1.

a) Write Syntax of Verilog Module.

```
module < module name >(terminal list);
----------
module internals
--------
endmodule
```

b) List out the various levels of abstraction.

- Behavioral level
- Data flow level
- Gate level
- switch level

c) How an array will be declared in Verilog HDL.

Syntax:

<data_type> <var_name>[start_idx : end_idx];

Examples:

integer count[0:7]; // An array of 8 count variables

d) Write the syntax to represent comments in Verilog HDL.

- a=b&&c;//one-line comment
- /*multiple line comment*/

e) Write the difference between $display and $monitor in Verilog HDL.

The need to call $display every time when we want to print values, but in the case of $monitor, we need to call it only one time, and it will print a value of a variable every time when its value is getting changed.

f) What are the logic values supported by Verilog HDL.

0, 1, X, Z

g) Write a Verilog HDL code for AND gate in Dataflow modelling.

```
module AND_2_data_flow (output Y, input A, B);
assign Y = A & B;
endmodule
```

h) Write about conditional Operator.

In Verilog, conditional statements are used to control the flow of execution based on certain conditions.

condition ? value_if_true : value_if_false

i) If A= 8'H32, B = 1'd0 and C = 1'b1, then find Y = {A[6:3],3{B},C}.

Y=01000001.

j) Define rise delay.

The time taken for the output of a gate to change from some value to 1 is called a rise delay.

k) Explain about implicit continuous assignment.

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

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

l) Why Initial block is not allowed in Designing Hardware in Verilog.

An initial block is not synthesizable and hence cannot be converted into a hardware schematic with digital elements.

m) Write a Verilog HDL code to swap two numbers without temporary variable.

always @ (posedge clock)
a<=b;
always @ (posedge clock)
b<=a;

n) Write syntax to define a task in Verilog HDL.

