got Value %d",
    count_compare, count);
    #5 -> terminate_sim;
 end
```

Now that we have the all the logic in place, we can remove $display and $monitor, as our testbench have become fully automatic, and we don't require to manually verify the DUT input and output. Try changing the count_compare = count_compare +2, and see how compare logic works.

This is just another way to see if our testbench is stable.

User Defined Primitives(UDP)

Verilog provides a standard set of primitives,such as and,nand,or,nor and not as a part of the language.These are also commonly know as built-in primitves.Designer 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.

There are two types of UDPS:Combinational and Sequential.

*Combinational UDPs are defined where the output is solely determined by a logical combination of the inputs.example:- 4-to-1 multiplexer.

*Sequential UDPs take the value of the current inputs and current output to determine the value of the next output.The value of the output is also the internal state of the UDP.Good example of sequential UDPs are latches and flipflops.

We can include timing information along with these UDP to model complete ASIC library models.

Syntax:-

UDP begins with reserve word primitive and ends with end primitive. Ports/terminals of primitive should follow. This is similar to what we do for module definition. UDPs should be defined outside module and endmodule.

```
//This code shows how input/output ports
// and primitve is declared
primitive udp_syntax (
a, // Port a
b, // Port b
c, // Port c
d  // Port d
);



output a;
input b,c,d;
// UDP function code here
endprimitive
```

In the above code, udp_syntax is the primitive name, it contains ports a, b,c,d.

The formal syntax of the UDP definition is as follows:

Old Style Port List

```
primitive primitive name (output, input, input, ... );
 output terminal_declaration;
  input terminal declarations;
  reg output terminal;
```

```
 initial output terminal = logic value;
table
 table entry;
 table entry;
endtable
endprimitive
```

ANSI-C Style Port List (added in Verilog-2001)

```
primitive primitive_name
( output reg = logic_value terminal_declaration,
input terminal_declarations );
table
 table_entry;
 table_entry;
  endtable
endprimitive
```

UDP Rules:-

1-UDPs can take only scalar input terminals (1bit).Multiple input terminal are permitted.

2-UDPs can have only one scalar output terminal(1bit).The output terminal must always appear first in the terminal list.Multiple output terminals are not allowed.

3-In the declarations section ,the output terminal is declared with the keyword output.since sequential UDPs store state ,the output terminal must also be declared as a reg.

4-The inputs are declared with the keyword input.

5-The state in a sequential UDP can be initialized with an initial statement is optional.A 1 bit value is assigned to the output,which is declared as reg

6-The state table entries can contain values 0,1 or x.UDPs donot handle z values .z values passed to a UDP are treated as x values.

7-UDPs defined at the same level as modules.UDPs cannot be defined inside modules.They can be instantiated only inside modules.UDPs are instantiated exactly like gate primitives.

8-UDPs donot support inout ports.

Both Combinational and sequential UDPs must follow the above rules.

Body:-

Functionality of primitive (both combinational and sequential) is described inside a table, and it ends with reserved word 'endtable' as shown in the code below. For sequential UDP, we can use initial to assign an initial value to output.

```
// This code shows how UDP body looks like
primitive udp_body (
a, // Port a
```

```
b, // Port b
c  // Port c
);
output a;
input b,c;
// UDP function code here
// A = B | C;
table
 // B  C    : A
    ?  1    : 1;
    1  ?    : 1;
    0  0    : 0;
endtable
endprimitive
```

Note: An UDP cannot use 'z' in the input table

TestBench to check the above UDP

```
`include "udp_body.v"
module udp_body_tb();
reg b,c;
wire a;
udp_body udp (a,b,c);

initial begin
  $monitor(" B = %b C = %b  A = %b",b,c,a);
  b = 0;
  c = 0;
  #1 b = 1;
  #1 b = 0;
  #1 c = 1;


  #1 b = 1'bx;
  #1 c = 0;
  #1 b = 1;
  #1 c = 1'bx;
  #1 b = 0;
  #1 $finish;
end
endmodule
```

Table:-

Table is used for describing the function of UDP. Verilog reserved word table marks the start of table and reserved word endtablemarks the end of table.

Each line inside a table is one condition; when an input changes, the input condition is

matched and the output is evaluated to reflect the new change in input.

Initial:-

Initial statement is used for initialization of sequential UDPs. This statement begins with the keyword 'initial'. The statement that follows must be an assignment statement that assigns a single bit literal value to the output terminal reg.

```
primitive udp_initial (a,b,c);
output a;
input b,c;
reg a;
// a has value of 1 at start of sim
initial a = 1'b1;
table
// udp_initial behaviour
endtable
endprimitive
```

Symbols:-

TruthTable Symbol Definition

0 ---------->logic 0 on input or output

1 ---------->logic 1 on input or output

x or X ---------->unknown on input or output

? ----------> don't care if an input is 0, 1, or X

b or B ---------->don't care if and input is 0 or 1

(vw) ----------> input transition from logic v to logic w
e.g.: (01) represents
a transition from 0 to 1

r or R ----------> rising input transition: same as (01)

f or F ---------->falling input transition: same as (10)

p or P ----------->positive input transition: (01), (0X) or (X1)

n or N ----------> negative input transition: (10), (1X) or (X0)

*

----------->Any possible input transition: same as (??)

Combinational UDPs:-

In combinational UDPs, the output is determined as a function of the current input. Whenever an input changes value, the UDP is evaluated and one of the state table rows is matched. The output state is set to the value indicated by that row. This is similar to condition statements: each line in table is one condition.

Combinational UDPs have one field per input and one field for the output. Input fields and output fields are separated with colon. Each row of the table is terminated by a semicolon. For example, the following state table entry specifies that when the three inputs are all 0, the output is 0.

```
primitive udp_combo (.....);


table
0 0 0 : 0;
...
endtable


endprimitive
```

The order of the inputs in the state table description must correspond to the order of the inputs in the port list in the UDP definition header. It is not related to the order of the input declarations.

Each row in the table defines the output for a particular combination of input states. If all inputs are specified as x, then the output must be specified as x. All combinations that are not explicitly specified result in a default output state of x.

Example:-

In the below example entry, the ? represents a don't-care condition. This symbol indicates iterative substitution of 1, 0, and x. The table entry specifies that when the inputs are 0 and 1, the output is 1 no matter what the value of the current state is.

You do not have to explicitly specify every possible input combination. All combinations that are not explicitly specified result in a default output state of x.

It is illegal to have the same combination of inputs, specified for different outputs.

```
// This code shows how UDP body looks like
primitive udp_body (
a, // Port a
b, // Port b
c  // Port c
);
```

```
output a;
input b,c;
// UDP function code here
// A = B | C;
table
 // B  C    : A
    ?  1    : 1;
    1  ?    : 1;
    0  0    : 0;
endtable
endprimitive
```

Testbench to check above UDP

```
`include "udp_body.v"
module udp_body_tb();

reg b,c;
wire a;

udp_body udp (a,b,c);

initial begin
  $monitor(" B = %b C = %b  A = %b",b,c,a);
  b = 0;
  c = 0;
  #1 b = 1;
  #1 b = 0;
  #1 c = 1;
  #1 b = 1'bx;
  #1 c = 0;
  #1 b = 1;
  #1 c = 1'bx;
  #1 b = 0;
  #1 $finish;
end
endmodule
```

Level Sensitive Sequential UDP

Level-sensitive sequential behavior is represented in the same way as combinational

behavior, except that the output is declared to be of type reg, and there is an additional field in each table entry. This new field represents the current state of the UDP.

The output is declared as reg to indicate that there is an internal state. The output value of the UDP is always the same as the internal state.

A field for the current state has been added. This field is separated by colons from the inputs and the output.

Sequential UDPs have an additional field inserted between the input fields and the output field, compared to combinational UDP. This additional field represents the current state of the UDP and is considered equivalent to the current output value. It is delimited by colons.

```
primitive udp_seq (.....);
table
0 0 0 : 0 : 0;
…..
endtable
endprimitive
```

Example:-

```
primitive udp_latch(q, clk, d) ;
output q;
input clk, d;
reg q;
table
  //clk d    q     q+
  0     1  : ? :   1   ;
  0     0  : ? :   0   ;
  1     ?  : ? :   -   ;
endtable
endprimitive
```

Edge-Sensitive UDPs:-

In level-sensitive behavior, the values of the inputs and the current state are sufficient to determine the output value. Edge-sensitive behavior differs in that changes in the output are triggered by specific transitions of the inputs.

As in the combinational and the level-sensitive entries, a ? implies iteration of the entry over the values 0, 1, and x. A dash (-) in the output column indicates no value change.

All unspecified transitions default to the output value x. Thus, in the previous example, transition of clock from 0 to x with data equal to 0 and current state equal to 1 result in the output q going to x.

All transitions that should not affect the output must be explicitly specified. Otherwise, they will cause the value of the output to change to x. If the UDP is sensitive to edges of any input, the desired output state must be specified for all edges of all inputs.

```
primitive udp_sequential(q, clk, d);
output q;
input clk, d;
reg q;
table
```

```
// obtain output on rising edge of clk
// clk        d       q      q+
   (01)        0   :   ?   :   0   ;
   (01)        1   :   ?   :   1   ;
   (0?)        1   :   1   :   1   ;
   (0?)        0   :   0   :   0   ;
// ignore negative edge of clk
   (?0)        ?   :   ?   :   -   ;
// ignore d changes on steady clk
   ?      (??)     :   ?   :   -   ;
 endtable
endprimitive
```

Example UDP with initial:-

