**Chapter 2: Hierarchical Modeling Concepts  
  
2.8 Exercises**  
**1. An interconnect switch (IS) contains the following components, a shared memory (MEM), a system controller (SC) and a data crossbar (Xbar).  
a) Define the modules MEM, SC, and Xbar, using the module/endmodule keywords. You do not need to define the internals. Assume that the modules have no terminal lists.**

**Ans**:   
module MEM; // No port list

MEM

// No internals  
endmodule  
  
module SC; // No port list

SC

// No internals  
endmodule   
  
module Xbar; // No port list

Xbar

// No internals  
endmodule  
  
**b) Define the module IS, using the module/endmodule keywords. Instantiate the modules MEM, SC, Xbar and call the instance mem1, sc1, and xbar1, respectively. You do not need to define the internals. Assume that the module IS has no terminals.**

**Ans:**

SC

sc1

IS

module IS;  
//instantiate the modules  
MEM mem1( );  
SC sc1( );  
Xbar xbar1( );

Xbar

Xbar1

MEM

mem1

endmodule  
 **c) Define a stimulus block (Top), using the module/endmodule keywords. Instantiate the design block IS and call the instance is1. This is the final step in building the simulation environment.**

**Ans:**  
module Top;  
//Top level block instantiates module IS  
IS is1;  
endmodule

IS

Is1

Top

**2. A 4-bit ripple carry adder (Ripple\_Add) contains four 1-bit full adders (FA).  
  
a) Define the module FA. Do not define the** **internals or the terminal list.**

**Ans:**

module FA;

FA

// No internals  
  
endmodule;  
  
**b) Define the module Ripple\_Add. Do not define the internals or the terminal list. Instantiate four full adders of the type FA in the module Ripple\_Add and call them fa0, fa1, fa2, fa3.**  
**Ans:**  
**module Ripple\_Add;  
  
FA fa0;  
FA fa1;  
FA fa2;  
FA fa3;  
  
endmodule**

FA

FA

fa0

FA

fa3

FA

fa2

FA

fa1

**Chapter 3: Chapter 3. Basic Concepts**  
  
**3.5 Exercises**  
  
**1. Practice writing the following numbers:  
  
a) Decimal number 123 as a sized 8-bit number in binary. Use \_ for readability.**

1. 123=8’b0111\_1011

**b) A 16-bit hexadecimal unknown number with all x’s.**

**A**) 16’hx **c) A 4-bit negative 2 in decimal. Write the 2’s complement form for this number.**

**A) -**4’d2=4’b1110 **d) An unsized hex number 1234.**

**A)** 32’h1234 or ‘h1234

**2. Are the following legal strings? If not, write the correct strings.  
  
a) “This is a string displaying the % sign”**

1. “This is a string displaying the %% sign”

**b) “out=in1+in2”  
A)** Right **c) “Please ring a bell \007”**

**A)** Right **d) “This is a backslash \ character\n**”

**A)** “This is a backslash \\ character”  
  
**3. Are these legal identifiers ?  
a) system1  
A) Right  
b) 1reg  
A) wrong  
c) $latch**

**A) wrong  
d) exec$**

**A) Right**

**4. Declare the following variables in Verilog:  
  
a) An 8-bit vector net called a\_in.  
A)** wire [7:0]a\_in;  
**b) A 32-bit storage register called address. Bit 31 must be the most significant bit. Set the value of the register to a 32-bit decimal number equal to 3.**

1. reg [31:0] address=32’d3;

**c) An integer called cout.**

**A)** integer cout;  
**d) A time variable called snap\_shot.**

**A)** time snap\_shot;  
**e) An array called delays. Array contains 20 elements of the type integer.**

**A)** integer delays [0:19];  
**f) A memory MEM containing 256 words of 64 bits each.**

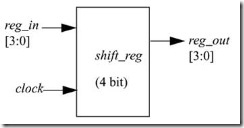
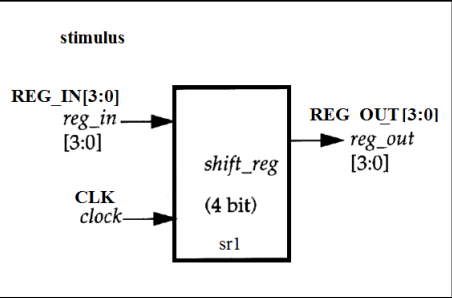
**A)** reg [63:0] MEM [0:255]; **g) A parameter cache\_size equal to 512.**

**A)** parameter cache\_size=512;  
  
**5. What would be the output/effect of the following statements ?  
  
a) latch = 4’d12;  
  
$display(“The current value of latch = %b\n”, latch);**

**A)** The current value of latch =4’b1100  
 **b) in\_reg=3’d2;  
  
$monitor($time, “ In register value = %b\n”, in\_reg[2:0]);**

**A)** b) 0 In register value = 3’b010 **c) `define MEM\_SIZE 1024  
  
$display(“ The maximum memory size is %h”, `MEM\_SIZE);**

**A)** The maximum memory size is ‘h400

**Chapter 4. Modules and Ports  
  
4.5 Exercises  
  
1. What are the basic components of a module? Which components are mandatory?**  
**A)** Module Name, Port List, Port Declarations, Parameters, Declarations of wires, regs and other variables, Data flow statements, Instantiation of lower-level modules, always and initial blocks, Tasks and functions, endmodule statement. module, module name,and endmodule.  
  
**2. Does a module that does not interact with its environment have any I/O ports? Does it have a port list in the module definition?**  
  
**A**) No need has ports., no  
  
**3. A 4-bit parallel shift register has I/O pins as shown in the figure bellow. Write the module definition for this module shift\_reg. Include the list of ports and port declarations. You do not need to show the internals.**  
  
**A) `**  
module shift\_reg (reg\_out, reg\_in, clock);  
  
output reg [3:0]reg\_out;  
  
input [3:0]reg\_in;  
  
input clock;  
//no internals  
endmodule  
**4. Declare a top-level module stimulus. Define REG\_IN(4-bit)and CLK(1-bit)as reg register variables and REG\_OUT(4-bit)as wire. Instanitiate the module shift\_reg and call it sr1. Connect the ports by ordered list.**  
  
  
**A)**module stimulus;  
  
reg [3:0]REG\_IN;  
  
reg CLK;  
  
wire [3:0]REG\_OUT;  
  
shift\_reg sr1(REG\_OUT, REG\_IN, CLK);

// initial block goes here  
  
endmodule  
**5. Connect the ports in Step 4 by name.**

**A)** shift\_reg( .reg\_out(REG\_OUT), .reg\_in(REG\_IN), .clock(CLK));  
**6.Write the hierarchical names for variables REG\_IN,CLK,REG\_OUT;  
A)** stimulus.REG\_IN,

stimulus.CLK,

stimulus.REG\_OUT  
**7. Write the hierarchical name for the instance sr1. Write the hierarchical names for its ports clock and reg\_in.**