task (name);
input port list;
output port list;
begin
statement
end
endtask

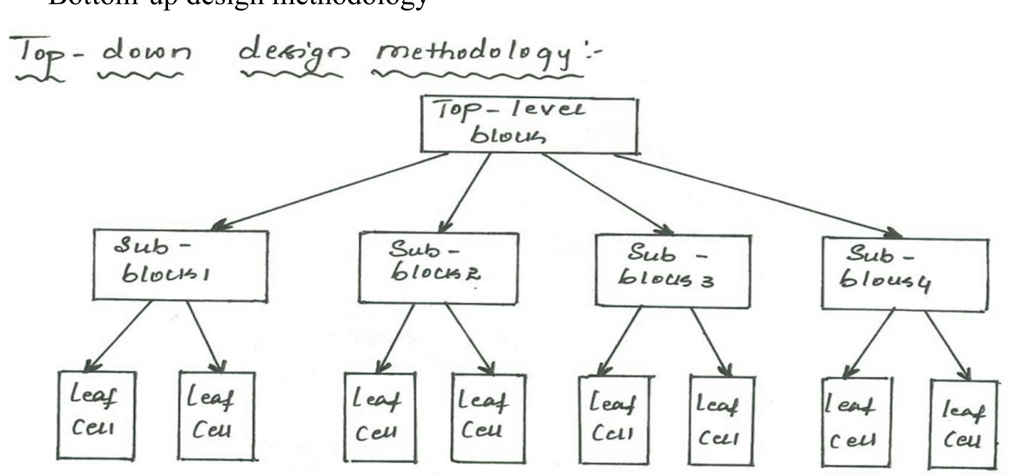
Unit-I

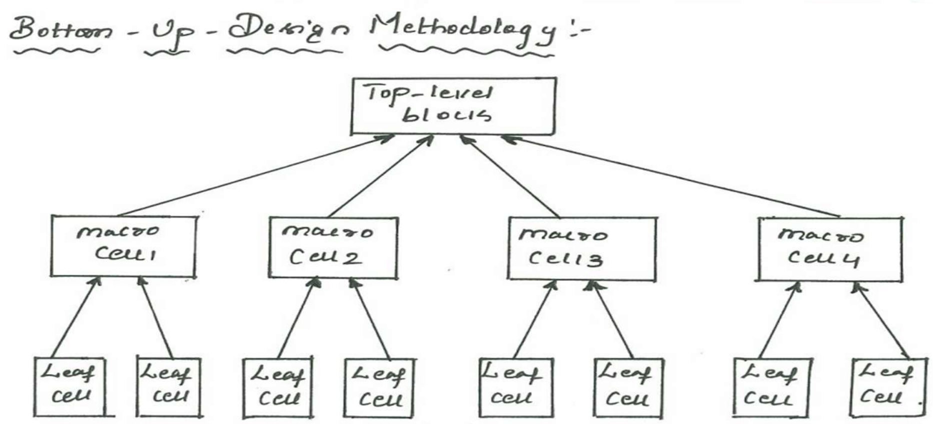
2 Illustrate digital design methodologies in Verilog HDL using a example.

Design methodologies:

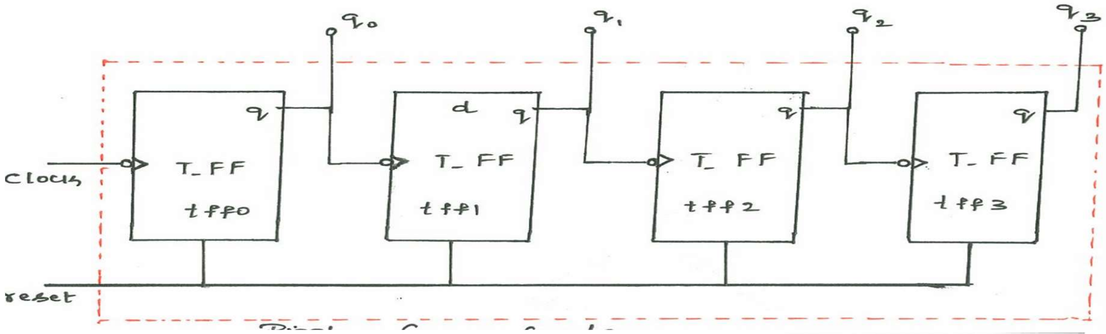
There are two types of design methodology.

