CBCS SCHEME

USN 18EC56

Fifth Semester B.E. Degree Examination, Feb./Mar. 2022
Verilog HDL

Time: 3 hrs. Max. Marks: 100

Note: *Answer any FIVE full questions, choosing ONE full question from each module.*

03-2022 08:47:08am CR - CR - CR

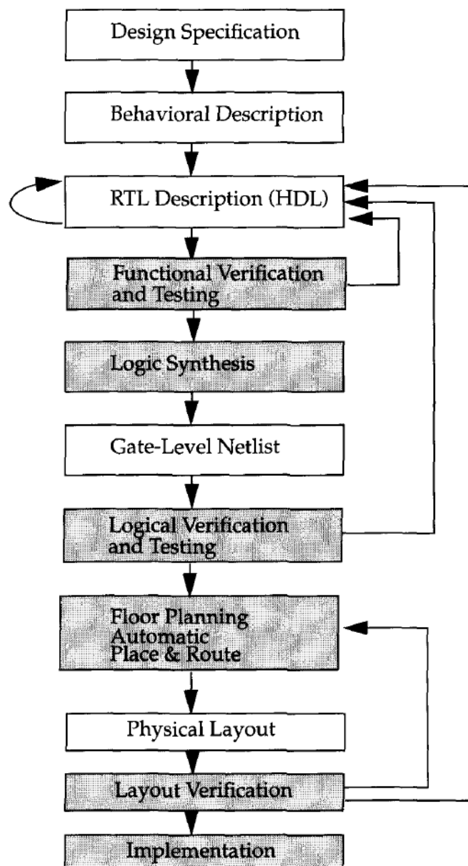
Module-1

1 a. Explain a typical design flow for designing VLSI IC circuits using the block diagram. (10 Marks)

b. Explain the importance of HDLs. (05 Marks)

c. Explain the trends in HDLs. (05 Marks)

1a. A typical design flow for designing VLSI IC circuits is shown in Figure below



In any design, specifications are written first. Specifications describe abstractly the functionality, interface, and overall architecture of the digital circuit to be designed. A behavioral description is then created to nalyse the design in terms of functionality, performance, compliance to standards, and other high-level issues. They are written using HDLs. The behavioral description is manually converted to an RTL description in an HDL. The designer has to describe the data flow that will implement the desired digital circuit. Logic synthesis tools convert the RTL description to a gate-level netlist. A gate-level netlist is a description of the circuit in terms of gates and connections between them. The gate-level netlist is input to an Automatic Place and Route tool, which creates a layout. The layout is verified and then fabricated on chip. Thus, most digital design activity is concentrated on manually optimizing the RTL description of the circuit. Behavioral synthesis tools have begun to emerge recently. These tools can create RTL descriptions from a behavioral or algorithmic description of the circuit.

1b.

HDLs have many advantages compared to traditional schematic-based design.

- Designs can be described at a very abstract level by use of HDLs. Designers can write their RTL description without choosing a specific fabrication technology. Logic synthesis tools can automatically convert the design to any fabrication technology. If a new technology emerges, designers do not need to redesign their circuit. They simply input the RTL description to the logic synthesis tool and create a new gate-level netlist, using the new fabrication technology. The logic synthesis tool will optimize the circuit in area and timing for the new technology.

- By describing designs in HDLs, functional verification of the design can be done early in the design cycle. Since designers work at the RTL level, they can optimize and modify the RTL description until it meets the desired functionality. Most design bugs are eliminated at this point. This cuts down design cycle time significantly because the probability of hitting a functional bug at a later time in the gate-level netlist or physical layout is minimized.

- Designing with HDLs is analogous to computer programming. A textual description with comments is an easier way to develop and debug circuits. This also provides a concise representation of the design, compared to gate-level schematics. Gate-level schematics are almost incomprehensible for very complex designs.

HDL-based design is here to stay.[3] With rapidly increasing complexities of digital circuits and increasingly sophisticated EDA tools, HDLs are now the dominant method for large digital designs. No digital circuit designer can afford to ignore HDL-based design.

1c

1. Start design of digital circuits using HDL at an RTL level, because logic synthesis tools can create gate-level netlists from RTL level des

2. Behavioral synthesis helps designers to design directly in terms of algorithms and the behavior of the circuit, and then use CAD tools to do the translation and optimization in each

phase of the design.
3. *Formal verification* techniques are also appearing on the horizon. Formal verification applies formal mathematical techniques to verify the correctness of Verilog HDL descriptions and to establish equivalency between RTL and gate-level netlists.
4. Designers can mix gate-level description directly into the RTL description to achieve optimum results.
5. System-level design can be a mixed bottom-up methodology where the designers use either existing Verilog HDL modules, basic building blocks, or vendor-supplied core blocks to quickly bring up their system simulation. This is done to reduce development costs and compress design schedules.

OR

2 a. Explain the different levels of abstraction used for programming in verilog. **(08 Marks)**
b. Write the verilog code for 4-bit ripple carry counter. Also write the stimulus. **(12 Marks)**

2a

Levels of abstraction: (9)

1. Behavioural level or algorithmic level.
- This is the highest level of abstraction provided by Verilog.
- In this the module can be implemented in terms of algorithm w/o concern of H/w Implementation details
- Similar to "C" program.

2. Dataflow level
- The module is designed by specifying the data flow.
- the designer is aware of how data is flowing b/w Register & H/w & How it is processed.

3. Gate level.
- Modules are implemented in terms of logic Gates & their interconnections b/w the Gates.
- This is same as Gate level design.