```
primitive udp_sequential_initial(q, clk, d);
output q;
input clk, d;
reg q;
initial begin
q = 0;
end
table
// obtain output on rising edge of clk
// clk        d       q      q+
   (01)        0   :   ?   :   0   ;
   (01)        1   :   ?   :   1   ;
   (0?)        1   :   1   :   1   ;
   (0?)        0   :   0   :   0   ;
// ignore negative edge of clk
   (?0)        ?   :   ?   :   -   ;
// ignore d changes on steady clk
   ?      (??)     :   ?   :   -   ;
 endtable
endprimitive
```

Verilog Coding Style:-

A Good Verilog Coding Style is a prime requirement in every Design, for predictable results, and reusing the codes for various applications.

- ✓ Indent the code so that the code is readable.
- ✓ Ensure the code is generic across all technologies and not specified to a single technology.
- ✓ Ensure the consistent signal names across the hierarchy.
- ✓ Ensure that each RTL file contains file headers as comment comprising of { Name of the RTL, Version tag, Authors involved and their email-id's , information about the parameters used in the constructs, last edited, Bug-fixes history, Brief about what the code is performing}.
- ✓ Ensure only one verilog statement per line.
- ✓ Ensure that speed-critical logics are in seperate module.
- ✓ Maximize the usuage of gated-clocks, as this saves power.
- ✓ Prefer clock-generating circuitary in seperate module files.
- ✓ Ensure one port declaration on one line, followed by comments about the port.
- ✓ Preserve port order.
- ✓ Declare the internal nets in the design.
- ✓ Limitation on the line length{for example:not more than 75}.
- ✓ Avoid the usage of 'include construct.
- ✓ Ensure that the power-downed signals are in an known state.
- ✓ Ensure that the coding style does not contains combinational feedback loops.
- ✓ Never assign "x" value to the signals.
- ✓ Maximize the usage of parameters instead of text-macros.
- ✓ Use parameters for state-encoders.
- ✓ Ensure that the design coding does not generate tri-state logic.
- ✓ Use the concept of base+ offset for coding address busses.
- ✓ Tie the un-used bits to a known value.
- ✓ Connect based on port-names while instantiating.
- ✓ Ensure that no access of nets and variables outside the module context.
- ✓ Avoid using ports of type "inout" in the design.
- ✓ Ensure that the latches are transparent during scan-phase.

- ✓ Ensure that the PLL bypass is present to verify with out PLL's.
- ✓ Minimize top-level glue logic coding style.
- ✓ Ensure only one module is defined in a file.
- ✓ Ensure that the active low signals are suffix'ed with '_b' and clocks named with domain names for example clock_a, clock_b, "_z" for high impedance signals, "_o" for output signals, "_i" for input signals, "_ns" for state-machine next states, "_se" for scan-enable signals.
- ✓ Prefer using case statements, instead of using long if and else statements.
- ✓ Ensure that default statement is used in the case statements.
- ✓ While designing FSM behaviours , prefer three always blocks(1. register definitions. 2. Defining next-logic. 3. Defining outputs).
- ✓ Ensure that the unused module inputs are driven and not floating.
- ✓ Ensure that the design has enough amount of spare-logic to incorporate the last minute design bug-fixes.
- ✓ Ensure that the same test-benches are used for RTL and gate level simulations.
- ✓ Ensure that the design does not contain initial blocks and delay elements.
- ✓ Ensure that the whole design is resetable to a known state. No internal logic generated asynchronous resets.
- ✓ Ensure the reset is not an late arriving signal w.r.t clocks due to dense reset trees.
- ✓ In case if the desired reset is an asynchronous signal, ensure that it is part of the sensitivity list as according to the verilog if the reset signal is not part of the sensitivity list then it is inferred as a synchronous reset.
- ✓ As in the case of an synchronous reset design, assist the synopsys synthesis tool with the comment //synopsys sync_set_reset_n "rst_n" , As for the synthesis tool reset is also treated as any other signal in case of synchronous-resets. In case of synchronous reset designs, in order to maintain the pulse-width of the signal, a small counter type of circuit is required.
- ✓ Extra care is required while designing with synchronous resets and also designs having clock-gating logic. Poor coding styles can make the design non-resetable as it may be blocking the clocks reaching .
- ✓ Ensure that a situation doesn't trigger an asynchronous set and asynchronous resets for the same flip-flop.

- ✓ Ensure that the de-assertion of the asynchronous resets is not close to the clock-edge so that the flip-flop could be prevented from being entering in to metastable state.
- ✓ Ensure that a recovery and removal timing checks are performed for resets to ensure the optimal behaviour.
- ✓ Even though the design is working with asynchronous reset style, Ensure that the resets are synchronized passing through synchronizers. Ensure that these synchronizer flops are not part of the scan-chain stitching in the design.
- ✓ Ensure that the resets are not used as data or clock signals in the design.
- ✓ Ensure that the reset is priority over any other signal in the case of an if-else construct.
- ✓ Ensure that the signals which cross the clock-domains are synchronized.
- ✓ Ensure that the design does not contain while statements.
- ✓ Understanding the signals & functionalities based on the signal names (like For Clock definitions naming the signal with "clk", naming resets with "res", and to know the whether the signal is active-high or active low triggerred(For example : resets which are active-low are named with res_n).
- ✓ Ensure that clock-dividers are by-passed during scan stitching.
- ✓ Coding style which is generic and re-usuable.
- ✓ Avoid using verilog UDP in the design.
- ✓ Coding style which will not have simulation and synthesis mis-match results.
- ✓ Coding style friendly to Synthesis and not having constructs which synthesis tool cannot understand.
- ✓ Coding style which ensures the correct digital logic will be generated after performing synthesis. The basic digital mismatches could be infering (For example : Latch for a Flip-flop requirement, Priority Encoder for a Mux Design requirement).
- ✓ Coding Style which are DFT(Design for Test) friendly.
- ✓ Usage of Blocking(=) & Non-blocking(<=) statements and Steps to prevent Raceconditions .
- ✓ Race conditions are situtations, the order of execution isn't always guaranteed within verilog.
- ✓ Use blocking statements for coding combinational logic.
- ✓ Use non-blocking statements for coding sequential logic.
- ✓ Never mix blocking and non-blocking in a same procedural block.

- ✓ Never assign a value to the same variable multiple times in different always blocks. Even though it is a non-blocking statements the design is prone to race-condition.
- ✓ To get to know more depth in to the concept Refer the paper:

- ✓ Be Sensitive about the Sensitivity list
- ✓ Synthesis tools assumes to generate combinational logic from an always block, if it does not contain statemens like "posedge" or "negedge".
- ✓ Ensure that a complete list of signals present in the sensitivity list in an always block, thereby the simulation results for a pre-synthesis versus the post-synthesis match.
- ✓ Ensure only one clock per sensitivity list.
- ✓ Synopsys Full-case and parallel case directives

Synopsys full_case directive :

When the Synthesis tool(Synopsys Design Compiler), comes across with the comment (//synopsys full_case, before case statements), the tool is guided stating that though all the cases are not mentioned , these are the possible cases design can honour so the tool for the rest of the non-listed cases the synthesis tool assumes that the outputs are don't cares.

(synopsys parallel_case, before case statements), the tool is guided stating that it optimizes the logic, assuming that the case statements would match only one case , prevents the synthesize tool from optimizing the un-necessary logic.

Various ways of Coding the same logic (for example :Mux)

```
// first example with continuous assignment
wire z;
assign z = sel ? x : y;
// second example with if and else
reg output;
always @ (x or y or sel)
if (sel)
output = x;
else
output = y;
```

**Unsupported Verilog Language Constructs

Unsupported definitions and declarations

Primitive definition

time declaration

Ranges and arrays for integer declarations.

event declarations
triand, trior, tri1, tri0 and trireg net type
Ranges and arrays for integers.

Statements not supported by synthesis tool

defparam statement
initial statement
repeat statement
delay control statement
event control
wait statement
fork statement
deassign statement
force statement
release statement

Unsupported operators

Case equality and inequality operators (=== and !==)
Division and modulus operators for variables

Unsupported gate-level constructs

nmos,cmos, pmos, rnmos, rpmos, rcmos
pullup, pulldown, tranif0, tranif1, rtran, rtainf0, and rtainf1 gate type signals
**Miscellaneous constructs, such as hierarchical names within a module
**Unsupported Simulation Directives
'timescale

Verilog Basic Examples:

1)AND GATE

```
//in data flow model
module and_gate(input a,b,output y);
//Above style of declaring ports is ANSI style.Verilog2001 Feature
assign y = a & b;
endmodule
```

TestBench

```
module tb_and_gate
reg A,B;
wire Y;
and_gate a1(.a(A),.b(B),.y(Y));
```

```
//Above style is connecting by names
initial begin
A =1'b0;
B= 1'b0;
#45 $finish;
end
always #6 A=~A;
always #3B =~B;

always@(Y)
$display("time =%0t \tINPUT VALUES: \t A=%b B =%b \t output value Y=%b",
$time,A,B,Y);
endmodule
```

2)XOR GATE

```
//in Structural model

module xor_gate(input a,b,outputy);

xor x1(y,a,b); //xor is a built in primitive. While using these primitives you should follow the connection rules. First signal should be output and then inputs.

endmodule
```

TestBench

```
module tb_and_gate;

reg A,B;
wire Y;
xor_gate a1(.a(A),.b(B),.y(Y));

initial begin
A=1'b0;
B=1'b0;
#45$finish;
end

always #6A=~A;
always #3B=~B;
always@(Y)
$display("time =%0t \tINPUT VALUES: \t A=%b B =%b \t output value Y =%b",
$time,A,B,Y);

endmodule
```

3)OR GATE

```
  //in Behaviour model
   module or_gate(input a,b,output reg  y);

    always @(a,b)

    y = a |b;

endmodule
```

TestBench

```
module tb_and_gate;
reg A,B;
wire Y;

or_gate a1 (.a(A) ,.b(B),.y(Y));

initial begin
A=1'b0;
B=1'b0;
#45 $finish;
```

```
end
always#6A=~A;
always#3B=~B;
always@(Y)
$display("time =%0t \tINPUT VALUES: \t A=%b B =%b \t output value Y =%b",
$time,A,B,Y);
endmodule
```

4)Half Adder

```
module half_adder(input a,b,output(sum,carry);
assign sum= a^b;
assign carry = a & b;
endmodule
```

TestBench

```
module tb_half_adder;
reg A,B;
wire SUM,CARRY;
half_adder HA(.a(A),.b(B),.sum(SUM),.carry(CARRY))
initial begin
A =1'b0;
B=1'b0;
#45 $finish;
end
always #6A=~A;
always #3B =~B;
alway@(SUM,CARRY)
$disjplay("time =%0t \tINPUT VALUES: \t A=%b B =%b \t output value SUM =%b CARRY
=%b ",$time,A,B,SUM,CARRY);
endmodule
```

5)Full Adder

```
module full_adder(input a,b,cin,output reg sum,cout);
always @(*)begin
sum=a^b^cin;
cout=(a&b)+(b&cin)+(cin&a);
end
endmodule
```

TestBench

```
module tb_full_adder;
reg A,B,CIN;
wire SUM,COUT;
full_adder FA (.a(A) ,.b(B),.sum(SUM),.cin(CIN),.cout(COUT));
initial begin
A =1'b0;
B= 1'b0;
```

```
CIN = 1'b0;
#45 $finish;
    end

     always #6 A =~A;
     always #3 B =~B;
     always #12 CIN = ~CIN;

     always @(SUM,COUT)
     $display("time =%0t \tINPUT VALUES: \t A =%b B =%b CIN =%b \t output value SUM
      =%b COUT =%b ",$time,A,B,CIN,SUM,COUT);
endmodule
```

6)Ripple Carry Adder(Parameterized and using generate)

Verilog design