- Top-down design methodology
- Bottom-up design methodology

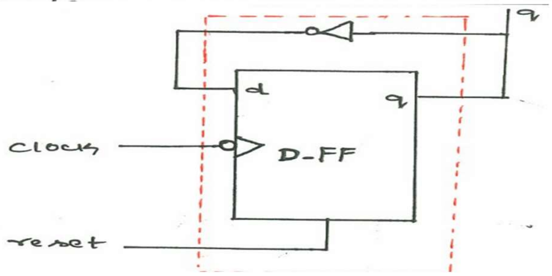


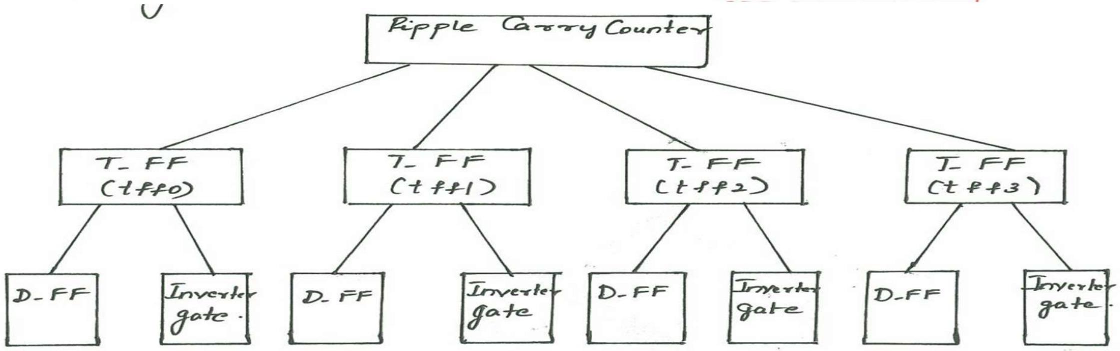


4-bit Ripple Carry Counter



reset	q_n	q_{n+1}
1	1	0
1	0	0
0	0	1
0	1	0
0	0	0






```
module ripple_carry_counter (q, clk, reset);
output [3:0] q ; // I/o Signals and vector declarations will be
                 // Explained in comming modules.
input clk, reset; // I/o  ,,      ,,       ,,
// Four instances of a module T_FF are created. Each has a
// unique name. Each instance is passed with a set of.
// Signals. Note that Each instance is copy of the module T_FF
  T_FF   tff0 (q[0], clk, reset);
  T_FF   tff1 (q[1], q[0], reset);
  T_FF   tff2 (q[2], q[1], reset);
  T_FF   tff3 (q[3], q[2], reset);
endmodule
```

```
module T_FF (q, clk, reset):                    T_FF
// Declarations will be Explained in Comming modules.
Output q ;
input clk, reset;
wire d ;
D_FF dff0 (q, d, clk, reset); // Instantiate D_FF
not n1 (d, q); // not gate is a Verilog primitive Explained.
               // in comming modules.
endmodule.
```

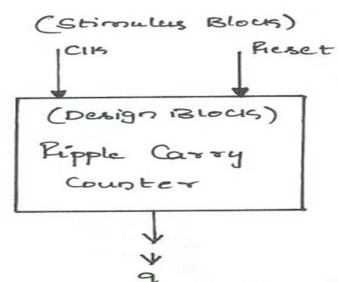
(OR)

3 a) Explain the components of simulation with a neat block diagram.

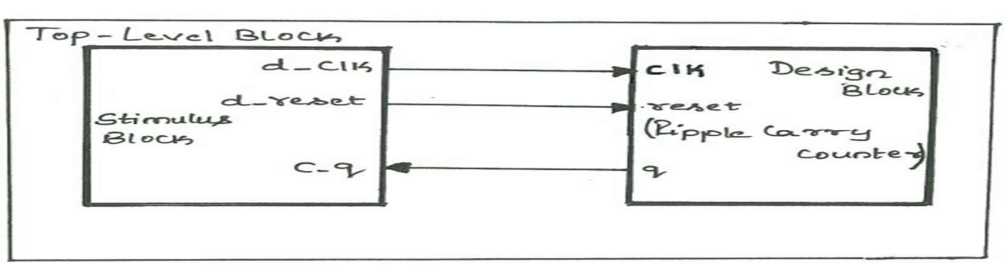
* Once a design block is completed, it must be tested
* The functionality of the design block can be tested by applying stimulus for checking results, such block is called as "Stimulus block"
* The stimulus block can be written in Verilog and it is a good practice to keep the stimulus and design blocks separate.
* The stimulus block is also called as "test bench".

=> There are two styles of stimulus

* In the first style, the stimulus block instantiat the design block and directly drives the signals in the design block.
* As shown in the figure below, the stimulus block becomes the top-level block. It manipulates signals clk and reset, and it checks and display output signal q.



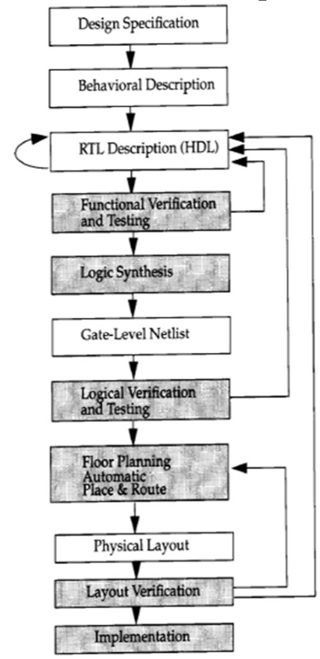
* The second style of applying stimulus is to instantiate both the stimulus and design blocks in a top-level dummy module.
* The stimulus block interacts with the design block. Only through the interface. as shown in the figure below.
* The stimulus module drives the signals d_clk and d_reset which are connected to the signals clk and reset in the design block.
* it also checks and displays signal c_q, which is connected to the signal q in the design block.



