	iii. Use the following input/output signals: ◦ clk – Clock input ◦ rst – Asynchronous active-high reset ◦ flip – 1-bit input (success = 1,fail = 0) win – 1-bit output, high only when the player wins.				
		OR			
9		Design the Verilog HDL for User Defined Primitive with example.	L4	CO4	10 M
		UNIT-V			
10	a)	Analyze the test bench to verify the functionality of 3 to 8 decoder with an enable pin.	L4	CO5	5 M
	b)	Explain DUT with block diagram.	L4	CO5	5 M
		OR			
11	a)	Design the test bench to verify the functionality of D Flip-flop with a reset pin.	L4	CO5	5 M
	b)	Explain the difference between output Q; and output reg Q;	L4	CO5	5 M

PVP 23

Code: 23EC4501A

III B.Tech - I Semester - Regular Examinations - NOVEMBER 2025

DIGITAL SYSTEM DESIGN THROUGH HDL
(ELECTRONICS & COMMUNICATION ENGINEERING)

Duration: 3 hours Max. Marks: 70

Note: 1. This question paper contains two Parts A and B.
2. Part-A contains 10 short answer questions. Each Question carries 2 Marks.
3. Part-B contains 5 essay questions with an internal choice from each unit. Each Question carries 10 marks.
4. All parts of Question paper must be answered in one place.

BL – Blooms Level CO – Course Outcome

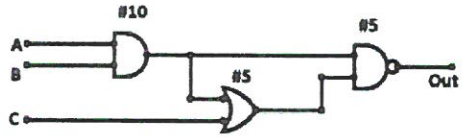
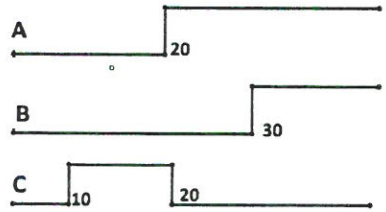
PART – A

		BL	CO
1.a)	Discuss the mean by Z in Verilog HDL, explain with a case study.	L2	CO1
1.b)	Describe the difference between the operators "&&" and "&".	L2	CO1
1.c)	Analyze the output of y1 and y2, if a = 110X and b = 110X assign y1 = (a ==b); assign y2 = (a===b);	L4	CO2
1.d)	Perform the binary addition of +6 and -3 and verify your answer.	L2	CO2
1.e)	What is lowest level of abstraction in Verilog HDL?	L2	CO3
1.f)	Predict the truth tables of OR gate including 0, 1, X and Z.	L2	CO3
1.g)	Identify the operators required to design a one-bit comparator.	L2	CO4
1.h)	Differentiate between blocking and non-blocking statements.	L2	CO4

1.i)	Is it possible to include both level-sensitive and edge-sensitive signals in the sensitivity list of a single always block.	L4	CO5
1.j)	Explain the Testing.	L2	CO5

PART – B

			BL	CO	Max. Marks
		UNIT-I			
2	a)	Discuss the different complier directives in Verilog HDL.	L2	CO1	5 M
	b)	Explain Module structure in Verilog HDL.	L2	CO1	5 M
		OR			
3	a)	Describe the different Data types in Verilog HDL.	L2	CO1	5 M
	b)	Implement XOR gate using basic gate primitives.	L2	CO1	5 M
		UNIT-II			
4	a)	Explain the differences between sequential and parallel blocks with a case study.	L2	CO2	5 M
	b)	Write a Verilog HDL code to implement 4 bit Register.	L2	CO2, CO3	5 M
		OR			
5	a)	Develop a Verilog code to implement the D flip flop. Name the module as D_FF.	L4	CO2	5 M
	b)	Design Verilog HDL program 4 bit binary counter using behavioural modelling.	L4	CO2, CO3	5 M
		UNIT-III			
6		Implement a mux_2to1 circuit using a conditional operator and then use the module instantiation to implement mux_4to1 using mux_2to1.	L4	CO2, CO3	10 M