1. stimulus.sr1,  
     
   stimulus.sr1.clock  
     
   stimulus.sr1.reg\_in

**Chapter 5. Gate-level Modeling  
5.4 Exercises  
1. Create your own 2-input Verilog gates called my\_or, my\_and and my\_not from 2-input nand gates. Check the functionality of these gates with a stimulus module.**

A)We can define the basic logic gates using NAND as follows:

1. NOT(A) = A NAND AB

A ~A

1. AND(A, B) = NOT(A NAND B) = (A NAND B) NAND (A NAND B)

 A

B

1. ****OR(A, B) = (NOT A) NAND (NOT B) = (A NAND A) NAND (B NAND B)

**Ans) //** Design code

module my\_not(output Y, input A);

nand(Y, A, A); // NOT A = A NAND A

endmodule

module my\_and(output Y, input A, B);

wire nand\_out;

nand(nand\_out, A, B); // NAND(A, B)

nand(Y, nand\_out, nand\_out); // NOT(NAND(A, B))

endmodule

module my\_or(output Y, input A, B);

wire not\_A, not\_B;

nand(not\_A, A, A); // NOT A

nand(not\_B, B, B); // NOT B

nand(Y, not\_A, not\_B); // OR(A, B) = (NOT A) NAND (NOT B)

endmodule

**//Stimulus**

`timescale 1ns / 1ps

module test\_my\_gates;

reg A, B;

wire Y\_not, Y\_and, Y\_or;

// Instantiate the modules

my\_not U1 (Y\_not, A);

my\_and U2 (Y\_and, A, B);

my\_or U3 (Y\_or, A, B)

initial begin

$dumpfile("test\_my\_gates.vcd");

$dumpvars(0, test\_my\_gates)

// Apply test cases

$display("A B | NOT(A) AND(A,B) OR(A,B)");

$monitor("%b %b | %b %b %b", A, B, Y\_not, Y\_and, Y\_or)

A = 0; B = 0; #10;

A = 0; B = 1; #10;

A = 1; B = 0; #10;

A = 1; B = 1; #10;

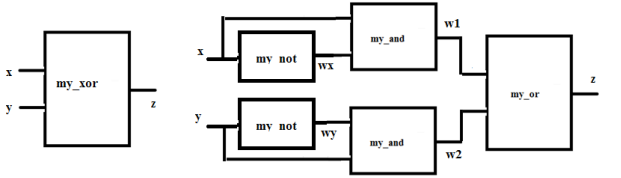
$finish;

end

endmodule

**2. A 2-input xor gate can be built from my\_and, my\_or and my\_not gates. Construct an xor module in Verilog that realizes the logic function, z=xy’+x’y. Inputs are x and y, and z is the output. Write a stimulus module that exercises all four combinations of x and y inputs.**

**A).**

****

module my\_xor(z,x, y);

output z;

input x, y;

wire wx,wy,w1,w2;

//implementation out = (in1' . in2')'

my\_not n1(wx,x);

my\_not n2(wy,y);

my\_and a1(w1, x, wy);

my\_and a2(w2, wx, y);

my\_or o1(z, w1, w2);

endmodule

module my\_and(out, in1, in2);

output out;

input in1, in2;

wire a;

//implementation out = (in1.in2)''

nand x1 (a, in1,in2);

nand x2(out, a, a);

endmodule

module my\_or(out, in1, in2);

output out;

input in1, in2;

wire a, b;

//implementation out = (in1' + in2')'

nand x1 (a, in1,in1);

nand x2(b, in2, in2);

nand x3(out, a, b);

endmodule

module my\_not(out, in1);

output out;

input in1;

//implementation out = (in1')

nand x1 (out, in1,in1);

endmodule

**// Stimulus (top-level module)**

module stimulus;

// Declare variables

reg x, y;

wire z;

// Instantiate the module D

my\_xor d1( z, x, y);

// Stimulate the inputs. Finish the simulation at 40 time units.

initial

begin

$monitor ($time, "\t x=%b y=%b z=%b\n",x,y,z);

x= 1'b0; y= 1'b0;

#10 x= 1'b0; y= 1'b1;

#10 x= 1'b1; y= 1'b0;

#10 x= 1'b1; y= 1'b1;

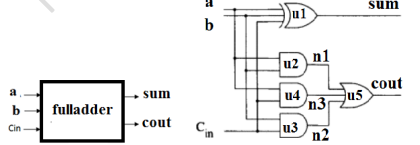
#20 $finish;

end

endmodule

**3) The 1-bit full adder described in the chapter can be expressed in a sum of products form. sum = a.b.cin + a'.b.cin' + a'.b'.cin + a.b'.cin' cout = a.b + b.cin + a.cin Assuming a, b, cin are the inputs and sum and cout are the outputs, design a logic circuit to implement the 1-bit full adder, using only and, not, and or gates. Write the Verilog description for the circuit. You may use up to 4-input Verilog primitive and and or gates. Write the stimulus for the full adder and check the functionality for all input combinations**

**A).**

****  
  
// Define a 1-bit full adder

module fulladder (sum,cout, a, b, cin);

output sum, cout;

input a, b,cin;

wire n1,n2,n3;

xor u1(sum,a,b,cin);

and u2(n1,a,b);

and u3(n2,b,cin);

and u4(n3,a,cin);

or u5(cout,n1,n2,n3);

endmodule

//Stimulus

module stimulus;

reg A,B,C\_IN;

wire SUM,C\_OUT;

begin

fulladder fa(SUM,C\_OUT,A,B,C\_IN);

$monitor($time,"\t A= %b, B=%b, C\_IN= %b, C\_OUT= %b, SUM= %b\n", A, B,

C\_IN, C\_OUT,

SUM);

initial

begin

A = 1'b0; B = 1'b0; C\_IN = 1'b0;

#10 A = 1'b0; B = 1'b0; C\_IN = 1'b1;

#10 A = 1'b0; B = 1'b1; C\_IN = 1'b0;

#10 A = 1'b0; B = 1'b1; C\_IN = 1'b1;

#10 A = 1'b1; B = 1'b0; C\_IN = 1'b0;

#10 A = 1'b1; B = 1'b0; C\_IN = 1'b1;

#10 A = 1'b1; B = 1'b1; C\_IN = 1'b0;

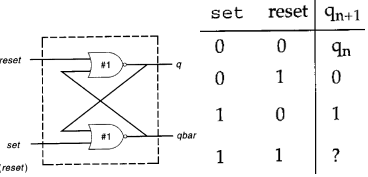
#10 A = 1'b1; B = 1'b1; C\_IN = 1'b1;

#10 $stop;

end

endmodule

**4. The logic diagram for an RS latch with delay is shown below. Write the Verilog description for the RS latch. Include delays of 1 unit when instantiating the nor gates. Write the stimulus module for the RS latch, using the following table, and verify the outputs.**

**A).**

module RS\_LATCH(q,qbar,set,reset);

output q,qbar;

input set,reset;

nor #1 x1(q,reset,qbar);

nor #1 x2(qbar,set,q);

endmodule

module stimulus;

wire q,qbar;

reg set,reset;

