USN										

Internal Assessment Test 3 – Nov. 2019

Sub:	Verilog HDL					Sub Code:	17EC53	Branch:	ECE
Date:	18/11/19	Duration:	90 min's	Max Marks:	50	Sem / Sec:	5th D		OBE

Answer any FIVE FULL Questions

		MARKS	CO	RBT
1.	**What are blocking and non-blocking assignment statements? Explain with examples.**	[10]	CO3	L2

Blocking Assignments
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. Sequential and Parallel Blocks. The '=' operator is used to specify blocking assignments.

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
     integer count;

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

Note that for procedural assignments to registers, if the right-hand side has more bits than the register variable, the right-hand side is truncated to match the width of the register variable. The least significant bits are selected and the most significant bits are discarded. If the right-hand side has fewer bits, zeros are filled in the most significant bits of the register variable.

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. Note that 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

```
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
```

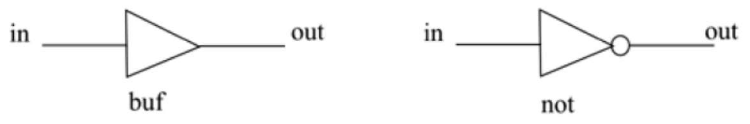
In this example, the statements x = 0 through reg_b = reg_a

2.	**Explain the different types of buffers and not gates supported by Verilog HDL. Also write the help of truth table, logic symbol and instantiation examples.**	**[10]**	CO3	L1

Buf/not gates have one scalar input and one or more scalar outputs. The last terminal in the port list is connected to the input. Other terminals are connected to the outputs.

Two basic buf/not gate primitives are provided in Verilog: `buf, not`

The symbols for these logic gates are shown in Figure



Gate Instantiations of Buf/Not Gates

```
buf b1(OUT1, IN);
not n1(OUT1, IN);
buf b1_2out(OUT1, OUT2, IN);
not (OUT1, IN); // legal gate instantiation
```

The truth tables for these gates are very simple. Truth tables for gates with one input and one output are

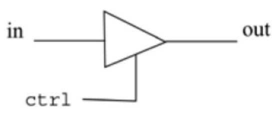
buf	in	out
	0	0
	1	1
	x	x
	z	x

not	in	out
	0	1
	1	0
	x	x
	z	x

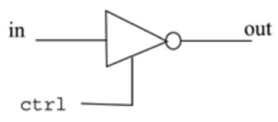
Bufif/notif:

Gates with an additional control signal on buf and not gates are also available: `bufif1, notif1, bufif0, notif0.`

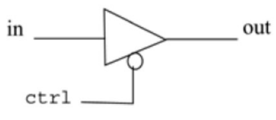
These gates propagate only if their control signal is asserted. They propagate **z** if their
control signal is deasserted:



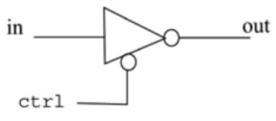
bufif1



notif1



bufif0



notif0

The truth tables for these gates are

	bufif1	ctrl 0	ctrl 1	ctrl x	ctrl z
in	0	z	0	L	L
in	1	z	1	H	H
in	x	z	x	x	x
in	z	z	x	x	x

	bufif0	ctrl 0	ctrl 1	ctrl x	ctrl z
in	0	0	z	L	L
in	1	1	z	H	H
in	x	x	z	x	x
in	z	x	z	x	x

	notif1	ctrl 0	ctrl 1	ctrl x	ctrl z
in	0	z	1	H	H
in	1	z	0	L	L
in	x	z	x	x	x
in	z	z	x	x	x

	notif0	ctrl 0	ctrl 1	ctrl x	ctrl z
in	0	1	z	H	H
in	1	0	z	L	L
in	x	x	z	x	x
in	z	x	z	x	x

```
//Instantiation of bufif gates.
bufif1 b1 (out, in, ctrl);
bufif0 b0 (out, in, ctrl);
//Instantiation of notif gates
notif1 n1 (out, in, ctrl);
notif0 n0 (out, in, ctrl);
```

