There are several algorithms for fault simulation: serial fault simulation, parallel fault simulation, and concurrent fault simulation. Next, we shall discuss each of these types of fault simulation in turn.

14.4.1 Serial Fault Simulation

Serial fault simulation is the simplest fault-simulation algorithm. We simulate two copies of the circuit, the first copy is a good circuit. We then pick a fault and insert it into the faulty circuit. In test terminology, the circuits are called **machines**, so the two copies are a **good machine** and a **faulty machine**. We shall continue to use the term *circuit* here to show the similarity between logic and fault simulation (the simulators are often the same program used in different modes). We then repeat the process, simulating one faulty circuit at a time. Serial simulation is slow and is impractical for large ASICs.

14.4.2 Parallel Fault Simulation

Parallel fault simulation takes advantage of multiple bits of the words in computer memory. In the simplest case we need only one bit to represent either a `'1'` or `'0'` for each node in the circuit. In a computer that uses a 32-bit word memory we can simulate a set of 32 copies of the circuit at the same time. One copy is the good circuit, and we insert different faults into the other copies. When we need to perform a logic operation, to model an AND gate for example, we can perform the operation across all bits in the word simultaneously. In this case, using one bit per node on a 32-bit machine, we would expect parallel fault simulation to be about 32 times faster than serial simulation. The number of bits per node that we need in order to simulate each circuit depends on the number of states in the logic system we are using. Thus, if we use a four-state system with `'1'`, `'0'`, `'X'` (unknown), and `'z'` (high-impedance) states, we need two bits per node.

Parallel fault simulation is not quite as fast as our simple prediction because we have to simulate all the circuits in parallel until the last fault in the current set is detected. If we use serial simulation we can stop as soon as a fault is detected and then start another fault simulation. Parallel fault simulation is faster than serial fault simulation but not as fast as concurrent fault simulation. It is also difficult to include behavioral models using parallel fault simulation.

14.4.3 Concurrent Fault Simulation

Concurrent fault simulation is the most widely used fault-simulation algorithm and takes advantage of the fact that a fault does not affect the whole circuit. Thus we do not need to simulate the whole circuit for each new fault. In concurrent simulation we first completely simulate the good circuit. We then inject a fault and resimulate a copy of only that part of the circuit that behaves differently (this is the **diverged circuit**). For example, if the fault is in an inverter that is at a primary out-

put, only the inverter needs to be simulated—we can remove everything preceding the inverter.

Keeping track of exactly which parts of the circuit need to be diverged for each new fault is complicated, but the savings in memory and processing that result allow hundreds of faults to be simulated concurrently. Concurrent simulation is split into several chunks, you can usually control how many faults (usually around 100) are simulated in each chunk or **pass**. Each pass thus consists of a series of test cycles. Every circuit has a unique **fault-activity signature** that governs the divergence that occurs with different test vectors. Thus every circuit has a different optimum setting for **faults per pass**. Too few faults per pass will not use resources efficiently. Too many faults per pass will overflow the memory.

14.4.4 Nondeterministic Fault Simulation

Serial, parallel, and concurrent fault-simulation algorithms are forms of **deterministic fault simulation**. In each of these algorithms we use a set of test vectors to simulate a circuit and discover which faults we can detect. If the fault coverage is inadequate, we modify the test vectors and repeat the fault simulation. This is a very time-consuming process.

As an alternative we give up trying to simulate every possible fault and instead, using **probabilistic fault simulation**, we simulate a subset or sample of the faults and extrapolate fault coverage from the sample.

In **statistical fault simulation** we perform a fault-free simulation and use the results to predict fault coverage. This is done by computing measures of observability and controllability at every node.

We know that a node is not stuck if we can make the node toggle—that is, change from a `'0'` to `'1'` or vice versa. A **toggle test** checks which nodes toggle as a result of applying test vectors and gives a statistical estimate of **vector quality**, a measure of faults detected per test vector. There is a strong correlation between high-quality test vectors, the vectors that will detect most faults, and the test vectors that have the highest **toggle coverage**. Testing for nodes toggling simply requires a single logic simulation that is much faster than complete fault simulation.