b) Discuss in detail about VLSI Design flow with neat flow chart and example

Typical Design Flow

- Design specification describe the functionality, interface &overall architecture.
- Behavioral description is created to analyze the design in terms of functionality, performance.
- This is converted into RTL description in an HDL.
- Logic synthesis converts the RTL description into gate-level net list.
- Gate-level net list is a description in terms of gates and connection between them.
- Synthesis tool ensure that gate-level net list needs timing, area and power specifications.
- Floor planning analysis of the design.
- Place and route: Placement of cells and connections to the target hardware.



UNIT-II

4 a) Explain different data types in Verilog HDL with examples

Value level

0
1
X
Z

Nets

- Used to represent connections between HW elements
- Values continuously driven on nets.
- Keyword: wire
- Default: One-bit values
- unless declared as vectors
- Default value: z
- except the trireg net, which defaults to x
- Examples:
- wire a;

Registers

- Registers represent data storage elements
- Retain value until next assignment
- This is not a hardware register or flipflop
- Keyword: reg
- Default value: x
- Values of registers can be changed anytime in a simulation by assigning a new value to the register.
- Example:

reg reset;

Vectors

- Net and register data types can be declared as vectors (multiple bit widths)
- Syntax:
- wire/reg [msb_index : lsb_index] data_id;
- Vectors can be declared at [high# : low#] or [low# : high#]
- Example

wire [0:3] a;
wire [7:0] bus;

Integer

- general purpose register data type used for manipulating quantities.
- integer variables are signed numbers.
- reg vectors are unsigned numbers.
- Keyword: integer

- Bit width: implementation-dependent (at least 32-bits)
- Designer can also specify a width:
  integer [7:0] tmp;
- Examples:
  integer counter;
  initial
  counter = -1;

Real

- Real : It is also a register data type.
- Keyword: real
- Values:
- Default value: 0
- Decimal notation: 12.24
- Scientific notation: 3e6 (=3x10^6)
- Cannot have range declaration
- Example:

```
real delta;
initial
begin
  delta=4e10;
  delta=2.13;
end
integer i;
initial
  i = delta; // i gets the value 2 (rounded value of 2.13)
```

Time

- Register data type used to store values of simulation time.
- Keyword: time
- Bit width: implementation-dependent (at least 64)
- $time system function gives current simulation time
- Example:

```
time save_sim_time;
initial
  save_sim_time = $time;
```

- Simulation time is measured in terms of simulation.

seconds. The unit is denoted by s, the same as real time.

Arrays

- One-dimensional arrays and multi-dimensional arrays are supported.
- Allowed for wire, reg, integer, time,real data types
- Syntax:
- <data_type> <var_name>**[start_idx : end_idx]**;
- Examples:
- integer count**[0:7]**; // An array of 8 count variables

Memories

- RAM, ROM, and register-files used many times in digital systems.
- Memory is an array of registers in Verilog
- Word is an element of the array
- Can be one or more bits
- Examples:

reg membit[0:1023];

Parameters

- But can be overridden for each module at compile-time
- cannot be used as variables.
- Syntax:

parameter <const_id>=<value>;

- Gives flexibility
- Allows to customize the module
- Example:

parameter port_id=5; // Defines a constant port_id
parameter cache_line_width=256;

Strings

- Strings are stored in reg variables.
- 8-bits required per character
- Example:

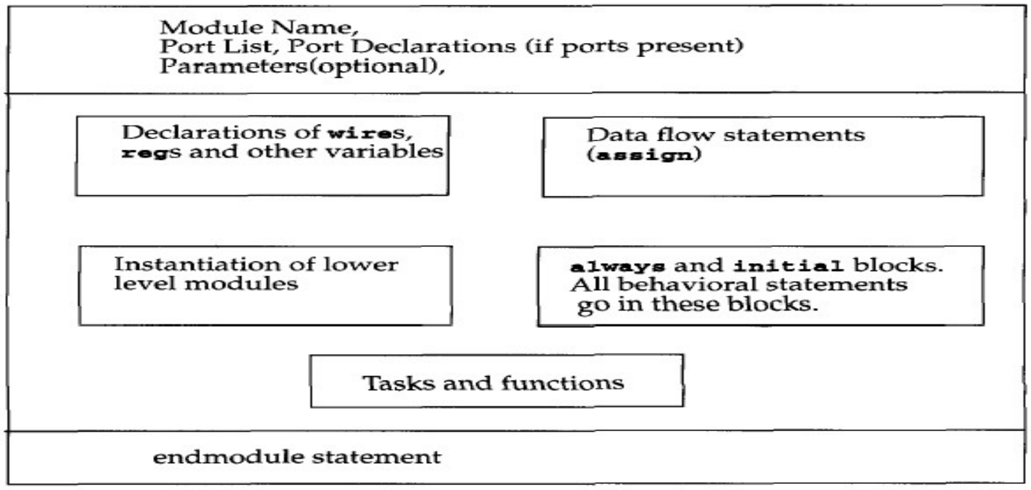
```
reg [8*18:1] string_value;
initial
  string_value = “Hello World!”;
```

(OR)

5 a) Explain about the components of the Verilog module.

- The module name, port list, port declarations, and optional parameters must come first in a module definition.

- Port list and port declarations are present only if the module has any ports to interact with the external environment.
- The components can be in any order and at any place in the module
- The endmodule statement must always come last in a module definition.
- All components except module, module name, and endmodule are optional and can be mixed and matched as per design needs.



b) Explain port declaration with an example using Verilog code.

- All ports in the list of ports must be declared in the module.
- Ports can be declared as follows:

• **Verilog keyword**	**types of port**
• Input	input port
• Output	output port
• Inout	bidirectional port

```
module fulladd4(sum, c_out, a, b, c_in);

//Begin port declarations section
output[3:0] sum;
output c_cout;

input [3:0] a, b;
input c_in;
//End port declarations section
...
<module internals>
...
endmodule
```

Unit-III

6 a) Design and write Verilog HDL code for a 4-bit Adder in gate level modelling.

Full Adder

```
module full_adder(
   input a,
   input b,
   input cin,
   output s,
   output cout,
   wire p,q,r );
xor(p,a,b);
and(r,a,b);
xor(sum,p,cin);
and(q,p,cin);
or(cout,q,r);
endmodule
```

4-bit Adder

```
module four_bit_adder(
```

```
    input [3:0] A,
    input [3:0] B,
    input C0,
    output [3:0] S,
    output C4 );
    wire C1,C2,C3;
    full_adder fa0 (A[0],B[0],C0,S[0],C1);
   full_adder fa1 (A[1],B[1],C1,S[1],C2);
   full_adder fa2 (A[2],B[2],C2,S[2],C3);
    full_adder fa3 (A[3],B[3],C3,S[3],C4);
endmodule
```

Test bench

```
module test_4_bit();
    reg [3:0] A;
    reg [3:0] B;
    reg C0;
    wire [3:0] S;
    wire C4;
    four_bit_adder dut(A,B,C0,S,C4);
    initial begin
    A = 4'b0011;B=4'b0011;C0 = 1'b0; #10;
    A = 4'b1011;B=4'b0111;C0 = 1'b1; #10;
    A = 4'b1111;B=4'b1111;C0 = 1'b1; #10;
    end
endmodule
```

(OR)