	OR			
7	Develop the Data flow Verilog model to design the circuit with delay constraints. Write stimulus block to excite the inputs shown below. Also draw the output waveform.  circuit  Input excitation (timescale is ns)	L4	CO2, CO3	10 M
	UNIT-IV			
8	In the traditional Korean game *Ddakji*, a player wins if they successfully flip the opponent's tile **three times in a row**. Each successful flip is represented by a binary 1, and each failed flip is represented by a binary 0. Design a Mealy/Moore **sequence detector** using **Verilog HDL** that detects three consecutive successful flips in a serial input stream. Design Requirements: i. The design should use a **Finite State Machine (FSM)**. ii. The detector should not allow **overlapping sequences**.	L4	CO4	10 M

1.a) Z meaning ----------1M
Explanation ----------1M

1.b) any one difference----------2M

1.c) y1 and y2----------2M

1.d) addition----------2M

1.e) name----------2M

1.f) OR Gate Symbol ----------1M
Table----------1M

1.g) Equations----------1M
Operators----------1M

1.h Any one difference----------2M

1.i) answer----------2M

1.j) definition----------2M

2.a) Types----------2M
explanation----------3M

2.b) module with example ----------3M
Diagram----------2M

3.a) Data types----------1M
Net data types & example----------2M
Registers or variable data types ----------2M

3.b) Logic Symbol and truth table ----------2M
Program(verilog code)----------3M
-

4.a) exaplanation of sequential ----------2M
exaplanation of parallel----------3M

4.b) Dff Program ----------2M
Register ----------3M

5. a) Logic Symbol and Circuite diagram ----------2M
Program ----------3M

5.b) iDiagram and Truth table ----------2M

Program ----------3M

6.) 2X1 MUX- diagram ana truth table----------2M
2x1 MUX program----------3M
4X1 MUX- diagram ana truth table----------2M
4x1 MUX program----------2M

7.) Program(verilog code)----------5M
Test bence----------5M

8. State diagram---------- ---------- 2M
Staetable----------3M
Program----------5M
9. Diaram and truth tabe----------5M
Program----------5M

10.a) Logic diagram ----------2M
Function table ----------1M
Program(verilog code)----------2M

10.b) block diagram ----------3M
explanation----------2M

11.a) Logic Symboland diaram----------2M
Program(verilog code)----------3M

11.b) any two differences with example program----------5M

III B.Tech I Sem Regular Examination November -2025
Digital System Design Through HDL
Scheme of valuation
ECE
Code: 23EC4501A

1.a

Z means High Impedance state and It means the signal **is not driving anything and the output behaves like it is disconnected or floating**

1.b

&& means- Logical AND

Result is **1 or 0 only**

& mean-s Bitwise AND

It works **bit-by-bit and used on vectors / multi-bit signals**

1.c

y1 = (a == b)

1,2,3,bit are same Last bit: **X == X → treated as don't care**

y1 **=1**

y2 = (a === b)

X === X → TRUE

y2=1

1.d

+6- 0110

-3 0011

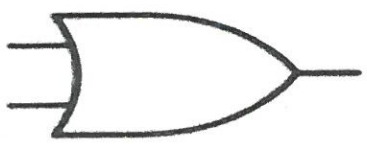
3 0011

1.e

Gatelevel modelling

1.f

or	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x



1.g

A **1-bit comparator** compares two single-bit inputs **A** and **B** and produces three outputs:

A = B

A > B

A < B

operators

NOT (~)

AND(&)

OR(|)

1.h

Blocking assignments = equals	non-blocking assignments <= is less than equals
Blocking assignments are excuted in the order they are specified in a sewuential block	non-blocking assignments allows scheduling of without blocking the execution of statements

1.i

NO

1.j

A test bench is a Verilog program often written to test the functionality of a design for all possible input combinations

This helps us to prevent the system from malfunctioning

2.a

All compiler directives are defined by using the <keyword> constructand two most complier directives

1.define

2.include

`define

The `define directive is used to define text macros in Verilog (see Example 3-6). This is similar to the *#define* construct in C. The defined constants or text macros are used in the Verilog code by preceding them with a ` (back tick). The Verilog compiler substitutes the text of the macro wherever it encounters a `*<macro_name>*.