3 (a) Write Verilog HDL program of 4-bit synchronous up counter. **[06]** CO3 L3

```
module counter(Q , clock, clear);
output [3:0] Q;
input clock, clear;
reg [3:0] Q;
always @( negedge clock)
begin
if (clear)
Q <= 4'd0;
else
Q <= Q + 1;
end
endmodule
```

(b) Write Verilog HDL program of 4:1 mux using If-else statement. **[04]** CO3 L3

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
output out;
input i0, i1, i2, i3;
input s1, s0;
reg out;
always @(s1 or s0 or i0 or i1 or i2 or i3)
begin
if(s1==0 && s0==0)
i0=1’b1;
else if(s1==0 && s0==1)
i1=1’b1
else if(s1==1 && s0==0)
i2=1’b1;
else if(s1==1 && s0==1)
i3=1’b1;
end
endmodule
```

4. Explain genvar, generate statement with syntax and an example. **[10]** CO3 L1

Generate statements allow Verilog code to be generated dynamically at elaboration time before the simulation begins. This facilitates the creation of parametrized models. Generate statements are particularly convenient when the same operation or module instance is repeated for multiple bits of a vector, or when certain Verilog code is conditionally included based on parameter definitions.
Generate statements allow control over the declaration of variables, functions, and tasks, as well as control over instantiations. All generate instantiations are coded with a module scope and require the keywords **generate - endgenerate**.

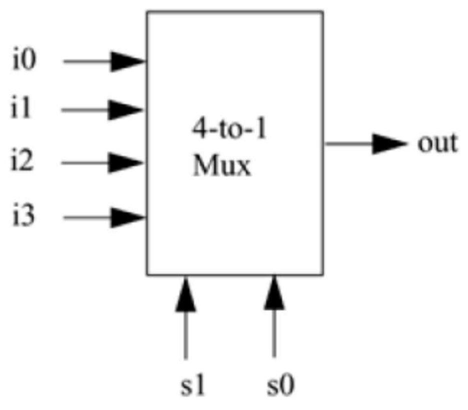
Generated instantiations can be one or more of the following types:
• Modules
• User defined primitives
• Verilog gate primitives
• Continuous assignments
• initial and always blocks
Generated declarations and instantiations can be conditionally instantiated into a design.
Generated variable declarations and can be referenced hierarchically.
Example:
```
module bitwise_xor (out, i0, i1);
parameter N = 32;
output [N-1:0] out;
input [N-1:0] i0, i1;
genvar j;
generate for (j=0; j<N; j=j+1) begin: xor_loop
xor g1 (out[j], i0[j], i1[j]);
end
endgenerate
endmodule
```

5 (a) 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). **[05]** CO3 L3

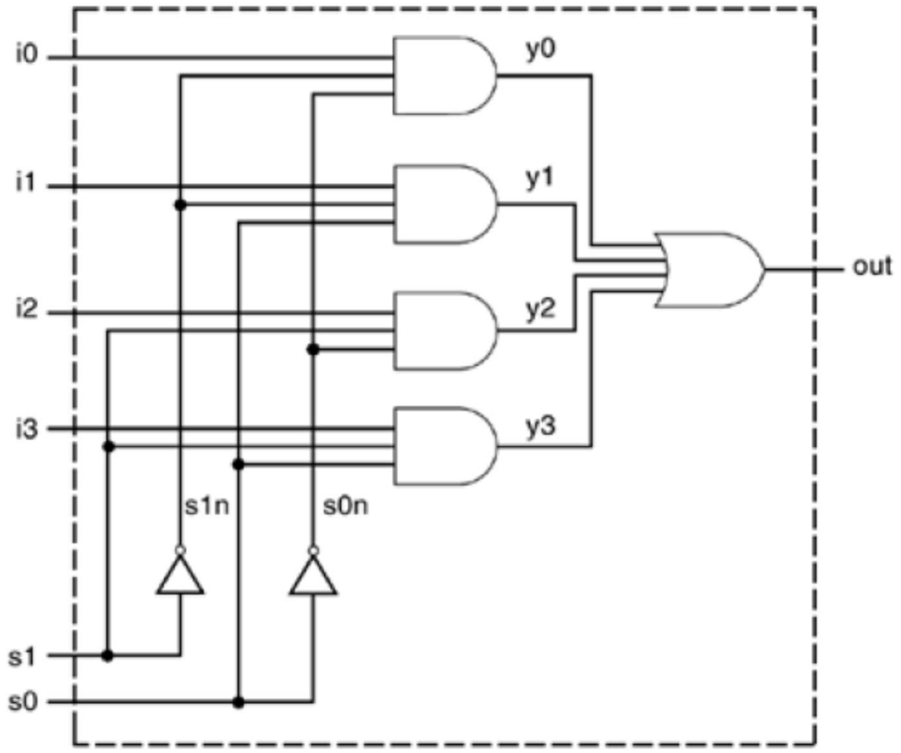
reg oscillate;
initial
begin
oscillate = 1’b0;
forever #30 oscillate = ~ oscillate;
end

(b) Write gate level description to implement function $y = a.b + c$, with 5 and 4 time units of gate delay for AND and OR gate respectively. **[05]** CO3 L3
```
module D (out, a, b, c);
output out;
input a,b,c;
wire e;
and #(5) a1(e, a, b); //Delay of 5 on gate a1
or #(4) o1(out, e,c); //Delay of 4 on gate o1
endmodule
```

6. Use gate level description of Verilog HDL to design 4 to 1 multiplexer. Write truth table, top-level block, logic expression and logic diagram. Also write the stimulus block for the same. **[10]** CO3 L3



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