```
`include "full_adder.v"
//Full_added.v contains above defined(Full ADDER) program
module ripple_carry(a,b,cin,cout,sum);
    parameter N=4;
    input   [N-1 :0] a,b;
    input   cin;
    output [N-1:0]sum;
    output cout;
    wire [N:0]carry;
    assign carry[0]=cin;


 //generate statement without using label is verilog-2005 feature. Generate statement is verilog-2001 feature.
    genvar i;
    generate for(i=0;i<N;i=i+1) begin
    full_adder FA (.a(a[i]),.b(b[i]),.cin(carry[i]),.sum(sum[i]),.cout(carry[i+1]));
    end
    endgenerate
    assign cout = carry[N];
endmodule
```

TestBench Using $random

```
module tb_ripple_carry;
    parameter N=4;
    reg [N-1:0]A,B;
    reg CIN;
    wire [N-1:0]SUM;
    wire COUT;
    ripple_carry RCA(.a(A),.b(B),.cin(CIN),.sum(SUM),.cout(COUT));
    initial begin
       A= 4'b0000;
       B= 4'b0000;
       CIN =1'b0;
       repeat(10)
       input_generate(A,B,CIN);
       #45 $finish;
```

```
    end

    task input_generate;
    output [N-1:0]A_t,B_t;
    output CIN_t;
    begin
        #4;
        A_t = $random % 4;
        //Above statement will generate Random values from -3 to +3.
        B_t = $random % 4;
        CIN_t =$random;
    end
    endtask

    task display;
    input [N-1:0] A_td,B_td,SUM_td;
    input CIN_td,COUT_td;

    $strobe("Time =%0t \t INPUT VALUES A=%b B=%b CIN =%b \t OUTPUT VALUES
SUM =%b COUT =%b",$time,A_td,B_td,CIN_td,SUM_td,COUT_td);

    endtask




always@(SUM,A,COUT)
    $display(A,B,SUM,CIN,COUT);
endmodule
```

7)Multiplexer(2:1)

```
module mux21(
input a,b,sel,
output y);

    assign y = sel ?b:a;

endmodule
```

TestBench

```
module tb_mux21;

    reg A,B,SEL;
    wire Y;

    mux21 MUX (.a(A) ,.b(B),.sel(SEL),.y(Y));

    initial begin
        A =1'b0;
        B= 1'b0;
        SEL =1'b0;
        #45 $finish;
    end

    always #6 A =~A;
    always #3 B =~B;
    always #5 SEL =~SEL;

    always @(Y)
    $display("time =%0t \tINPUT VALUES: \t A=%b B =%b SEL =%b \t output value Y =%b
",$time,A,B,SEL,Y);

endmodule
```

8)Multiplexer(4:1)

```
module mux41(
    input i0,i1,i2,i3,sel0,sel1,
    output reg y);

    always @(*)   //It includes all Inputs. You can use this instead of specifying all inputs
in //sensivity list.Verilog-2001 Feature
    begin
       case ({sel0,sel1})




      2'b00 : y = i0;
       2'b01 : y = i1;
       2'b10 : y = i2;
       2'b11 : y = i3;
       endcase
    end

endmodule
```

TestBench

```
module tb_mux41;

   reg I0,I1,I2,I3,SEL0,SEL1;
   wire Y;

   mux41 MUX (.i0(I0),.i1(I1),.i2(I2),.i3(I3),.sel0(SEL0),.sel1(SEL1),.y(Y));

   initial begin
       I0 =1'b0;
       I1= 1'b0;
       I2 =1'b0;
       I3 =1'b0;
       SEL0 =1'b0;
       SEL1 =1'b0;
       #45 $finish;
   end

   always #2 I0 = ~I0;
   always #4 I1 =~I1;
   always #6 I2 =~I1;
   always #8 I3 =~I1;
   always #3 SEL0 = ~SEL0;
   always #3 SEL1 = ~SEL1;

   always @(Y)
   $display("time =%0t INPUT VALUES: \t I0=%b I1 =%b I2 =%b I3 =%b SEL0 =%b SEL1
=%b \t output value Y =%b ",$time,I0,I1,I2,I3,SEL0,SEL1,Y);

endmodule
```

9)Encoder(8:3)

*Disadvantage of Encoder is that at a time only one Input is active.
*Output is zero for when all inputs are zero and when enable is zero

Verilog design

```
module encoder83(
input  en,
input  [7:0]in,
output reg [2:0]out);
```

```
always@(*)
  begin
     if(!en) //Active low enable
        out = 0;
     else begin
        case ({in})
        8'b0000_0001 : out =3'b000;
        8'b0000_0010 : out =3'b001;
        8'b0000_0100 : out =3'b010;
        8'b0000_1000 : out =3'b011;
        8'b0001_0000 : out =3'b100;
        8'b0010_0000 : out =3'b101;
        8'b0100_0000 : out =3'b110;
        8'b1000_0000 : out =3'b110;
        default  : out =3'bxxx;
        endcase
     end
  end

endmodule
```

TestBench using $random and Tasks

```
module tb_encoder83;

    reg en;
    reg  [7:0]in;
    wire [2:0] out;

    encoder83 ENC (.en(en),.in(in),.out(out));

    initial begin
        en =0;
        in =0;
        repeat(10)
        random_generation(in,en);
        #45 $finish;
    end

    task random_generation;
    output [7:0]in_t;
    output en_t;
    begin
        #4;
        in_t = $random % 8;
        en_t =$random;
    end
    endtask









  task display;
    input en_t;
    input [7:0]in_t;
    input [2:0]out_t;
        $display("time =%0t \t INPUT VALUES \t en =%b in =%b \t OUTPUT VALUES out =
%b",$time,en_t,in_t,out_t);
    endtask

    always@(out)
```

```
      display(en,in,out);
endmodule
```

10)Priority Encoder(8:3)

Priority Encoder overcomes all drawbacks of encoder.
* At a time more than one input can be active, Based on priority output will come.
* "v" is a valid Indicator, it become HIGH only when at least one input is active. You can differentiate the output when enable is zero and when only LSB (in0) is active

Verilog design

```
module priority_enco(
input en,
input [3:0]in,
output reg v,
output reg [1:0]out );

    integer i;

    always@(*) begin
       if(!en) begin
            out = 2'b00;
            v =1'b0;
       end
       else
       begin :block1
           for (i=3; i>=0; i= i-1) begin
                //Priority Logic. each Time It will check Whether the MSB bit is active, If so it will break //the loop. Otherwise It will decrement and continue the same
                if (in[i]==1'b1) begin
                case (i)
                     3: begin out = 2'b11; v= 1'b1; end
                     2: begin out = 2'b10; v= 1'b1; end
                     1: begin out = 2'b01; v= 1'b1; end
                     0: begin out = 2'b00; v= 1'b1; end
                    default :begin out = 2'bxx; v= 1'bx; end
                endcase

                disable block1;
                //Disable statement is synthesizible

                end
           end

       end



  end
endmodule
```

TestBench using $random and Tasks

```
module tb_prior_enco ;

    reg en;
    reg  [2:0]in;
    wire [1:0] out;
    wire v;

    priority_enco  PRIOR_ENC (.en(en),.in(in),.out(out),.v(v));

    initial begin
        en =0;
```

```
        in =0;
        repeat(19)
        random_generation(in,en);
        #65 $finish;
    end

    task random_generation;
    output [3:0]in_t;
    output en_t;
    begin
        #4;
        in_t = $random % 4;
        en_t =$random;
    end
    endtask

    task display;
    input en_t;
    input [3:0]in_t;
    input [1:0]out_t;
    input v_t;

        $display("time =%0t \t INPUT VALUES \t en =%b in =%b \t OUTPUT VALUES out =
%b v =%b",$time,en_t,in_t,out_t,v_t);

    endtask

    always@(out)
      display(en,in,out,v);

endmodule
```

11)Decoder(8:3)

Verilog design

```
module decoder38(
    input  en,
    input  [2:0]in,
    output  reg [7:0]out);

    always@(*)
    begin
      if(!en)
          out  = 0;
      else begin
          case ({in})
          3'b000  :  out = 8'b0000_0001;
          3'b001  :  out = 8'b0000_0010;
          3'b010  :  out = 8'b0000_0100;
          3'b011  :  out = 8'b0000_1000;
          3'b100  :  out = 8'b0001_0000;
          3'b101  :  out = 8'b0010_0000;
          3'b110  :  out = 8'b0100_0000;
          3'b111  :  out = 8'b1000_0000;
          default :  out = 8'bxxxx_xxxx;
          endcase
      end
    end

endmodule
```

TestBench using $random and Tasks

```
module tb_decoder38;
```

```
    reg en_tb;
    reg  [2:0]in_tb;
    wire [7:0] out_d;
    reg  [7:0] out_tb;

    decoder38  DEC (.en(en_tb),.in(in_tb),.out(out_d));

    initial begin
       en_tb  =0;
       in_tb  =0;
       repeat(10)
       random_generation(in_tb,en_tb) ;
       #45 $finish;
    end

    //Below Block is used to generate expected outputs in Test bench only. These
outputs //are used to compare with DUT output. You have Checker task (ScoreBoard in
SV), for //that you need Reference output

    always@(in_tb,en_tb)
    begin
        if(!en_tb)
          out_tb  = 0;
        else begin
          case ({in_tb})




          3'b000  :  out_tb  = 8'b0000_0001;
          3'b001  :  out_tb  = 8'b0000_0010;
          3'b010  :  out_tb  = 8'b0000_0100;
          3'b011  :  out_tb  = 8'b0000_1000;
          3'b100  :  out_tb  = 8'b0001_0000;
          3'b101  :  out_tb  = 8'b0010_0000;
          3'b110  :  out_tb  = 8'b0100_0000;
          3'b111  :  out_tb  = 8'b1000_0000;
          default :  out_tb  = 8'bxxxx_xxxx;
          endcase
        end
    end


    task random_generation;
    output [2:0]in_t;
    output en_t;
    begin
        #4;
        in_t = $random % 3;
        en_t =$random;
    end
    endtask

    task checker;
    //In this block reference value and generated output are compared
    input [7:0]outd_t;
    input [7:0]outtb_t;
    begin
        if(outd_t === outtb_t)
            $display("time =%0t \t  DUT VALUE =%b TB VALUE =%b \tDUT and TB VALUES
ARE MATCHED ",$time,outd_t,outtb_t);
        else
             $display("time =%0t \tDUT and TB VALUES ARE NOT  MATCHED ",$time);
    end
    endtask

    always@(out_d,out_tb)
    checker(out_d,out_tb);

endmodule
```

TestBench using $random and Tasks

12)D-Latch

Verilog design

```
module d_latch(
input en,d,
output reg q);
   always@(en,d)
   begin
     if(en)



    q <= d;
   end
endmodule
```

TestBench

```
module tb_latch;
    reg en,d;
    wire q;

    d_latch DLATCH (.en(en) ,.d(d) ,.q(q));

    initial begin
        en =1'b0;
        d =1'b1;
        #45 $finish;
    end

    always #6 en =~ en;
    always #3 d =~d;

    always@( q , en )
        $display("time =%0t \t INPUT VALUES \t en =%b d =%b \t OUTPUT VALUES q=%b",
$time,en,d,q);
endmodule
```

13)D-FlipFlop(Asynchronous Reset)

Verilog design