We can obtain a considerable improvement in fault simulation speed by putting the high-quality test vectors at the beginning of the simulation. The sooner we can detect faults and eliminate them from having to be considered in each simulation, the faster the simulation will progress. We take the same approach when running a production test and initially order the test vectors by their contribution to fault coverage. This assumes that all faults are equally likely. Test engineers can then modify the test program if they discover vectors late in the test program that are efficient in detecting faulty chips.

14.4.5 Fault-Simulation Results

The output of a fault simulator separates faults into several **fault categories**. If we can detect a fault at a location, it is a **testable fault**. A testable fault must be placed

The Lord is my shepherd
I shall not lack anything.

14.7.07.

Timing Controls.

↳ Timing controls provides a way to specify the simulation time at which procedural statements will execute.

↳ Three methods of timing control.

1). delay-based timing control (delay control)
2). event-based timing control (event control)
3). level-sensitive timing control (condition).

1). Delay-based.

↳ Delay-based timing control in an expression specifies the time duration b/w when the statement is encountered and when it is executed.

↳ Delays are specified by the symbol #.

1). inter-statement-
2). intra-statement-
3). zero delay [explicit zero delay]

Zero delay control.

↳ It is a method to ensure that a statement is executed last, after all other statements in that simulation time are executed.

↳ This is used to eliminate race conditions.

↳ Ex:

```
initial
  begin
    x=0;
    y=0;
  end
initial
  begin
    #0 x=1;
    #0 y=1;
  end
```

// zero delay.

↳ x=1, y=1 having #0, will be executed last. (i.e) at the end of time 0..

↳ The order in which x=1 & y=1 are executed is not deterministic.

Note: * The practice of assigning two different values to a variable in a single time step is generally not recommended and may cause race conditions in the design.

* #0 provides a useful mechanism to control the order of execution of statements in a simulation.

Event-based Timing control

↳ An event is the change in the value on a register or a net.

↳ Events can be utilized to trigger execution of a statement or a block of statements.

↳ Four types of event control.

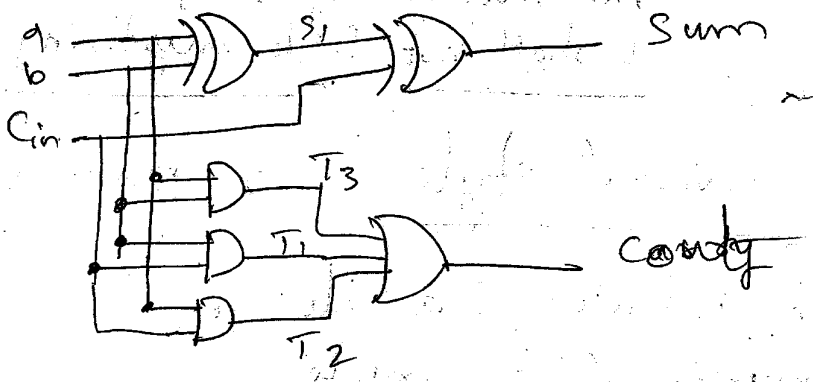
1). regular event control (edge-triggered)
2). named event control
3). event OR control
4). level-sensitive timing control

The Lord is my shepherd
I shall not lack anything. ps:23:1

Behavioral style → process flow

* The behavior of a design is described using procedural constructs.
* These are
  (i) Initial statement : executes only once.
  (ii) Always statement : always executes in a loop, (ie) the statement is executed repeatedly.
* only a register data type can be assigned a value in either of these statements.
* Register data type retains its value until a new value is assigned.
* All initial statements and always statements begin execution at time 0 concurrently.

Ex:




```
module FA_seq (A, B, Cin, Sum, Cout);
input A, B, Cin ;
output sum, Cout;
reg sum, Cout;
reg T1, T2, T3 ;
always
@ (A or B or Cin) begin
        sum = (A ^ B) ^ Cin ;
        T1 = b & Cin;
        T2 = a & cin;
        T3 = a & b;
      Cout = (T1 | T2) | T3 ;
end
endmodule
```

event control

sequential block

Note: * sum, Cout, T_1, T_2, T_3 are assigned values within always statement, so they are declared to be reg data type.