Example 3-6 *`define Directive*

```
//define a text macro that defines default word size
//Used as `WORD_SIZE in the code
`define WORD_SIZE 32

//define an alias. A $stop will be substituted wherever `S appears
`define S $stop;
```

`include

The `include directive allows you to include entire contents of a Verilog source file in another Verilog file during compilation. This works similarly to the *#include* in the C programming language. This directive is typically used to include header files, which typically contain global or commonly used definitions (see Example

```
`include header.v
...
...
<Verilog code in file design.v>
...
...
```

2.b) explain Module structure in Verilog HDL

- A module is declared by the keyword ***"module"*** It must have a name (identifier) and terminal list (inputs, outputs). Every Verilog program start with a keyword module and end with a keyword endmodule

module *<module_name> (<module terminal list>);*

statement 1;

statement 2;

....

Endmodule

- A module in Verilog consists of distinct parts

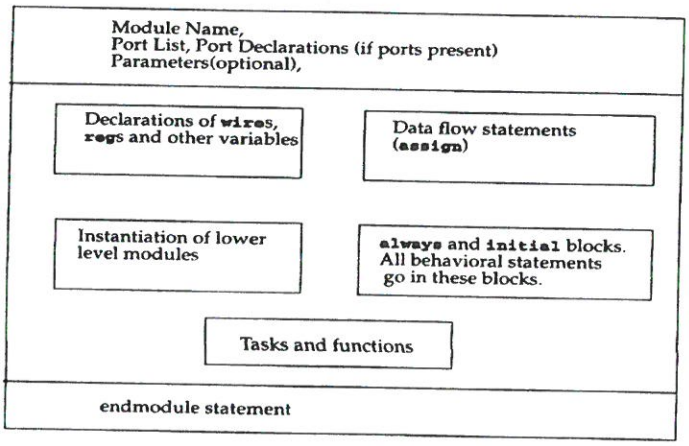


Figure 4-1 Components of a Verilog Module

3.a) Explain Data types in Verilog HDL

Ans) *There are two main groups of data types*

- **Nets**
- **Registers or variable data types**

NET:

- ***Nets*** represent physical connections between structural entities such as gates
- A ***Net*** is short form of network.
- ***Net***work is a group of devices that share a common connection
- ***Nets*** do not store values (except – **trireg**)
- If a ***net*** has no driver, it gets the value of z(high impedance) unless the net is a ***trireg***, in which case it shall hold the previously driven value and net is not a keyword
-

Net Data Type	Functionality
wire, tri	Interconnecting wire - no special resolution function.
wor, trior	Wired outputs OR together
wand, triand	Wired outputs AND together
tri0, tri1	Net pulls-down or pulls-up when not driven
supply0, supply1	Net has a constant logic 0 or logic 1 (supply strength)
trireg	Retains last value, when driven by z (tristate)

example

wire a

Registers or variable data type:

- A variable is an abstraction for a **storage device**
- A variable can be declared through the keyword ***reg*** and stores the value of a logic level 0, 1, x, and z
- Registers retain value until another value is placed onto them
- The value stored in a ***reg*** is changed through a fresh assignment in the program
- Unlike a ***wire,*** a ***reg*** doesn't need a driver
- Ex reg a;

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

- An ***integer*** is a general purpose register data type used for manipulating quantities and Integers are declared by the keyword ***integer***

  integer counter;

  counter = -1;

- The ***time*** register is a 64-bit wide data type that is used to store simulation time and to check timing dependence

  time savetime;

  savetime= $time;

- **Real** number constants and real register data types are declared with the keyword ***real and*** They can be specified in decimal notation (e.g., 3.14)

  real a;

  a = 4.56;

3.b)

module xor_gate(A,B,Y);

Input A,B;

output Y;

xor (Y,A,B);

endmodule

2 input XOR gate

A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

4.a)

Sequential block — delimited by begin ... end

Statements inside a sequential block execute **one after the other**, in the order they are written.