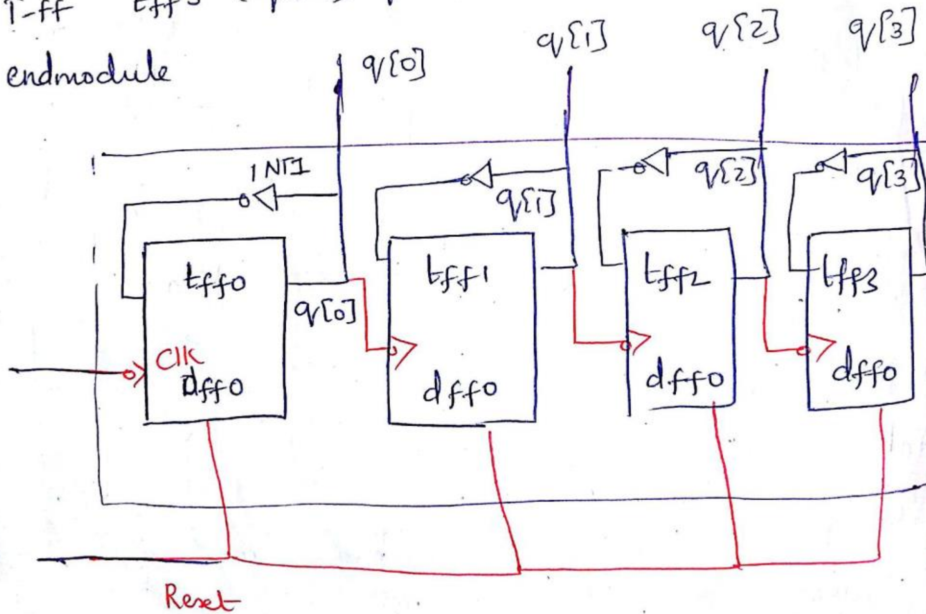
4. Switch level.
- This is the lowest level of abstraction.
- They are implemented Using Switch, storage nodes & Interconnection b/w them.

2b

(12)

Code for 4-bit Ripple Carry Using T-ff designed intern Using d-ff & NOT.

```
Module Ripple_carry ( q, Clk, reset);
output [ 3: 0]   q;
 input  Clk, reset;

T-ff  dff0( q[0], clk, reset);
T-ff  dff1 ( q[1], q[0], reset);
T-ff  tff2 ( q[2], q[1], reset);
T-ff  tff3 (q[3], q[2], reset);
endmodule
```



Reset

Stimulus Block

// Where instantiation is done by stimulus block. (20)

Ex)

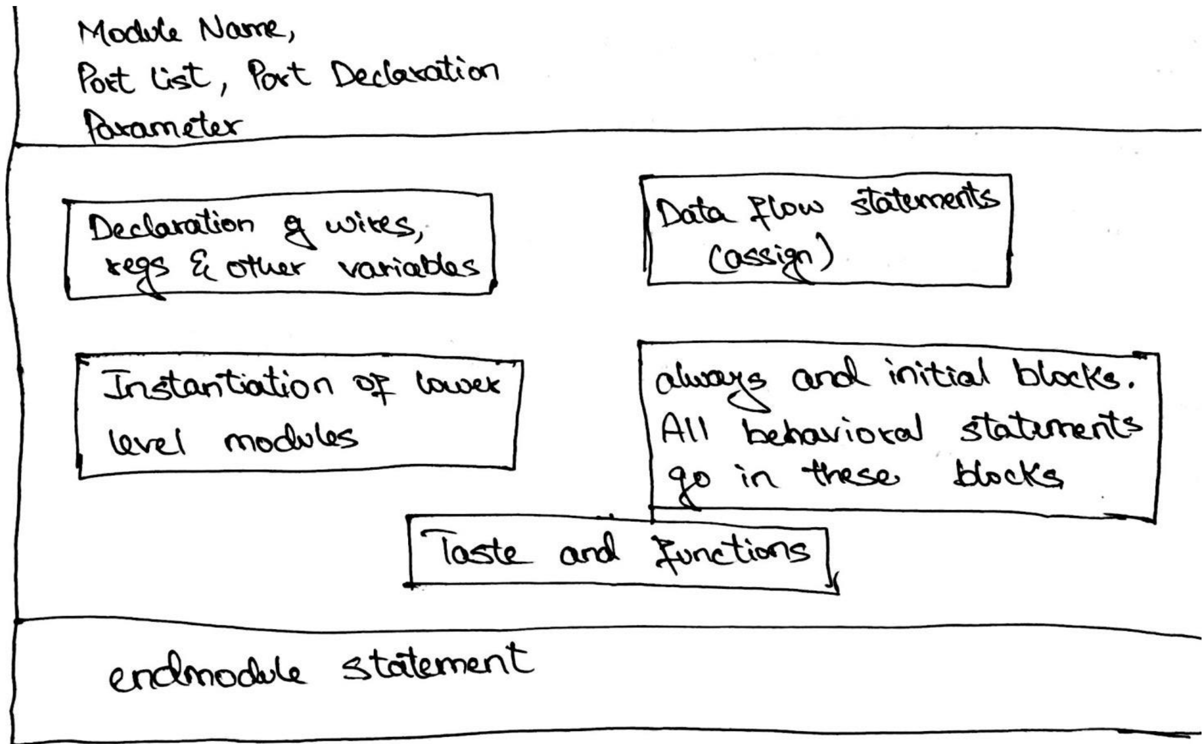
```
module stimulus ;          // module name
                           // outputs of stimulus is
reg   clk;                 //   reg.
reg   reset;
wire [3:0] q;              // output of design block.

                           // instantiation of design block
Ripple-carry r(q, clk, reset);
                           // Clk Control Cycle time = 10.
initial
   clk = 1'b0;             // sets clk = 0;
always
   #5 clk = ~clk;          // Toggles every 5 times units.
                           // Controls the Reset signal

initial
begin
   reset = 1'b1;
   #15 reset = 1'b0;
195  #180 reset = 1'b1;
205  # 10 reset = 1'b0;
#20 $finish ;
end                        // Monitor o/p.
initial
   $monitor ($time, " Output q= %d", q);
endmodule .
```

Module-2

3 a. Explain the components of verilog module with block diagram. (06 Marks)

b. Explain the following data types with an example in verilog.
i) Registers ii) Arrays iii) Parameters iv) Nets v) Integers. (10 Marks)

c. Explain the port connection rules in verilog. (04 Marks)



A module definition always begin with the keyword module. The module name, port list, port declaration, & optional parameters must come first in a module definition.