```
module d_ff (
input clk,d,rst_n,
output reg q);
   //Here is reset is Asynchronous, You have include in sensitivity list
   always@(posedge clk ,negedge rst_n)
   begin
     if(!rst_n)
        q <= 1'b0;
     else
        q <= d;
   end
endmodule
```

TestBench

```
module tb_dff;
    reg RST_n, CLK,D;
    wire Q;
```

```
    d_ff DFF (.clk(CLK) ,.rst_n(RST_n) ,.q(Q),.d(D));



     initial begin
        RST_n = 1'b0;
        CLK =1'b0;
        D =1'b0;
        #5 RST_n = 1'b1;
        #13 RST_n = 1'b0;
        #7 RST_n = 1'b1;
        #45 $finish;
    end

    always #3 CLK = ~CLK;
    always #6 D = ~D;

    always @(posedge CLK, negedge RST_n)
    $strobe("time =%0t \t INPUD VALUES \t D =%b RST_n =%b \t OUDPUD VALUES Q =
%d",$time,D,RST_n,Q);
    //$strobe will execute as a last statement in current simulation.
endmodule
```

time =39 INPUT VALUES D =0 RST_n =1 OUTPUT VALUES Q =0

14)D-FlipFlop(Synchronous Reset)

Verilog design

```
module d_ff (
input clk,d,rst_n,
output reg q);

    //In Synchronous Reset, Reset condition is verified wrt to clk.Here It is verified at
every //posedge of clk.
    always@(posedge clk )
    begin
        if (!rst_n)
            q <= 1'b0;
        else
            q <= d;
    end

endmodule
```

TestBench

```
module tb_dff;
    reg RST_n, CLK,D;
    wire Q;

    d_ff DFF (.clk(CLK) ,.rst_n(RST_n) ,.q(Q),.d(D));



  initial begin
        RST_n = 1'b0;
        CLK =1'b0;
        D =1'b1;
        #5 RST_n = 1'b1;
        #7 RST_n = 1'b0;
        #7 RST_n = 1'b1;
```

```
        #45 $finish;
    end

    always #4 CLK = ~CLK;
    always #6 D = ~D;

    always @(posedge CLK )
    $strobe("time =%0t \t INPUT VALUES \t D =%b RST_n =%b \t OUDPUT VALUES Q =
%d",$time,D,RST_n,Q);

endmodule
```

15)T-FlipFlop

Verilog design

```
module t_ff (
input clk,t,rst_n,
output reg q);

    always@(posedge clk ,negedge rst_n)
    begin
        if (!rst_n)
            q <= 1'b0;
        else if(t)
            q <= ~q;
        else
            q <= q;
    end

endmodule
```

TestBench

```
module tb_tff;
    reg RST_n, CLK,T;
    wire Q;

    t_ff TFF (.clk(CLK) ,.rst_n(RST_n) ,.q( Q ),.t(T));

    initial begin
        RST_n = 1'b0;
        CLK=1'b0;
        T =1'b0;
        #5 RST_n = 1'b1;
        #13 RST_n = 1'b0;
        #7 RST_n = 1'b1;
        #45 $finish;


  end

    always #3 CLK = ~CLK;
    always #6 T = ~T;

    always @(posedge CLK ,negedge RST_n)
    $strobe("time =%0t \t INPUT VALUES \t T =%b RST_n =%b \t OUTPUT VALUES Q =
%d",$time,T,RST_n,Q);

endmodule
```

16)3-Bit Counter

//Used Structural Model in RTL and Behavior Model in Test bench

Verilog design

```
module t_ff(
output reg q,
```

```
input t, rst_n, clk);

always @ (posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0;
    else if (t) q <= ~q;

endmodule

          //Standard counters are designed using either T or JK F/F.

module counter (
    output [2:0] q,
    input rst_n, clk);

    wire t2;

    t_ff ff0 ( q[0], 1'b1, rst_n, clk);
    t_ff ff1 ( q[1], q[0], rst_n, clk);
    t_ff ff2 ( q[2], t2,   rst_n, clk);
    and a1 (t2, q[0], q[1]);

endmodule
```

TestBench

```
module tb_counter_3bit;
    reg clk,rst_n;
    wire [2:0] q;
    reg [2:0] count;

    counter CNTR (.clk(clk),.rst_n(rst_n),.q(q));

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

    initial



   begin
        rst_n <= 0;
        @(posedge clk);
        @(negedge clk);
        rst_n <= 1;
        repeat (10)   @(posedge clk);
        $finish;
    end

    //Below always block represents the 3-bit counter in behavior style.
    //Here it is used to generate reference output
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n)
            count <= 0;
        else
            count <= (count + 1);
    end

    always @( q ) scoreboard(count);

    //Below task is used to compare reference and generated output. Similar to score board
//in SV Test bench

    task scoreboard;
    input [2:0]count;
    input [2:0] q;
    begin
      if (count == q)
          $display ("time =%4t q = %3b count = %b match!-:)",
                              $time, q, count);
      else
          $display ("time =%4t q = %3b count = %b <-- no match",
                              $time, q, count);
```

```
    end
    endtask
endmodule
```

output

```
time =   0 q = 000 count = 000 match!-:)
time =  15 q = 001 count = 001 match!-:)
time =  25 q = 010 count = 010 match!-:)
time =  35 q = 011 count = 011 match!-:)
time =  45 q = 100 count = 100 match!-:)
time =  55 q = 101 count = 101 match!-:)
time =  65 q = 110 count = 110 match!-:)
time =  75 q = 111 count = 111 match!-:)
time =  85 q = 000 count = 000 match!-:)
time =  95 q = 001 count = 001 match!-:)
```

17. Gray code counter (3-bit) Using FSM.

It will have following sequence of states. It can be implemented without FSM also.

000
001

011
010
110
111
101
100

FSM Design IN VERILOG

There are many ways of designing FSM.Most efficient are
(i)Using Three always Block (ex: Gray code counter)
(ii)Using Two always block (Ex: divide by 3 counter)

Verilog Code

```
module greycode_counter_3bit(
    input clk,rst_n,
    output reg [2:0]count);
    reg [2:0] pr_state,nx_state;

    parameter cnt0 =3'b000,
              cnt1 =3'b001,
              cnt2 =3'b011,
              cnt3 =3'b010,
              cnt4 =3'b110,
              cnt5 =3'b111,
              cnt6 =3'b101,
              cnt7 =3'b100;

    always@(posedge clk, negedge rst_n) begin // FIRST ALWAYS BLOCK
    //This always block is used for State assignment. Sequential always block.
    if(!rst_n)
        pr_state <= cnt0;
    else
        pr_state <=nx_state;
    end

    always@(pr_state) begin  //SECOND ALWAYS BLOCK
        //this always block used for next state logic, Combinational
        case (pr_state)
        cnt0 : nx_state = cnt1;
        cnt1 : nx_state = cnt2;
        cnt2 : nx_state = cnt3;
        cnt3 : nx_state = cnt4;
        cnt4 : nx_state = cnt5;
```

```
        cnt5 : nx_state = cnt6;
        cnt6 : nx_state = cnt7;
        cnt7 : nx_state = cnt0;
        default : nx_state = cnt0;
        endcase
    end

    always@(posedge clk ,negedge rst_n) begin //THIRD ALWAYS BLOCK
        //this always block used for output assignment,Sequential
        if(!rst_n)
            count <= 3'b000;
        else begin



     case (pr_state)
            cnt0 : count <= 3'b000;
            cnt1 : count <= 3'b001;
            cnt2 : count <= 3'b011;
            cnt3 : count <= 3'b010;
            cnt4 : count <= 3'b110;
            cnt5 : count <= 3'b111;
            cnt6 : count <= 3'b101;
            cnt7 : count <= 3'b100;
            default :count <=3'bxxx;
            endcase
        end
    end
endmodule
```

TestBench

```
module tb_greycode_counter;
    reg clk,rst_n;
    wire [2:0] count;

    greycode_counter_3bit COUNTER(.clk(clk),.rst_n(rst_n),.count(count));

    initial begin
        clk =1'b0;
        rst_n = 1'b0;
        @(posedge clk);
        @(posedge clk);
        rst_n = 1'b1;
        repeat(9) @(posedge clk);
        $finish;
    end

    always #5 clk = ~clk;
    always@(count)
    $display("time =%0t \t rst_n =%b count =%b",$time,rst_n,count);

endmodule
```

output

```
time =0         rst_n =0 count =000
time =25         rst_n =1 count =001
time =35         rst_n =1 count =011
time =45         rst_n =1 count =010
time =55         rst_n =1 count =110
time =65         rst_n =1 count =111
time =75         rst_n =1 count =101
time =85         rst_n =1 count =100
time =95         rst_n =1 count =000
```

19)Divide by 2 clk

```
module div_2clk(
input clk,rst_n,
output reg clk_out);
```

```
always@(posedge clk,negedge rst_n) begin
   if(rst_n)
      clk_out <= 1'b0;
   else
      clk_out <= ~clk_out;
   end
endmodule
```

Various Examples:-

1)Following is the Verilog code for flip-flop with a positive-edge clock.

```
module flop (clk, d, q);
input  clk, d;
output q;
reg    q;

always @(posedge clk)
begin
  q <= d;
end
endmodule
```

2)Following is Verilog code for a flip-flop with a negative-edge clock and asynchronous clear.

```
module flop (clk, d, clr, q);
input  clk, d, clr;
output q;
reg    q;
always @(negedge clk or posedge clr)
begin
   if (clr)
     q <= 1'b0;
   else
     q <= d;
end
endmodule
```

3)Following is Verilog code for the flip-flop with a positive-edge clock and synchronous set.

```
module flop (clk, d, s, q);
input  clk, d, s;
```

```
output q;
  reg    q;
  always @(posedge clk)
  begin
    if (s)
      q <= 1'b1;
    else
      q <= d;
  end
  endmodule
```

4)Following is Verilog code for the flip-flop with a positive-edge clock and clock enable.