RS\_LATCH r1(q,qbar,set,reset);

initial

begin

$monitor($time, "\t set=%b reset=%b q=%b qbar=%b\n",set,reset,q,qbar);

set=1'b0; reset=1'b0;

#10 set=1'b0; reset=1'b1;

#10 set=1'b1; reset=1'b0;

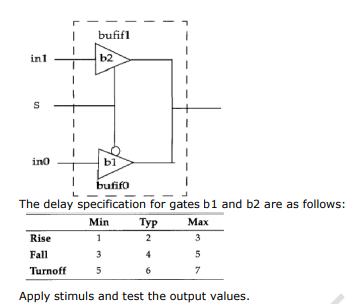
#10 set=1'b1; reset=1'b1;

#20 $stop;

end

endmodule

**5) Design a 2-to-1 multiplexer using bufif0 and bufif1 gates as shown below.**

****

**A)**

module mux2\_1(out, in0, in1, s);

output out;

input in0,in1,s;

bufif0 #(1:2:3,3:4:5,5:6:7) b1(out,in0,s);

bufif1 #(1:2:3,3:4:5,5:6:7) b2(out,in1,s);

endmodule

**//Stimulus**

module stimulus;

wire out;

reg in0,in1,s;

mux2\_1 x1(out,in0,in1,s);

initial

begin

$monitor($time, "\t in0=%b in1=%b s=%b out=%b\n",in0,in1,s,out);

in0=1'b0;in1=1'b1;s=1'b0;

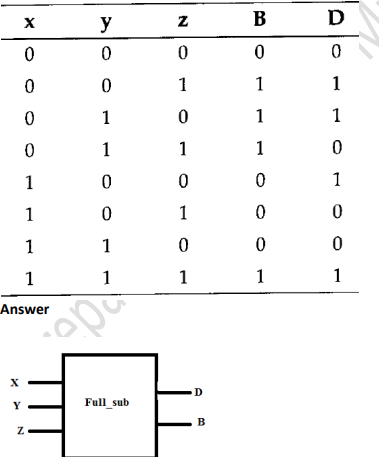
#10 s=1'b1;

#10 s=1'b0;

#20 $stop;

end

endmodule

**A). **

module Full\_sub(D,B,x,y,z);

output D,B;

input x,y,z;

//D = x'.y'.z + x'.y.z' + x.y'.z' + x.y.z

assign D =((~x)&(~y)&z) | ((~x)&y&(~z)) |(x&(~y)&(~z)) | (x&y&z);

//B = x'.y + x'.z +y.z

assign B =(~x&y) | (~x&z) | (y&z);

endmodule

**//TEST BENCH**

module stimulus;

reg x,y z;

wire D, B;

Full\_sub uut ( D,B,x,y,z);

initial

begin

$monitor($time, "\tinput xyz=%b output BD=%b\n",{x,y,z},{B,D});

// Initialize Inputs

x = 0;

y = 0;

z = 0;

#10 {x,y,z} = 3'b001;

#10 {x,y,z} = 3'b010;

#10 {x,y,z} = 3'b011;

#10 {x,y,z} = 3'b100;

#10 {x,y,z} = 3'b101;

#10 {x,y,z} = 3'b110;

#10 {x,y,z} = 3'b111;

#20 $stop;

end

endmodule

**2. A magnitude comparator checks if one number is greater than or equal to or**

**less than another number. A 4-bit magnitude comparator takes two 4-bit**

**numbers, A and B, as input. We write the bits in A and B as follows. The**

**leftmost bit is the most significant bit.**

**A=A(3)A(2)A(1)A(0)**

**B=B(3)B(2)B(1)B(0)**

**The magnitude can be compared by comparing the numbers bit by bit, starting**

**with the most significant bit. If any bit mismatches, the number with bit 0 is**

**the lower number. To realize this functionality in logic equations, let us define**

**an intermediate Variable. Notice that the function below is an xnor function**

**x(i)=A(i)B(i)+A(i)’B(i)’**

**The three outputs of the magnitude comparator are A\_gt\_B,A\_lt\_B,A\_eq\_B.**

**They are define with the following logic equations:**

**A\_gt\_B=A(3)B(3)’ + x(3).A(2).B(2)' + x(3).x(2).A(1).B(1)' + x(3).x(2).x(1).A(0).B(0)'**

**A\_lt\_B = A(3)'.B(3) + x(3).A(2)'.B(2) + x(3).x(2).A(1)'.B(1) +**

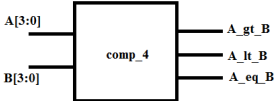
**x(3).x(2).x(1).A(0)'.B(0)**

**A\_eq\_B = x(3).x(2).x(1).x(0)**

**Write the Verilog description of the module magnitude\_comparator.**

**Instantiate the magnitude comparator inside the stimulus module and try out a**

**few combinations of A and B**

****

**A).**

module comp\_4(A\_gt\_B,A\_lt\_B,A\_eq\_B,A,B);

output A\_gt\_B,A\_lt\_B,A\_eq\_B;

input [3:0]A,B;

wire [3:0] x;

assign x = ~(A ^ B);

assign A\_gt\_B=(A[3]&(~B[3])) | (x[3]&A[2]&(~B[2])) | (x[3]&x[2]&A[1]&(~B[1]))

| (x[3]&x[2]&x[1]&A[0]&(~B[0]));

assign A\_lt\_B=((~A[3])&B[3]) | (x[3]&(~A[2])&B[2]) | (x[3]&x[2]&(~A[1])&B[1])

| (x[3]&x[2]&x[1]&(~A[0])&B[0]);

assign A\_eq\_B = &x;

endmodule

**//TestBench**

module stimulus;

reg [3:0] A,B;

wire A\_gt\_B,A\_lt\_B,A\_eq\_B;

// Instantiate the Unit Under Test (UUT)

comp\_4 x1 (A\_gt\_B,A\_lt\_B,A\_eq\_B,A,B);

initial begin

// Initialize Inputs

A = 4'b0000;

B = 4'b0000;

$monitor($time, "\t A=%b B=%b A\_gt\_B=%b A\_lt\_B=%b

A\_eq\_B=%b\n",A,B,A\_gt\_B,A\_lt\_B,A\_eq\_B);

// Wait 100 ns for global reset to finish

#10 A=4'b1010; B=4'b0111;

#10 A=4'b1010; B=4'b1111;

#10 A=4'b0000; B=4'b0111;

#10 A=4'b1010; B=4'b1101;

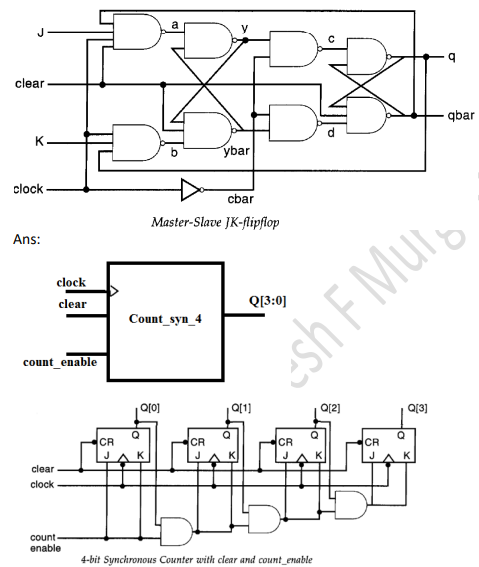
#10 A=4'b0000; B=4'b0000;