The five components within a module are: 'variable declaration', 'dataflow statement', 'instantiation of lower modules', 'behavioral blocks' & 'tasks' or 'functions'.

These components can be in any order & at any place in the module definition. The 'endmodule' statements must always come last in a module definition.

2. Explain the different types of ports supported by Verilog HDL with examples

3bExplain the following data types with an example in verilog:

(i) Nets: Nets represent connections between hardware elements. Just as in real circuits, nets have values continuously driven on them by the outputs of devices that they are connected to. Nets are declared primarily with the keyword wire.
Example:
wire a; //Declare net a for the above circuit
wire b,c; //Declare two wires b,c for the above circuit
wire d = l'b0; //Net d is fixed to logic value 0 at declaration.

(ii) Register: *Registers* represent data storage elements. Registers retain value until another value is placed onto them. Unlike a net, a register does not need a driver. Verilog registers do not need a clock as hardware
registers do. Values of registers can be changed anytime in a simulation by assigning a new value to the register. Register data types are commonly declared by the keyword reg. The default value for a reg data type is 'x'.
Example:
reg reset; //declare a variable reset that can hold its value
initial //this construct will be discussed later
begin
reset = l'b1; //initialize reset to 1 to reset the digital circuit.
#100 reset = l'b0; //after 100 time units reset is deasserted.
End

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

quantities.
Example:
integer counter; //general purpose variable used as a counter.
Initial
counter = -1;

Arrays: Collection of more than one elements of same datatype.
Arrays are allowed in verilog for reg, integer, time.
and vectored register data types.
Arrays are not for real types.

```
ex: integer count[0:7]; //array of 8 elements(integer)
    reg bool[31:0]; //array of 32 elements each element
                    // is a binary digit.
```

parameters:

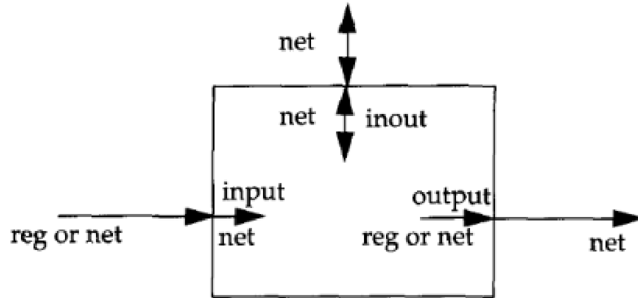
Verilog allows constants to be defined in a module by the keyword parameter. They cannot be used as a variables.

– 01mark

```
ex: parameter port_id = 5;
```

3c

A port consisting of two units, one unit that is *internal* to the module another that is *external* to the module. The internal and external units are connected.



Internally, input ports must always be of the type net. Externally, the inputs can be connected to a variable which is a reg or a net.

02 marks

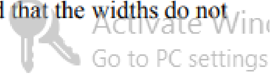
Internally, outputs ports can be of the type reg or net. Externally, outputs must always be connected to a net. They cannot be connected to a reg.

02 marks

Internally, inout ports must always be of the type net. Externally, inout ports must always be connected to a net.

02 marks

It is legal to connect internal and external items of different sizes when making inter-module port connections. However, a warning is typically issued that the widths do not match.



Verilog allows ports to remain unconnected.

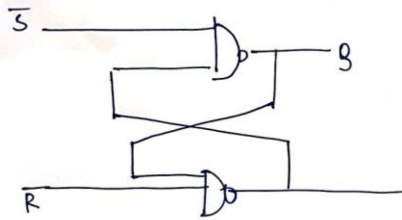
OR

4 a. Write the verliog description of SR latch. Also write stimulus code. **(10 Marks)**

b. Explain $display, $monitor, $finish and $stop system tasks with examples. **(10 Marks)**

4a

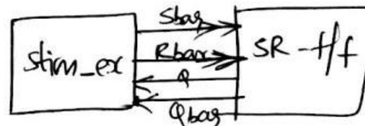
SR - latch



S	R	S̄	R̄	Q	Q̄
0	0	1	1	No change.	
0	1	1	0	0	1
1	0	0	1	1	0
1	1	0	0	Invalid state	

```
module sr_en (Q, Qbar, Sbar, Rbar);
    input Sbar, Rbar;
    output Q, Qbar;
    nand n1 (Q, Sbar, Qbar);
    nand n2 (Qbar, Rbar, Q);
    end module.
```

```
// module name and port list
module SR(Q, Qbar, Sbar, Rbar);
// port declaration
output  Q, Qbar;
input   Sbar, Rbar;

// instantiate verilog primitive nand gates
nand n1 (Q, Sbar, Qbar);
nand n2 (Qbar, Rbar, Q);

// end module statement
endmodule
```




```
// stimulus module
module stim_ex;
// declaration of ports & wires
  reg   Sbar, Rbar;
  wire  Qbar, Q;
// instantiate
  SR   SR1 ( Q, Qbar, Sbar, Rbar);
// behavioural block, initial
initial
begin
#10     Sbar = 1'b1 ; Rbar = 1'b1;
#10     Sbar = 1'b1 ; Rbar = 1'b0;
```

```
#10   Sbar = 1'b0 ;   Rbar = 1'b1;
#10   Sbar = 1'b1 ;   Rbar = 1'b1;
end $finish
initial
$monitor ($time ," input  Sbar = %b , Rbar = %b  and outp
   Q = %b , Qbar = %b , Sbar, Rbar, Q , Qbar);

// endmodule statement
endmodule
```

4b

System task and Compiler Directives

System tasks

→ Verilog provides standard System tasks for Certain Routine.