If you use timing delays (e.g., #5, #10, or event controls) for some statements, each delay is relative to when the *previous* statement in the block completed.

For specifying a sequence of actions in known order — e.g. in testbench initialization, stimulus generation, or certain behavioral sequences in simulation.

example

initial begin

a = 0;

#5 b = 1;

#10 c = a & b;

End

Parallel Block delimited by fork...join

All statements inside a parallel block are **started simultaneously** when the block is entered — they run *in parallel.*

If statements have delays or event controls, each delay is measured relative to the time when the block was entered not relative to previous statements

The order the statements are written inside the block does **not** determine execution order. Instead, timing controls decide when each statement executes.

Control leaves the block only after all concurrently started statements complete (after their respective delays or events

```
initial begin
 fork
  #5 a = 1;
  #10 b = 0;
  #15 c = a | b;
 join
end
```

4.b)

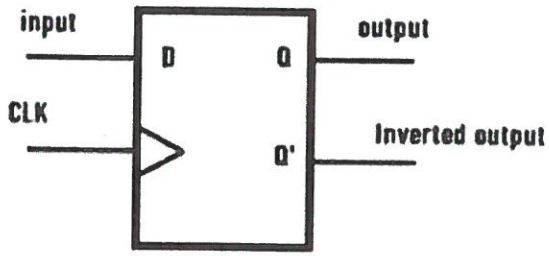
First define D FLIP-FLOP

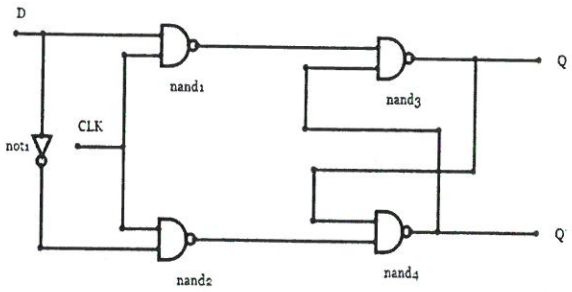
```
module dff(d,clk,q,qb);
    input d,clk;
    output q,qb;
    reg q,qb;
    initial
    begin
      q=1'b0;
      qb=1'b1;
    end

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

```
module four_bit_register(A, clk, q0, q1, q2, q3);
input [0:3] A;
input clk;
output wire q0, q1, q2, q3;
wire qb0, qb1, qb2, qb3;
reg d0, d1, d2, d3;
dff df0(d0, clk, q0, qb0);
dff df1(d1, clk, q1, qb1);
dff df2(d2, clk, q2, qb2);
dff df3(d3, clk, q3, qb3);
always @(posedge clk)
    if (clk)
        begin
            d0 = A[0];
            d1 = A[1];
            d2 = A[2];
            d3 = A[3];
        end
endmodule
```

5.a)





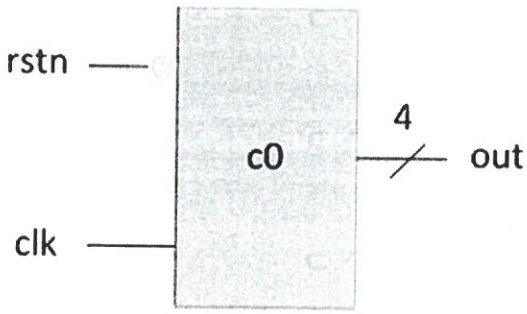

```
module D_FF (Q,D,CLK);
output Q;
input D,CLK;
reg Q;
always @(posedge CLK)
Q = D;
endmodule
```

(OR)