```
  module flop (clk, d, ce, q);
  input  clk, d, ce;
  output q;
  reg    q;
  always @(posedge clk)
begin
    if (ce)
    q <= d;
  end
endmodule
```

5)Following is Verilog code for a 4-bit register with a positive-edge clock, asynchronous set and clock enable.

```
  module flop (clk, d, ce, pre, q);
  input        clk, ce, pre;
  input  [3:0] d;
  output [3:0] q;
  reg    [3:0] q;
  always @(posedge clk or posedge pre)
begin
    if (pre)
      q <= 4'b1111;
    else if (ce)
      q <= d;
end
endmodule
```

6)Following is the Verilog code for a latch with a positive gate.

```
  module latch (g, d, q);
input  g, d;
output q;
```

```
  reg    q;
  always @(g or d)
begin
  if (g)
    q <= d;
end
endmodule
```

7)Following is the Verilog code for a latch with a positive gate and an asynchronous clear.

```
module latch (g, d, clr, q);
input  g, d, clr;
output q;
reg    q;
  always @(g or d or clr)
begin
  if (clr)
    q <= 1'b0;
  else if (g)
    q <= d;
end
endmodule
```

8)Following is Verilog code for a 4-bit latch with an inverted gate and an asynchronous preset.

```
module latch (g, d, pre, q);
input       g, pre;
input  [3:0] d;
output [3:0] q;
reg    [3:0] q;
always @(g or d or pre)
begin
  if (pre)
    q <= 4'b1111;
  else if (~g)
    q <= d;
end
endmodule
```

9)Following is Verilog code for a tristate element using a combinatorial process and always block.

```
module three_st (t, i, o);
input  t, i;
output o;
```

```
reg    o;
   always @(t or i)
   begin
     if (~t)
       o = i;
     else
       o = 1'bZ;
   end
   endmodule
```

10)Following is the Verilog code for a tristate element using a concurrent assignment.

```
   module three_st (t, i, o);
   input  t, i;
   output o;
     assign o = (~t) ? i: 1'bZ;
   endmodule
```

11)Following is the Verilog code for a 4-bit unsigned up counter with asynchronous clear.

```
   module counter (clk, clr, q);
   input       clk, clr;
   output [3:0] q;
   reg    [3:0] tmp;
   always @(posedge clk or posedge clr)
   begin
     if (clr)
       tmp <= 4'b0000;
     else
       tmp <= tmp + 1'b1;
   end
     assign q = tmp;
   endmodule
```

12)Following is the Verilog code for a 4-bit unsigned down counter with synchronous set.

```
   module counter (clk, s, q);
   input       clk, s;
   output [3:0] q;
   reg    [3:0] tmp;
   always @(posedge clk)
   begin
```

```
if (s)
    tmp <= 4'b1111;
  else
    tmp <= tmp - 1'b1;
end
  assign q = tmp;
endmodule
```

13)Following is the Verilog code for a 4-bit unsigned up counter with an asynchronous load from the primary input.

```
module counter (clk, load, d, q);
input       clk, load;
input  [3:0] d;
output [3:0] q;
reg    [3:0] tmp;
always @(posedge clk or posedge load)
begin
  if (load)
    tmp <= d;
  else
    tmp <= tmp + 1'b1;
end
  assign q = tmp;
endmodule
```

14)Following is the Verilog code for a 4-bit unsigned up counter with a synchronous load with a constant.

```
  module counter (clk, sload, q);
  input       clk, sload;
  output [3:0] q;
  reg    [3:0] tmp;
  always @(posedge clk)
  begin
    if (sload)
     tmp <= 4'b1010;
    else
      tmp <= tmp + 1'b1;
  end
    assign q = tmp;
endmodule
```

15)Following is the Verilog code for a 4-bit unsigned up counter with an asynchronous clear and a clock enable.

```
module counter (clk, clr, ce, q);
input       clk, clr, ce;
output [3:0] q;
reg    [3:0] tmp;
always @(posedge clk or posedge clr)
begin
  if (clr)
    tmp <= 4'b0000;
  else if (ce)
    tmp <= tmp + 1'b1;
end
  assign q = tmp;
endmodule
```

16)Following is the Verilog code for a 4-bit unsigned up/down counter with an asynchronous clear.

```
module counter (clk, clr, up_down, q);
input       clk, clr, up_down;
output [3:0] q;
reg    [3:0] tmp;
always @(posedge clk or posedge clr)
begin
  if (clr)
    tmp <= 4'b0000;
  else if (up_down)
    tmp <= tmp + 1'b1;
  else
    tmp <= tmp - 1'b1;
end
  assign q = tmp;
endmodule
```

17)Following is the Verilog code for a 4-bit signed up counter with an asynchronous reset.

```
module counter (clk, clr, q);
input              clk, clr;
output signed [3:0] q;
reg    signed [3:0] tmp;
always @ (posedge clk or posedge clr)
begin
  if (clr)
    tmp <= 4'b0000;
  else
    tmp <= tmp + 1'b1;
```

```
    end

assign q = tmp;
    endmodule
```

18)Following is the Verilog code for a 4-bit signed up counter with an asynchronous reset and a modulo maximum.

```
    module counter (clk, clr, q);
    parameter MAX_SQRT = 4, MAX = (MAX_SQRT*MAX_SQRT);
    input               clk, clr;
    output [MAX_SQRT-1:0] q;
    reg    [MAX_SQRT-1:0] cnt;
    always @ (posedge clk or posedge clr)
    begin
      if (clr)
        cnt <= 0;
      else
        cnt <= (cnt + 1) %MAX;
    end
      assign q = cnt;
    endmodule
```

19)Following is the Verilog code for a 4-bit unsigned up accumulator with an asynchronous clear.

```
    module accum (clk, clr, d, q);
    input       clk, clr;
    input  [3:0] d;
    output [3:0] q;
    reg    [3:0] tmp;
    always @(posedge clk or posedge clr)
    begin
      if (clr)
        tmp <= 4'b0000;
      else
        tmp <= tmp + d;
    end
      assign q = tmp;
    endmodule
```

20)Following is the Verilog code for an 8-bit shift-left register with a positive-edge clock, serial in and serial out.

```
      module shift (clk, si, so);
      input       clk,si;
      output       so;
```

```
    reg    [7:0] tmp;


always @(posedge clk)
    begin
      tmp    <= tmp << 1;
      tmp[0] <= si;
    end
      assign so = tmp[7];
  endmodule
```

21)Following is the Verilog code for an 8-bit shift-left register with a negative-edge clock, a clock enable, a serial in and a serial out.

```
    module shift (clk, ce, si, so);
    input       clk, si, ce;
    output       so;
    reg    [7:0] tmp;
    always @(negedge clk)
    begin
      if (ce) begin
        tmp    <= tmp << 1;
        tmp[0] <= si;
      end
    end
      assign so = tmp[7];
  endmodule
```

22)Following is the Verilog code for an 8-bit shift-left register with a positive-edge clock, asynchronous clear, serial in and serial out.

```
    module shift (clk, clr, si, so);
    input       clk, si, clr;
    output       so;
    reg    [7:0] tmp;
    always @(posedge clk or posedge clr)
    begin
      if (clr)
        tmp <= 8'b00000000;
      else
        tmp <= {tmp[6:0], si};
    end
      assign so = tmp[7];
  endmodule
```

23)Following is the Verilog code for an 8-bit shift-left register with a positive-edge

clock, a synchronous set, a serial in and a serial out.

```
module shift (clk, s, si, so);


input       clk, si, s;
   output      so;
   reg    [7:0] tmp;
   always @(posedge clk)
   begin
     if (s)
       tmp <= 8'b11111111;
     else
       tmp <= {tmp[6:0], si};
   end
     assign so = tmp[7];
endmodule
```

24)Following is the Verilog code for an 8-bit shift-left register with a positive-edge clock, a serial in and a parallel out.

```
module shift (clk, si, po);
input       clk, si;
output [7:0] po;
reg    [7:0] tmp;
always @(posedge clk)
begin
  tmp <= {tmp[6:0], si};
end
  assign po = tmp;
endmodule
```

25)Following is the Verilog code for an 8-bit shift-left register with a positive-edge clock, an asynchronous parallel load, a serial in and a serial out.

```
module shift (clk, load, si, d, so);
input       clk, si, load;
input  [7:0] d;
output      so;
reg    [7:0] tmp;
always @(posedge clk or posedge load)
begin
  if (load)
   tmp <= d;
  else
    tmp <= {tmp[6:0], si};
end
```

```
    assign so = tmp[7];
endmodule
```

26)Following is the Verilog code for an 8-bit shift-left register with a positive-edge clock, a synchronous parallel load, a serial in and a serial out.

```
module shift (clk, sload, si, d, so);
input       clk, si, sload;
input  [7:0] d;
output      so;
reg    [7:0] tmp;
always @(posedge clk)
begin
  if (sload)
    tmp <= d;
  else
    tmp <= {tmp[6:0], si};
end
  assign so = tmp[7];
endmodule
```

27)Following is the Verilog code for an 8-bit shift-left/shift-right register with a positive-edge clock, a serial in and a serial out.

```
  module shift (clk, si, left_right, po);
  input       clk, si, left_right;
  output      po;
  reg    [7:0] tmp;
  always @(posedge clk)
  begin
    if (left_right == 1'b0)
      tmp <= {tmp[6:0], si};
    else
      tmp <= {si, tmp[7:1]};
  end
    assign po = tmp;
endmodule
```

28)Following is the Verilog code for a 4-to-1 1-bit MUX using an If statement.

```
  module mux (a, b, c, d, s, o);
  input       a,b,c,d;
  input  [1:0] s;
  output      o;
```

```
    reg         o;
    always @(a or b or c or d or s)
    begin
      if (s == 2'b00)


  o = a;
      else if (s == 2'b01)
        o = b;
      else if (s == 2'b10)
        o = c;
      else
        o = d;
    end
  endmodule
```

29)Following is the Verilog Code for a 4-to-1 1-bit MUX using a Case statement.

```
    module mux (a, b, c, d, s, o);
    input       a, b, c, d;
    input  [1:0] s;
    output      o;
    reg         o;
    always @(a or b or c or d or s)
    begin
      case (s)
        2'b00   : o = a;
        2'b01   : o = b;
        2'b10   : o = c;
        default : o = d;
      endcase
    end
endmodule
```

30)Following is the Verilog code for a 3-to-1 1-bit MUX with a 1-bit latch.