→ all System tasks appear from $

→ Operations such as displaying on the Screen, monitoring values of nets, stopping & finishing.

Display information

$ is the main system task for displaying values or strings or expressions

Usage: $display(p1, p2, p3 - - pn)

p1, p2, p3 : pn can be quoted as strings, Variables or expressions.

Ex: → $display("Hello world");

→ $display($time);

//

→ reg [4:0] port_id;

$display(" ID of Port is %b", port_id);

-- ID of Port is 00101

Monitoring information

→ Verilog provides a mechanism to monitor when a signals value changes.

(38)

```
$monitor (p1, p2, -- pn)
```

NOTE: monitor needs to be invoked only once,

Two tasks to switch monitor on & off

```
$monitoron ;
$monitoroff ;
```

Ex:

```
Initial
  begin
    $monitor ($time, "Value of signals
            Clock =%b reset=%b", Clock, reset);
```

o/p!

```
10   value of signals Clock=0 reset=1.
20   value of    "         "  = 1 reset=1
30    "     "    "         "  = 0  "  = 1
```

Stopping & finishing in a Simulation

The task $stop is provided to stop the simulation.

(39)

$stop

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

$finish

- terminates the simulation.

//Example of stop & finish
Stops at time 100 in the simulation & examines the result // finishes the simulation at 1000

```
initial
 begin
  clock = 0;
  reset = 1;
  #100 $stop ; //This will suspend the simulation at time 100
  #1000 $finish     // terminates the simulation at time 1000.
 end
```

Compiler directive

Compiler directive uses ' keyword construct (40)

'define :

→ This 'define directive is used to define macro text

→ macro substitution happens whenever it encounters the '<macro-name>.

→ This is similar to # define in C.

Ex:

```
        // where ever word-size is recognized 32 is replaced.
'define      word-size    32 ;
        // whereever 'S' is encountered $stop is substituted
'define      S      $stop ;
Illarly whenever ex1 is recognized 15 is substituted
'define      ex1      15;
```

'include

we can include a Verilog file in another main file using 'include.

5a

Module-3

5 a. What are rise, fall and turn off delays? How they are specified in verilog? **(06 Marks)**

b. What would be the output of the following for A = 4′b0111 an dB = 4′b1001.

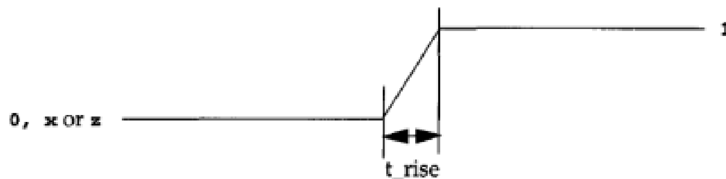
i) &B ii) A<<<2 iii) {A, B} iv) {2{B}} v) A^B vi) A||B vii) A*B viii) A < = B. **(08 Marks)**

c. Mention the symbol, truth table and an example for BUFIF1 and BUFIF0 primitive gates. **(06 Marks)**

5a

Rise delay:

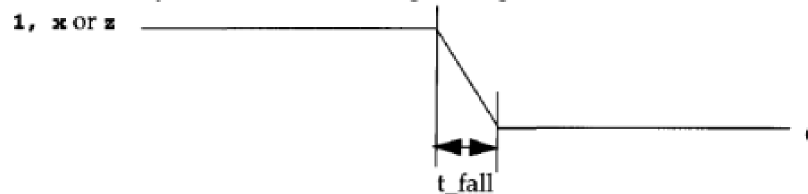
The rise delay is associated with a gate output transition to a 1 from another value.



01 marks

Fall delay

The fall delay is associated with a gate output transition to a 0 from another value.



01 marks

Turn-off delay

The turn-off delay is associated with a gate output transition to the high impedance value (z) from another value. If the value changes to x, the minimum of the three delays is considered.

01 marks

Three types of delay specifications are allowed.

If only *one* delay is specified, this value is used for all transitions.

```
// Delay of delay_time for all transitions
and #(delay_time) a1(out, i1, i2);
```

and #(5) al(out, il, i2);

If *two* delays are specified, they refer to the rise and fall delay values. The turn-off delay is the minimum of the two delays.

```
// Rise and Fall Delay Specification.
and #(rise_val, fall_val) a2(out, i1, i2);
```

and #(4,6) a2(out, il, i2);

If all *three* delays are specified, they refer to rise, fall, and turn-off delay values. If no delays are specified, the default value is zero.

```
// Rise, Fall, and Turn-off Delay Specification
bufif0 #(rise_val, fall_val, turnoff_val) b1 (out, in, control);
```

Bufif0 # (3,4,5) bl (out, in, control) ;