```
module D_FF (Q,D,CLK);
  input CLK,RST,D;
  output Q;
  reg Q;

  always@(posedge CLK or posedge RST)
    if (RST) Q<=0;
    else     Q<=D;
endmodule
```

5.b)



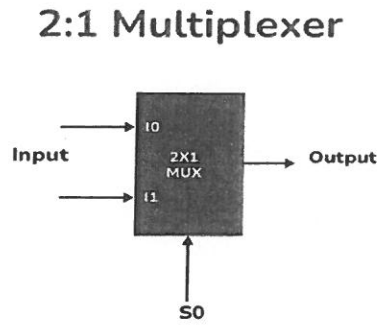
CLK	O3	O2	O1	O0
↑	0	0	0	0
↑	0	0	0	1
↑	0	0	1	0
↑	0	0	1	1
↑	0	1	0	0
↑	0	1	0	1
↑	0	1	1	0
↑	0	1	1	1
↑	1	0	0	0
↑	1	0	0	1
↑	1	0	1	0
↑	1	0	1	1
↑	1	1	0	0
↑	1	1	0	1
↑	1	1	1	0
↑	1	1	1	1
↑	0	0	0	0

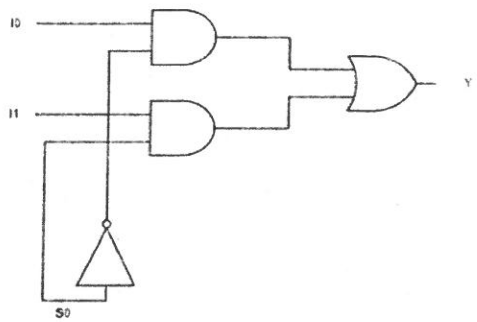
Out=Count

```
module counter (  input clk,
                  input rstn,
                  output reg[3:0]  count);
always @ (posedge clk)
begin
  if (! rstn)
    count <= 0;
```

```
    else
      count <= count + 1;
  end
endmodule
```

6.



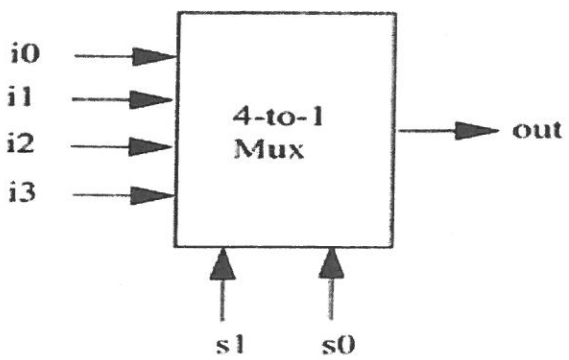


Y=S0.I0+S0.I1

```
module MUX_2x1(S,Y,I0,I1);
input S,I0,I1;
output Y;
assign Y = S? I1 : I0
```

sel	y
0	i_0
1	i_1

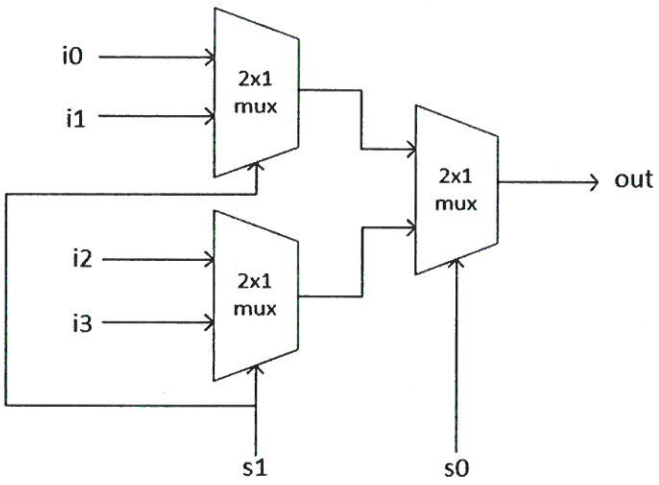
4x1 Multiplexer



s1	s0	out
0	0	I0
0	1	I1
1	0	I2
1	1	I3

4x1 multiplexer using instantiation

```
module mux2x1(s,y,io,i1);
input s,io,i1;
output Y;
wire y1,y2;
not (so,so');
and (y1, so',io);
and (y2,s0,i1);
or (y,y1,y2);
```