```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
output out;
input i0, i1, i2, i3;
input s1, s0;
wire s1n, s0n;
wire y0, y1, y2, y3;
not (s1n, s1);
not (s0n, s0);
and (y0, i0, s1n, s0n);
and (y1, i1, s1n, s0);
and (y2, i2, s1, s0n);
and (y3, i3, s1, s0);
or (out, y0, y1, y2, y3);
endmodule
```

Stimulus for Multiplexer

```
module stimulus;
reg IN0, IN1, IN2, IN3;
reg S1, S0;
wire OUTPUT;
mux4_to_1 mymux(OUTPUT, IN0, IN1, IN2, IN3, S1, S0);
initial
begin
IN0 = 1; IN1 = 0; IN2 = 1; IN3 = 0;
#1 $display("IN0= %b, IN1= %b, IN2= %b, IN3=
%b\n",IN0,IN1,IN2,IN3);
S1 = 0; S0 = 0;
#1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0,
OUTPUT);
S1 = 0; S0 = 1;
#1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0,
OUTPUT);
S1 = 1; S0 = 0;
#1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0,
OUTPUT);
S1 = 1; S0 = 1;
#1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0,
OUTPUT);
end
endmodule
```

7. A full subtractor has three l-bit inputs *x, y,* and z (previous borrow) and two 1-bit outputs D (difference) and B (borrow). The logic equations for D and B are as follows: CO3 L3

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

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

Write the full Verilog description for the full subtractor module using data-flow modeling, including I/0 ports (Remember that + in logic equations corresponds to a logical 'or' operator (|) in dataflow). Instantiate the subtractor inside a stimulus block and test all eight possible combinations of *x*, y, and *z*.