5b

b. What would be the output of the following for A = 4′b0111 an dB = 4′b1001.
i) &B ii) A<<<2 iii) {A, B} iv) {2{B}} v) A^B vi) A||B vii) A*B viii) A < = B. **(08 Marks)**

i) 0&1&1&1=0

ii) 1100

iii) 10011001

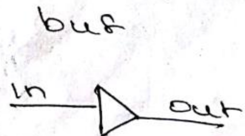
iv) 1110

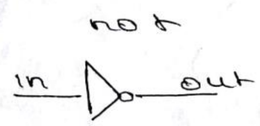
v) 1111

vi) 111111=63d

vii) 1

5c

buf 

not 

Gate instantiation

```
buf b1(out1,in);  } with an instance
buf b2(out2,in);  }

not (out,in); //no instance
```

Truth table

buf	in	out
	0	0
	1	1
	X	X
	Z	X

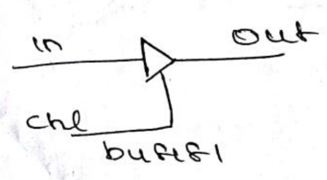
not	in	out
	0	1
	1	0
	X	X
	Z	X

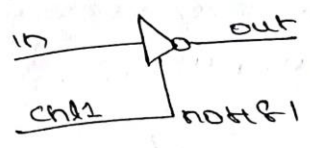
Bufif/ Notif: Gates with an additional control signal on buf & not gates

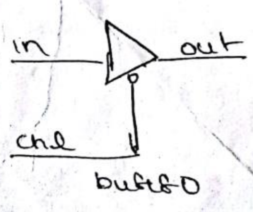
bufif0 notif0

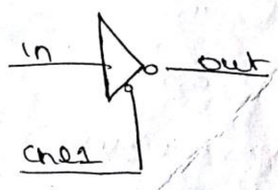
bufif1 notif1

These gates will operate only if their control signal is properly asserted. They propagate Z if their control signal is deasserted.









TRUTH TABLE

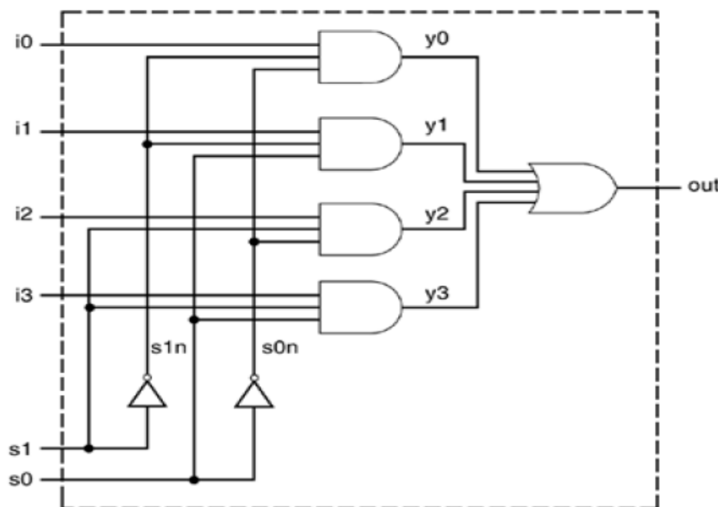
bufif1	0	1	x	z
0	z	0	L	L
1	z	1	H	H
x	z	x	x	x
z	z	x	x	x

bufif0	0	1	x	z
0	0	Ø	L	L
1	1	z	H	H
x	x	z	x	x
z	x	z	x	x

6 a. Design AOI based 4 to 1 multiplexer and write the verilog description and its stimulus. **(10 Marks)**

b. Write the verilog data flow description for 4-bit full adder with carry look –ahead logic. **(10 Marks)**

Ans. 6a.



Verilog Code:-

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
```

```
input s1, s0;
// Internal wire declarations
wire s1n, s0n;
wire y0, y1, y2, y3;
// Gate instantiations
// Create s1n and s0n signals.
not (s1n, s1);
not (s0n, s0);
// 3-input and gates instantiated
and (y0, i0, s1n, s0n);
and (y1, i1, s1n, s0);
and (y2, i2, s1, s0n);
and (y3, i3, s1, s0);
// 4-input or gate instantiated
or (out, y0, y1, y2, y3);
endmodule
```

Stimulus:-

```
module stimulus;
// Declare variables to be connected
// to inputs
reg IN0, IN1, IN2, IN3;
84
reg S1, S0;
// Declare output wire
wire OUTPUT;
// Instantiate the multiplexer
mux4_to_1 mymux(OUTPUT, IN0, IN1, IN2, IN3, S1, S0);
// Stimulate the inputs
// Define the stimulus module (no ports)
initial
begin
// set input lines
IN0 = 1; IN1 = 0; IN2 = 1; IN3 = 0;
#1 $display("IN0= %b, IN1= %b, IN2= %b, IN3= %b\n",IN0,IN1,IN2,IN3);
// choose IN0
S1 = 0; S0 = 0;
#1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);
// choose IN1
S1 = 0; S0 = 1;
#1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);
// choose IN2
S1 = 1; S0 = 0;
#1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);
```

```
// choose IN3
S1 = 1; S0 = 1;
#1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);
end
endmodule
```

Output:-

The output of the simulation is shown below. Each combination of the select signals is tested.
IN0= 1, IN1= 0, IN2= 1, IN3= 0
S1 = 0, S0 = 0, OUTPUT = 1
S1 = 0, S0 = 1, OUTPUT = 0
S1 = 1, S0 = 0, OUTPUT = 1
S1 = 1, S0 = 1, OUTPUT = 0

Ans.6b

Gate-level diagram has to be made first and then code has to be written as:-