* The always statement has sequential block (begin-end pair) associated with an event control (the expression following the @ character.

Association b/w seq. block & event control

↳ Whenever an event occurs on A, B, or Cin, the sequential block is executed.

↳ Statements within a sequential block executes sequentially & the execution suspends after the last statement in the sequential block has executed.

↳ After the sequential block completes execution, the always statement again waits for an event to occur on A, B, or Cin.

↳ The statements that appear within the sequential block are examples of blocking procedural assignments.

↳ The blocking procedural assignments completes execution before the next statement executes.

↳ A procedural assignment may optionally have a delay.

(i) Inter-statement delay

(ii) Intra-statement delay.

Inter-statement delay:

↳ The delay by which a statement's execution is delayed.

↳ Ex:
```
Sum = (A ^ B) ^ Cin ;
#4 T1 = A & Cin ;
```

↳ the delay in the 2nd statement specifies that the execution of the assignment is to be delayed by 4 time units. and then executes the second assignment.

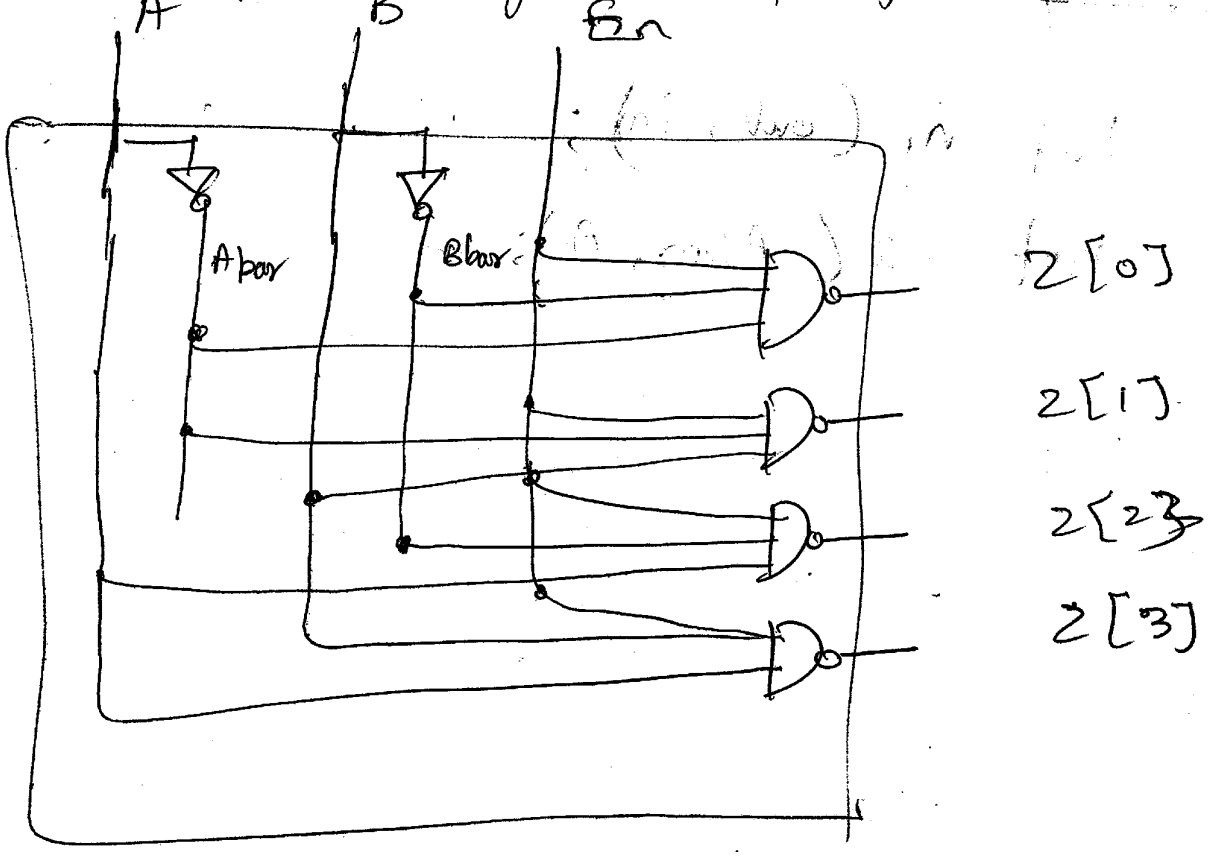
Intra-segment delay:

↳ the delay b/w computing the value of the right-hand side expression and its assignment to the left-hand side.

Ex:
```
Sum = #3 (A ^ B) ^ Cin ;
```

↳ the delay in this assignment means that the value of the RHS expression is to be computed first wait for 3 time units, and then assign the value to the Sum.

↳ If no delays are specified in a




```
module decoder2x4 (A, B, En, z);
input A, B, En;
output [0:3]z;
wire Abar, Bbar;
assign #1 Abar = ~A;
assign #1 Bbar = ~B;
assign #2 z[0] = ~( Abar & Bbar & En);
assign #2 z[1] = ~ (Abar & B & En);
assign #2 z[2] = ~ (A & Bbar & En);
assign #2 z[3] = ~ (A & B & En);
endmodule
```

buf/ not gates.

```
buf n1 (out, in);
not n2 ( Abar, A);
```

The Lord is my shepherd
I shall not lack anything. Ps: 23:1

Initial statement

↳ It executes only once.

↳ It ~~executes~~ begins its execution at start of simulation which is at time 0.

↳ The syntax for the initial statement is:

```
initial
    [ timing_control] procedural_statement
```

* procedural statement can be one of the following statements.

procedural_assignment (blocking or non-blocking)
procedural_continuous_assignment
Conditional_statement
case_statement
loop_statement
wait_statement
disable_statement
event_trigger
Sequential_block
Parallel_block
task_enable (user or system)

↳ The sequential block is the most commonly used procedural statement.
[begin.... end]

↳ The timing control can be a

1) delay control : wait for a certain time.

2). event control : wait for an event to occur or

3) condition to become true.

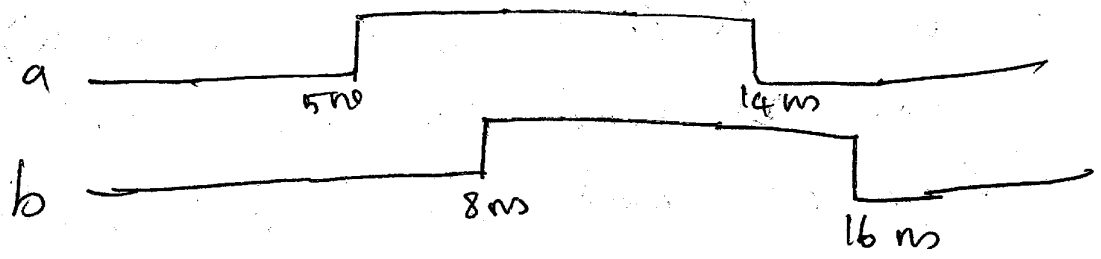
↳ The execution of an initial statement causes the procedural statement to execute once.

↳ The initial statement starts execution at time 0.

Ex:

```
' timescale 1ns / 1ns
module Test (a, b);
output a, b;
reg a, b;
initial
  begin
      a = 0 ;
      b = 0 ;
      a = #5 1 ;
      b = #3 1 ;
      a = #6 0 ;
      b = #2 0 ;
  end
endmodule.
```

↳ This module generates the following waveform as follows.



↳ The initial statement contains a sequential block which starts execution at time 0ns and after it completes executing all statements within the sequential block, the initial statement suspends forever.

↳ In example, the sequential block contains blocking procedural assignments with intra-statement delays specified.

Mixed-design style.