```
module mux4x2(y,i0,i1,i2,i3,s1,s0);
 input i0,i1,i2,i3,s1,s0;
output y;
 wire mux1, mux2;
mux2x1  mux_1(mux1,i0,i1,s1);
 mux2x1  mux_2(mux2,i2,i3,s1);
 mux2x1  mux_3(y,mux1,mux2,s0);
endmodule
```

7)

```
module circuit_df (A, B, C, OUT);
 input  A, B, C;
   output OUT;
   wire X, Y;
   assign #10 X = A | B;     // OR with 10 ns delay
   assign #5  Y = X & C;     // AND with 5 ns delay
   assign #5  OUT = Y | C;   // OR with 5 ns delay
endmodule
```

TEST BENCH

```
module_ tb;
    reg A, B, C;
    wire OUT;
    circuit_df uut (A, B, C, OUT);
    initial begin
      // Initial inputs (0 ns)
      A = 0; B = 0; C = 1;
      // At 10 ns
      #10 C = 0; B = 1;
      // At 20 ns
      #10 A = 1; C = 1;
// At 30 ns
      #10 A = 0; B = 0; C = 1;
      #20 $finish;
   end
   initial begin
      $dumpfile("wave.vcd");
      $dumpvars(0, tb);
   end
endmodule
```

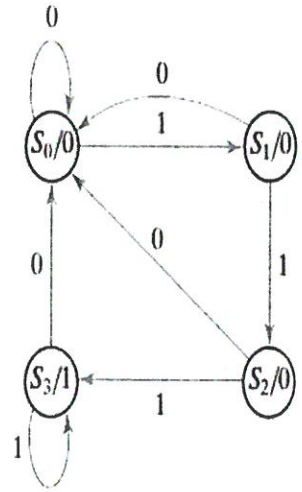
8.

Input:

- 1 = successful flip
- 0 = failed flip
- **FSM Design (Moore Machine)**

State	Meaning
S0	no successes yet (start)
S1	1 consecutive success (1)
S2	2 consecutive successes (11)
S3	3 consecutive successes detected (111 → output=1)

```
module seq111_detector(
    input clk,
    input reset,
    input din,
    output reg out
);
    // Encoding states
    typedef enum reg [1:0] {
        S0 = 2'b00,   // no 1s yet
        S1 = 2'b01,   // 1
        S2 = 2'b10,   // 11
        S3 = 2'b11    // 111 detected (output = 1)
    } state_t;
    state_t current, next;
    // STATE REGISTER
    always @(posedge clk or posedge reset) begin
        if (reset)
            current <= S0;
        else
            current <= next;
    end
    // NEXT STATE LOGIC
    always @(*) begin
        case (current)
            S0: next = (din ? S1 : S0);
            S1: next = (din ? S2 : S0);
            S2: next = (din ? S3 : S0);
            S3: next = S0;          // NON-OVERLAPPING reset
        endcase
```

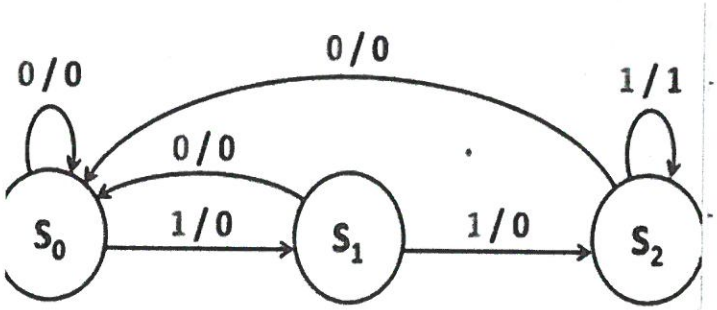


```
    end
    // OUTPUT LOGIC (Moore: depends only on state)
    always @(*) begin
        out = (current == S3);
    end
endmodule
```

mealay Machine