```
module fulladd4(sum, c_out, a, b, c_in);
// Inputs and outputs
output [3:0] sum;
output c_out;
input [3:0] a,b;
input c_in;
// Internal wires
wire p0,g0, p1,g1, p2,g2, p3,g3;
wire c4, c3, c2, c1;
// compute the p for each stage
assign p0 = a[0] ^ b[0],
p1 = a[1] ^ b[1],

p2 = a[2] ^ b[2],
p3 = a[3] ^ b[3];
// compute the g for each stage
assign g0 = a[0] & b[0],
g1 = a[1] & b[1],
g2 = a[2] & b[2],
g3 = a[3] & b[3];
// compute the carry for each stage
// Note that c_in is equivalent c0 in the arithmetic equation for
// carry lookahead computation
assign c1 = g0 | (p0 & c_in),
c2 = g1 | (p1 & g0) | (p1 & p0 & c_in),
c3 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & c_in),
c4 = g3 | (p3 & g2) | (p3 & p2 & g1) | (p3 & p2 & p1 & g0) |
(p3 & p2 & p1 & p0 & c_in);
// Compute Sum
assign sum[0] = p0 ^ c_in,
```

```
sum[1] = p1 ^ c1,
sum[2] = p2 ^ c2,
sum[3] = p3 ^ c3;
// Assign carry output
assign c_out = c4;
endmodule
```

Module-4

7 a. Explain blocking and non-blocking assignments with an example. **(10 Marks)**

b. Write a verilog code for clock generation with a period of 20 units using forever loop. **(05 Marks)**

c. Write the differences between the tasks and functions. **(05 Marks)**

Ans. 7a.

Blocking assignment

Blocking assignment statements are executed in the order they are specified in a sequential block. A blocking assignment will not block execution of statements that follow in a parallel block. The = operator is used to specify blocking assignments.

Example:- Blocking Statements

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
//All behavioral statements must be inside an initial or always block
initial
begin
x = 0; y = 1; z = 1; //Scalar assignments
count = 0; //Assignment to integer variables
reg_a = 16'b0; reg_b = reg_a; //initialize vectors
#15 reg_a[2] = 1'b1; //Bit select assignment with delay
#10 reg_b[15:13] = {x, y, z} //Assign result of concatenation
to
// part select of a vector
count = count + 1; //Assignment to an integer (increment)
end
```

The statement y = 1 is executed only after x = 0 is executed. The behavior in a particular block is sequential in a begin-end block if blocking statements are used, because the statements can execute only in sequence. The statement count = count+ 1 is executed last. The simulation times at which the statements are executed are as follows:

- All statements x = 0 through reg_b = reg_a are executed at time 0
- Statement reg_a[2] = 0 at time = 15
- Statement reg_b[15:13] = {x, y, z} at time = 25
- Statement count = count + 1 at time = 25
- Since there is a delay of 15 and 10 in the preceding statements, count = count + 1 will be executed at time = 25 units

Nonblocking Assignments

Nonblocking assignments allow scheduling of assignments without blocking execution of the statements that follow in a sequential block. A <= operator is used to specify nonblocking assignments. This operator has the same symbol as a relational operator, less_than_equal_to. The operator <= is interpreted as a relational operator in an expression and as an assignment operator in the context of a nonblocking assignment. To illustrate the behavior of nonblocking statements and its difference from blocking statements, let us consider example where we convert some blocking assignments to nonblocking assignments, and observe the behavior.

Example Nonblocking Assignments

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
//All behavioral statements must be inside an initial or always block
initial
begin
x = 0; y = 1; z = 1; //Scalar assignments
count = 0; //Assignment to integer variables
reg_a = 16'b0; reg_b = reg_a; //Initialize vectors
reg_a[2] <= #15 1'b1; //Bit select assignment with delay
reg_b[15:13] <= #10 {x, y, z}; //Assign result of concatenation
//to part select of a vector
count <= count + 1; //Assignment to an integer (increment)
end
```

In this example, the statements x = 0 through reg_b = reg_a are executed sequentially at time 0. Then the three nonblocking assignments are processed at the same simulation time.

1. reg_a[2] = 0 is scheduled to execute after 15 units (i.e., time = 15)
2. reg_b[15:13] = {x, y, z} is scheduled to execute after 10 time units (i.e., time = 10)
3. count = count + 1 is scheduled to be executed without any delay (i.e., time = 0)

Thus, the simulator schedules a nonblocking assignment statement to execute and continues to the next statement in the block without waiting for the nonblocking statement to complete execution. Typically, nonblocking assignment statements are executed last in the time step in which they are scheduled, that is, after all the blocking assignments in that time step are executed.

Ans. 7b.

Code for Clock generation

```
module gen_clk(clock);
reg clock;
initial
begin
clock = 1'b0;
forever #10 clock = ~clock; //Clock with period of 20 units
end
endmodule
```

Ans. 7c

Tasks and functions serve different purposes in Verilog.

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

8 a. Discuss sequential and parallel blocks with examples. **(10 Marks)**
b. Write a verilog program for 8 : 1 multiplexer using case statement. **(10 Marks)**

Ans.8a

Block statements are used to group multiple statements to act together as one.
There are two types of blocks: sequential blocks and parallel blocks.
Sequential blocks
The keywords begin and end are used to group statements into sequential blocks. Sequential blocks have the following characteristics:
• The statements in a sequential block are processed in the order they are specified. A statement is executed only after its preceding statement completes execution (except for nonblocking assignments with intra-assignment timing control).
• If delay or event control is specified, it is relative to the simulation time when the previous statement in the block completed execution.

Statements in the sequential block execute in order. In Illustration 1, the final values are x = 0, y= 1, z = 1, w = 2 at simulation time 0. In Illustration 2, the final values are the same except that the simulation time is 35 at the end of the block.

Example Sequential Blocks
```
//Illustration 1: Sequential block without delay
reg x, y;
```