7 a) Design and write Verilog HDL code for 4:16 decoder in dataflow modelling style

```
module Decoder4x16 (input [3:0] select,
input enable, output reg [16:0] out);
always @(select, enable)
begin
if(enable == 1'b0)
out = 16'b0000000000000000;
else if(enable == 1'b1)
if(select == 4'b0000)
out <= 16'b0000000000000001;
else if(select == 4'b0001)
out <= 16'b0000000000000010;
else if(select == 4'b0010)
out <= 16'b0000000000000100;
else if(select == 4'b0011)
out <= 16'b0000000000001000;
else if(select == 4'b0100)
out <= 16'b0000000000010000;
else if(select == 4'b0101)
out <= 16'b0000000000100000;
else if(select == 4'b0110)
out <= 16'b0000000001000000;
else if(select == 4'b0111)
out <= 16'b0000000010000000;
else if(select == 4'b1000)
out <= 16'b0000000100000000;
else if(select == 4'b1001)
out <= 16'b0000001000000000;
else if(select == 4'b1010)
out <= 16'b0000010000000000;
else if(select == 4'b1011)
out <= 16'b0000100000000000;
else if(select == 4'b1100)
out <= 16'b0001000000000000;
else if(select == 4'b1101)
out <= 16'b0010000000000000;
else if(select == 4'b111)
out <= 16'b0100000000000000;
else if(select == 4'b1111)
out <= 16'b1000000000000000;
end
endmodule
```

Testbench

```
module Decoder4x16_test;
reg [3:0] select;
reg enable;
wire [16:0] out;
parameter sim_time = 2800;
Decoder4x16 decoder(select, enable, out);
initial #sim_time $finish;
initial
begin
select = 4'b0000;
enable = 1'b0;
repeat(16) #10 begin
enable = 1'b1;
#85 $display("select = %b \t out = %b",
select, out);
select = select + 4'b0001;
end
end
endmodule
```

b) Explain about Verilog HDL Operators with examples.

Arithmetic Operators

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

Binary operators

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

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

D = 6; E = 4; F=2// D and E are integers

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

D / E // Divide D by E. Evaluates to 1. Truncates any fractional part.

A + B // Add A and B. Evaluates to 4'b0111

B - A // Subtract A from B. Evaluates to 4'b0001

Logical Operators

Logical operators are logical-and (&&), logical-or (||) and logical-not (!). Operators && and || are binary operators.

1. Logical operators always evaluate to a 1-bit value, 0 (false), 1 (true), or x (ambiguous).
2. If an operand is not equal to zero, it is equivalent to a logical 1 (true condition).
3. Logical operators take variables or expressions as operands.

// Logical operations

A = 3; B = 0;

A && B // Evaluates to 0. Equivalent to (logical-1 && logical-0)

A || B // Evaluates to 1. Equivalent to (logical-1 || logical-0)

Relational Operators

Relational operators are greater-than (>), less-than (<), greater-than-or-equal-to (>=), and less-than-or-equal-to (<=).

If relational operators are used in an expression, the expression returns a logical value of 1 if

// A = 4, B = 3

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

A <= B // Evaluates to a logical 0

A > B // Evaluates to a logical 1

Equality Operators

Equality operators are logical equality (==), logical inequality (!=), case equality (===), and case inequality (!==). When used in an expression, equality operators return logical value 1 if true, 0 if false.

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

// A = 4, B = 3

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

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

A == B // Results in logical 0

X != Y // Results in logical 1

Bitwise Operators

• Bitwise operators are negation (~), and(&), or (|), xor (^), xnor (^~, ~^). Bitwise operators perform a bit-by-bit operation on two operands.

Example:

// X = 4'b1010, Y = 4'b1101
// Z = 4'b10x1
~X // Negation. Result is 4'b0101
X & Y // Bitwise and. Result is 4'b1000

Reduction Operators

• Reduction operators are and (&), nand (~&), or (|), nor (~|), xor (^), and xnor (~^, ^~).

• Reduction operators take only one operand.