```
    module mux (a, b, c, d, s, o);
    input       a, b, c, d;
    input  [1:0] s;
    output      o;
    reg         o;
    always @(a or b or c or d or s)
    begin
      if (s == 2'b00)
        o = a;
```

```
  else if (s == 2'b01)
    o = b;
  else if (s == 2'b10)
    o = c;
end
endmodule
```

31)Following is the Verilog code for a 1-of-8 decoder.

```
module mux (sel, res);
input  [2:0] sel;
output [7:0] res;
reg    [7:0] res;
always @(sel or res)
begin
  case (sel)
    3'b000  : res = 8'b00000001;
    3'b001  : res = 8'b00000010;
    3'b010  : res = 8'b00000100;
    3'b011  : res = 8'b00001000;
    3'b100  : res = 8'b00010000;
    3'b101  : res = 8'b00100000;
    3'b110  : res = 8'b01000000;
    default : res = 8'b10000000;
  endcase
end
endmodule
```

32)Following Verilog code leads to the inference of a 1-of-8 decoder.

```
  module mux (sel, res);
  input  [2:0] sel;
  output [7:0] res;
  reg    [7:0] res;
  always @(sel or res) begin
    case (sel)
      3'b000  : res = 8'b00000001;
      3'b001  : res = 8'b00000010;
      3'b010  : res = 8'b00000100;
      3'b011  : res = 8'b00001000;
      3'b100  : res = 8'b00010000;
      3'b101  : res = 8'b00100000;
      // 110 and 111 selector values are unused
      default : res = 8'bxxxxxxxx;
    endcase
```

```
    end
  endmodule
```

33)Following is the Verilog code for a 3-bit 1-of-9 Priority Encoder.

```
    module priority (sel, code);
    input  [7:0] sel;


output [2:0] code;
    reg    [2:0] code;
    always @(sel)
    begin
      if (sel[0])
        code = 3'b000;
      else if (sel[1])
        code = 3'b001;
      else if (sel[2])
        code = 3'b010;
      else if (sel[3])
        code = 3'b011;
      else if (sel[4])
        code = 3'b100;
      else if (sel[5])
        code = 3'b101;
      else if (sel[6])
        code = 3'b110;
      else if (sel[7])
        code = 3'b111;
      else
        code = 3'bxxx;
    end
  endmodule
```

34)Following is the Verilog code for a logical shifter.

```
    module lshift (di, sel, so);
  input  [7:0] di;
    input  [1:0] sel;
    output [7:0] so;
    reg    [7:0] so;
    always @(di or sel)
    begin
      case (sel)
```

```
        2'b00   : so = di;
        2'b01   : so = di << 1;
        2'b10   : so = di << 2;
        default : so = di << 3;
      endcase
    end
endmodule
```

35)Following is the Verilog code for an unsigned 8-bit adder with carry in.

```
module adder(a, b, ci, sum);
input  [7:0] a;
input  [7:0] b;
input        ci;
output [7:0] sum;

  assign sum = a + b + ci;

endmodule
```

36)Following is the Verilog code for an unsigned 8-bit adder with carry out.

```
module adder(a, b, sum, co);
input  [7:0] a;
input  [7:0] b;
output [7:0] sum;
output       co;
wire   [8:0] tmp;

  assign tmp = a + b;
  assign sum = tmp [7:0];
  assign co  = tmp [8];

endmodule
```

37)Following is the Verilog code for an unsigned 8-bit adder with carry in and carry out.

```
module adder(a, b, ci, sum, co);
input        ci;
input  [7:0] a;
input  [7:0] b;
output [7:0] sum;
```

```
output       co;
wire   [8:0] tmp;

  assign tmp = a + b + ci;
  assign sum = tmp [7:0];
  assign co  = tmp [8];

endmodule
```

38)Following is the Verilog code for an unsigned 8-bit adder/subtractor.

```
  module addsub(a, b, oper, res);
  input        oper;
  input  [7:0] a;
  input  [7:0] b;
  output [7:0] res;
  reg    [7:0] res;
  always @(a or b or oper)
  begin
    if (oper == 1'b0)
      res = a + b;
    else
      res = a - b;
end
endmodule
```

39)Following is the Verilog code for an unsigned 8-bit greater or equal comparator.

```
  module compar(a, b, cmp);
  input  [7:0] a;
  input  [7:0] b;
  output       cmp;

    assign cmp = (a >= b) ?  1'b1 : 1'b0;

endmodule
```

40)Following is the Verilog code for an unsigned 8x4-bit multiplier.

```
module compar(a, b, res);
input  [7:0]  a;
input  [3:0]  b;
```

```
output [11:0] res;

  assign res = a * b;

endmodule
```

41)Following Verilog template shows the multiplication operation placed outside the always block and the pipeline stages represented as single registers.

```
module mult(clk, a, b, mult);
input         clk;
input  [17:0] a;
input  [17:0] b;



output [35:0] mult;
reg    [35:0] mult;
reg    [17:0] a_in, b_in;
wire   [35:0] mult_res;
reg    [35:0] pipe_1, pipe_2, pipe_3;

  assign mult_res = a_in * b_in;

always @(posedge clk)
begin
    a_in   <= a;
  b_in   <= b;
  pipe_1 <= mult_res;
  pipe_2 <= pipe_1;
  pipe_3 <= pipe_2;
  mult   <= pipe_3;
end
endmodule
```

42)Following Verilog template shows the multiplication operation placed inside the always block and the pipeline stages are represented as single registers.

```
module mult(clk, a, b, mult);
input         clk;
input  [17:0] a;
input  [17:0] b;
output [35:0] mult;
reg    [35:0] mult;
reg    [17:0] a_in, b_in;
reg    [35:0] mult_res;
```

```
reg    [35:0] pipe_2, pipe_3;
always @(posedge clk)
begin
    a_in     <= a;
  b_in     <= b;
  mult_res <= a_in * b_in;
  pipe_2   <= mult_res;
  pipe_3   <= pipe_2;
  mult     <= pipe_3;
end
endmodule
```

43)Following Verilog template shows the multiplication operation placed outside the always block and the pipeline stages represented as single registers.

```
module mult(clk, a, b, mult);
  input         clk;
  input  [17:0] a;
  input  [17:0] b;
  output [35:0] mult;
  reg    [35:0] mult;
  reg    [17:0] a_in, b_in;
  wire   [35:0] mult_res;
  reg    [35:0] pipe_1, pipe_2, pipe_3;

    assign mult_res = a_in * b_in;

  always @(posedge clk)
  begin
    a_in   <= a;
    b_in   <= b;
    pipe_1 <= mult_res;
    pipe_2 <= pipe_1;
    pipe_3 <= pipe_2;
    mult   <= pipe_3;
  end
endmodule
```

44)Following Verilog template shows the multiplication operation placed inside the always block and the pipeline stages are represented as single registers.

```
module mult(clk, a, b, mult);
```

```
input         clk;
input  [17:0] a;
input  [17:0] b;
output [35:0] mult;
reg    [35:0] mult;
reg    [17:0] a_in, b_in;
reg    [35:0] mult_res;
reg    [35:0] pipe_2, pipe_3;
always @(posedge clk)
begin
  a_in     <= a;
  b_in     <= b;
  mult_res <= a_in * b_in;
  pipe_2   <= mult_res;
  pipe_3   <= pipe_2;
  mult     <= pipe_3;


end
endmodule
```

45)Following Verilog template shows the multiplication operation placed outside the always block and the pipeline stages represented as shift registers.

```
module mult3(clk, a, b, mult);
input         clk;
input  [17:0] a;
input  [17:0] b;
output [35:0] mult;
reg    [35:0] mult;
reg    [17:0] a_in, b_in;
wire   [35:0] mult_res;
reg    [35:0] pipe_regs [3:0];

  assign mult_res = a_in * b_in;

always @(posedge clk)
begin
  a_in <= a;
  b_in <= b;
  {pipe_regs[3],pipe_regs[2],pipe_regs[1],pipe_regs[0]} <=
{mult, pipe_regs[3],pipe_regs[2],pipe_regs[1]};
end
endmodule
```

46)Following templates to implement Multiplier Adder with 2 Register Levels on Multiplier Inputs in Verilog.

```
module mvl_multaddsub1(clk, a, b, c, res);
input        clk;
input  [07:0] a;
input  [07:0] b;
input  [07:0] c;
output [15:0] res;
reg    [07:0] a_reg1, a_reg2, b_reg1, b_reg2;
wire   [15:0] multaddsub;
always @(posedge clk)
begin
  a_reg1 <= a;
  a_reg2 <= a_reg1;
  b_reg1 <= b;
  b_reg2 <= b_reg1;
end



assign multaddsub = a_reg2 * b_reg2 + c;
    assign res = multaddsub;
endmodule
```

47)Following is the Verilog code for resource sharing.

```
module addsub(a, b, c, oper, res);
input       oper;
input  [7:0] a;
input  [7:0] b;
input  [7:0] c;
output [7:0] res;
reg    [7:0] res;
always @(a or b or c or oper)
begin
  if (oper == 1'b0)
    res = a + b;
  else
    res = a - c;
end
endmodule
```

48)Following templates show a single-port RAM in read-first mode.

```
module raminfr (clk, en, we, addr, di, do);
input       clk;
```

```
input       we;
input       en;
input  [4:0] addr;
input  [3:0] di;
output [3:0] do;
reg    [3:0] RAM [31:0];
reg    [3:0] do;
always @(posedge clk)
begin
  if (en) begin
    if (we)
        RAM[addr] <= di;

   do <= RAM[addr];
  end
end
endmodule
```

49)Following templates show a single-port RAM in write-first mode.

```
module raminfr (clk, we, en, addr, di, do);
input       clk;
input       we;
input       en;
input  [4:0] addr;
input  [3:0] di;
output [3:0] do;
reg    [3:0] RAM [31:0];
reg    [4:0] read_addr;
always @(posedge clk)
begin
  if (en) begin
    if (we)
        RAM[addr] <= di;
   read_addr <= addr;
  end
end
  assign do = RAM[read_addr];
endmodule
```

50)Following templates show a single-port RAM in no-change mode.

```
module raminfr (clk, we, en, addr, di, do);
input       clk;
input       we;
input       en;
input  [4:0] addr;
input  [3:0] di;
output [3:0] do;
reg    [3:0] RAM [31:0];
reg    [3:0] do;
always @(posedge clk)
begin
  if (en) begin
    if (we)
        RAM[addr] <= di;
    else
        do <= RAM[addr];
  end
end
endmodule
```

51)Following is the Verilog code for a single-port RAM with asynchronous read.

```
module raminfr (clk, we, a, di, do);
input       clk;
input       we;
input  [4:0] a;
input  [3:0] di;
output [3:0] do;
reg    [3:0] ram [31:0];
always @(posedge clk)
begin
    if (we)
     ram[a] <= di;
end
  assign do = ram[a];
endmodule
```

52)Following is the Verilog code for a single-port RAM with "false" synchronous read.