#10 A=4'b1110; B=4'b1111;

#20 $stop;

end

endmodule

**3) A synchronous counter can be designed by using master-slave JK flipflops. Design a 4-bit synchronous counter. Circuit diagrams for the synchronous counter and the JK flipflop are given below. The clear signal is active low. Data gets latched on the positive edge of clock, and the output of the flipflop appears on the negative edge of clock. Counting is disabled when count\_enable signal is low. Write the dataflow description for the synchronous counter. Write a stimulus file that exercises clear and count\_enable. Display the output count Q[3:0].**

**A).**

module JK\_FF(q,qbar,J,K,clock,clear);

output q,qbar;

input J,K,clock,clear;

wire a, b, c, d, y, ybar, cbar;

assign a= ~(J & clear & clock & qbar);

assign b= ~(K & clock & q);

assign cbar= ~clock;

assign y=~(a & ybar);

assign ybar= ~(b & clear & y);

assign c= ~(y & cbar);

assign d= ~(ybar & cbar);

assign q=~(c & qbar);

assign qbar=~(d & q & clear);

endmodule

**//TestBench**

module Count\_syn\_4(Q, clock, clear, count\_enable);

output [3:0]Q;

input clock, clear, count\_enable;

wire w1,w2,w3;

JK\_FF jk1(Q[0], , count\_enable, count\_enable,clock,clear);

and a1(w1,count\_enable,Q[0]);

JK\_FF jk2(Q[1], , w1,w1,clock,clear);

and a2(w2,w1,Q[1]);

JK\_FF jk3(Q[2], , w2,w2,clock,clear);

and a3(w3,w2,Q[2]);

JK\_FF jk4(Q[3], , w3,w3,clock,clear);

endmodule

Chapter-7

1. **Declare a register called oscillate. Initialize it to 0 and make it toggle every 30 time units. Do not use always statement (Hint: Use the forever loop).**
2. module task;

reg oscillate;

oscillate=0;

initial

begin

forever #30 oscillate=~oscillate;

end

#500 $finish;

endmodule

1. **Design a clock with time period = 40 and a duty cycle of 25% by using the always and initial statements. The value of clock at time = 0 should be initialized to 0.**
2. module ex1;

initial clk = 0; //Time period (T) = 40 time units, Duty cycle = 25%

always begin

#10 clk = 1; // High for 10-time units, (25/100) × 40 = 10 time units

#30 clk = 0; // Low for 30-time units, 40 - 10 = 30 time units

end

endmodule

1. **Given below is an initial block with blocking procedural assignments. At what simulation time is each statement executed? What are the intermediate and final values of a, b, c, d? initial begin a= 1'b0; b = #10 1'b1; c = #5 1'b0; d = #20 {a, b, c}; end**
   * Time 0: a = 0
   * Time 10: b = 1
   * Time 15: c = 0
   * Time 35: d = {a, b, c} = 3'b010
2. **Repeat exercise 3 if nonblocking procedural assignments were used.**

**initial**

**begin**

**a <= 1'b0;**

**b <= #10 1'b1;**

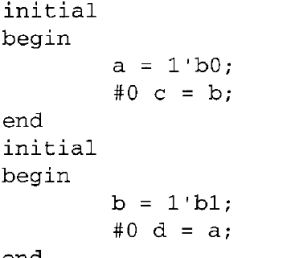
**c <= #5 1'b0;**

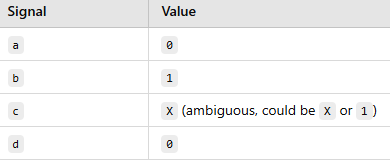
**d <= #20 {a, b, c};**

**end**

**A).**

1. At time 0:
   * The simulator schedules:
     + a to be updated immediately to 0
     + b to be updated at time 10 with 1
     + c to be updated at time 5 with 0
     + d to be updated at time 20 with {a, b, c} (values from time 0 are used!)
2. At time 5:
   * c is updated to 0.
3. At time 10:
   * b is updated to 1.
4. At time 20:
   * d is assigned {a, b, c} using values from time 0:
     + a = 0 (from time 0)
     + b = 1 (scheduled at time 10, but irrelevant since d uses time 0 values)
     + c = 0 (scheduled at time 5, but irrelevant since d uses time 0 values)
   * So, d = {0, 0, 0} = 3'b000 at time 20!
5. **What is the order of execution of statements in the following Verilog code? Is there any ambiguity in the order of execution? What are the final values of a,b,c,d?**

****

**A).**

module sample;

reg a,b,c,d;

initial

begin

a = 1'b0;

#5 c = b;

end

initial

begin

b = 1'b1;

#10 d = a;

end

initial $monitor($time, "\ta= %b c=%b b=%b d=%b \n",a,c,b,d);

endmodule

output: 0 a= 0 c=1 b=1 d=0

1. **What is the final value of d in the following example? (Hint: See intraassignment delays.) initial begin b = 1'b1; c = 1'b0; #10 b = 1'b0; initial 173 begin d = #25 (b | c); end**

**A).**

**7.** **Design a negative edge-triggered D-flipflop (D\_FF) with synchronous clear,**

**active high (D\_FF clears only at a negative edge of clock when clear is high).**

**Use behavioral statements only. (Hint: Output q of D\_FF must be declared as**

**reg). Design a clock with a period of 10 units and test the D\_FF.**

**A)**

module D\_FF (

input wire clk, // Clock signal

input wire d, // Data input

input wire clr, // Synchronous clear (active high)

output reg q // Flip-flop output

);

always @(negedge clk) begin

if (clr)

q <= 1'b0; // Clear output when clr is high

else

q <= d; // Otherwise, store D input

end

endmodule

module stimulus;

wire q;

reg clk,clr,d;

d\_ff x1(q,clk,clr,d);

initial

begin

clk=1’b0

forever #5 clk=~clk;

end

initial

begin

$monitor($time, “\t clr=%b d=%b q=%b\n”,clr,d,q);

clr=1’b1; d=1’b1;

#5 clr=1’b0;

#100 clr=1’b1;

# 50 $finish;

end

endmodule

**8. Design the D-flipflop in exercise 7 with asynchronous clear (D\_FF clears whenever clear goes high. It does not wait for next negative edge). Test the D\_FF.**