// X = 4'b1010
&X //Equivalent to 1 & 0 & 1 & 0. Results in 1'b0
|X//Equivalent to 1 | 0 | 1 | 0. Results in 1'b1

Concatenation Operator

• The concatenation operator ({, }) provides a mechanism to append multiple operands.

// A = 1'b1, B = 2'b00, C = 2'b10, D = 3'b110
Y = {B , C} // Result Y is 4'b0010

Replication Operator

• Repetitive concatenation of the same number can be expressed by using a replication constant.

• A replication constant specifies how many times to replicate the number inside the brackets ({ })

reg A;
reg [1:0] B, C;
reg [2:0] D;
A = 1'b1; B = 2'b00; C = 2'b10; D = 3'b110;
Y = { 4{A} } // Result Y is 4'b1111

Conditional Operator

The conditional operator(?:) takes three operands.

Usage: condition_expr ? true_expr : false_expr ;

Unit-IV

8 a) Design decade counter using behavioural style-based Verilog HDL with testbench.

Decade Counter Verilog Code

```
module decade_counter(en, clock, count);
input en, clock;
output reg [3:0] count;
always @( posedge clock)
begin
if(en)
begin
if ( count>=4'd0 && count<4'd10)
count<=count+4'd1;
else
count<=4'd0;
end
else
count<=4'd0;
end
endmodule
```

Testbench:

```
module decadecounter_tb;
wire [3:0] count;
reg en,clock;
decade_counter dut(.en(en), .clock(clock),
.count(count));
initial begin
$display($time," Starting the Simulation");
en=0;
clock=0;
```

```
#20 en=1'd1;
end
always
#5 clock=~clock;
```

```
initial
$monitor ( $time , "clock= %b, count=
%d, en= %b",    clock,count, en);
endmodule
```

b) Develop a Verilog HDL code for 4:2 encoder using case statement.

4:2 encoder

```
module 4_2_ENC(
input [3:0]din,
output [1:0]dout );
reg [1:0]dout;
always @ (din)
case (din)
1 : dout[0] = 0;
2 : dout[1] = 1;
4 : dout[2] = 2;
8 : dout[3] = 3;
default : dout = 2'bxx;
endcase
endmodule
```

Testbench

```
initial begin
// Initialize Inputs
din = 0;
// Wait 100 ns for global reset to finish
#100;
#100; din=1;
#100; din=2;
#100; din=4;
#100; din=8;
end
initial begin
#100
$monitor("din=%b, dout=%b", din, dout);
end
endmodule
```

(OR)

9 a) Distinguish blocking and non-blocking assignments with examples.

There are two types of Procedural assignments : Blocking and Non-Blocking assignments.

Blocking assignments:

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

Example: initial

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

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

Non-blocking assignments:

The nonblocking assignment allows assignment scheduling without blocking the procedural flow. The nonblocking assignment statement can be used whenever several variable assignments within the same time step can be made without regard to order or dependence upon each other. Non-blocking assignments are made using the operator <=.comparison operator and not as non-blocking assignment.

Example: initial

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

b) Explain about functions in Verilog HDL and write the difference between task and function.

The purpose of a function is to return a value that is to be used in an expression. A function definition always start with the keyword `function` followed by the return type, name and a port list enclosed in parantheses. Verilog knows that a function definition is over when it finds the `endfunction` keyword. Note that a function shall have atleast one input declared and the return type will be `void` if the function does not return anything.

Syntax

```
function [automatic] [return_type] name ([port_list]);
    [statements]
endfunction
```

Functions	Tasks
A function can enable another function but not another task.	A task can enable other tasks and functions.
Functions always execute in 0 simulation time.	Tasks may execute in non-zero simulation time.
Functions must not contain any delay, event, or timing control statements.	Tasks may contain delay, event, or timing control statements.
Functions must have at least one **input** argument. They can have more than one **input**.	Tasks may have zero or more arguments of type **input, output or inout**.
Functions always return a single value. They cannot have **output** or **inout** arguments.	Tasks do not return with a value but can pass multiple values through **output** and **inout** arguments.