↳ Within a module, structural and behavioral constructs can be mixed freely.

↳ A module can contain a mixture of gate instantiations, module instantiations, continuous assignments & always & initial statements amongst others.

↳ Values from always and initial statements can drive gates or switches

↳ Values from gates or continuous assignments (can only drive nets) can in turn be used to trigger always statements and initial statements.

1-bit FA.

```
module FA_mix (A, B, Cin, Sum, Cout);
 input A, B, Cin;
 output sum, Cout;
 reg Cout;
 reg T1, T2, T3;
 wire S1;

 xor x1 (S1, A, B);          // Gate instantiation

 always                      // always statement
     @ (A or B or Cin) begin
     T1 = A & Cin;
     T2 = B & Cin;
     T3 = A & B;
     Cout = (T1 | T2) | T3;
    end
 assign sum = S1 ^ Cin;      // continuous Assignment
 endmodule
```

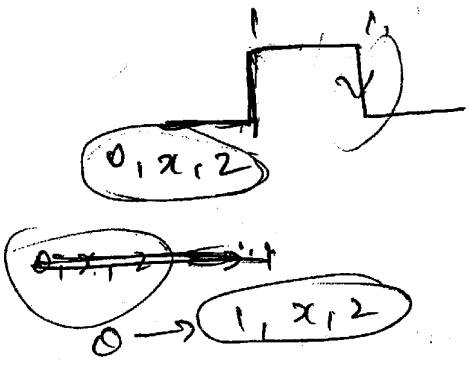
Regular event control

↳ @ symbol is used to specify an event control

↳ statements can be executed on changes in signal value or at +ve or -ve transition of the signal value.

Eg:

```
@ (clock) q = d ;
@ (posedge clock)  q = d;
@ (negedge clock)  q = d;
q = @ (posedge clock)  d;   // d is assigned to q at +ve edge of clock.
```



1 → 0
x → 0
z → 0

1 → 0, x, z.

x → 1
z → 1

Named event control.

↳ verilog provides the capability to declare an event and then trigger and recognize the occurance 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 @.

Exc: Data buffer storing data after the last packet of data has arrived.

```
event received_data ;
always @(posedge clock)
begin
   if ( last_data_packet)
        -> received_data ;

   end
always @ (received_data)
   data_buf = { data_pkt[0], data_pkt[1],
        data_pkt[2], data_pkt[3] };
```

Event OR control.

↳ A transition on any one of multiple signals 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 known as a sensitivity list.

↳ The keyword or is used to specify multiple triggers

Ex:

```
always @( reset or clock or d) :
begin
    if ( reset)
        q = 1'b0;
    else if (clock)
        q = d;
end
```

Level-sensitive Timing Control.

↳ The ability to wait for a certain condition to be true before a statement or a block of statements is executed.

↳ The keyword wait is used for level-sensitive constructs.

```
always
    wait (count_enable) #20 count=count+1;
```

↳ count_enable is monitored continuously.

* If count_enable is 0 the statement is not executed.
* If it is logical 1, the statement count = count+1 is executed after 20 time unit

Conditional statements.

↳ used for making decisions based upon certain conditions.

↳ used to decide whether or not a statement should be executed

↳ keyword if and else are used for conditional statements.

Syntax:

```
if (expression) statement;
```

```
if (exp) statement-1;
   else statement2;
```

```
if (exp1) statement 1;
else if(exp2) statement-2;
else if (exp3) statem 3;
else default statement;
```

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 a different value.
- ↳ not like continous assignment The value of net is continously placed.

Two types:

(1). Blocking assignment

(2) non-Blocking assignment

Blocking:

↳ Blocking assignment statements are executed in the order they are specified in a sequential block.

↳ These statements will not block execution of statements that follow in a parallel block.

↳ The operator = is used.