**A)** module D\_FF (

input wire clk, // Clock signal

input wire d, // Data input

input wire clr, // Asynchronous clear (active high)

output reg q // Flip-flop output

);

always @(negedge clk or posedge clr) begin

if (clr)

q <= 1'b0; // Clear output immediately when clr is high

else

q <= d; // Otherwise, store D input on negative edge of clock

end

endmodule

**9. Using the wait statement, design a level-sensitive latch that takes clock and**

**d as inputs and q as output. q=d whenever clock=1.**

**Answer)**

module D\_Latch (

input wire clk, // Level-sensitive clock

input wire d, // Data input

output reg q // Output

);

always @(\*) begin

wait (clk == 1) q = d; // Update q when clk is high

end

endmodule

module stimulus;

wire q;

reg clk,d;

d\_latch x1(q,d,clk);

always #10 clk=~clk;

initial

begin

clk=1’b0;

d=1’b1;

#20 d=1’b0;

#80 d=1’b1;

#100 $finish;

end

endmodule

**10. Design the 4-to-1 multiplexer in Example 7-19 by using if and else statements. The port interface must remain the same.**

**A).** module mux4to1 (

input wire [1:0] sel, // 2-bit select input

input wire a, b, c, d, // 4 data inputs

output reg y // Output

);

always @(\*) begin

if (sel == 2'b00)

y = a;

else if (sel == 2'b01)

y = b;

else if (sel == 2'b10)

y = c;

else

y = d;

end

endmodule

**11. Design the traffic signal controller discussed in this chapter by using if and else statements.**

**A).** **module traffic\_signal (**

**input wire clk, // Clock signal (assume 1-second period for simplicity)**

**input wire rst, // Reset signal (active high)**

**output reg [1:0] main\_light, // Main road light (00=Red, 01=Yellow, 10=Green)**

**output reg [1:0] side\_light // Side road light (00=Red, 01=Yellow, 10=Green)**

**);**

**// State encoding**

**parameter RED = 2'b00, YELLOW = 2'b01, GREEN = 2'b10;**

**// Timing counter**

**reg [3:0] counter;**

**always @(posedge clk or posedge rst) begin**

**if (rst) begin**

**// Reset to default state (Main Road Green, Side Road Red)**

**main\_light <= GREEN;**

**side\_light <= RED;**

**counter <= 0;**

**end**

**else begin**

**counter <= counter + 1;**

**if (counter == 5) begin // Stay green for 5 cycles**

**main\_light <= YELLOW;**

**side\_light <= RED;**

**end**

**else if (counter == 7) begin // Yellow for 2 cycles**

**main\_light <= RED;**

**side\_light <= GREEN; // Side road green**

**end**

**else if (counter == 12) begin // Side road green for 5 cycles**

**main\_light <= RED;**

**side\_light <= YELLOW;**

**end**

**else if (counter == 14) begin // Side road yellow for 2 cycles**

**main\_light <= GREEN;**

**side\_light <= RED;**

**counter <= 0; // Restart the cycle**

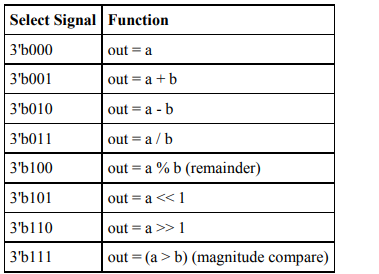
**end**

**end**

**end**

**endmodule**

**12: Using a case statement, design an 8-function ALU that takes 4-bit inputs a and b and a 3-bit input signal select, and gives a 5-bit output out. The ALU implements the following functions based on a 3-bit input signal select. Ignore any overflow or underflow bits**.

****

**A)**

module alu\_8function (

input wire [3:0] a, b, // 4-bit inputs

input wire [2:0] select, // 3-bit select signal

output reg [4:0] out // 5-bit output to handle division and shift

);

always @(\*) begin

case (select)

3'b000: out = a; // Pass-through

3'b001: out = a + b; // Addition

3'b010: out = a - b; // Subtraction

3'b011: out = a / b; // Division

3'b100: out = a % b; // Modulus (Remainder)

3'b101: out = a << 1; // Left Shift by 1

3'b110: out = a >> 1; // Right Shift by 1

3'b111: out = (a > b); // Magnitude Comparison (1-bit result)

default: out = 5'b00000; // Default case (avoid latches)

endcase

end

endmodule

**13. Using a while loop, design a clock generator. Initial value of clock is 0. Time period for the clock is 10**

**A).** module clock\_generator;

reg clk; // Clock signal

initial begin

clk = 0; // Initial value of clock

while (1) begin // Infinite loop to generate clock

#5 clk = ~clk; // Toggle clock every 5 time units

end

end

endmodule

**14. : Using the for loop, initialize locations 0 to 1023 of a 4-bit register array cache\_var to 0.**

A). module cache\_initializer;

reg [3:0] cache\_var [0:1023]; // 4-bit register array with 1024 locations

integer i; // Loop variable

initial begin

for (i = 0; i < 1024; i = i + 1) begin

cache\_var[i] = 4'b0000; // Initialize each location to 0

end

end

endmodule

**15. Using a forever statement, design a clock with time period = 10 and duty cycle = 40%. Initial value of clock is 0.**

**A).** module clock\_generator;

reg clk; // Clock signal

initial begin

clk = 0; // Initial value of clock

forever begin

#4 clk = 1; // HIGH for 4 time units (40% duty cycle)

#6 clk = 0; // LOW for 6 time units (60% duty cycle)

end

end

endmodule

**16. Using the repeat loop, delay the statement a = a + 1 by 20 positive edges of clock.**

**A).** module repeat\_example;

reg clk;

reg [7:0] a; // 8-bit register to hold value of 'a'

// Clock generation (Time period = 10 units)

initial begin

clk = 0;

forever #5 clk = ~clk; // Toggle every 5 units (Period = 10)

end

initial begin

a = 0; // Initialize a

// Wait for 20 positive edges of clk before executing a = a + 1

repeat (20) @(posedge clk);

a = a + 1;

end

endmodule

17. **Below is a block with nested sequential and parallel blocks. When does the block finish and what is the order of execution of events? At what simulation times does each statement finish execution? initial begin x = 1'b0; #5 y = 1'b1; fork #20 a = x; #15 b = y; join #40 x = 1'b1; fork #10 p = x; begin #10 a = y; #30 b = x; end #5 m = y; join end**

**A**). initial

begin

x = 1'b0; // Time = 0

#5 y = 1'b1; // Time = 5

fork

#20 a = x; // Time = 25

#15 b = y; // Time = 20

join

#40 x = 1'b1; // Time = 65

fork

#10 p = x; // Time = 75

begin

#10 a = y; // Time = 75

#30 b = x; // Time = 95

end

#5 m = y; // Time = 70

join

end

**18. Design an 8-bit counter by using forever loop, named block, and disabling of named block. The counter starts counting at count=5 and finishes at count=67. The count is incremented at positive edge of clock. The clock has a time period of 10. The counter counts through the loop only once and then is disabled.(Hint: Use the disable statement).**