```
module raminfr (clk, we, a, di, do);
input       clk;
input       we;
input  [4:0] a;
```

```
input  [3:0] di;
output [3:0] do;
reg    [3:0] ram [31:0];
reg    [3:0] do;
always @(posedge clk)
begin
  if (we)
    ram[a] <= di;
  do <= ram[a];
end
endmodule
```

53)Following is the Verilog code for a single-port RAM with synchronous read (read through).

```
module raminfr (clk, we, a, di, do);
input       clk;
input       we;
input  [4:0] a;
input  [3:0] di;
output [3:0] do;


reg    [3:0] ram [31:0];
reg    [4:0] read_a;
always @(posedge clk)
begin
  if (we)
    ram[a] <= di;
  read_a <= a;
end
  assign do = ram[read_a];
endmodule
```

54)Following is the Verilog code for a single-port block RAM with enable.

```
module raminfr (clk, en, we, a, di, do);
input       clk;
input       en;
input       we;
input  [4:0] a;
input  [3:0] di;
output [3:0] do;
reg    [3:0] ram [31:0];
reg    [4:0] read_a;
always @(posedge clk)
```

```
begin
  if (en) begin
    if (we)
      ram[a] <= di;
    read_a <= a;
  end
end
  assign do = ram[read_a];
endmodule
```

55)Following is the Verilog code for a dual-port RAM with asynchronous read.

```
module raminfr (clk, we, a, dpra, di, spo, dpo);
input       clk;
input       we;
input  [4:0] a;
input  [4:0] dpra;
input  [3:0] di;
output [3:0] spo;
output [3:0] dpo;
reg    [3:0] ram [31:0];
```

```
always @(posedge clk)
  begin
    if (we)
      ram[a] <= di;
  end
    assign spo = ram[a];
    assign dpo = ram[dpra];
endmodule
```

56)Following is the Verilog code for a dual-port RAM with false synchronous read.

```
module raminfr (clk, we, a, dpra, di, spo, dpo);
input       clk;
input       we;
input  [4:0] a;
input  [4:0] dpra;
input  [3:0] di;
output [3:0] spo;
output [3:0] dpo;
reg    [3:0] ram [31:0];
```

```
reg    [3:0] spo;
reg    [3:0] dpo;
  always @(posedge clk)
begin
  if (we)
    ram[a] <= di;

  spo = ram[a];
  dpo = ram[dpra];
end
endmodule
```

57)Following is the Verilog code for a dual-port RAM with synchronous read (read through).

```
module raminfr (clk, we, a, dpra, di, spo, dpo);
input       clk;
input       we;
input  [4:0] a;
input  [4:0] dpra;
input  [3:0] di;
output [3:0] spo;
output [3:0] dpo;
```

```
reg    [3:0] ram [31:0];
   reg    [4:0] read_a;
   reg    [4:0] read_dpra;
   always @(posedge clk)
   begin
     if (we)
       ram[a] <= di;
     read_a <= a;
     read_dpra <= dpra;
   end
     assign spo = ram[read_a];
     assign dpo = ram[read_dpra];
endmodule
```

58)Following is the Verilog code for a dual-port RAM with enable on each port.

```
module raminfr (clk, ena, enb, wea, addra, addrb, dia, doa, dob);
input       clk, ena, enb, wea;
```

```
input  [4:0] addra, addrb;
input  [3:0] dia;
output [3:0] doa, dob;
reg    [3:0] ram [31:0];
reg    [4:0] read_addra, read_addrb;
always @(posedge clk)
begin
  if (ena) begin
    if (wea) begin
        ram[addra] <= dia;
    end
  end
end

always @(posedge clk)
begin
  if (enb) begin
    read_addrb <= addrb;
  end
end
  assign doa = ram[read_addra];
  assign dob = ram[read_addrb];
endmodule
```

59)Following is Verilog code for a ROM with registered output.

```
module rominfr (clk, en, addr, data);
input      clk;
input      en;
input [4:0] addr;
output reg [3:0] data;
  always @(posedge clk)
begin
  if (en)
    case(addr)
      4'b0000: data <= 4'b0010;
      4'b0001: data <= 4'b0010;
      4'b0010: data <= 4'b1110;
      4'b0011: data <= 4'b0010;
      4'b0100: data <= 4'b0100;
      4'b0101: data <= 4'b1010;
      4'b0110: data <= 4'b1100;
      4'b0111: data <= 4'b0000;
      4'b1000: data <= 4'b1010;
```

```
            4'b1001: data <= 4'b0010;
            4'b1010: data <= 4'b1110;
            4'b1011: data <= 4'b0010;
            4'b1100: data <= 4'b0100;
            4'b1101: data <= 4'b1010;
            4'b1110: data <= 4'b1100;
            4'b1111: data <= 4'b0000;
            default: data <= 4'bXXXX;
          endcase
      end
      endmodule
```

60)Following is Verilog code for a ROM with registered address.

```
      module rominfr (clk, en, addr, data);
      input       clk;
      input       en;
      input [4:0] addr;
      output reg [3:0] data;
      reg   [4:0] raddr;
      always @(posedge clk)
      begin
        if (en)
          raddr <= addr;
      end
```

```
always @(raddr)
      begin
        if (en)
          case(raddr)
              4'b0000: data = 4'b0010;
              4'b0001: data = 4'b0010;
              4'b0010: data = 4'b1110;
              4'b0011: data = 4'b0010;
              4'b0100: data = 4'b0100;
              4'b0101: data = 4'b1010;
              4'b0110: data = 4'b1100;
              4'b0111: data = 4'b0000;
              4'b1000: data = 4'b1010;
              4'b1001: data = 4'b0010;
              4'b1010: data = 4'b1110;
```

```
            4'b1011: data = 4'b0010;
            4'b1100: data = 4'b0100;
            4'b1101: data = 4'b1010;
            4'b1110: data = 4'b1100;
            4'b1111: data = 4'b0000;
            default: data = 4'bXXXX;
        endcase
    end
endmodule
```

61)Following is the Verilog code for an FSM with a single process.

```
module fsm (clk, reset, x1, outp);
input       clk, reset, x1;
output      outp;
reg         outp;
reg   [1:0] state;
parameter s1 = 2'b00; parameter s2 = 2'b01;
parameter s3 = 2'b10; parameter s4 = 2'b11;
always @(posedge clk or posedge reset)
begin
  if (reset) begin
    state <= s1; outp <= 1'b1;
  end
  else begin
    case (state)



          s1: begin
              if (x1 == 1'b1) begin
                state <= s2;
          outp  <= 1'b1;
              end
              else begin
                state <= s3;
          outp  <= 1'b1;
              end
            end
        s2: begin
                state <= s4;
         outp  <= 1'b0;
            end
        s3: begin
```

```
                    state <= s4;
                outp  <= 1'b0;
                  end
              s4: begin
                    state <= s1;
                outp  <= 1'b1;
                  end
          endcase
        end
      end
    endmodule
```

63)Following is the Verilog code for an FSM with two processes.

```
    module fsm (clk, reset, x1, outp);
    input       clk, reset, x1;
    output      outp;
    reg         outp;
    reg   [1:0] state;
    parameter s1 = 2'b00; parameter s2 = 2'b01;
    parameter s3 = 2'b10; parameter s4 = 2'b11;
    always @(posedge clk or posedge reset)
    begin
      if (reset)
        state <= s1;
      else begin
        case (state)




              s1: if (x1 == 1'b1)
                    state <= s2;
                  else
                    state <= s3;
              s2: state <= s4;
              s3: state <= s4;
              s4: state <= s1;
        endcase
      end
    end
    always @(state) begin
      case (state)
        s1: outp = 1'b1;
```

```
        s2: outp = 1'b1;
        s3: outp = 1'b0;
        s4: outp = 1'b0;
      endcase
    end
  endmodule
```

63)Following is the Verilog code for an FSM with three processes.

```
    module fsm (clk, reset, x1, outp);
    input       clk, reset, x1;
    output      outp;
    reg         outp;
    reg   [1:0] state;
    reg   [1:0] next_state;
    parameter s1 = 2'b00; parameter s2 = 2'b01;
    parameter s3 = 2'b10; parameter s4 = 2'b11;
    always @(posedge clk or posedge reset)
    begin
      if (reset)
        state <= s1;
      else
        state <= next_state;
    end

    always @(state or x1)
    begin
      case (state)
        s1: if (x1 == 1'b1)
              next_state = s2;
            else
              next_state = s3;
        s2: next_state = s4;
        s3: next_state = s4;
        s4: next_state = s1;
      endcase
  end
```

64)Multiplexer Built with Always

```
module mux(f, a, b, sel);

output f;

input a, b, sel;
```

```
reg f;

always @(a or b or sel)

  if (sel) f = a;

  else f = b;

endmodule
```

65)Mux with Continuous Assignment

```
 module mux(f, a, b, sel);

output f;

input a, b, sel;

assign  f = sel ? a : b;

endmodule
```

66)Example for Counter with testbench

```
module ctr( q, clk, r );

   output [3:0] q;
   input  clk;
   input  r;



 reg [3:0] q;


   always @( posedge clk )
     if ( r )
       q = 4'b0000;
     else
       q = q + 1;

endmodule // ctr
```

```
module top();

  reg clk;
  reg r;
  wire [3:0] q;

  ctr c0( q, clk, r );
  initial
    clk = 1'b0;
  always
    #5 clk = ~clk;

  initial
    begin
      r = 1'b1;
      #15 r = 1'b0;
      #180 r = 1'b1;
      #10 r = 1'b0;
      #20 $finish;
   end
```

67)MUX Description Using If... Else Statement

```
module mux4 (sel, a, b, c, d, outmux);
input [1:0] sel;
input [1:0] a, b, c, d;
output [1:0] outmux;
reg [1:0] outmux;

always @(sel or a or b or c or d)
begin
if (sel[1])
if (sel[0])
outmux = d;
else
outmux = c;
else
if (sel[0])
outmux = b;
else
outmux = a;
end
endmodule
```

68)MUX Description Using Case Statement

```
module mux4 (sel, a, b, c, d, outmux);
input [1:0] sel;
input [1:0] a, b, c, d;
output [1:0] outmux;
reg [1:0] outmux;

always @(sel or a or b or c or d)
begin
case (sel)
2'b00:  outmux = a;
2'b01:  outmux = b;
2'b10:  outmux = c;
default: outmux = d;
endcase
end
endmodule
```

69)Function Declaration and Function Call

```
module comb15 ( A, B, CIN, S, COUT);
input [3:0] A, B;
input CIN;
output [3:0] S;
output COUT;
wire [1:0] S0, S1, S2, S3;
function [1:0] ADD;
input A, B, CIN;
reg S, COUT;
begin
S = A ^ B ^ CIN;
COUT = (A&B) | (A&CIN) | (B&CIN);
ADD = {COUT, S};
end
endfunction

assign S0 = ADD ( A[0], B[0], CIN),
S1 = ADD ( A[1], B[1], S0[1]),
S2 = ADD ( A[2], B[2], S1[1]),
S3 = ADD ( A[3], B[3], S2[1]),
S = {S3[0], S2[0], S1[0], S0[0]},
COUT = S3[1];
endmodule
```

70)Task Declaration and Task Enable

```
module EXAMPLE ( A, B, CIN, S, COUT);
input [3:0] A, B;
```

```
input CIN;
output [3:0] S;
output COUT;
reg [3:0] S;
reg COUT;
reg [1:0] S0, S1, S2, S3;

task ADD;
input A, B, CIN;
output [1:0] C;
reg [1:0] C;
reg S, COUT;

begin
S = A ^ B ^ CIN;
COUT = (A&B) | (A&CIN) | (B&CIN);
C = {COUT, S};
end
endtask

always @( A or B or CIN)
begin
ADD ( A[0], B[0], CIN, S0);
ADD ( A[1], B[1], S0[1], S1);
ADD ( A[2], B[2], S1[1], S2);

S = {S3[0], S2[0], S1[0], S0[0]};
COUT = S3[1];
end
endmodule
```

71)Encoder - Using case Statement.