~~Non~~

Ex:

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
initial
begin
  x=0; y=1; z=1;
  count=0;
  reg_a = 16'b0; reg_b = reg_a;
  #15 reg_a[2] = 1'b1
  #10 reg_b[15:13] = {x, y, z}
  count = count+1;
end
```

-15
-10
-25

Non-blocking.

↳ It allow scheduling of assignments without blocking execution of the statements that follow in a sequential block.

↳ A <= operator is used to specify nonblocking assignments.

Ex:

```
reg  x, y, z ;
reg [15:0] reg_a, reg_b;
integer count;
initial
  begin
     x=0; y=1; z=1;
     count = 0;
     reg_a = 16'b0; reg_b = reg_a;
     reg_a[2] <= #15 1'b1;
     reg_b[15:13] <= #10 {x,y,z};
     count <= count + 1;
  end.
```

Initial statement.

* Syntax:
* Timing control. : delay, event, condition.
* procedural statements.

DFF

```
module dff (d, clk, q);
input d;
input clk;
output q;
reg q;
always @ (posedge clk)
q=d;
endmodule
```

delay control.

- is a method to ensure that a statement is executed last, after

zero

↳ It
st

St. Joseph's College of Engineering
Department of ECE

EC1401: VLSI DESIGN
Question Bank

PART-A	
UNIT-I	
1	What are the materials used for masks in IC technology?
2	What are the advantages of twin tub process?
3	What are the advantages of EBL pattern generation?
4	What are the types of oxidation?
5	What are the types of etching process?
6	What are the advantages of SOI CMOS process?
7	What is a thinox?
8	How is the channel stop implant made?
9	What are the types of interconnect?
10	What is BiCMOS?
11	What is latch-up?
12	List the two techniques that can be used to prevent latch-up.
13	What is guard ring?
14	How is a capacitor created in CMOS fabrication process?
15	How is a resistor created in CMOS fabrication process?
16	What is the objective of the layout rules?
17	What are the types of layout design rules?
18	What is scribe line?
19	What is passivation or overglass?
20	List the four main CMOS technologies.
21	Draw the circuit of a CMOS 2-input NAND gate.
22	Distinguish electrically alterable and non-electrically alterable ROM.
23	List the sequence of steps to create the physical layout of an inverter.
24	Draw the physical layout of 2-input NOR gate
25	What is Lambda-Based design rule?
UNIT-II	
1	Draw the structure of a nmos enhancement transistor.
2	Draw the characteristics of a n- channel & p – channel enhancement transistor.
3	What are the advantages of SiO_2 as a dielectric?
4	State the parameters on which threshold voltage is dependent on.
5	What is a field induced junction?
6	What are the factors that influence the drain current.
7	Define cut – off and saturated regions in the characteristics of an MOS transistor.
8	Define threshold voltage

9	Define Body effect.
10	What is channel length modulation?
11	Draw the small signal model of a MOS transistor.
12	Define noise margin. Illustrate how it can be obtained from the transfer characteristics of a CMOS Inverter.
13	What is a transmission gate.
14	Why is the transmission of logic 1 degraded as it passes through a nmos pass transistor.
15	Why is the transmission of logic 0 degraded as it passes through a pmos pass transistor?
16	Draw the structure & symbol of a CMOS tri - state inverter.
17	Write the equation for total power dissipation.
18	Define rise time (t_r).
19	Define fall time (t_f)
20	Define delay time (t_d).
21	Compare NMOS & PMOS devices
22	Compare enhancement and depletion mode devices.
23	What is static power dissipation?
24	What is dynamic power dissipation?
25	List three modes of MOS transistor.
	UNIT-III
1	What is meant by continuous assignment statement in Verilog HDL?
2	List out different data types in verilog.
3	What are the types of modeling in verilog?
4	What is HDL? Name two types of HDLs.
5	What is Module?
6	What is a task in Verilog?
7	What is the key difference between an initial statement and an always statement?
8	List any two capabilities of Verilog.