```
module seq111_mealy (
    input  clk,
    input  reset,
    input  din,
    output reg out
);
    // Mealy state encoding
    typedef enum reg [1:0] {
        S0 = 2'b00,   // no 1s yet
        S1 = 2'b01,   // 1
        S2 = 2'b10    // 11
    } state_t;
    state_t current, next;
    // STATE REGISTER
    always @(posedge clk or posedge reset) begin
        if (reset)
            current <= S0;
        else
            current <= next;
    end
always @(*) begin
        out = 0;    // default
```

```
case (current)
  S0: begin
    if (din)  next = S1;
    else      next = S0;
  end
  S1: begin
    if (din)  next = S2;
    else      next = S0;
  end
  S2: begin
    if (din) begin
      next = S0;   // non-overlapping
      out  = 1;    // third consecutive 1
    end
    else begin
      next = S0;
    end
  end
```



State	Present State		(Present Input)	Next State		Output
	Q1	Q0	X	Q1+	Q0+	Z
S0	0	0	0	0	0	0
	0	0	1	0	1	0
S1	0	1	0	0	0	0
	0	1	1	1	0	0
S2	1	0	0	0	0	0
	1	0	1	1	0	1

9.

primitive udp_and(a,b,out);

input a,b;

output out;table

```
    // a    b    :    out;
       0    0    :    0;
       0    1    :    0;
       1    0    :    0;
       1    1    :    1;

endtable //end state table definition

endprimitive //end of udp_and definition
```

Note: Consider Any Combinational circuit UDP

10.a)

```
module tb_decoder;

reg a,b,c,en;

wire [3:0]y;

decoder24 uut(.en(en),.a(a),.b(b),.c(c),.y(y));
initial

begin

$monitor("en=%b a=%b b=%b y=%b",en,a,b,y);

en=1;a=0;b=0; c=0#5

en=1;a=0;b=0; c=1#5

en=1;a=0;b=1; c=0 #5

en=1;a=0;b=1; c=1 #5

en=1;a=1;b=0; c=0 #5

en=1;a=1;b=0; c=1#5

en=1;a=1;b=1; c=0 #5

en=1;a=1;b=1; c=1 #5


$finish;