```
reg [1:0] z, w;
initial
begin
x = 1'b0;
y = 1'b1;
z = {x, y};
w = {y, x};
152
end
//Illustration 2: Sequential blocks with delay.
reg x, y;
reg [1:0] z, w;
initial
begin
x = 1'b0; //completes at simulation time 0
#5 y = 1'b1; //completes at simulation time 5
#10 z = {x, y}; //completes at simulation time 15
#20 w = {y, x}; //completes at simulation time 35
end
```

Parallel blocks

Parallel blocks, specified by keywords fork and join, provide interesting simulation features.
Parallel blocks have the following characteristics:

- Statements in a parallel block are executed concurrently.
- Ordering of statements is controlled by the delay or event control assigned to each statement.
- If delay or event control is specified, it is relative to the time the block was entered.

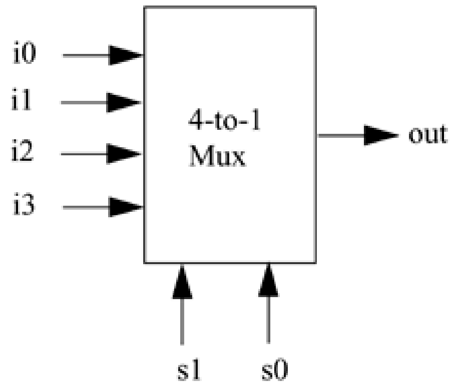
All statements in a parallel block start at the time when the block was entered. Thus, the order in which the statements are written in the block is not important.

Example Parallel Blocks

```
//Example 1: Parallel blocks with delay.
reg x, y;
reg [1:0] z, w;
initial
fork
x = 1'b0; //completes at simulation time 0
#5 y = 1'b1; //completes at simulation time 5
#10 z = {x, y}; //completes at simulation time 10
#20 w = {y, x}; //completes at simulation time 20
join
```

The result of simulation remains the same except that all statements start in parallel at time 0. Hence, the block finishes at time 20 instead of time 35.
Parallel blocks provide a mechanism to execute statements in parallel. However, it is important to be careful with parallel blocks because of implicit race conditions that might arise if two statements that affect the same variable complete at the same time.

Ans. 8b



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

4-to-1 Multiplexer with Case Statement

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;
reg out;
always @(s1 or s0 or i0 or i1 or i2 or i3)
case ({s1, s0}) //Switch based on concatenation of control signals
2'd0 : out = i0;
2'd1 : out = i1;
2'd2 : out = i2;
2'd3 : out = i3;
default: $display("Invalid control signals");
endcase
endmodule
```

Module-5

9 a. Write the verilog description for D – flipflop using assign and deassign procedural continuous assignments. **(10 Marks)**

b. Explain defparam statement with an example. **(10 Marks)**

Ans.9a

The keywords assign and deassign are used to express the first type of procedural continuous assignment. The left-hand side of procedural continuous assignments can be only be a register or a concatenation of registers. It cannot be a part or bit select of a net or an array of registers. Procedural continuous assignments override the effect of regular procedural assignments. Procedural continuous assignments are normally used for controlled periods of time.

D-Flipflop with Procedural Continuous Assignments

```
module edge_dff(q, qbar, d, clk, reset);
output q,qbar;
input d, clk, reset;
reg q, qbar; //declare q and qbar are registers
always @(negedge clk) //assign value of q & qbar at active edge of clock.
begin
```

```
q = d;
qbar = ~d;
end
always @(reset) //Override the regular assignments to q and qbar
//whenever reset goes high. Use of procedural continuous assignments.
if(reset)
begin
assign q = 1'b0;
assign qbar = 1'b1;
end
else
begin //If reset goes low, remove the overriding values by deassigning the
//registers. After this the regular assignments q = d and qbar = ~d will be able
//to change the registers on the next negative edge of clock.
deassign q;
deassign qbar;
end
endmodule
```

In above code, we overrode the assignment on q and qbar and assigned new values to them when the reset signal went high. The register variables retain the continuously assigned value after the deassign until they are changed by a future procedural assignment. The assign and deassign constructs are now considered to be a bad coding style and it is recommended that alternative styles be used in Verilog HDL code.

Ans. 9b
Parameters can be defined in a module definition However, during compilation of Verilog modules, parameter values can be altered separately for each module instance. This allows us to pass a distinct set of parameter values to each module during compilation regardless of predefined parameter values. There are two ways to override parameter values: through the defparam statement or through module instance parameter value assignment.

defparam Statement

Parameter values can be changed in any module instance in the design with the keyword defparam. The hierarchical name of the module instance can be used to override parameter values.

Example Defparam Statement

```
//Define a module hello_world
module hello_world;
parameter id_num = 0; //define a module identification number = 0
initial //display the module identification number
$display("Displaying hello_world id number = %d", id_num);
endmodule

//define top-level module
module top;
//change parameter values in the instantiated modules
//Use defparam statement
defparam w1.id_num = 1, w2.id_num = 2;
//instantiate two hello_world modules
hello_world w1();
hello_world w2();
endmodule
```

In above example, the module hello_world was defined with a default id_num = 0.

However, when the module instances w1 and w2 of the type hello_world are created, their id_num values are modified with the defparam statement. If we simulate the above design, we would get the following output:

```
Displaying hello_world id number = 1
Displaying hello_world id number = 2
```

Multiple defparam statements can appear in a module. Any parameter can be overridden with the defparam statement. The defparam construct is now considered to be a bad coding style and it is recommended that alternative styles be used in Verilog HDL code.