```
module encoder_using_case(binary_out , encoder_in , enable );
output [3:0] binary_out  ;



input  enable ;
input [15:0] encoder_in ;

reg [3:0] binary_out ;

always @ (enable or encoder_in)
begin
  binary_out = 0;
```

```
 if (enable) begin
   case (encoder_in)
     16'h0002 : binary_out = 1;
     16'h0004 : binary_out = 2;
     16'h0008 : binary_out = 3;
     16'h0010 : binary_out = 4;
     16'h0020 : binary_out = 5;
     16'h0040 : binary_out = 6;
     16'h0080 : binary_out = 7;
     16'h0100 : binary_out = 8;
     16'h0200 : binary_out = 9;
     16'h0400 : binary_out = 10;
     16'h0800 : binary_out = 11;
     16'h1000 : binary_out = 12;
     16'h2000 : binary_out = 13;
     16'h4000 : binary_out = 14;
     16'h8000 : binary_out = 15;
  endcase
 end
end
endmodule
```

72)Encoder - Using if-else Statement.

```
module encoder_using_if(binary_out , encoder_in ,enable);
output [3:0] binary_out  ;
input  enable ;
input [15:0] encoder_in ;
reg [3:0] binary_out ;
always @ (enable or encoder_in)
 begin
  binary_out = 0;
  if (enable) begin
    if (encoder_in == 16'h0002) begin
     binary_out = 1;
    end  if (encoder_in == 16'h0004) begin


    binary_out = 2;
    end  if (encoder_in == 16'h0008) begin
     binary_out = 3;
    end  if (encoder_in == 16'h0010) begin
     binary_out = 4;
    end  if (encoder_in == 16'h0020) begin
     binary_out = 5;
```

```
    end  if (encoder_in == 16'h0040) begin
     binary_out = 6;
    end  if (encoder_in == 16'h0080) begin
     binary_out = 7;
    end  if (encoder_in == 16'h0100) begin
     binary_out = 8;
    end  if (encoder_in == 16'h0200) begin
     binary_out = 9;
    end if (encoder_in == 16'h0400) begin
     binary_out = 10;
    end  if (encoder_in == 16'h0800) begin
     binary_out = 11;
    end  if (encoder_in == 16'h1000) begin
     binary_out = 12;
    end  if (encoder_in == 16'h2000) begin
     binary_out = 13;
    end  if (encoder_in == 16'h4000) begin
     binary_out = 14;
    end if (encoder_in == 16'h8000) begin
     binary_out = 15;
    end
  end
end

endmodule
```

73)Pri-Encoder - Using if-else Statement

```
module pri_encoder_using_if (binary_out , encoder_in , enable);
output [3:0] binary_out ;
input  enable ;
input [15:0] encoder_in ;

reg [3:0] binary_out ;

always @ (enable or encoder_in)
begin




binary_out = 0;
  if (enable) begin
   if (encoder_in == {{14{1'bx}},1'b1,{1{1'b0}}}) begin
    binary_out = 1;
   end else if (encoder_in == {{13{1'bx}},1'b1,{2{1'b0}}}) begin
```

```
   binary_out = 2;
  end else if (encoder_in == {{12{1'bx}},1'b1,{3{1'b0}}}) begin
   binary_out = 3;
  end else if (encoder_in == {{11{1'bx}},1'b1,{4{1'b0}}}) begin
   binary_out = 4;
  end else if (encoder_in == {{10{1'bx}},1'b1,{5{1'b0}}}) begin
   binary_out = 5;
  end else if (encoder_in == {{9{1'bx}},1'b1,{6{1'b0}}}) begin
   binary_out = 6;
  end else if (encoder_in == {{8{1'bx}},1'b1,{7{1'b0}}}) begin
   binary_out = 7;
  end else if (encoder_in == {{7{1'bx}},1'b1,{8{1'b0}}}) begin
   binary_out = 8;
  end else if (encoder_in == {{6{1'bx}},1'b1,{9{1'b0}}}) begin
   binary_out = 9;
  end else if (encoder_in == {{5{1'bx}},1'b1,{10{1'b0}}}) begin
   binary_out = 10;
  end else if (encoder_in == {{4{1'bx}},1'b1,{11{1'b0}}}) begin
   binary_out = 11;
  end else if (encoder_in == {{3{1'bx}},1'b1,{12{1'b0}}}) begin
   binary_out = 12;
  end else if (encoder_in == {{2{1'bx}},1'b1,{13{1'b0}}}) begin
   binary_out = 13;
  end else if (encoder_in == {{1{1'bx}},1'b1,{14{1'b0}}}) begin
   binary_out = 14;
  end else if (encoder_in == {1'b1,{15{1'b0}}}) begin
   binary_out = 15;
  end
 end
end

endmodule
```

74)Encoder - Using assign Statement

```
module pri_encoder_using_assign (
binary_out , //  4 bit binary output
encoder_in , //  16-bit input
enable       //  Enable for the encoder
);

output [3:0] binary_out ;
input  enable ;
input [15:0] encoder_in ;
```

```
wire [3:0] binary_out ;

assign  binary_out  = (!enable) ? 0 : (
   (encoder_in == 16'bxxxx_xxxx_xxxx_xxx1) ? 0 :
   (encoder_in == 16'bxxxx_xxxx_xxxx_xx10) ? 1 :
   (encoder_in == 16'bxxxx_xxxx_xxxx_x100) ? 2 :
   (encoder_in == 16'bxxxx_xxxx_xxxx_1000) ? 3 :
   (encoder_in == 16'bxxxx_xxxx_xxx1_0000) ? 4 :
   (encoder_in == 16'bxxxx_xxxx_xx10_0000) ? 5 :
   (encoder_in == 16'bxxxx_xxxx_x100_0000) ? 6 :
   (encoder_in == 16'bxxxx_xxxx_1000_0000) ? 7 :
   (encoder_in == 16'bxxxx_xxx1_0000_0000) ? 8 :
   (encoder_in == 16'bxxxx_xx10_0000_0000) ? 9 :
   (encoder_in == 16'bxxxx_x100_0000_0000) ? 10 :
   (encoder_in == 16'bxxxx_1000_0000_0000) ? 11 :
   (encoder_in == 16'bxxx1_0000_0000_0000) ? 12 :
   (encoder_in == 16'bxx10_0000_0000_0000) ? 13 :
   (encoder_in == 16'bx100_0000_0000_0000) ? 14 : 15);

endmodule
```

75)Decoder - Using case Statement

```
module decoder_using_case (
binary_in   , //  4 bit binary input
decoder_out , //  16-bit  out
enable        //  Enable for the decoder
);
input [3:0] binary_in  ;
input  enable ;
output [15:0] decoder_out ;

reg [15:0] decoder_out ;

always @ (enable or binary_in)
begin
  decoder_out = 0;
  if (enable) begin
    case (binary_in)



      4'h0 : decoder_out = 16'h0001;
      4'h1 : decoder_out = 16'h0002;
      4'h2 : decoder_out = 16'h0004;
```

```
     4'h3 : decoder_out = 16'h0008;
     4'h4 : decoder_out = 16'h0010;
     4'h5 : decoder_out = 16'h0020;
     4'h6 : decoder_out = 16'h0040;
     4'h7 : decoder_out = 16'h0080;
     4'h8 : decoder_out = 16'h0100;
     4'h9 : decoder_out = 16'h0200;
     4'hA : decoder_out = 16'h0400;
     4'hB : decoder_out = 16'h0800;
     4'hC : decoder_out = 16'h1000;
     4'hD : decoder_out = 16'h2000;
     4'hE : decoder_out = 16'h4000;
     4'hF : decoder_out = 16'h8000;
   endcase
  end
end

endmodule
```

76)Example for Full Adder

```
module fulladder(a,b,c,sum,carry);
input a,b,c;
output sum,carry;
wire sum,carry;
assign sum=a^b^c;       // sum bit
assign carry=((a&b) | (b&c) | (a&c));    //carry bit
endmodule
testbench for fulladder
module main;
reg a, b, c;
wire sum, carry;
fulladder add(a,b,c,sum,carry);
always @(sum or carry)
begin

$display("time=%d:%b + %b + %b = %b, carry =
%b\n",$time,a,b,c,sum,carry);
end
initial



begin
   a = 0;   b = 0; c = 0;
   #5
```

```
   a = 0;   b = 1; c = 0;
   #5
   a = 1;   b = 0; c = 1;
   #5
   a = 1;   b = 1; c = 1;
end
endmodule
```

77) Example for Adder/Subtractor

```
module addSub(A, B, sel, Result);
input sel;
input [3:0] A,B;
output [3:0] Result;
wire [3:0] Result;
assign Result = (sel)? A + B : A - B;
endmodule
testbench for addsub
module main;
reg    [3:0] A, B;
reg    sel;
wire   [3:0] Result;
addSub as1(A, B, sel, Result);
initial begin
    A = 4'b0001;
    B = 4'b1010;
end
nitial begin
    forever begin
       #10
       A = A + 1'b1;
       B = B + 1'b2;
    end
end
initial begin
    sel = 1;
    #200
    sel = 0;
end
endmodule
```