**A).** module counter\_8bit;

reg clk; // Clock signal

reg [7:0] count; // 8-bit counter

initial count = 5; // Start counting from 5

// Clock generator (Period = 10, Toggle every 5)

initial begin

clk = 0;

forever #5 clk = ~clk; // Clock toggles every 5 time units

end

initial begin: COUNT\_BLOCK // Named block "COUNT\_BLOCK"

forever begin // Infinite loop for counting

@(posedge clk); // Wait for positive edge of clock

count = count + 1; // Increment counter

if (count == 67) begin

disable COUNT\_BLOCK; // Stop the counter

end

end

end

endmodule

**Chapter 8. tasks and functions  
  
8.5 Exercises**  
**1. Define a function to calculate the factorial of a 4-bit number. The output is a 32- bit value. Invoke the function by using stimulus and check results.**

A).

module factorial\_test;

// Function to calculate factorial of a 4-bit number

function automatic [31:0] factorial(input [3:0] n);

if (n == 0)

factorial = 1; // Base case: 0! = 1

else

factorial = n \* factorial(n - 1); // Recursive call

endfunction

// Testbench

reg [3:0] test\_input;

wire [31:0] test\_output;

integer i;

initial begin

$display("Factorial Calculation Test");

for (i = 0; i <= 10; i = i + 1) begin

test\_input = i;

#1;

$display("Factorial of %d = %d", test\_input, factorial(test\_input));

end

$finish;

end

endmodule

**2.Define a function to multiply two 4-bit numbers a and b. The output is an 8-bit value. Invoke the function by using stimulus and check results.**

**A).**

module multiply\_test;

// Function to multiply two 4-bit numbers

function automatic [7:0] multiply(input [3:0] a, input [3:0] b);

multiply = a \* b; // Simple multiplication

endfunction

// Testbench

reg [3:0] a, b;

wire [7:0] result;

integer i, j;

initial begin

$display("4-bit Multiplication Test");

for (i = 0; i < 16; i = i + 1) begin

for (j = 0; j < 16; j = j + 1) begin

a = i;

b = j;

#1;

$display("%d \* %d = %d", a, b, multiply(a, b));

end

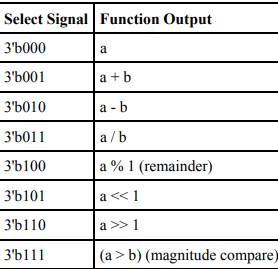
end

$finish;

end

endmodule

**3. Define a function to design an 8-function ALU that takes two 4-bit numbers a and b and computes a 5-bit result out based on a 3-bit select signal. Ignore overflow or underflow bits.**

****

**A).**

module alu\_test;

// Function to implement the 8-function ALU

function automatic [4:0] alu(

input [3:0] a,

input [3:0] b,

input [2:0] sel

);

case (sel)

3'b000: alu = a; // Pass a

3'b001: alu = a + b; // Addition

3'b010: alu = a - b; // Subtraction

3'b011: alu = (b != 0) ? (a / b) : 5'b0; // Division (avoid divide by zero)

3'b100: alu = a % 1; // Remainder when divided by 1 (always 0)

3'b101: alu = a << 1; // Left shift by 1

3'b110: alu = a >> 1; // Right shift by 1

3'b111: alu = (a > b) ? 5'b00001 : 5'b00000; // Magnitude compare

default: alu = 5'b00000; // Default case

endcase

endfunction

// Testbench

reg [3:0] a, b;

reg [2:0] sel;

wire [4:0] result;

integer i, j, k;

initial begin

$display("8-Function ALU Test");

for (i = 0; i < 16; i = i + 1) begin

for (j = 0; j < 16; j = j + 1) begin

for (k = 0; k < 8; k = k + 1) begin

a = i;

b = j;

sel = k;

#1;

$display("a=%d, b=%d, sel=%b -> out=%d", a, b, sel, alu(a, b, sel));

end

end

end

$finish;

end

endmodule

**4. Define a task to compute the factorial of a 4-bit number. The output is a 32-bit value. The result is assigned to the output after a delay of 10 time units.**

**A).**

module factorial\_task\_test;

reg [3:0] num;

reg [31:0] result;

// Task to compute factorial with a 10-time unit delay

task factorial(input [3:0] n, output reg [31:0] fact\_out);

integer i;

begin

fact\_out = 1; // Initialize result

for (i = 1; i <= n; i = i + 1) begin

fact\_out = fact\_out \* i;

end

#10; // 10 time unit delay

end

endtask

// Testbench to verify factorial computation

integer i;

initial begin

$display("Factorial Task Test (with 10 time unit delay)");

for (i = 0; i <= 10; i = i + 1) begin

num = i;

factorial(num, result);

#1; // Small delay to ensure task execution

$display("Factorial of %d = %d (computed after delay)", num, result);

end

$finish;

end

endmodule

**5: Define a task to compute even parity of a 16-bit number. The result is a 1-bit value that is assigned to the output after three positive edges of clock. (Hint: Use a repeat loop in the task.)**

A). module even\_parity\_task\_test;

reg [15:0] data;

reg clk;

reg parity\_out;

// Task to compute even parity after 3 positive clock edges

task even\_parity(input [15:0] num, output reg parity);

integer i;

begin

parity = 0; // Initialize parity to 0

for (i = 0; i < 16; i = i + 1) begin

parity = parity ^ num[i]; // XOR all bits to get odd parity

end

parity = ~parity; // Invert to get even parity

// Wait for three positive edges of the clock

repeat (3) @(posedge clk);

end

endtask

// Clock generation (50% duty cycle)

always #5 clk = ~clk;

// Testbench to verify even parity computation

initial begin

clk = 0;

$display("Even Parity Task Test (Output after 3 positive clock edges)");

data = 16'b1010\_1100\_1111\_0001; // Example test case

even\_parity(data, parity\_out);

$display("Time=%0t | Data=%b | Even Parity=%b", $time, data, parity\_out);

#50; // Wait before finishing simulation

$finish;

end endmodule  
  
**6: Using named events, tasks, and functions, design the traffic signal controller in Traffic Signal Controller on page 160.**

A).

module traffic\_signal\_controller;

// Signal definitions

reg clk;

reg [1:0] ns\_light, ew\_light; // 00: Red, 01: Yellow, 10: Green

// Named events for state transitions

event ns\_go, ns\_warn, ew\_go, ew\_warn;

// Function to display current signal status

function void display\_status();

$display("Time=%0t | NS Light: %s | EW Light: %s",

$time, light\_to\_string(ns\_light), light\_to\_string(ew\_light));

endfunction

// Function to convert light states to string

function [15:0] light\_to\_string(input [1:0] light);

case (light)

2'b00: light\_to\_string = "RED";

2'b01: light\_to\_string = "YELLOW";

2'b10: light\_to\_string = "GREEN";

default: light\_to\_string = "UNK";

endcase

endfunction

// Task to control NS Green phase

task ns\_green\_phase;

begin

ns\_light = 2'b10; ew\_light = 2'b00; // NS Green, EW Red

display\_status();