end

endmodule
```

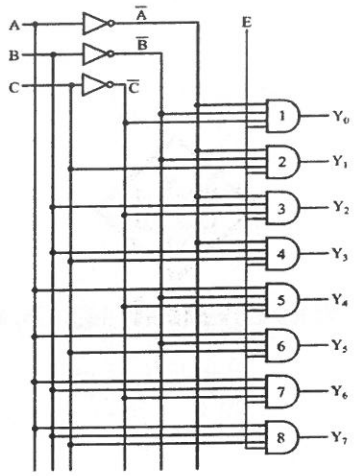
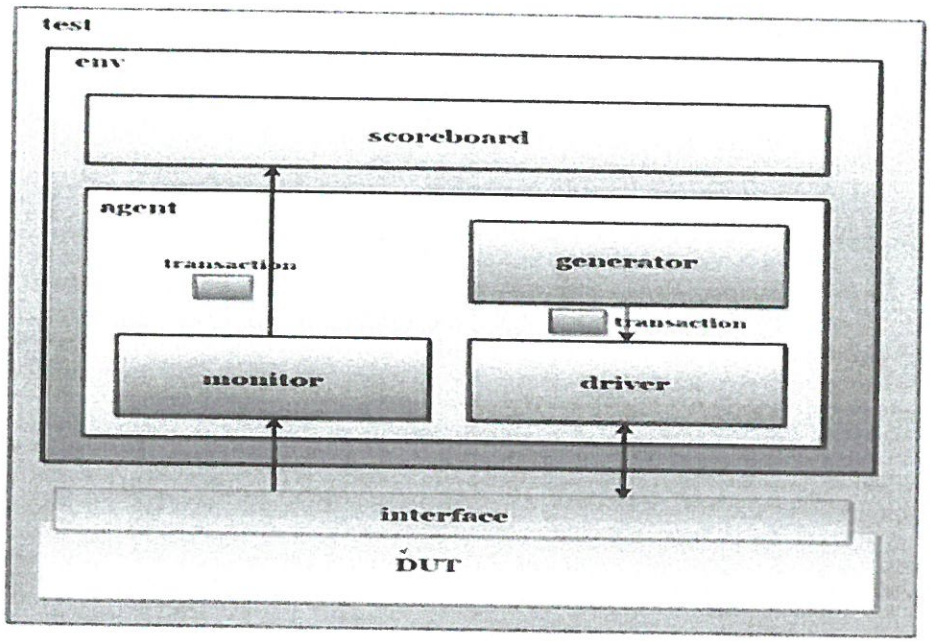


Figure 5 - 3 to 8 Decoder

Inputs				Outputs							
E	A	B	C	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	X	X	X	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

10.b)



- **Transaction**

  The transaction is a packet that is driven to the DUT or monitored by the monitor as a pin-level activity.

  In simple terms, the transaction is a class that holds a structure that is used to communicate with DUT.

- **Generator**

  The generator creates or generates randomized transactions or stimuli and passes them to the driver.

- **Driver**

  The driver interacts with DUT. It receives randomized transactions from the generator and drives them to the driven as a pin level activity.

- **Monitor**

  The monitor observes pin-level activity on the connected interface at the input and output of the design. This pin-level activity is converted into a transaction packet and sent to the scoreboard for checking purposes

Agent

- An agent is a container that holds the generator, driver, and monitor. This is helpful to have a structured hierarchy based on the protocol or interface requirement.

Scoreboard

- The scoreboard receives the transaction packet from the monitor and compares it with the reference model. The reference module is written based on design specification understanding and design behavior.

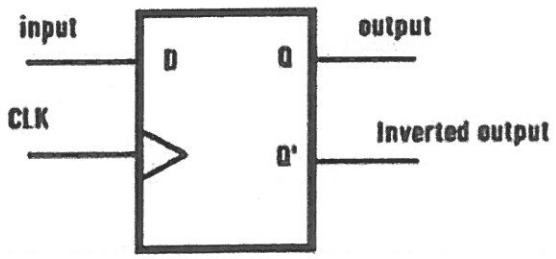
Test

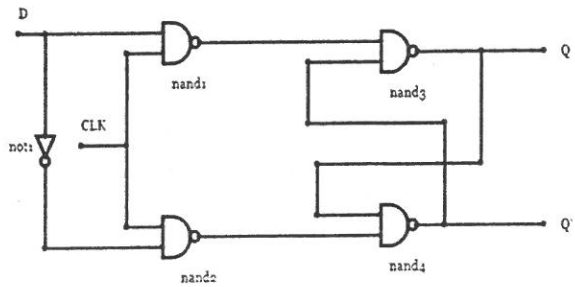
- The **top-level test class** that controls the simulation.
- Creates and configures the entire **environment (env)**.
- Starts stimulus sequences, controls test flow, and collects results.
- **DUT (Design Under Test)**
- The RTL block being verified.
- The DUT interacts with the **interface**, which is controlled by the driver and observed by the monitor.
- **Environment (env)**
- The **main container** that holds all major verification components:
    - Agents
    - Scoreboard
    - Coverage collectors

11.a)

```
module tb_dff;
  reg clk;
  reg reset;
  reg d;
  wire q;
  dff_async uut (
    .clk(clk),
    .reset(reset),
    .d(d),
    .q(q)
  );

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





```
        reset = 1;
        d = 0;
        #10 reset = 0;
        // Apply some test values
        #10 d = 1;
        #10 d = 0;
        #10 d = 1;
        #20 $finish;
    End
Endmodule
```

11.b)

output Q;

Here, **Q is not Register data type and just act as wire**

it canot store the values

Example

```
assign Q = D & clk;
```

utput reg Q;

Q is an **output port** and Q is a **reg type**, meaning it can store a value and be assigned inside an **always block**.

Example

```
output reg Q;
 always @(posedge clk) begin
   Q <= D;
 end
```