9	What is value set?
10	Define net.
11	What is UDP?
12	What are the types of event control?
13	Define an identifier
14	Define a port. How is it declared?
15	What are the types of structural modeling?
16	List the types of Gate delay.
17	What are the types of procedural assignment?
18	What is the difference between reduction operator and bitwise operator?
19	What is inter-statement delay?
20	What is intra-statement delay?
21	Give the structural coding of an Half adder.
22	Give the behavioral coding of 2-input AND gate.
23	Give the symbol, Truth table and syntax of bufif1.
24	Write the syntax for gate primitive instantiation.
25	What is replication operator?
	UNIT-IV

1	Draw the symbols of N & P – switches
2	What is a crowbarred state?
3	Draw a CMOS NAND gate with its pull – up & pull – down Truth table.
4	Draw the symbol of a 2 i/p CMOS Mux.
5	Draw a CMOS positive level sensitive D latch.
6	What is 22V10?
7	What is CLB?
8	When is a full custom ASIC designed?
9	What is a wafer lot?
10	What is a mega cell?
11	What is a gate array?
12	What are the types of Masked gate arrays
13	List the features of Channeled gate array, Channelless gate array, and Structured gate array.
14	List the features of PLD.
15	What are the types of PLD's?
16	What are the Essential features of FPGA?
17	What FAMOS?
18	What is the difference between Full-custom ASIC and semi-custom ASIC?
19	What is the difference between CBIC and MGA?
20	What is SOG?
21	What is Antifuse?
22	What is PLICE?
23	What is ViaLink?
24	What is the disadvantage of Embedded Gate Array
25	Mention the characteristics of 22V10
	UNIT-V
1	List some typical defects in the manufacturing of IC.
2	What are SA0 & SA1 faults?
3	Define observability and controllability.
4	What is fault sampling?
5	What is a sensitized path?
6	What are primary i/p's and o/p's?
7	State the 3 types of fault simulation process.
8	What is delay fault testing?
9	What is ILA testing?
10	What are the advantages & disadvantages of IDDQ?
11	What are the various connections of a Test Access Port.? (TAP).
12	Describe the test architecture of a boundary scan using TAP.
13	What is a TAP controller?
14	Explain BYPASS ,EXTEST, SAMPLE /PRELOAD.
15	What is a test DR?
16	What are boundary scan registers?
17	What is short circuit and open circuit faults?
18	What is JTAG?

19	What is Fault coverage?
20	Expand ATPG & SCOAP
21	How are sequential faults caused in CMOS?
22	What is BIST?
23	What is LSSD?
24	What is concurrent simulation?
25	List the three approaches of Design for testability
	PART-B
	UNIT-I
1	(i) Draw and explain the n-well process (10) (ii) Explain the twin tub process with a neat diagram.(6)
2	Explain the various steps involved in p-well CMOS fabrication with necessary diagrams (16)
3	Explain the terms (16) (a) Oxidation (b) Epitaxy (c) Ion implantation (d) Diffusion
4	(i) Explain SOI process in detail. (10) (ii) What are the advantages of SOI process? (6)
5	(i) What is interconnect? Explain different types of interconnect. (8) (ii) Explain the fabrication of circuit elements in detail. (8)
6	(i) Discuss the origin of latch-up problems in CMOS circuits with necessary diagrams. Explain the remedial measures.(10) (ii) Draw and explain briefly the n-well CMOS design rules. (6)
7	Explain a sequence of steps to create the physical layout of an inverter. NAND & NOR gates with neat diagram.(16)
8	Explain Layout design rules. (16)
9	Write notes on (i) Design rule Backgrounder(6) (ii) CAD tool sets(5) (iii)Design Hierarchy(5)
10	(i) What is latch-up & how is it triggered? (ii) State & Explain the condition for prevention of latch-up.
	UNIT-II
1	Explain the operation of NMOS enhancement MOSFET with neat characteristics curves. (16)
2	Explain the CMOS inverter DC characteristics.(16)
3	Write notes on (i) The transmission gate. (8) (ii) Noise margin for a CMOS Inverter. (8)
4	Derive the expression for the fall time and rise time of a CMOS inverter. (16)