#10 -> ns\_warn; // After 10 time units, transition to Yellow

end

endtask

// Task to control NS Yellow phase

task ns\_yellow\_phase;

begin

ns\_light = 2'b01; ew\_light = 2'b00; // NS Yellow, EW Red

display\_status();

#3 -> ew\_go; // After 3 time units, transition to EW Green

end

endtask

// Task to control EW Green phase

task ew\_green\_phase;

begin

ns\_light = 2'b00; ew\_light = 2'b10; // NS Red, EW Green

display\_status();

#10 -> ew\_warn; // After 10 time units, transition to Yellow

end

endtask

// Task to control EW Yellow phase

task ew\_yellow\_phase;

begin

ns\_light = 2'b00; ew\_light = 2'b01; // NS Red, EW Yellow

display\_status();

#3 -> ns\_go; // After 3 time units, transition back to NS Green

end

endtask

// Clock generation

always #5 clk = ~clk;

// Initial block to trigger the state transitions

initial begin

clk = 0;

ns\_light = 2'b00; ew\_light = 2'b00; // Default state: Both Red

// Trigger first event

-> ns\_go;

// Main control loop

forever begin

@(ns\_go) ns\_green\_phase();

@(ns\_warn) ns\_yellow\_phase();

@(ew\_go) ew\_green\_phase();

@(ew\_warn) ew\_yellow\_phase();

end

end

endmodule

**Chapter -9: useful modelling techniques**

**9.7 Exercises**

**1: Using assign and deassign statements, design a positive edge-triggered Dflipflop with asynchronous clear (q=0) and preset (q=1).**

**A).**

module dff\_async (

input clk, // Clock signal

input d, // Data input

input clear, // Asynchronous clear (active high)

input preset, // Asynchronous preset (active high)

output reg q // Flip-flop output

);

always @(posedge clk or posedge clear or posedge preset) begin

if (clear) begin

assign q = 0; // Asynchronous Clear (q = 0)

end

else if (preset) begin

assign q = 1; // Asynchronous Preset (q = 1)

end

else begin

deassign q; // Remove any forced assignment

q <= d; // Latch the D input on positive edge of clk

end

end

endmodule

**2. Using primitive gates, design a 1-bit full adder FA. Instantiate the full adder inside a stimulus module. Force the sum output to a & b & c\_in for the time between 15 and 35 units.**

**A).**

module full\_adder (

input a, b, c\_in, // Inputs

output sum, c\_out // Outputs

);

wire s1, c1, c2;

xor (s1, a, b); // First XOR for sum

xor (sum, s1, c\_in); // Second XOR for sum

and (c1, a, b); // First AND for carry-out

and (c2, s1, c\_in); // Second AND for carry-out

or (c\_out, c1, c2); // Final OR for carry-out

endmodule

**3. A 1-bit full adder FA is defined with gates and with delay parameters as shown**

**below.**

**// Define a 1-bit full adder**

**module fulladd(sum, c\_out, a, b, c\_in);**

**parameter d\_sum = 0, d\_cout = 0;**

**// I/O port declarations**

**output sum, c\_out;**

**input a, b, c\_in;**

**// Internal nets**

**wire s1, c1, c2;**

**// Instantiate logic gate primitives**

**xor (s1, a, b);**

**and (c1, a, b);**

**xor #(d\_sum) (sum, s1, c\_in); //delay on output sum is d\_sum**

**and (c2, s1, c\_in);**

**or #(d\_cout) (c\_out, c2, c1); //delay on output c\_out is d\_cout**

**endmodule**

**Define a 4-bit full adder fulladd4 as shown in Example 5-8 on page 77, but pass**

**the following parameter values to the instances, using the two methods**

**discussed in the book:**

**Instance Delay Values**

**fa0**

**fa1**

**fa2**

**fa3**

**d\_sum=1, d\_cout=1**

**d\_sum=2, d\_cout=2**

**d\_sum=3, d\_cout=3**

**d\_sum=4, d\_cout=4**

**213**

**a. Build the fulladd4 module with defparam statements to change instance**

**parameter values. Simulate the 4-bit full adder using the stimulus shown**

**in Example 5-9 on page 77. Explain the effect of the full adder delays on**

**the times when outputs of the adder appear. (Use delays of 20 instead of**

**5 used in this stimulus.)**

**A).**

module fulladd4 (

output [3:0] sum,

output c\_out,

input [3:0] a, b,

input c\_in

);

wire c1, c2, c3; // Internal carry signals

// Instantiate four 1-bit full adders with default parameters

fulladd fa0(sum[0], c1, a[0], b[0], c\_in);

fulladd fa1(sum[1], c2, a[1], b[1], c1);

fulladd fa2(sum[2], c3, a[2], b[2], c2);

fulladd fa3(sum[3], c\_out, a[3], b[3], c3);

// Use defparam to modify instance parameters

defparam fa0.d\_sum = 1, fa0.d\_cout = 1;

defparam fa1.d\_sum = 2, fa1.d\_cout = 2;

defparam fa2.d\_sum = 3, fa2.d\_cout = 3;

defparam fa3.d\_sum = 4, fa3.d\_cout = 4;

endmodule

**b. Build the fulladd4 with delay values passed to instances fa0, fa1, fa2,**

**and fa3 during instantiation. Resimulate the 4-bit adder, using the**

**stimulus above. Check if the results are identical.**

**A).**

module fulladd4\_param (

output [3:0] sum,

output c\_out,

input [3:0] a, b,

input c\_in

);

wire c1, c2, c3; // Internal carry signals

// Instantiate four 1-bit full adders with parameterized delays

fulladd #(.d\_sum(1), .d\_cout(1)) fa0(sum[0], c1, a[0], b[0], c\_in);

fulladd #(.d\_sum(2), .d\_cout(2)) fa1(sum[1], c2, a[1], b[1], c1);

fulladd #(.d\_sum(3), .d\_cout(3)) fa2(sum[2], c3, a[2], b[2], c2);

fulladd #(.d\_sum(4), .d\_cout(4)) fa3(sum[3], c\_out, a[3], b[3], c3);

endmodule

**4: Create a design that uses the full adder example above. Use a conditional compilation (`ifdef). Compile the fulladd4 with defparam statements if the text macro DPARAM is defined by the `define statement; otherwise, compile the fulladd4 with module instance parameter values.**

**A).**

`timescale 1ns/1ps

`define DPARAM // Uncomment this line to use defparam method, comment to use instance parameter values

// 1-bit Full Adder Module (Same as before)

module fulladd(sum, c\_out, a, b, c\_in);

parameter d\_sum = 0, d\_cout = 0;

output sum, c\_out;

input a, b, c\_in;

wire s1, c1, c2;

xor (s1, a, b);

and (c1, a, b);

xor #(d\_sum) (sum, s1, c\_in);

and (c2, s1, c\_in);

or #(d\_cout) (c\_out, c2, c1);

endmodule

// Conditional Compilation for fulladd4

`ifdef DPARAM // Use defparam method

module fulladd4 (sum, c\_out, a, b, c\_in);

output [3:0] sum;

output c\_out;

input [3:0] a, b;

input c\_in;

wire c1, c2, c3;