10 a. What is logic synthesis? Explain the flow diagram for the designer's mind as the logic synthesis tool. **(10 Marks)**

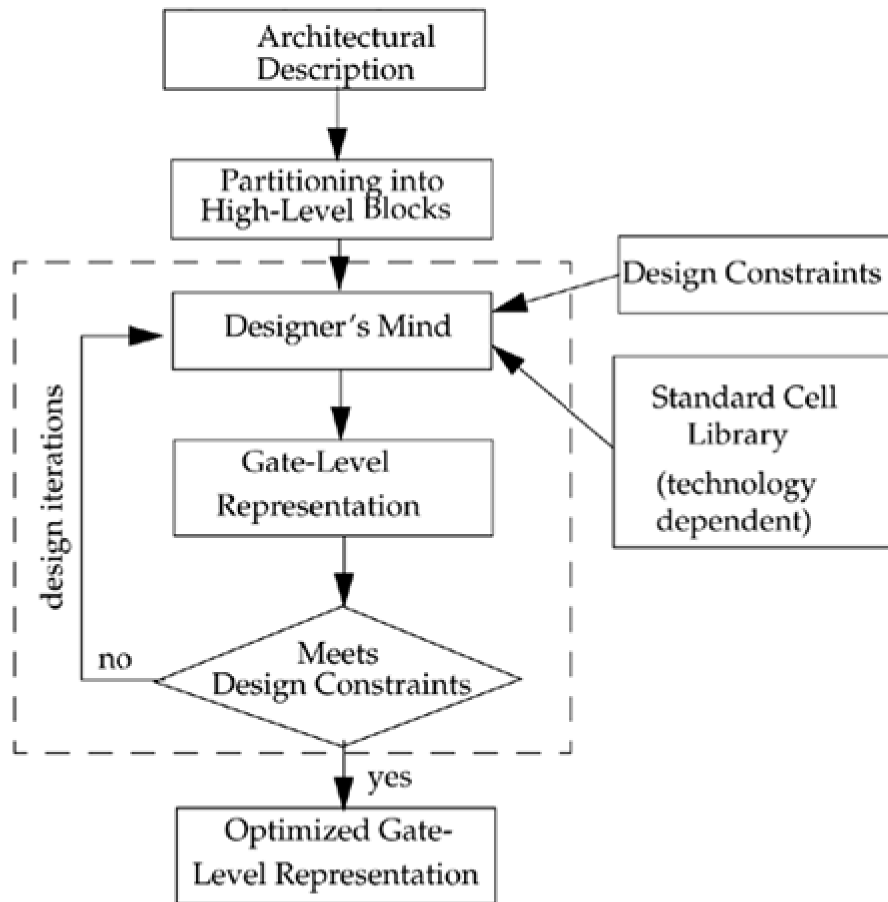
b. What will be the following statements translate to when run on a logic synthesis tool :
Assign {C-out, sum } = a + b + C in ;
Assign out = (s) ? i1 : i0 ; **(10 Marks)**

Ans. 10a

Logic Synthesis:-

Logic synthesis is the process of converting a high-level description of the design into an optimized gate-level representation, given a standard cell library and certain design constraints. A standard cell library can have simple cells, such as basic logic gates like and, or, and nor, or macro cells, such as adders, muxes, and special flipflops. A standard cell library is also known as the technology library.

Flow Diagram for designer's mind as the logic synthesis tool:



Logic synthesis always existed even in the days of schematic gate-level design, but it was always done inside the designer's mind. The designer would first understand the architectural description. Then he would consider design constraints such as timing, area, testability, and power. The designer would partition the design into high-level blocks, draw them on a piece of paper or a computer terminal, and describe the functionality of the circuit. This was the high-level description. Finally, each block would be implemented on a hand-drawn schematic, using the cells available in the standard cell library. The last step was the most complex process in the design flow and required
several time-consuming design iterations before an optimized gate-level representation that met all design constraints was obtained. Thus, the designer's mind was used as the
logic synthesis tool, as illustrated in above Fig.

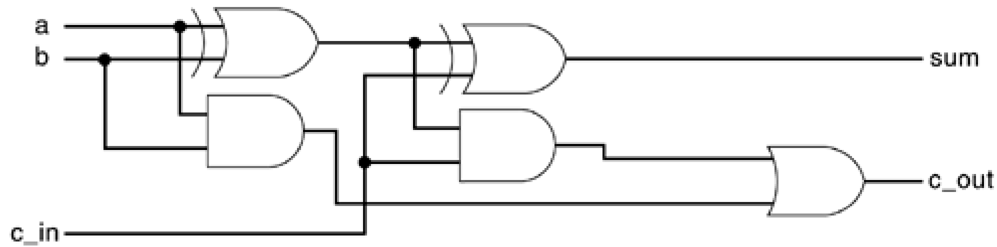
Ans. 10b

Logic synthesis tools frequently interpret the constructs and translate them to logic gates.

assign statement is one those construct.
The assign construct is the most fundamental construct used to describe combinational logic at an RTL level.
If arithmetic operators are used, each arithmetic operator is implemented in terms of arithmetic hardware blocks available to the logic synthesis tool. A 1-bit full adder is implemented below:-

```
assign {c_out, sum} = a + b + c_in;
```

Assuming that the 1-bit full adder is available internally in the logic synthesis tool, the above assign statement is often interpreted by logic synthesis tools as follows:



If a multiple-bit adder is synthesized, the synthesis tool will perform optimization and the designer might get a result that looks different from the above figure.

If a conditional operator ? is used, a multiplexer circuit is inferred.

```
assign out = (s) ? i1 : i0;
```

It frequently translates to the gate-level representation as:-