5	Explain the operation of PMOS enhancement MOSFET with neat characteristics curves. (16)
6	(i) Explain body effect. (6) (ii) Derive the expression for threshold voltage for nMOS transistor

7	Derive the DC equations for a NMOS transistor (10).	
8	(i)	Discuss the small signal model of an MOS transistor .(8)
	(ii)	Explain Static power dissipation in a CMOS inverter(8)
9	Explain power dissipation in a CMOS inverter. (16)	
10	Write notes on	
	(i)	Channel length modulation & Mobility variation (8)
	(ii)	MOS models & tristate inverter (8)
	UNIT-III	
1	(i)	Give a Verilog structural gate level description of 2-bit magnitude comparator. (10)
	(ii)	Give a brief account of timing control and Gate delay in Verilog. (6)
2	Write notes on (i) VLSI Design flow. (8) (ii) Design Hierarchy. (8)	
3	(i)	Explain about the switch primitives used in Verilog HDL (8)
	(ii)	Write the behavioral and structural description of 4X1 MUX in Verilog HDL. (8)
4	(i)	Explain Behavioral RTL Modeling with an example. (8)
	(ii)	Explain switch level modeling with an example. (8)
5	(i)	Construct CMOS 2-input NAND and NOR gate with required truth tables. Write the verilog code using NMOS and PMOS primitives.(12)
	(ii)	Explain how to model resistive elements and power supply in verilog HDL.(4)
6	(i)	Give a Verilog structural gate level description of a ripple carry adder.(10)
	(ii)	Write a brief note on the conditional statements available in Verilog. (6)
7	(i)	Write Verilog code in structural and Behavioral modeling to describe the operation of a priority encoder(10)
	(ii)	Write Verilog code in data flow model to describe the operation of equality detector. (6)
8	Explain different types of operators in Verilog HDL with example(16)	
9	(i)	Explain the number specification methodology with examples for verilog HDL. (6)
	(ii)	Explain major capabilities of verilog HDL. (10)
10	(i)	Describe the operation of D-FF in Behavioral Modeling. (6)
	(ii)	Describe the operation of Full adder in data flow modeling.(6)
	(iii)	Write notes on procedural assignments. (4)
	UNIT-IV	
1	(i)	Draw the ASIC design flow and explain(8)
	(ii)	Draw the structure of Xilinx Configurable Logic block (8)
2	Explain the types of ASICs with neat diagram (16)	
3	Write notes on programmable logic devices. (16)	
4	Explain the architecture of reprogrammable gate array(16)	
5	(i) Write short notes on Antifuse and ViaLink (8)	

	(ii) Explain the working of a programmable interconnect using pass transistors. (8)
6	(i) Explain Hot Electron Injection with neat diagram(8) (ii) Implement the function (a) Z=((AB+C)D)' (4) (b) Z=(A+B+C+D)' (4)
7	(i) Design a positive edge triggered D register using Transmission gates & explain its operation (8) (ii) Design CMOS logic gates for the given function Z=((A.B)+C.(A+B))' (8)
8	Construct and explain 2X1 MUX and 4X1 MUX using transmission gates
9	(i) Explain the architecture of 22V10 (8) (ii) Explain the programming techniques of PALs(8)
10	(i) Explain Actel Programmable I/O Pad(8) (ii) Explain the structure of Actel logic element and QuickLogic logic cell. (8)
	UNIT-V
1	(i) What is the need for Testing? (4) (ii) Write notes on (i) Functionality Test (6) (ii) Manufacturing Test (6)
2	(i) Explain the different fault models.(8) (ii) Explain ATPG. (8)
3	Explain self-test Techniques.(16)
4	Explain the principles of Ad-hoc Testing. (16)
5	Explain system level test techniques. (16)
6	(i) Explain the SCOAP algorithm for assigning controllabilities & observabilities of a logic circuit (8) (ii) Explain Fault Grading and Fault simulation (8)
7	(i) What is DFT? Explain Statistical Fault Analysis and Fault Sampling.(12) (ii) What is IDDQ testing? (4)
8	(i) Explain LSSD.(8) (ii) Explain how serial scan testing is implemented.(8)
9	Write notes on (i) Partial Serial scan(8) (ii) Parallel scan(8)
10	Explain the chip level test techniques. (16)