```
module full_subtractor(output B, D, input x, y, z);
assign D = (~x & ~y & z) | (~x & y & ~z )| (x & ~y & ~z) | (x & y & z);
assign B = (~x & y) | (~x & z )| (y & z);
endmodule
```

Stimulus:

```
module stimulus;
reg x, y;
reg z;
wire D, B;
full_subtractor FS (B, D, x, y, z);
initial
begin
```

[10]

```
$monitor($time," x= %b, y=%b, z= %b, --- B= %b, D=
%b\n",x, y, z, B, D);
end
initial
begin
x = 0; y = 0; z = 0;
#5 x = 0; y = 0; z = 1;
#5 x = 0; y = 1; z = 0;
#5 x = 0; y = 1; z = 1;
#5 x = 1; y = 0; z = 0;
#5 x = 1; y = 0; z = 1;
#5 x = 1; y = 1; z = 0;
#5 x = 1; y = 1; z = 1;
end
endmodule
```

8. With syntax explain conditional, branching and loop statements available in Verilog HDL behavioral description. CO3 L1

Conditional statements are used for making decisions based upon certain conditions. These conditions are used to decide whether or not a statement should be executed.
Formal Syntax Definition.

```
//Type 1 conditional statement. No else statement.
//Statement executes or does not execute.
if (<expression>) true_statement ;
//Type 2 conditional statement. One else statement
//Either true_statement or false_statement is evaluated
if (<expression>) true_statement ; else false_statement
;
//Type 3 conditional statement. Nested if-else-if.
//Choice of multiple statements. Only one is executed.
if (<expression1>) true_statement1 ;
else if (<expression2>) true_statement2 ;
else if (<expression3>) true_statement3 ;
else default_statement ;
```

Conditional Statement Examples

```
//Type 1 statements
if(!lock) buffer = data;
if(enable) out = in;
//Type 2 statements
if (number_queued < MAX_Q_DEPTH)
begin
data_queue = data;
number_queued = number_queued + 1;
end
else
$display("Queue Full. Try again");
//Type 3 statements
//Execute statements based on ALU control signal.
if (alu_control == 0)
```

[10]

```
y = x + z;
else if(alu_control == 1)
y = x - z;
else if(alu_control == 2)
y = x * z;
else
$display("Invalid ALU control signal");
```

Multiway Branching

The keywords case, endcase, and default are used in the case statement..

```
case (expression)
alternative1: statement1;
alternative2: statement2;
alternative3: statement3;
...
...
default: default_statement;
endcase
```

Each of statement1, statement2 , default_statement can be a single statement or a block
of multiple statements. A block of multiple statements must be grouped by keywords
begin and end. The expression is compared to the alternatives in the order they are written. For the first alternative that matches, the corresponding statement or block is
executed. If none of the alternatives matches, the default_statement is executed.

```
reg [1:0] alu_control;
...
...
case (alu_control)
2'd0 : y = x + z;
2'd1 : y = x - z;
2'd2 : y = x * z;
default : $display("Invalid ALU control signal");
endcase
```

There are two variations of the case statement. They are denoted by keywords, casex and
casez.

• casez treats all z values in the case alternatives or the case expression as don't cares. All bit positions with z can also represented by '?' in that position.
• casex treats all x and z values in the case item or the case expression as don't cares.

Loops

While Loop

The keyword while is used to specify this loop. The while loop executes until the while expression is not true. If the loop is entered when the while-expression is not true, the loop is not executed at all.

```
integer count;
```

```
initial
begin
count = 0;
while (count < 128) //Execute loop till count is 127.
//exit at count 128
begin
$display("Count = %d", count);
count = count + 1;
end
end
```

For Loop

The keyword for is used to specify this loop. The for loop contains three parts:

• An initial condition
• A check to see if the terminating condition is true
• A procedural assignment to change value of the control variable

```
integer count;
initial
for ( count=0; count < 128; count = count + 1)
$display("Count = %d", count);
```

The initialization condition and the incrementing procedural assignment are included in the for loop and do not need to be specified separately. Thus, the for loop provides a more compact loop structure than the while loop. Note, however, that the while loop is more general-purpose than the for loop. The for loop cannot be used in place of the while loop in all situations.

Repeat Loop

The keyword repeat is used for this loop. The repeat construct executes the loop a fixed number of times.

```
integer count;
initial
begin
count = 0;
repeat(128)
begin
$display("Count = %d", count);
count = count + 1;
end
end
```

Forever loop

The keyword forever is used to express this loop. The loop does not contain any expression and executes forever until the $finish task is encountered.

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