// Instantiate four 1-bit full adders

fulladd fa0(sum[0], c1, a[0], b[0], c\_in);

fulladd fa1(sum[1], c2, a[1], b[1], c1);

fulladd fa2(sum[2], c3, a[2], b[2], c2);

fulladd fa3(sum[3], c\_out, a[3], b[3], c3);

// Define instance-specific parameters using defparam

defparam fa0.d\_sum = 1, fa0.d\_cout = 1;

defparam fa1.d\_sum = 2, fa1.d\_cout = 2;

defparam fa2.d\_sum = 3, fa2.d\_cout = 3;

defparam fa3.d\_sum = 4, fa3.d\_cout = 4;

endmodule

`else // Use parameterized instance method

module fulladd4 (sum, c\_out, a, b, c\_in);

output [3:0] sum;

output c\_out;

input [3:0] a, b;

input c\_in;

wire c1, c2, c3;

// Instantiate fulladd instances with inline parameter values

fulladd #(.d\_sum(1), .d\_cout(1)) fa0(sum[0], c1, a[0], b[0], c\_in);

fulladd #(.d\_sum(2), .d\_cout(2)) fa1(sum[1], c2, a[1], b[1], c1);

fulladd #(.d\_sum(3), .d\_cout(3)) fa2(sum[2], c3, a[2], b[2], c2);

fulladd #(.d\_sum(4), .d\_cout(4)) fa3(sum[3], c\_out, a[3], b[3], c3);

endmodule

`endif

**5: Identify the files to which the following display statements will write:**

**//File output with multi-channel descriptor**

**module test;**

**integer handle1,handle2,handle3; //file handles**

**//open files**

**initial**

**begin**

**handle1 = $fopen("f1.out");**

**handle2 = $fopen("f2.out");**

**handle3 = $fopen("f3.out");**

**end**

**//Display statements to files**

**initial**

**begin**

**//File output with multi-channel descriptor**

**#5;**

**$fdisplay(4, "Display Statement # 1");**

**$fdisplay(15, "Display Statement # 2");**

**$fdisplay(6, "Display Statement # 3");**

**$fdisplay(10, "Display Statement # 4");**

**$fdisplay(0, "Display Statement # 5");**

**end**

**endmodule**

**A).** $fdisplay(4, "Display Statement # 1"); // Writes to file descriptor 4 → f2.out

$fdisplay(15, "Display Statement # 2"); // Invalid descriptor (ignored or error)

$fdisplay(6, "Display Statement # 3"); // Invalid descriptor (ignored or error)

$fdisplay(10, "Display Statement # 4"); // Invalid descriptor (ignored or error)

$fdisplay(0, "Display Statement # 5"); // Writes to console (stdout)

**6: What will be the output of the $display statement shown below?**

**module top;**

**A a1();**

**endmodule**

**module A;**

**214**

**B b1();**

**endmodule**

**module B;**

**initial**

**$display("I am inside instance %m");**

**Endmodule**

**A).** I am inside instance top.a1.b1

**7: Consider the 4-bit full adder in Example 6-4 on page 108. Write a stimulus file to do random testing of the full adder. Use a random number generator to generate a 32-bit random number. Pick bits 3:0 and apply them to input a; pick bits 7:4 and apply them to input b. Use bit 8 and apply it to c\_in. Apply 20 random test vectors and observe the output.**

**A).**

`timescale 1ns / 1ps

module fulladd4\_tb;

// Inputs

reg [3:0] a, b;

reg c\_in;

// Outputs

wire [3:0] sum;

wire c\_out;

// Instantiate the 4-bit Full Adder

fulladd4 uut (

.a(a),

.b(b),

.c\_in(c\_in),

.sum(sum),

.c\_out(c\_out)

);

// Random test stimulus

integer i;

reg [31:0] rand\_num;

initial begin

$display("Time | A | B | C\_in | Sum | C\_out");

$display("--------------------------------------------------");

for (i = 0; i < 20; i = i + 1) begin

rand\_num = $random; // Generate a 32-bit random number

a = rand\_num[3:0]; // Extract bits [3:0] for A

b = rand\_num[7:4]; // Extract bits [7:4] for B

c\_in = rand\_num[8]; // Extract bit 8 for Carry-in

#10; // Wait 10 time units for stable output

$display("%4t | %b | %b | %b | %b | %b",

$time, a, b, c\_in, sum, c\_out);

end

$stop; // Stop simulation

end

endmodule

**8: Use the 8-byte memory initialization example in Example 9-14 on page 205. Modify the file to read data in hexadecimal. Write a new data file with the following addresses and data values. Unspecified locations are not initialized. Location Address Data 1 2 4 5 6 33 66 z0 0z 01**

A).

`timescale 1ns / 1ps

module memory\_init;

reg [7:0] memory [0:255]; // 256-byte memory

integer i, file;

initial begin

// Initialize memory to zero

for (i = 0; i < 256; i = i + 1)

memory[i] = 8'h00;

// Read memory from the file (hex format)

$readmemh("mem\_init.hex", memory);

// Open a new file to write the modified memory content

file = $fopen("mem\_out.hex", "w");

// Writing specific addresses with values

memory[1] = 8'h01;

memory[2] = 8'h02;

memory[4] = 8'h04;

memory[5] = 8'h05;

memory[6] = 8'h06;

memory[33] = 8'h33;

memory[66] = 8'h66;

memory['hZ0] = 8'hZZ; // Undefined (Verilog allows 'z' for high-impedance)

memory['h0Z] = 8'hZZ;

// Write back the modified memory to the output file

for (i = 0; i < 256; i = i + 1) begin

if (memory[i] !== 8'h00) // Only write initialized values

$fdisplay(file, "%h : %h", i, memory[i]);

end

$fclose(file);

$display("Memory initialization completed. Output written to mem\_out.hex");

end

endmodule

**9: Write an initial block that controls the VCD file. The initial block must do the**

**following:**

**• Set myfile.dmp as the output VCD file.**

**• Dump all variables two levels deep in module instance top.a1.b1.c1.**

**• Stop dumping to VCD at time 200.**

**• Start dumping to VCD at time 400.**

**• Stop dumping to VCD at time 500.**

**• Create a checkpoint. Dump the current value of all VCD variables to the**

**dumpfile.**

**A).**

`timescale 1ns / 1ps

module vcd\_control;

initial begin

// Set the VCD output file

$dumpfile("myfile.dmp");

// Dump all variables two levels deep in module instance top. a1.b1.c1

$dumpvars(2, top.a1.b1.c1);

// Stop dumping at time 200

#200 $dumpoff;

$display ("Stopped VCD dumping at time 200");

// Start dumping again at time 400

#200 $dumpon;

$display ("Resumed VCD dumping at time 400");

// Stop dumping at time 500

#100 $dumpoff;

$display ("Stopped VCD dumping at time 500");

// Create a checkpoint and dump all current values

$dumpflush;

$display ("Dump checkpoint created at time %t", $time);

end

endmodule